POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints

Connor Imes
University of Chicago
ckimes@cs.uchicago.edu

David H.K. Kim
University of Chicago
hongk@cs.uchicago.edu

Martina Maggio
Lund University
martina@control.lth.se

Henry Hoffmann
University of Chicago
hankhoffmann@cs.uchicago.edu

***Abstract*—Embedded real-time systems must meet timing constraints while minimizing energy consumption. To this end, many energy optimizations are introduced for specific platforms or specific applications. These solutions are not portable, however, and when the application or the platform change, these solutions must be redesigned. Portable techniques are hard to develop due to the varying tradeoffs experienced with different application/platform configurations. This paper addresses the problem of finding and exploiting general tradeoffs, using control theory and mathematical optimization to achieve energy minimization under soft real-time application constraints. The paper presents POET, an open-source C library and runtime system that takes a specification of the platform resources and optimizes the application execution. We test POET's ability to portably deliver predictable timing and energy reduction on two embedded systems with different tradeoff spaces – the first with a mobile Intel Haswell processor, and the second with an ARM big.LITTLE System on Chip. POET achieves the desired latency goals with small error while consuming, on average, only 1.3% more energy than the dynamic optimal oracle on the Haswell and 2.9% more on the ARM. We believe this open-source, library-based approach to resource management will simplify the process of writing portable, energy-efficient code for embedded systems.**

I. Introduction

Many embedded developers must design applications to meet timing constraints while minimizing energy consumption. To accommodate these goals, embedded hardware platforms are becoming increasingly configurable. Almost all support dynamic voltage and frequency scaling (DVFS), which trades processor performance for reduced power consumption [39]. Other designs aggressively power-gate cores, so that unused cores can be idled to further reduce power consumption [42]. Recent designs, such as the Exynos5 SoC (in the Samsung Galaxy line), support heterogeneous cores where different cores have different performance and energy tradeoffs [27, 45].

The abundance of configurable resources in embedded multicores creates a wide range of power and timing tradeoffs. Unfortunately, the more configurable the system, the harder it is to schedule resource usage to meet the required timing constraints and turn power/timing tradeoffs into energy reduction. While many energy-aware resource strategies exist, they tend to be specialized for a particular application (*e.g.,* a web-browser [50, 58], a wireless sensor network [10], a web server [4, 44]), or a specific embedded platform (*e.g.,* a particular phone [46] or processor [40, 52]).

Creating application and platform-independent resource allocation frameworks is a challenging problem. Various applications use resources with differing energy consumptions. Different platforms expose a diversity of resources. Furthermore, it is not just the resources that differ from platform to platform – the whole shape of the timing/energy tradeoff space can change (see Section II). As a result, most embedded developers design resource allocation for particular applications and platforms. If the application is moved to a new platform, much of it must be redesigned and re-implemented to use the new resources efficiently.

A. A Library and Runtime for Portable Energy Optimization

This paper addresses the need for a *portable* resource allocation scheme that achieves predictable application timing with minimal energy consumption by presenting POET: the Performance with Optimal Energy Toolkit. POET is a C library and runtime which automatically allocates resources in embedded platforms. POET is portable – a POET application will run on different embedded platforms with no code changes and achieve both predictable timing and near optimal energy. At runtime, POET measures current timing and power consumption, uses feedback control to ensure the timing goals are met, and solves a linear optimization problem to select minimal energy resource allocations based on a user-provided specification. Adding POET to an application requires only a few (12 - 30) additional lines of code, but once integrated, these applications achieve portable energy reduction while meeting soft real-time constraints.

B. Summary of Results

We implement POET and evaluate it using eight standard benchmarks and two embedded systems – a Sony Vaio tablet with an embedded Intel Haswell processor and an ODROID development board with an ARM big.LITTLE processor. We make the library implementation, the benchmark patches, and all configuration files available as open source so that others can evaluate or use POET themselves[1]. These benchmarks were not originally designed to provide any timing predictability, yet we find that POET achieves:

This work was funded by the U.S. Government under the DARPA PERFECT program, by the Dept. of Energy under DOE DE-AC02-06CH11357, and by the NSF under CCF 1439156. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. This work was also partially supported by the Swedish Research Council (VR) with the projects "Cloud Control" and "Power and temperature control for large-scale computing infrastructures", and through the LCCC Linnaeus and ELLIIT Excellence Centers.

[1]Everything is available by following links from the project web page at http://poet.cs.uchicago.edu/

TABLE I: Two embedded platforms with different configurable components.

Platform	Processor	Cores	Core Types	Speeds (GHz)	TurboBoost	HyperThreads	Idle Power (W)	Configurations
Sony Vaio	Intel Haswell	2	1	.6–1.5	yes	yes	2.5	45
ODROID-XU+E	Samsung Exynos5 Octa	8	2 (A15 & A7)	.8–1.6 (A15) .5-1.2 (A7)	no	no	0.12	69

- **Ease of Use**: Adding POET to applications requires only a few additional lines of code. (See Section V.)
- **Predictable Latency**: We set various latency targets and measure the error between the desired latency and POET's achieved latency. On both the Vaio and the ODROID, the average error across all targets and all benchmarks is less than 2%. (See Section VI-A.)
- **Energy Savings**: We run all applications in all configurations for both platforms, and we use this data to compute the minimal possible energy consumption for every input. This calculation tells us the true minimal energy given perfect knowledge of the application and platform. On the Vaio, POET exceeds this minimal energy by only 1.3% on average; on the ODROID it exceeds minimal by only 2.9%. These values include the overhead of the POET runtime. (See Section VI-B.) In addition, we compare POET's general approach to one that uses just DVFS. POET's energy savings compared to DVFS vary tremendously for different platforms, applications and latency targets, but POET can provide as much as a 7× reduction in energy compared to approaches that use DVFS only. (See Section VI-C.)
- **Adaptability**: For applications with distinct phases, POET turns periods of reduced computational demand into considerable energy savings by reducing resource usage. (See Section VI-D). When POET applications are co-scheduled with other applications, the POET runtime automatically adjusts resource usage to ensure the latency goals are met. (See Section VI-E.)

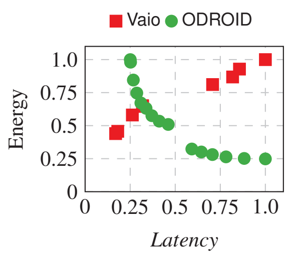
C. *This paper makes the following contributions*

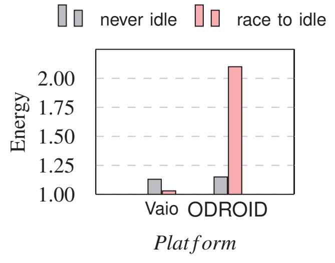
- Design for portable, energy-aware resource allocation across embedded platforms.
- Description of POET's design and implementation.
- Empirical evaluation of POET on two embedded platforms with eight different applications.
- Open-source release of POET, with patches to apply it to common benchmarks.

The rest of this paper is organized as follows. Section II presents background on embedded platforms and motivates the need for portable resource allocation. Section III presents the mathematical framework behind POET's control and optimization strategies. Section IV describes how this framework is implemented. Section V illustrates how we apply POET to real applications and systems and Section VI presents our empirical evaluation. Section VII compares POET to prior work. The paper concludes in Section VIII.

II. Background and Motivation

This section motivates the need for portable, energy-aware resource management. We evaluate the timing and energy tradeoffs of a video encoder on two embedded platforms, a Sony Vaio tablet and an ODROID development board. The two platforms not only have different resources for management, but also have latency and energy tradeoffs with





(a) Energy/Latency Tradeoffs (b) Energy Consumption

Fig. 1: Energy/latency tradeoffs.

different topologies. Thus, resource allocation strategies that save energy on one are wasteful on the other.

Our example features a video encoder, composed of jobs, where each job encodes a frame. We instrument the encoder to report job latency and we measure the platform's energy consumption over time. The two platforms have different configurable resources, shown in Table I. The Vaio allows configuration of the number of active cores, the number of hyperthreads per core, the speed of the processors, and the use of TurboBoost. The ODROID, supports configuration of the number of active cores, their clock speed, and whether the application uses the "big" (Cortex-A15 high performance, high power) or "LITTLE" (Cortex-A7 low performance, low power) cores.

Figure 1a shows the tradeoffs between energy consumption and latency. The x-axis shows the average latency (normalized to 1 – the empirically determined worst case). The y-axis shows energy (normalized to 1 – the highest measured energy). The two plots show the very different tradeoffs for the Vaio and the ODROID with each point representing a different configuration. For the Vaio, energy increases as frame latency increases; *i.e.,* a slower job wastes energy. For the ODROID, energy decreases as frame latency increases; *i.e.,* slower encodings save energy.

The different shapes of these tradeoff spaces lead to different optimal resource allocation strategies. Empirical studies show that the *race to idle* heuristic, which makes all resources available and then idles after completing a job, is near optimal on systems like the Vaio [3, 19, 21, 24, 38]. On systems like the ODROID, recent approaches save energy by keeping the system constantly busy and *never idle* [11, 21, 24, 29, 32].

To demonstrate the importance of choosing the right strategy, we analyze the two heuristics on both platforms and compare their energy consumption to optimal. We set a latency target equal to twice the minimum latency and measure the energy consumption of encoding 500 video frames using each heuristic. Figure 1b shows the results, normalized to the optimal energy found by measuring every possible resource configuration. Both heuristics meet the latency target, but their energy consumptions vary tremendously. On the Vaio, *race to idle* is near optimal, but *never idle* consumes 13% more energy. Conversely, *never idle* is near optimal for the ODROID, but *race to idle* consumes 2× more energy.

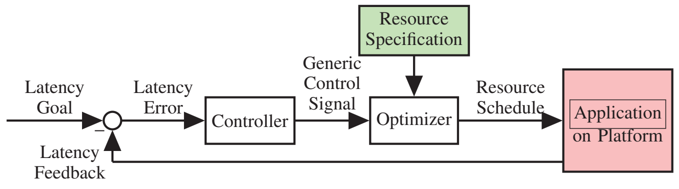


Fig. 2: Overview of the POET runtime.

These results demonstrate that resource allocation strategy greatly affects energy consumption, and more importantly, that heuristic solutions are **not portable** across devices. *These two points motivate the need for an approach like POET, which provides near optimal resource allocation while remaining platform-independent.* POET's runtime uses control theory to meet timing constraints and linear programming to minimize energy consumption. A POET user does not need to be a control or optimization expert, but simply make small changes to their application code. POET makes it easy for embedded developers to write portable applications providing predictable timing and minimal energy across a range of devices.

III. General and Portable Resource Allocation

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

Figure 2 illustrates our approach. The application informs the runtime of its target job latency. Measuring each job start and completion time, POET's runtime computes a latency error and passes it to a **controller**. The controller uses the error to calculate a generic control signal, indicating how much the application speed should be altered. This signal is used by the **optimizer**, together with the specification of available resources, to schedule resource usage so that the desired speed is achieved and energy consumption is minimized. Both the controller and the optimizer are designed independently of any particular application and system. The only assumption made is that applications are composed of repeated jobs, each with a (soft real-time) deadline, or desired latency. As we target multicore platforms, we assume each job may be processed by multiple, communicating threads.

A. Controller

The controller cancels the error between the desired job deadline d_r and its measured latency $d_m(t)$ at time t. We consider the error $e(t)$ using the abstraction of the job speed, where the required speed is $1/d_r$ and the measured speed at time t is $1/d_m(t)$.

$$e(t) = \frac{1}{d_r} - \frac{1}{d_m(t)} \tag{1}$$

POET models latency as

$$d_m(t) = \frac{1}{s(t-1) \cdot b(t-1)} \tag{2}$$

where $s(t)$ is the speedup to achieve with respect to $b(t)$, the base application speed, *i.e.,* the speed of the application when it uses the minimum amount of resources. POET's controller uses the error computed with Eqn. 1 to calculate the control signal $s(t)$ in Eqn. 2 so that the speedup cancels the error. The controller intervenes at discrete time instants and implements the *integral control law* [18]:

$$s(t) = s(t-1) + (1-p) \cdot \frac{e(t)}{b(t)} \tag{3}$$

where p is a configurable *pole* of the closed loop characteristic equation [14]. To ensure the controller reaches a steady state where the error is eliminated without oscillations, the value of p should lay in the unit circle, *i.e.,* $0 \leq p < 1$. p is user-configurable. A small p makes the controller highly reactive, while a large p makes it slow to respond to external changes. However, a large p ensures robustness with respect to transient fluctuations and may be beneficial for very noisy systems. A small p will cause the controller to react quickly, potentially producing a noisy control signal.

The parameter $b(t)$ represents the application's base speed, which directly influences the controller. Different applications will have different base speeds. Applications may also experience *phases*, where base speed changes over time. To accommodate these situations, POET continually estimates base speed using a Kalman filter [53], which adapts $b(t)$ of Eqn. 3 to the current application behavior. More details on the Kalman filter are presented in Appendix A (Eqn. 10).

POET's control formulation is independent of a particular application as it uses the Kalman filter to estimate the application base speed. Unlike prior work, this controller does not reason about a particular set of resources, but computes a generic control signal $s(t)$. POET provides formal guarantees about its steady-state convergence and robustness without requiring users to understand control theory.

B. Optimizer

The optimizer turns the generic control signal computed by the controller into a system-specific resource allocation strategy, translating the speedup $s(t)$ computed with Eqn. 3 into a *schedule* for the available resources. The schedule is computed for the next τ time units. To meet the requirement on the target latency and avoid deadline misses, POET ensures that the application completes $I(t)$ jobs in the next interval[2], with $I(t) = \tau \cdot s(t) \cdot b(t)$.

As shown in Figure 2, the optimizer takes, as input, a resource specification containing the set of available system configurations. There are C possible configurations in the system and by convention, we number the configurations from 0 to $C-1$. We use $c = 0$ to indicate the configuration where the least amount of resources is given to the application, corresponding to a low-power idle state or sleep state when available. In contrast, configuration $C-1$ maximizes the resource availability. Each configuration c is associated with a power consumption p_c and speedup s_c.

[2]In our implementation, both the jobs to be completed and an acceptable scheduling period τ are specified by the application.

Algorithm 1 Finding a Minimal-Energy Schedule.

```
Require: $C$                                   ▷ system configurations
Require: $s(t)$                                ▷ given by Eqn. 3
Require: $\tau$                                ▷ given by application
  $under = \{c \mid s_c \leq s(t)\}$
  $over = \{c \mid s_c > s(t)\}$
  $candidates = \{\langle u, o\rangle \mid u \in under, o \in over\}$
  $energy = \infty$
  $optimal = \langle -1, -1\rangle$
  $schedule = \langle 0, 0\rangle$

  for $\langle u, o\rangle \in candidates$ do                ▷ loop over all pairs
    $\tau_u = \frac{\tau \cdot s(t) - \tau \cdot s_o}{s_u - s_o}$   ▷ Compute time to spend in each config in pair
    $\tau_o = \tau - \tau_u$                          ▷ Compute energy of this pair
    $newEnergy = \tau_u \cdot p_u + \tau_o \cdot p_o$
    if $newEnergy < energy$ then     ▷ Compare energy to best found so far
      $energy = newEnergy$
      $optimal = \langle u, o\rangle$
      $schedule = \langle \tau_u, \tau_o\rangle$
    end if
  end for

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

Given this information, POET schedules for each configuration c an execution time τ_c, ensuring that the $I(t)$ iterations complete and the total energy consumption is minimized. To do so, POET solves the following optimization problem:

$$minimize \quad \sum_{c=0}^{C-1} \tau_c \cdot p_c \tag{4}$$

$$s.t. \quad \sum_{c=0}^{C-1} \tau_c \cdot s_c \cdot b(t) = I(t) \tag{5}$$

$$\sum_{c=0}^{C-1} \tau_c = \tau \tag{6}$$

$$0 \leq \tau_c \leq \tau, \qquad \forall c \in \{0, \dots, C-1\} \tag{7}$$

Eqn. 4 minimizes the total energy consumption. Eqn. 5 constrains all jobs to complete within the next control period. Eqn. 6 ensures that the time is fully scheduled and Eqn. 7 imposes that a non-negative time is assigned to each configuration. Solving linear optimization problems is, in general, hard. However, this particular problem has a structure that makes it practical to solve. Feasible solutions are confined to a polytope in the positive quadrant defined by the two constraints Eqns. 5 and 6. Thus, linear programming theory states an optimal solution exists for this problem when all the τ_c are equal to zero except for (at most) two configurations [7].

Algorithm 1 takes the set of configurations, the controller's speedup, and the time interval τ specified by the application. It then divides the configurations in two distinct sets. The first set contains all configurations with a speedup less than or equal to the target. The second contains the remaining configurations; *i.e.,* those with speedups greater than required. Subsequently, Algorithm 1 loops over all possible pairs of configurations, with one from each set, to determine how much time should be spent in each configuration given the deadline. If the energy of the pair is lower than any previous energy, the algorithm stores the current best pair, its energy, and its schedule. When the algorithm terminates, its output is the pair of chosen configurations and their assigned times. The algorithm tests all possible pairs from the two sets, each of which contains at most C elements, so an upper bound to the algorithm complexity is $O(C^2)$. We know that there is an optimal solution to the linear program with at most two non-zero τ_c (as the dual problem has two dimensions [7]) and Algorithm 1 tests all pairs of configurations. Therefore, *Algorithm 1 will find a minimal-energy schedule*.

C. Discussion of Generality and Robustness

The controller and the optimizer both reason about speedup instead of absolute performance or latency. The absolute performance of the application, measured by the average latency of its jobs, will vary as a function of the application itself and the platform it executes on. However, speedup is a general concept and can be applied to any application and system, providing a more general metric for control. Moreover, the controller customizes the behavior of a specific application using the estimate of its base speed produced by the Kalman filter. The optimizer operates in a platform-independent manner, using the available configurations provided as input to find the optimal solution, without relying on a particular heuristic that may be system-specific or application-dependent. Finally, the customizable pole p in Eqn. 3 allows for flexibility and robustness to inaccuracies and noise.

The ability to control robustness to inaccuracies and model errors is a major advantage of feedback control systems [14]. In particular, POET is stable and converges to the desired latency without oscillations provided that $0 \leq p < 1$. Formal analysis of this behavior can be obtained by applying standard control techniques – see Appendix B for further details.

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

IV. Implementation

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

A. POET's External Inputs

POET requires three pieces of information from users. First, it needs the available system configurations and their associated timing and power characteristics. Second, it needs a means to measure timing and power consumption during runtime. Third, it requires the application to specify the desired latency target.

The first input is the specification of available system configurations. POET separates the system configurations into

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

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

Fig. 3: Example of POET system-agnostic (left) and system-specific (right) configuration files.

two data structures. The first is system-agnostic and contains a configuration identifier along with **speedup** and **powerup** values. The second is system-specific and can take any form a developer considers appropriate to define a system configuration. To simplify the programmer's job, POET includes a default format for this second structure which contains a configuration identifier, the DVFS frequency to apply, and the number of cores to use. Snippets of actual configuration files representing both these data structures are presented in Figure 3. Section V describes how to characterize a system's timing and power behavior. These results can be used to create configurations if they are not already available.

To measure both latency and power, we modify the Heartbeats API [20] to record power data along with timing statistics. We then modify applications to issue heartbeats at appropriate points during processing, typically after the completion of every job. The issued heartbeats contain power and timing data that POET queries at runtime.

The third necessary input is a latency target. The user provides the target through the Heartbeats API, specifying a minimum and maximum latency goal. The timing targets can change during runtime, and POET will take care of the adaptation automatically.

B. POET's Interface

Users interact with only three POET functions. `poet_init` initializes POET and returns a `poet_state` data structure reference. `poet_apply_control` executes the controller, runs Algorithm 1, and configures the platform. `poet_destroy` cleans up the `poet_state` data structure.

POET's initialization function takes, as parameters, references to: the heartbeat data structure used to store the timing and power of the application, the system's configurations, and the function that applies the given system configurations. It also receives an optional reference to the function that determines the system's current state and a log file name. The first system configuration data structure (system-agnostic) is of type `poet_control_state_t`, and the second (system-specific) has type `void`.

The two functions passed by reference are the only ones that need to know the format of the second data structure and are therefore passed the `void` type reference given to `poet_init` as parameters. The first of these two functions must have a signature that matches the `poet_apply_func` definition and the second must match the `poet_curr_state_func` definition.

The other two API functions, `poet_apply_control` and `poet_destroy`, take the `poet_state` reference as their only parameter. This variable contains all the control state required to implement the framework described in Section III.

Auxiliary functions are also provided to load system configurations from files, discover the initial system configuration, and apply system configurations. The latter two of these meet the `poet_curr_state_func` and `poet_apply_func` definitions, respectively, and can be passed to `poet_init`. These auxiliary functions are platform-dependent and thus kept separate to maintain portability, allowing users to easily substitute their own versions. They are, however, generic enough that most Linux users do not need to write their own.

C. POET's Runtime

After an application signals its power and latency with a heartbeat, it makes a call to `poet_apply_control`, which contains POET's core logic. Heartbeats are initialized with a *window size* value that indicates the number of jobs to complete in a given *time interval*. The window size is analogous to $I(t)$ from Eqn. 5, while the time interval is equivalent to τ from Eqn. 7 and Algorithm 1. At the completion of a window, the POET runtime calculates the estimated base speed with Eqn. 10, then computes the latency error with Eqn. 1. It subsequently applies the controller of Eqn. 3 to determine the speedup necessary to eliminate the computed error. Finally it determines the energy-minimal resource schedule that achieves the necessary speedup using Algorithm 1 and applies the first configuration in the schedule by executing the provided `poet_apply_func` function.

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

The only remaining task is to apply the second configuration in the schedule at the appropriate time during the next window. When the computed number of heartbeats to wait passes, the `poet_apply_func` function executes again, but no further computation is performed. At the completion of the heartbeat window, the control process repeats.

V. Using POET

This section details the platforms and applications used to evaluate POET.

A. Testing Platforms

We use two modern embedded devices with different hardware. We selected these two platforms because prior work has shown that they expose different timing and energy tradeoffs [24]. Table II shows the hardware details for both, highlighting the configurable resources, the cardinality of the set of alternatives for each, and the maximum speedup achievable by manipulating that resource alone. Both platforms run Ubuntu Linux 14.04. The Vaio uses kernel 3.13.0, while the ODROID runs kernel 3.4.104. In both cases, `cpufrequtils` controls processor clock speeds. A **configuration** is a unique combination of allowable values for the system resources.

TABLE II: System configurations.

	Resource	Settings	Max Speedup
Vaio	cores	2	1.81
	core speeds	11	2.72
	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 III: 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

While the Vaio claims to support different frequency settings on different virtual cores, our experience leads us to conclude that this is not the case. Thus, we allow only configurations where all cores are set to the same frequency.

The ODROID's version of the Exynos5 Octa does not support executing on the big and LITTLE clusters simultaneously, and all cores in a cluster must operate at the same frequency. We use a mainline Linux kernel with the default In Kernel Switcher for managing cluster migration.

To capture power measurements on the Vaio, we use the Model-Specific Register (MSR) of the Haswell processor [42]. On the ODROID, we poll INA-231 power sensors [25] available on the XU+E model to capture power data for the A15 and A7 clusters as well as for the DRAM and for the GPU. Basic power figures of the two platforms are shown in Table III.

Capturing these metrics naturally requires hardware resources that expose power or energy data to software. The modified version of Heartbeats includes energy readers for some common hardware (*e.g.,* the MSR) and exposes a simple interface for extending to new hardware. Collecting power data on new platforms with different power or energy monitors is easy and does not require any modifications to POET.

B. Applications

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

The first five applications are included in the PARSEC benchmark suite [5]. Specifically, we use `blackscholes`, `bodytrack`, `facesim`, `ferret`, and `x264`. Bodytrack and x264 process video input and could be required to match the frame rate of a live feed (*e.g.,* from an on-board camera). Ferret is a toolkit for content-based similarity search of non-text data and should satisfy a latency requirement on how fast results are returned to users. Facesim creates realistic animations of a human face from a model and time sequence of muscle movements, and must maintain a real-time frame rate. The sixth and seventh applications are `dijkstra` and `sha`, from the ParMiBench benchmark suite [26]. Dijkstra[3] computes single-source shortest paths in a graph. SHA (Secure Hash Algorithm) is used for secure storage and transmission of data and must maintain response time to ensure timely communication.

[3]We use the parallelized multiple queue implementation provided with the ParMiBench benchmark suite.

The last application is `STREAM` [36], a synthetic benchmark for measuring sustainable memory bandwidth, representing a variety of memory-bound applications.

All benchmarks are modified as discussed in Section IV, adding Heartbeats and POET calls. The code snippet shown in Listing 1 provides an example of application code, highlighting the POET function calls.

Adding POET to existing applications does not require many modifications to the original code. The complete modification of an existing application requires nine function calls plus associated variable declarations, for a total of 14 lines of code. The user provides a desired latency target via the Heartbeats API using the `min_heartrate` and `max_heartrate` variables, which represent a desired minimum and maximum speed in terms of jobs completed per second. POET simply takes the average of these two values, meaning they can be the same. Given $I(t)$ jobs in a window and a target latency τ, the desired rates are easily computed as:

$$min\_heartrate = max\_heartrate = \frac{I(t)}{\tau} \qquad (8)$$

As demonstrated below, Heartbeats initialization also accepts requests for minimum and maximum accuracy and power – POET does not use these, so they can safely be set to any value. When initializing POET, the user should specify the system's configurations, encoded in `control_states` and `cpu_states`.

```
// 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 1: Example of POET application code.

C. Application Inputs

Table IV shows the inputs used for each of the applications. All inputs used are packaged with the original benchmarks, with the exception of the x264 input which comes from a set of standard test sequences. Recall that these applications were not originally designed to provide predictable timing. We quantify this inherent unpredictability by measuring the latency of each job and computing the standard deviation and mean over all jobs in an application. Figure 4 shows the ratio of standard deviation to mean for each application when running without POET. The figure shows that our applications have a range of natural behavior from low variance (implying natural predictability, *e.g.,* blackscholes) to high variance (meaning that the application naturally has widely distributed latencies, *e.g.,* x264). The variability in the applications is largely the

TABLE IV: 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

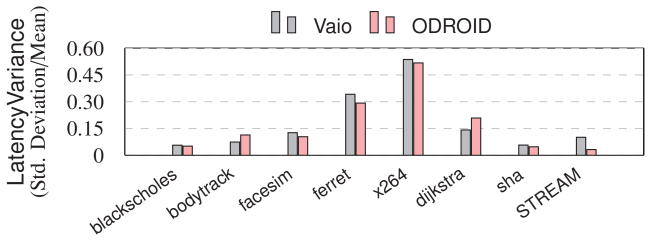


Fig. 4: Application Latency Variability.

same across platforms, indicating that it is a fundamental property of the applications and not the devices.

VI. Experimental Evaluation

The experimental evaluation of POET is divided into five parts. First, we demonstrate POET's ability to meet the latency requirements. Next, we quantify the energy consumption of the resulting system, then compare the energy of POET's general approach to one that controls latency and energy tradeoffs using just DVFS. We then evaluate POET's ability to adapt to input with multiple phases, and finally, its ability to run subject to the interference of other applications.

A. Meeting Latency Targets

To test POET's ability to meet latency targets, we run each application i in all possible configurations on both systems to determine the minimum average latency m_i. For each application, we impose four latency targets. The targets cover a wide range of achievable goals, from 25% to 95% of each system's performance capacity, *i.e.,* a 25% goal means that the target is set to $4 \times m_i$.

We quantify POET's ability to meet the latency goals by measuring each job's latency and comparing it to the goal. We then compute the Mean Absolute Percentage Error (MAPE), a standard metric in control theory to evaluate the behavior of controllers [14]. For an application composed of n jobs:

$$MAPE = 100\% \cdot \frac{1}{n}\sum_{i=1}^{n}\begin{cases} d_m(i) > d_r : & \left|\frac{d_m(i)-d_r}{d_r}\right| \\ d_m(i) \le d_r : & 0 \end{cases} \quad (9)$$

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

Figure 5 presents the MAPE values for each application for the four latency targets on both the Vaio and the ODROID. In general, the larger the variance in the application behavior, the larger the error. This is not surprising since more volatile applications are harder to control. However, the results indicate that MAPE values are generally low. On the Vaio, the average

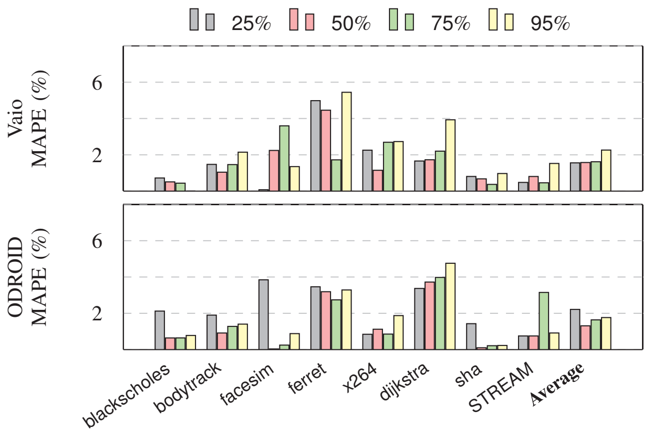


Fig. 5: Latency error (lower is better, 0 is optimal).

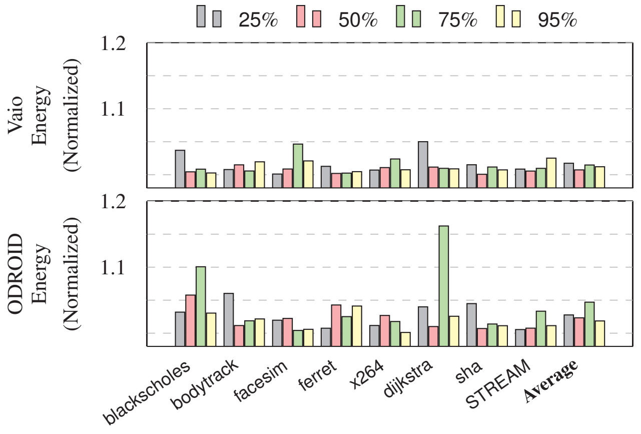


Fig. 6: Energy (lower is better, 1 is optimal).

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

B. Energy Minimization

This section evaluates POET's energy minimization strategy. As discussed, we have measured latency and energy consumption for all applications in all configurations and therefore have perfect knowledge of each application's behavior. We use this data to compute the minimal energy required for a latency target, *i.e.,* the energy consumed when we choose the best configuration for each job with foreknowledge of that job's needs and no overhead. We quantify POET's energy consumption by comparing its achieved energy to this effective minimal energy. Optimal energy is not attainable in practice as it would require knowledge of the future and no overhead.

For each application, we run POET for each latency target and record the achieved energy consumption. We then compute the ratio of the energy consumption with POET to the effective minimal energy. Unity represents minimal energy and values greater than 1 show energy consumption above the optimal. Figure 6 presents the normalized energy data for each application on both the Vaio and the ODROID. The data

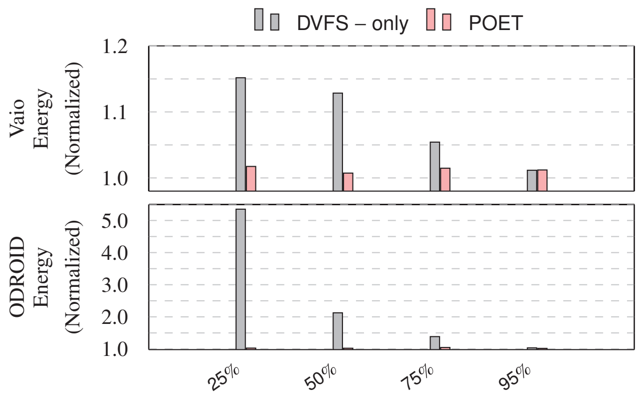


Fig. 7: Comparison of average energy consumption with DVFS-only versus POET (lower is better, 1 is optimal).

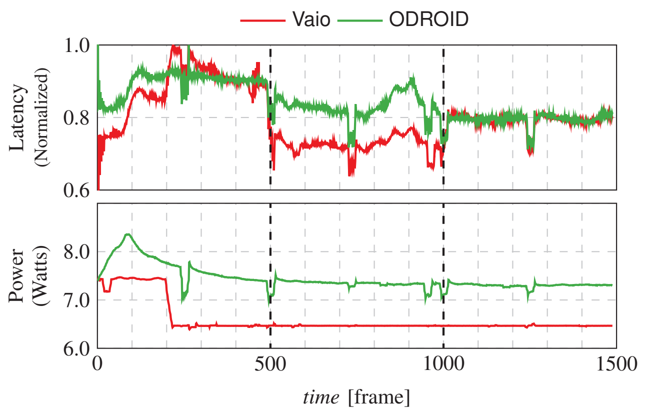


Fig. 8: x264 processing of input with distinct phases.

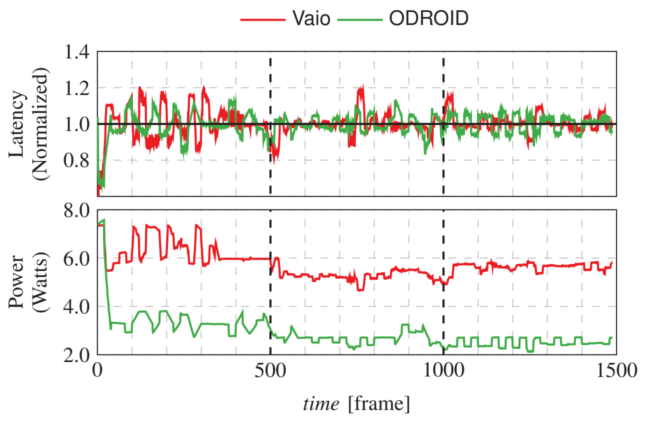


Fig. 9: POET behavior for x264 input with distinct phases.

includes the overhead of POET's runtime, which consumes additional energy executing the control and optimization tasks. On average across all applications and targets, POET's energy consumption exceeds optimal by 1.3% on the Vaio and by 2.9% on the ODROID. These results demonstrate that POET achieves near-optimal energy consumption in practice.

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

C. Comparison with DVFS

Several energy management approaches have been proposed that optimally tune DVFS settings to meet timing constraints while reducing energy consumption [2]. In this section, we compare POET's energy consumption to an approach which only uses DVFS. Specifically, we develop system configuration files that only specify changes in DVFS settings and deploy POET on both hardware platforms with these configurations. We compare this *DVFS-only* approach to POET's more general approach which coordinates multiple resource types and uses different resources on different platforms.

Figure 7 summarizes the data comparing a DVFS-only approach to POET. The charts show latency targets on the x-axes and energy consumption normalized to optimal on the y-axes (for POET, this is the same data shown in Figure 6). For each latency target, the figure shows the average (across all benchmarks) energy over optimal for both DVFS-only and POET on both platforms. At the higher latency (lower performance, *e.g.,* 25%) targets, POET saves substantial energy. The energy savings are especially high on the ODROID as POET is able to take advantage of cluster migration and the low-power LITTLE cores, whereas a DVFS-only approach cannot exploit this feature. This data clearly demonstrates that systems that are provisioned for a rarely seen worst case latency can greatly benefit from POET's generalized approach. This is also further confirmation of prior studies showing that DVFS by itself is not optimal [23, 37].

The exact energy savings compared to DVFS vary considerably for each application and latency target. Therefore, we include more detailed charts in Appendix D.

D. Responding to Application Phases

In this test, we examine POET's ability to cope with input whose workload varies with time. We execute the `x264` application using an input that is a combination of three videos of varying difficulty. The input thus has three distinct phases, each composed of 500 jobs (frames). Figure 8 shows time series data for both latency and power for the Vaio and the ODROID when they run without POET in their highest performing configurations. Latency is normalized to the maximum latency measured for any iteration (*i.e.,* the empirically determined worst case). We use this worst case result to derive the latency target for the POET tests. Frames that need fewer resources to achieve the latency goal present opportunities to save energy. We present the raw data for power; energy is the integral of those curves.

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

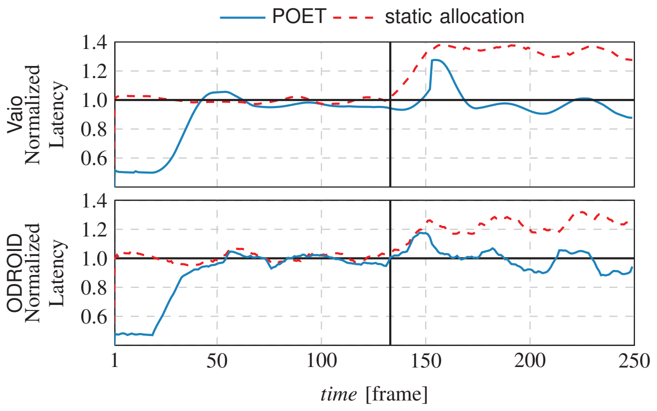


Fig. 10: POET running with another application.

Vaio, while the third one is the easiest for the ODROID.

We enable POET using the maximum measured latency identified in the first experiment as the target. The results of the executions are shown in Figure 9. POET is able to meet the latency target on both systems. Dips and spikes are visible at the beginning of each phase, showing the change in behavior of the input and POET adapting to the change. Despite these variations, latency goals are respected: MAPE is 2.2% on the Vaio and 2.0% on the ODROID. At the same time, energy is near minimal over the course of execution: energy is 1.7% greater than optimal on the Vaio and 3.6% over optimal on the ODROID.

E. *Adapting to Other Applications*

This test shows POET's behavior when other applications are present in the system. This external load is not under the direct control of POET. We launch a POET-enabled application with a target latency. Halfway through its execution, we launch another application. This second application consumes resources, slowing down the POET-enabled application. POET then assigns more resources to its own application so that it continues to meet its latency target. We have tested this capability on all applications, but we only report the results for `bodytrack` due to space limitations. The results for the other applications are similar.

Figure 10 shows POET's behavior, the top half displaying the Vaio and the bottom half the ODROID. The thick vertical lines show when the second application was launched. In both cases, we see the latency temporarily increase before POET adjusts the resource allocation. To show the benefits of POET, the charts also show a statically managed system that just selects the resources to be given to the application at the beginning of its execution. In the static case, the introduction of the new application dramatically increases the job latency. To quantify this effect, we compute the MAPE metric for both the static allocation and with POET. On the Vaio, POET's MAPE is 2.3% over the entire execution (including the period of adjustment to the new load), while the static case has a MAPE of 16%. On the ODROID, POET's MAPE is 2.4%, while the static case achieves 12%.

F. *Discussion of Results and Limitations*

Our results show that POET achieves the goals of providing predictable timing and near minimal energy across multiple platforms. These results are obtained despite the facts that 1) the tested applications were not originally designed to offer predictable latency and 2) the test platforms have completely different latency/energy tradeoffs. The applications require only minimal modifications to run with POET, but no other changes are needed to exploit the different resources and latency/energy tradeoffs that different platforms offer. In summary, POET achieves our design goal of enabling predictable timing with near-optimal energy in a portable library. The code for POET and the configurations used for the experiments are available to reproduce the results.

These results also demonstrate some limitations of the approach. POET supports only soft real-time constraints. The controller is guaranteed to converge to the desired latency and is provably robust to errors, but latency goals may be violated during the settling time, as seen in Figure 10 when POET adapts to the presence of the new application. In addition, highly variable applications can still cause temporary latency violations before the control action settles again, as seen in Figure 9 when controlling the high-variance x264 application. This is further evidence that there is a tension between timeliness and energy reduction [8] – the tremendous energy savings on the ODROID come at a cost of some latency errors compared to *race to idle*.

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

Finally, POET currently assumes that only one of the running applications (consisting of multiple, possibly communicating threads) should meet a deadline. POET's Kalman filter guarantees that even when other applications are present in the system, the controller will compute the correct speedup to be applied, as demonstrated in Figure 10. However, future work could extend POET with a priority scheme allowing multiple POET-enabled applications to work concurrently. In that scheme, high priority applications would be allocated the needed resources and lower priority applications would run in best-effort mode.

VII. Related Work

Many empirical studies have shown that it is more energy efficient to coordinate multiple resources than to manage any one alone [3, 12, 54, 55]. For example, Dubach et al. coordinate several microarchitectural features [13], Petrucci et al. coordinate thread scheduling and the use of heterogeneous cores [40], while Maggio et al. coordinate core allocation and clockspeed [34]. Bertini et al. coordinate tiers of a multi-tier webserver for e-commerce [4]. AbouGhazaleh et al. coordinate the speed of the processor and cache [1], while Yun et al. also coordinate the speeds of multiple on-chip components [55]. Liu et al. coordinate job scheduling and clock speed on clusters [33]. Bitirgen et al. coordinate clockspeed, cache, and memory bandwidth in a multicore [6]. The METE system manages clockspeed, memory bandwidth, and core usage [43]. Sinangil et al. co-design processor architecture which exposes both monitoring and configurable resources with an operating system that dynamically manages those resources [47]. All of these approaches coordinate multiple resources, but do so using system-specific implementations. For example, METE's controller would have to be redesigned to work with Bitirgen et al.'s architecture.

Several researchers have proposed frameworks that coordinate general sets of resources specified at compile or run time. Rajkumar et al. propose a general framework (with system specific implementation) for allocating resources to achieve real-time requirements, but this approach is not energy-aware [41]. Sojka et al. proposed a portable middleware layer for allocating resources to meet soft real-time constraints, but this system does not minimize energy [49]. ControlWare is another middleware approach that uses control theory to meet quality-of-service constraints, but does not address energy concerns [56]. Fu et al. describe a method for managing both real-time and thermal constraints on embedded multicores [16]. While thermal dissipation is often related to power and energy consumption, the two problems are different and resource schedules that respect temperature constraints do not always result in minimal energy consumption. The prior approach most similar to POET is PTRADE, which also uses control theory to manage general collections of resources [20]. PTRADE is designed to minimize power consumption, but not energy. In addition, PTRADE uses heuristic optimizations, while POET uses a true minimal-energy scheduling algorithm.

There is another class of management that delivers predictable timing behavior with minimal energy: *cross-layer approaches* [15, 22, 28, 30, 35, 51]. These approaches coordinate system resource usage with application-level adaptations that change the application's result. For example, a cross-layer approach to video-encoding might simultaneously adjust resource usage to save energy while using a simple encoding algorithm to meet timing constraints [51]. POET differs from such approaches because it only manages resource usage and does not require applications to change their internal algorithms. POET could be used as a resource manager in a cross-layer approach, but that is beyond the scope of this work. Instead, POET provides energy-minimal resource allocations while requiring only small code changes, rather than the large scale changes needed to apply cross-layer adaptation.

We note two related, complementary approaches. Zhao et al. manage processor speed to meet both reliability and timing constraints with minimal energy [57]. This problem is complicated by the effect DVFS scaling has on hardware reliability. It is possible that POET's general approach to resources other than DVFS might allow additional energy savings if incorporated in Zhao et al.'s work. He et al. propose adaptive energy management in the power circuitry itself to meet timing constraints while adapting the delivered energy to increase battery efficiency [17]. It is possible that combining POET's runtime-level resource management with this supply-level approach would further increase efficiency.

Like POET, a number of the approaches listed above use feedback control to manage timing constraints [4, 16, 20, 22, 30, 31, 34, 35, 51, 56]. Control techniques provide a formal framework for reasoning about the dynamic behavior of the system. POET is shown to be stable and provably robust to model inaccuracies. One limitation to feedback based methods is the need for instrumentation. For example, POET requires applications to signal the completion of jobs. Other approaches use predictive models that do not require feedback mechanisms [13, 48]. Despite not requiring instrumentation, these approaches do not provide formal guarantees.

In summary, POET is most related to prior approaches that abstract resource management into a middleware or runtime, like [20, 41, 49, 56]. POET is unique in its energy awareness, in its design for portability, and the incorporation of a true minimal-energy resource allocation algorithm.

VIII. Conclusion

This paper presents POET, an open source library for meeting timing constraints and minimizing energy consumption. POET uses a combination of control theory and mathematical optimization to generalize the problem of energy-aware resource allocation on embedded systems. We test POET on modern embedded multicore devices and demonstrate that applications controlled with POET achieve their latency goals with low error and near-optimal energy consumption. To foster result reproducibility and further development, we release POET's code, configurations, and benchmark patches used in this paper as open source.

Acknowledgments: We thank Haryadi Gunawi and the anonymous reviewers, whose comments helped us significantly improve this paper.

References

[1] N. AbouGhazaleh et al. "Integrated CPU and L2 Cache Voltage Scaling Using Machine Learning". In: *LCTES*. 2007.
[2] S. Albers. "Algorithms for Dynamic Speed Scaling". In: *STACS*. 2011, pp. 1–11.
[3] L. A. Barroso and U. Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (2007).
[4] L. Bertini et al. "Statistical QoS Guarantee and Energy-Efficiency in Web Server Clusters". In: *ECRTS*. 2007.
[5] C. Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *17th Conference on Parallel Architectures and Compilation Techniques*. 2008.
[6] R. Bitirgen et al. "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach". In: *MICRO*. 2008.
[7] S. Bradley et al. *Applied mathematical programming*. 1977.
[8] G. C. Buttazzo et al. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, 2006.
[9] L. Cao and H. M. Schwartz. "Analysis of the Kalman Filter Based Estimation Algorithm: An Orthogonal Decomposition Approach". In: *Automatica* 40.1 (2004).
[10] Q. Cao et al. "Virtual Battery: An Energy Reserve Abstraction for Embedded Sensor Networks". In: *RTSS*. 2008.
[11] A. Carroll and G. Heiser. "Mobile Multicores: Use Them or Waste Them". In: *HotPower*. 2013.
[12] H. Cheng and S. Goddard. "SYS-EDF: a system-wide energy-efficient scheduling algorithm for hard real-time systems". In: *IJES* 4.2 (2009).
[13] C. Dubach et al. "A Predictive Model for Dynamic Microarchitectural Adaptivity Control". In: *MICRO*. 2010.
[14] A. Filieri et al. "Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees". In: *ICSE*. 2014.
[15] J. Flinn and M. Satyanarayanan. "Managing Battery Lifetime with Energy-aware Adaptation". In: *ACM Trans. Comput. Syst.* 22.2 (2004).
[16] Y. Fu et al. "Feedback Thermal Control of Real-time Systems on Multicore Processors". In: *EMSOFT*. 2012.
[17] L. He et al. "Exploring Adaptive Reconfiguration to Optimize Energy Efficiency in Large-Scale Battery Systems". In: *RTSS*. 2013.
[18] J. L. Hellerstein et al. *Feedback Control of Computing Systems*. 2004.
[19] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. 2009.
[20] H. Hoffmann et al. "A generalized software framework for accurate and efficient management of performance goals". In: *EMSOFT*. 2013.

[21] H. Hoffmann. "Racing and pacing to idle: an evaluation of heuristics for energy-aware resource allocation". In: *HotPower*. 2013.
[22] H. Hoffmann. "CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems". In: *ECRTS*. 2014.
[23] H. Hoffmann et al. "Self-aware Computing in the Angstrom Processor". In: *49th Annual Design Automation Conference*. 2012.
[24] C. Imes and H. Hoffmann. "Minimizing Energy Under Performance Constraints on Embedded Platforms: Resource Allocation Heuristics for Homogeneous and Single-ISA Heterogeneous Multi-Cores". In: *EWiLi*. 2014.
[25] T. Instruments. http://www.ti.com/product/ina231.
[26] S. Iqbal et al. "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems". In: *Computer Architecture Letters* 9.2 (2010).
[27] B. Jeff. "Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration". In: *DAC*. 2012.
[28] M. Kim et al. "xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems". In: *ACM TECS* 11.4 (2013).
[29] E. Le Sueur and G. Heiser. "Slow Down or Sleep, That is the Question". In: *USENIX ATC*. 2011.
[30] B. Li and K. Nahrstedt. "A control-based middleware framework for quality-of-service adaptations". In: *IEEE Journal on Selected Areas in Communications* 17.9 (1999).
[31] X. Li et al. "Cross-component Energy Management: Joint Adaptation of Processor and Memory". In: *ACM Trans. Archit. Code Optim.* 4.3 (2007).
[32] J. D. Lin et al. "Real-energy: A New Framework and a Case Study to Evaluate Power-aware Real-time Scheduling Algorithms". In: *ISLPED*. 2010.
[33] C. Liu et al. "PASS: power-aware scheduling of mixed applications with deadline constraints on clusters". In: *ICCCN*. 2008.
[34] M. Maggio et al. "Power Optimization in Embedded Systems via Feedback Control of Resource Allocation". In: *IEEE TCST* 21.1 (2013).
[35] M. Maggio et al. "A Game-Theoretic Resource Manager for RT Applications". In: *Conference on Real-Time Systems (ECRTS)*. 2013.
[36] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE TCCA Newsletter* (1995).
[37] D. Meisner et al. "Power Management of Online Data-intensive Services". In: *38th Annual Symposium on Computer Architecture*. 2011.
[38] A. Miyoshi et al. "Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling". In: *ICS*. 2002.
[39] T. Pering et al. "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms". In: *ISLPED*. 1998.
[40] V. Petrucci et al. "Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems". In: *HotPower*. 2012.
[41] R. Rajkumar et al. "A Resource Allocation Model for QoS Management". In: *RTSS*. 1997.
[42] E. Rotem et al. "Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge". In: *Hot Chips*. 2011.
[43] A. Sharifi et al. "METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management". In: *SIGMETRICS PER* 39.1 (2011).
[44] V. Sharma et al. "Power-aware QoS Management in Web Servers". In: *RTSS*. 2003.
[45] Y. Shin et al. "28nm high- metal-gate heterogeneous quad-core CPUs for high-performance and energy-efficient mobile application processor". In: *ISSCC*. 2013.
[46] A. Shye et al. "Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures". In: *MICRO*. 2009.
[47] Y. Sinangil et al. "A self-aware processor SoC using energy monitors integrated into power converters for self-adaptation". In: *Symposium on VLSI*. 2014.
[48] D. C. Snowdon et al. "Koala: A Platform for OS-level Power Management". In: *EuroSys*. 2009.
[49] M. Sojka et al. "Modular software architecture for flexible reservation mechanisms on heterogeneous resources". In: *Journal of Systems Architecture - Embedded Systems Design* 57.4 (2011).
[50] N. Thiagarajan et al. "Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption". In: *WWW*. 2012.
[51] V. Vardhan et al. "GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy". In: *IJES* 4.2 (2009).
[52] M. Wang et al. "Real-Time Loop Scheduling with Leakage Energy Minimization for Embedded VLIW DSP Processors". In: *RTCSA*. 2007.
[53] G. Welch and G. Bishop. *An Introduction to the Kalman Filter*. Tech. rep. TR 95-041. UNC Chapel Hill, Department of Computer Science.
[54] C.-Y. Yang et al. "System-Level Energy-Efficiency for Real-Time Tasks". In: *SOCRTD*. 2007.
[55] H. Yun et al. "System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks". In: *ECRTS*. 2010.
[56] R. Zhang et al. "ControlWare: a middleware architecture for feedback control of software performance". In: *ICDCS*. 2002.
[57] B. Zhao et al. "Energy Management Under General Task-Level Reliability Constraints". In: *RTAS*. 2012.
[58] Y. Zhu and V. J. Reddi. "High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems". In: *HPCA*. 2013.

Appendix

This appendix contains additional details about the POET control system and some additional results of the experimental evaluation. First, it specifies the Kalman filter. It then analyzes the convergence of the controller. Next, it discusses robustness to inaccuracies in the user-specified configurations. Finally, it presents additional data comparing POET to an approach that only uses DVFS to meet latency goals while minimizing energy consumption.

A. The POET Kalman Filter Formulation

POET's controller customizes itself to the behavior of the application under control. This customization is achieved at runtime by estimating the key parameter of the controller, $b(t)$ in Eqn. 3, using a Kalman filter. Denoting the application timing variance as $q_b(t)$ and assuming minimal measurement variance (*i.e.,* the application may be noisy, but the signaling framework does not add additional noise), the Kalman filter formulation is standard [53].

$$\begin{cases} \hat{b}^{-}(t) &=& \hat{b}(t-1) \\ e_b^{-}(t) &=& e_b(t-1) + q_b(t) \\ k_b(t) &=& \dfrac{e_b^{-}(t)\cdot s(t)}{[s(t)]^2\cdot e_b^{-}(t)} \\ \hat{b}(t) &=& \hat{b}^{-}(t) + k_b(t)\left[\dfrac{1}{d_m(t)} - s(t)\cdot \hat{b}^{-}(t)\right] \\ e_b(t) &=& [1 - k_b(t)\cdot s(t-1)]\, e_b^{-}(t) \end{cases} \tag{10}$$

In this formulation, $k_b(t)$ is the Kalman gain for the latency, $\hat{b}^{-}(t)$ and $\hat{b}(t)$ are the *a priori* and *a posteriori* estimates of $b(t)$, and $e_b^{-}(t)$ and $e_b(t)$ are the *a priori* and *a posteriori* estimates of the error variance.

POET uses a Kalman filter because it produces a statistically optimal estimate of the system's parameters and is provably exponentially convergent [9]. The user does not need to have prior knowledge on Kalman filtering – all the filter parameters are computed by the controller (speedup $s(t)$), measured (latency $d_m(t)$, latency variance $q_b(t)$), or derived (all others).

B. Formal Analysis of POET Convergence

Control-theoretical adaptation mechanisms are useful since they provide formal guarantees on the behavior of the system under control during runtime. Usually, for linear time-invariant systems, this analysis is performed in the transfer function domain, because it greatly simplifies the mathematical approach. For continuous-time systems, the Laplace transform is used, while for discrete-time systems, the analysis is typically performed using the Z-transform [18]. In the Z-transform domain, the operator z^{-k} is a k-unit delay.

We want to prove that the controller computes the correct speedup to cancel the latency error. Therefore, we assume that, whichever speedup is computed, the optimizer translates it into a schedule with no error. We also know that the Kalman filter converges to its estimated value. Therefore we prove the convergence of the controller to the correct control signal when the Kalman filter has already reached its correct estimate b. Since the input and output signals are bounded, this proof suffices to show that the entire system converges.

Eqn. 2 can be transformed into its Z-transform equivalent

$$A(z) = \frac{b}{z} \tag{11}$$

where $A(z)$ is the transform of the effect of the input $s(t)$ on the output $d_m(t)$. Similarly, the controller equation (Eqn. 3)

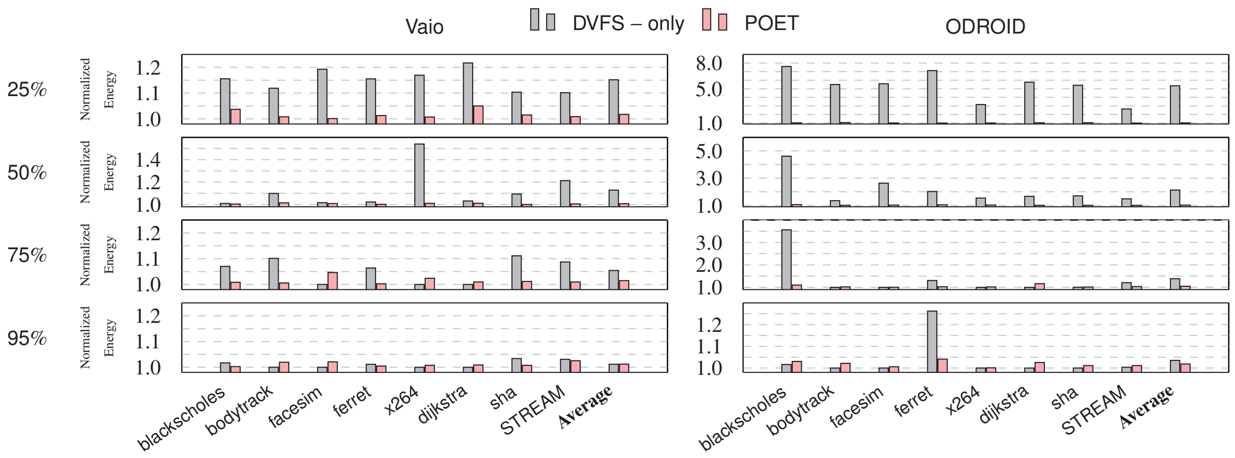


Fig. 11: Comparison of energy consumption with DVFS-only versus POET (lower is better, 1 is optimal).

can be transformed into the corresponding one in the transfer function domain, and becomes

$$C(z) = \frac{(p-1)\cdot z}{b\cdot(z-1)} \tag{12}$$

where p is the controller pole and z^{-1} is the unit delay. The Z-transform of the closed loop system represented in Figure 2 is

$$F(z) = \frac{C(z)\cdot A(z)}{1 + C(z)\cdot A(z)} \tag{13}$$

and can be rewritten as

$$F(z) = \frac{\frac{(p-1)\cdot z}{b\cdot(z-1)}\cdot\frac{b}{z}}{1+\frac{(p-1)\cdot z}{b\cdot(z-1)}\cdot\frac{b}{z}} = \frac{1-p}{z-p} \tag{14}$$

The closed loop system has a pole p. The system also has a static gain of 1, therefore the input (the desired latency) is translated directly into the output without modifications. Provided that $0 \le p < 1$, the system is stable by design.

C. Formal Analysis of POET Robustness

To analyze POET's robustness to error, suppose that the speedup values provided in the resource specification are incorrect. Let s_c be the specified speedup with the largest error. We write the true speedup as $\bar{s}_c = \Delta \cdot s_c$, with Δ being a multiplicative error term. For example, $\Delta = 5$ indicates that the specified speedup is off by a factor of 5 and $\Delta = 0.5$ indicates that the real speedup is only half of that specified.

Due to the relationship between base speed b and speedup s_c an error in the speedup term is equivalent to the same error in the estimate of base speed. Therefore, we determine POET's robustness to errors in the specified speedup by substituting $\Delta \cdot b = b$ into $F(z)$.

$$\begin{aligned} F(z) &= \frac{C(z)\cdot A(z)}{1+C(z)\cdot A(z)} \\ &= \frac{\frac{(p-1)\cdot z}{b\cdot(z-1)}\cdot\frac{b\cdot\Delta}{z}}{1+\frac{(p-1)\cdot z}{b\cdot(z-1)}\cdot\frac{b\cdot\Delta}{z}} \\ &= \frac{(1-p)\cdot\Delta}{1+\Delta\cdot(1-p)-z} \end{aligned} \tag{15}$$

The closed loop represented by Eqn. 15 is stable if and only if its pole is within the unit circle. Assuming the pole is real, its absolute value should be between 0 and 1. Then, for a stable system, $-1 < \Delta \cdot p + \Delta + 1 < 1$. The first part, $-1 < \Delta \cdot p + \Delta + 1$, translates to $\Delta < \frac{2}{(1-p)}$. The second part, $\Delta \cdot p + \Delta + 1 < 1$, imposes $\Delta > 0$ and $p < 1$. The second constraint is already verified in the controller design. As a final result, the system is stable when $0 < \Delta < \frac{2}{(1-p)}$. This means that if $p = 0.1$, the maximum Δ that the system can stand is 2.2, while if $p = 0.9$, the maximum Δ is 20.

D. Additional Results for DVFS-only Comparison

Section VI-C presented an overview comparing POET to an approach that uses only DVFS to trade latency for energy. Figure 11 shows the detailed results of this comparison for all four latency targets on both platforms. The Vaio results are shown on the left side and the ODROID results on the right. Each row represents a different latency target (from 25% to 95% of each system's performance capacity). The benchmark names are shown on the x-axes and the energy, normalized to optimal, is shown on the y-axes. Unity represents minimal energy.

These results provide further evidence of POET's benefits compared to prior approaches that rely solely on DVFS. For every latency target on the ODROID and every latency target but 95% on the Vaio, there is at least one benchmark for which the DVFS-only approach consumes more than 10% energy over optimal. The differences are larger for the less aggressive latency targets. At these targets, substantial energy savings can be achieved through a combination of both DVFS and reduction in cores usage (on the Vaio) or cluster migration (on the ODROID). We anticipate future systems will have increasing numbers of resources available for management and resource management approaches that only address one resource will become obsolete.