Portable Multicore Resource Management for Applications with Performance 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*—Many modern software applications have performance requirements, like mobile and embedded systems that must keep up with sensor data, or web services that must return results to users within an acceptable latency bound. For such applications, the goal is not to run as fast as possible, but to meet their performance requirements with minimal resource usage, the key resource in most systems being energy. Heuristic solutions have been proposed to minimize energy under a performance constraint, but recent studies show that these approaches are not portable – heuristics that are near-optimal on one system can waste integer factors of energy on others. The POET library and runtime system provides a portable method for resource management that achieves near-optimal energy consumption while meeting soft real-time constraints across a range of devices. Although POET was originally designed and tested on embedded and mobile platforms, in this paper we evaluate it on a manycore server-class system. The larger scale of manycore systems adds some overhead to adjusting resource allocations, but POET still meets timing constraints and achieves near-optimal energy consumption. We demonstrate that POET achieves portable energy efficiency on platforms ranging from low-power ARM big.LITTLE architectures to powerful x86 server-class systems.**

I. Introduction

Portability has long been a design goal for software systems – when new hardware platforms become available, we would ideally reuse software without modification. As software performance became increasingly important, attention turned to *performance portability*. Today, energy is increasingly becoming a key concern for developers, making *energy portability* another important design consideration for software. Put simply, energy-portable software should achieve near-minimal energy across a range of devices without requiring software rewrites or platform-specific code optimizations.

As energy concerns have come to dominate computer system designs, architects have responded by making processors increasingly *configurable*. For example, current processors expose multiple processor speeds, multiple cores, and even different core types. All these resources must be managed by software. While increasing configurability increases the potential energy savings, it can also reduce portability if a software's resource management strategies are specific to a particular architecture, platform, or system design implementation. A recent study shows that on heterogeneous multicores, *e.g.,* ARM big.LITTLE [24], cores should be kept busy much of the time [6]. Another study compares an Intel mobile Haswell processor to a Samsung ARM big.LITTLE System-on-Chip and demonstrates that resource allocation heuristics that are near-optimal for one can be extremely inefficient on the other [21]. Even different models of processor intended for the same market and built by the same manufacturer can radically differ in their response to heuristic resource allocation strategies [25]. Thus, it is currently up to software developers to understand the nuances of energy consumption on target platforms and write code that can achieve good energy consumption on all possible target platforms. This problem becomes even more challenging if attempting to "future-proof" software so that it will achieve good energy efficiency on platforms that do not even exist yet. Clearly, it is necessary to support energy-portable code and relieve software developers from this burden.

Recent work proposed POET, the Performance with Optimal Energy Toolkit, to enable energy portability for applications with performance constraints[1]. With POET, application developers specify performance requirements through a software interface and available resources using configuration files. The POET runtime system is linked into applications and then automatically manages resources to meet goals. POET uses control theory to meet performance goals and mathematical optimization to determine minimal-energy resource schedules.

While POET has demonstrated energy portability on embedded and mobile systems, it has not been evaluated on large-scale multicores (or manycores) which are currently used in server-class processors and will become increasingly prevalent in other systems as well – e.g., the TILEPro architecture supports up to 64 cores and is designed to run embedded workloads [18]. This paper evaluates POET on a large multicore system. *We find that, despite an order of magnitude increase in configurability, POET is still able to meet performance goals with minimal energy.*

We use a dual-socket Intel Xeon system with 16 physical cores, hyperthreading, and 16 different DVFS clockspeeds, including TurboBoost. Whereas the prior evaluation platforms had at most 68 configurations, this evaluation platform has 512. Naturally, the larger configuration space results in higher overhead. However, we find that the overhead comes predominantly from the time it takes the system to change configurations, which any adaptive resource scheduling strategy requires, not from POET itself. *We conclude that future multicore and*

This work was funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award. 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]POET is open source – the code is available by following links on the project web page at http://poet.cs.uchicago.edu/

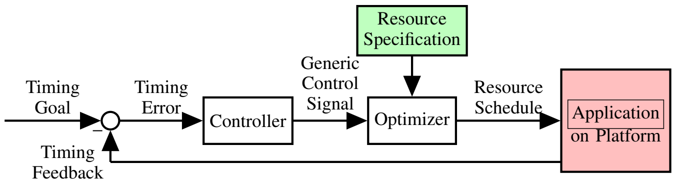

Fig. 1: Overview of the POET runtime.

manycore systems must not only support configurability, but must make changing configurations efficient and low-overhead.

II. General and Portable Resource Allocation

This section reviews POET's design, originally presented in [22]. POET has two design goals: (1) providing predictable timing and (2) minimizing energy consumption given the timing constraint. POET addresses the first goal using control theory and the second using mathematical optimization.

Figure 1 illustrates POET's design. The application specifies a target job latency. POET's runtime measures job latency and the error between the desired and measured latency. The **controller** then calculates a generic *speedup* signal. The **optimizer** treats the speedup signal as a constraint and then determines a minimal-energy resource allocation that will respect the specified speedup constraint. For portability, both the controller and the optimizer are independent of any particular application and system. POET assumes applications are composed of repeated jobs with soft deadlines. POET targets many and multicore platforms, so 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 by Eqn. 1 to calculate the speedup signal $s(t)$ in Eqn. 2. The controller acts at discrete time intervals and implements the *integral control law* [13]:

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

where p is a user-configurable *pole* of the closed loop characteristic equation [11]. To ensure the controller reaches a steady state without oscillations, we enforce $0 \le p < 1$. A small p will cause the controller to react quickly, potentially producing a noisy speedup signal. A large p ensures robustness with respect to transient fluctuations, making it slow to respond to external changes, and may be beneficial for very noisy systems.

The application's base speed is represented by $b(t)$. Different applications will have different base speeds and may even have *phases*, where base speed changes over time. Therefore, POET continually estimates base speed using a Kalman filter [38], which adapts $b(t)$ to the current application behavior. More details on the Kalman filter are presented in the original POET paper [22].

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, the POET controller does not reason about a particular set of resources, but computes a generic control signal $s(t)$.

B. Optimizer

The optimizer turns the speedup signal into a system-specific resource allocation strategy, producing a *schedule* for the available resources. To translate the continuous signal into a schedule for discrete resources, the optimizer considers the next τ time units. Specifically, POET completes $I(t)$ jobs in the next interval, with $I(t) = \tau \cdot s(t) \cdot b(t)$. Both the number of jobs to be completed and an acceptable scheduling period τ are specified by the application.

As shown in Figure 1, the POET optimizer is given a resource specification that defines the available configurations. There are C possible configurations in the system and we number the configurations from 0 to $C-1$. $c = 0$ corresponds to the minimal-resource configuration, while configuration $C-1$ makes all resources available. Each configuration c has a power consumption p_c and speedup s_c, normalized to $c = 0$.

POET assigns a time τ_c to spend in each configuration c such that $I(t)$ iterations complete and the total energy consumption is minimized. Formally, POET solves the following optimization problem:

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

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

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

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

Eqn. 4 minimizes the total energy consumption. Eqn. 5 constrains all jobs to complete within the next period. Eqn. 6 ensures that the time is fully scheduled and Eqn. 7 imposes that a non-negative time is assigned to each configuration. Recent work shows that an optimal solution to this problem will correspond to at most two $\tau_c \neq 0$ [25]. Furthermore, one configuration will be the most energy-efficient configuration above the required speedup, while the other will be the most energy efficient configuration below the required speedup.

C. Portability

The controller and the optimizer both reason about speedup instead of absolute performance or latency. The application's absolute performance, 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 base speed estimate produced by the Kalman filter. The optimizer operates in a platform-independent manner, using the available

```
#id     speedup     powerup
0       1.000       1.000
1       1.078       1.020
2       1.157       1.029
3       1.973       1.056
...
109     34.935      5.271
110     35.223      5.296
111     35.501      5.483
```

```
#id     frequency     cores
0       1200000       0
1       1300000       0
2       1400000       0
3       1200000       1
...
109     2901000       28
110     2901000       25
111     2901000       15
```

Fig. 2: Example of POET system-independent (left) and system-specific (right) configuration files.

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.

III. Implementation

This section describes how the POET framework is realized in a C library and runtime system.

A. POET's External Inputs

POET requires three user-specified inputs: (1) the available system configurations, (2) timing and power measurement capabilities, and (3) the performance target[2].

Two data structures track system configurations. The first is system-independent and contains a configuration identifier and **speedup** and **powerup** values. The second is system-specific and can take any form a developer considers appropriate to define a system configuration. In our evaluation, we specify the configuration identifier, the DVFS setting, and the number of processor cores to execute on. Figure 2 shows samples of actual configuration files representing these data structures.

POET uses the Heartbeats API [14, 17] to monitor performance and power. Applications are modified to emit heartbeats at key intervals. POET then queries the heartbeat data structure to extract the average job performance and power consumption between two consecutive heartbeats over the previous window period. The user provides the performance target through the Heartbeats API, which is described in more detail in Section IV-B. The timing targets can change during runtime, and POET will adapt automatically.

B. POET's Interface

Users interact with three POET functions. `poet_init` initializes POET and returns a `poet_state` data structure reference. `poet_apply_control` executes the controller, computes the optimal-energy configuration schedule, and configures the platform. `poet_destroy` cleans up the `poet_state` data structure.

POET's initialization function requires references to: the heartbeat data structure, the system's configurations, and the function that applies the given configurations. It also receives an optional reference to the function that determines the system's current state and a log file name. The first configuration data structure (system-independent) 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 second data structure's format, and are

[2]Performance is easily derived from a latency target or timing deadline.

TABLE I: System power characteristics.

Idle Power	Min Power	Max Power
17.90 W	37.80 W	199.26 W

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 II.

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 issuing a heartbeat, the application calls the `poet_apply_control` function, which contains POET's core logic. Heartbeats are initialized with a *window size* indicating how many jobs to complete in a given *time interval*. The window size is the interval $I(t)$ from Eqn. 5, while the time interval is τ from Eqn. 7. When the window completes, POET estimates base speed, computes error with Eqn. 1, and computes the speedup control signal with Eqn. 3. Having computed the speedup signal, POET uses mathematical optimization to determine the resource configuration schedule [25]. Once POET has determined the schedule, it puts the system in the scheduled configuration by calling the `poet_apply_func` function at the appropriate work interval.

IV. Using POET

A. Testing Platform

We evaluate POET on a dual-socket server system, where each socket contains 8 cores. With hyperthreading, the system exposes 32 virtual cores. There are 16 DVFS settings available, including TurboBoost. A **configuration** is a unique combination of allowable values for the system resources. The system runs Ubuntu 14.04 LTS with Linux kernel 3.13.0. We control core allocation with `taskset` and DVFS settings with `cpufrequtils`. For both simplicity and consistency, we set the DVFS frequency on all cores, regardless if a particular core is being used.

We capture runtime energy data from each socket's Model-Specific Register (MSR) [34]. Capturing power metrics naturally requires hardware resources that expose power or energy data to software. The Heartbeats implementation we use 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

Our analysis uses the same eight benchmarks that POET was originally evaluated with [22]. None of the applications were originally written to provide predictable performance, which challenges POET's approach as much as possible.

The first five applications are from the PARSEC benchmark suite [3]. Specifically, we use `blackscholes`, `bodytrack`, `facesim`, `ferret`, and `x264`. Blackscholes uses partial different equations to price financial investment portfolios. Both bodytrack and x264 process video input. Ferret performs content-based similarity search of non-text data. Facesim animates a human face from a model and time sequence of muscle movements. The next two applications are from the ParMiBench benchmark suite [23] – `dijkstra` and `sha`. Dijkstra computes single-source shortest paths in graphs. SHA is a hashing algorithm used for secure data transmission and storage. The sha application is also unique in that it only supports up to 8 threads, so we do not execute it on more cores than that. The final application is `STREAM` [30], which represents memory-bound applications.

The applications were modified as discussed in Section III, and remain unchanged from POET's original evaluation on embedded systems. The following snippet is an example of application code, highlighting the POET function calls.

```
// 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.

POET requires only minimal changes to application code. A trivial example requires only an additional 14 lines of code: nine function calls and associated variable declarations. The user provides a desired performance target via the Heartbeats API using the `min_heartrate` and `max_heartrate` variables. These variables represent a desired minimum and maximum speed in terms of jobs completed per second. POET simply averages these two values, so in practice they can be the same. Given $I(t)$ jobs in a window period and a target job latency τ, the performance values are simply computed as:

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

As demonstrated above, the Heartbeats initialization also accepts requests for minimum and maximum accuracy and power. Since POET does not use these fields, they can safely be set to any value, *e.g.,* 0. When initializing POET, the user specifies the system's configurations, which are encoded in the `control_states` and `cpu_states` variables. The former is an array of type `poet_control_state_t`. As described in Section III, the latter can be of any type the developer sees fit – in our evaluation, it is an array of type `poet_cpu_state_t`.

TABLE II: Input and Configuration Details.

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

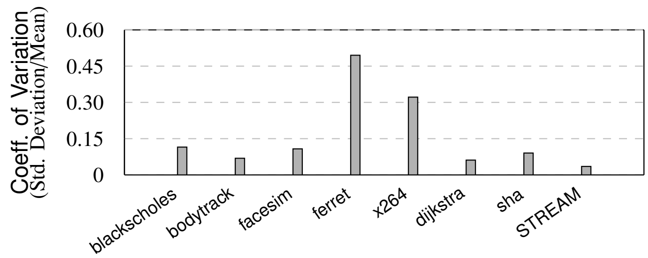

Fig. 3: Application Latency Variability.

C. Application Inputs

Table II lists the inputs used for each application. All inputs are packaged with the original benchmarks, except for the x264 input which comes from a set of standard test sequences. The server-class system used in this paper is significantly more powerful than the embedded systems POET was originally evaluated with. The overhead of changing resource allocations is also higher due to the larger core count. As a result, we increased both the size or length of some inputs and the window period size.

We quantify the inherent unpredictability of the applications by measuring the each job's latency, then computing the standard deviation and mean over all jobs in an application. Figure 3 demonstrates the ratio of standard deviation to mean for each application when running without POET. The applications have a range of natural behavior, from low variance which implies natural predictability (*e.g.,* bodytrack and dijkstra), to high variance which means that the application naturally has widely distributed latencies (*e.g.,* ferret and x264).

V. Experimental Evaluation

POET's experimental evaluation on the manycore server is divided into four parts. First, we demonstrate POET's ability to meet the latency requirements, then compare the energy consumption results to optimal. Next, we evaluate POET's ability to adapt to input with multiple phases, and finally, its ability to run subject to interference from another application.

A. Meeting Latency Targets

We demonstrate that POET is able to meet latency targets for each application on the manycore server system. First, we *characterize* each application i by executing in all possible configurations without POET. With these results, we determine the minimum average job latency m_i for each application and derive an oracle to be used for our analysis. This oracle determines an optimal resource schedule for each target without missing any deadlines, and has no computation or configuration switching overhead. Then we set latency targets for each application that range from 25% to 95% of their respective performance capacities. For example, a 25% goal

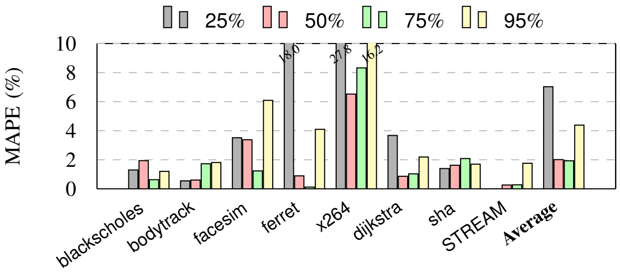

Fig. 4: Latency error (lower is better, 0 is optimal).

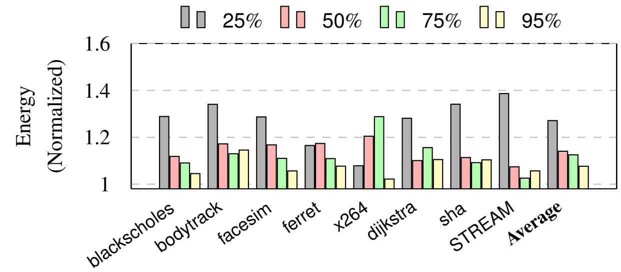

Fig. 5: Energy (lower is better, 1 is optimal).

means that the target is set to $4 \times m_i$. Applications are launched in the maximum-resource configuration (configuration $C - 1$ as described in Section II-B). POET observes application behavior during the first window period, then begins applying system changes.

To quantify POET's ability to meet the latency goals, we measure each job's latency and compare it to the goal. As was done in POET's prior analysis, we compute the Mean Absolute Percentage Error (MAPE), a standard metric in control theory for evaluating controller behavior [11]. For an application composed of n jobs:

$$MAPE = 100\% \cdot \frac{1}{n} \sum_{i=1}^{n} \begin{cases} d_m(i) > d_r : & \dfrac{d_m(i) - d_r}{d_r} \\ d_m(i) \leq d_r : & 0 \end{cases} \tag{9}$$

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

Figure 4 presents the MAPE values for each application for the four latency targets. The relationship between application variability (Figure 3 in Section IV-C) and MAPE is clear – higher variance typically results in higher MAPE. More volatile applications are more unpredictable and therefore more difficult to control. Still, the error values are generally low – on par with POET's behavior on embedded systems [22]. There are a few outliers, particularly with ferret and x264, which are both high variability applications. Ferret's threads are asynchronous, so work continues to be performed while system changes are being applied which introduces unpredictability into timing measurements. X264 is continuously creating and destroying threads, sometimes causing errors when assigning threads to cores with `taskset`. Further increasing the size of the window period or adjusting the pole value in POET's controller helps offset these kinds of issues, and is described further in Section V-E. MAPE penalizes every latency target violation, and error cannot be made up later by finishing jobs early. Still, POET achieves low MAPE for most executions on the manycore server system, despite the applications not being originally designed to provide predictable timing.

B. Energy Minimization

We now evaluate POET's energy efficiency in meeting the four latency targets by using the oracle derived from each application's characterization. Although the oracle has zero overhead, meeting all latency targets while simultaneously achieving optimal energy is not actually possible in practice as it would require knowledge of the future and no overhead.

Figure 5 presents the ratio of energy consumption to optimal for each latency target, where unity is optimal and values greater than 1 show energy consumed over optimal. Executions include the POET runtime overhead and the overhead imposed by changing system configurations. There are more resources to manage on the manycore server than on the embedded systems POET was originally evaluated on, which increases time and energy overhead. Despite the challenges, POET still achieves near-optimal energy consumption.

The 25% target is clearly the most inefficient, and in fact is not actually achievable for STREAM without idling, which POET does not support. Ferret and x264 appear to be efficient at the 25% target, but this is just a side effect of their high MAPE. These observations are consistent with studies on server-class systems that demonstrate how inefficient these machines are when running at low utilizations [2, 41].

C. Responding to Application Phases

We examine POET with an application input that exhibits changes in its behavior over time. In POET's prior analysis, we executed a video with the `x264` application that was a combination of three videos of varying encoding difficulty. This analysis is the same, except that we have increased each video phase length so that each phase is 1,500 jobs (frames), for a total of 4,500 jobs.

Figure 6 shows the time series data for latency and power consumption when running the application without POET in the highest resource configuration ($C - 1$). We normalize latency to the maximum recorded value. Frames that take less time are easier to encode, and require fewer system resources to meet a performance target compared to the frame that takes the most time. The phases are clearly distinguishable by the change in latency at frames 1,500 and 3,000. In the prior evaluation, we noted that the two embedded systems did not process each phase with the same relative latency. The first phase was the most difficult (highest latency) for both systems, but the second phase was the easiest (lowest latency) on one while the third was the easiest on the other. Now on our server system we find that the first and third phases are just about the same level of difficulty, and the second phase is easiest.

Figure 7 demonstrates enabling POET with a target that is about half of the system's maximum performance (twice the minimum latency). We launch the application in the highest resource configuration. During the first 100 frames, POET observes the application behavior, hence the low latency and higher power consumption. The first resource adjustment overshoots the latency target, reducing power consumption below where it will stabilize. Latency and power settle around frame 300, or the end of the second adjustment period. Later fluctuations are a result of variability in the input video (`x264` inputs exhibit high variance – see Figure 3). There is a

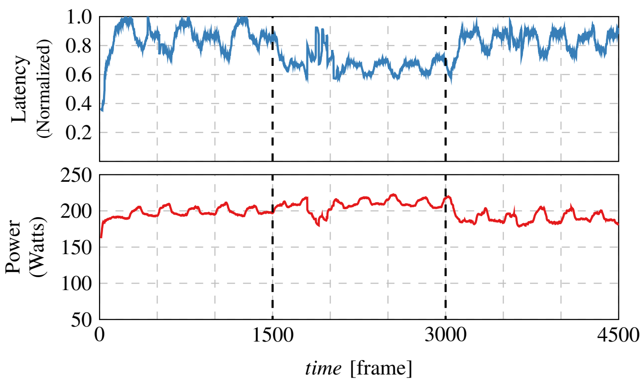

Fig. 6: Processing x264 input with distinct phases.

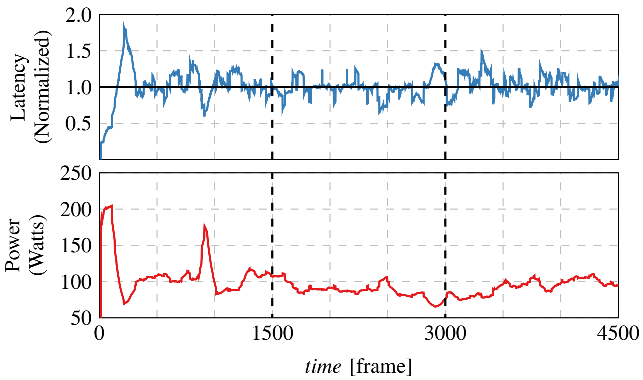

Fig. 7: Processing x264 input with distinct phases using POET.

discernible drop in power after frame 1,500, indicating the start of the second phase where fewer resources are required to meet the latency target. Power then increases after frame 3,000 when the processing again becomes more difficult. Despite these variations, latency goals are respected: MAPE is 5.6% and energy is 20.2% greater than optimal, which are similar to the `x264` results in Sections V-A and V-B.

D. Adapting to Other Applications

Finally, we demonstrate POET adapt to changes in system resource behavior at runtime. For this experiment, we launch the `bodytrack` application with POET and a performance target of about 50% capacity. Halfway through the execution, we launch an application in the background that does not use POET. This second application consumes system resources, slowing down the POET-controlled bodytrack execution. POET adapts by allocating more system resources, *i.e.,* increasing the DVFS speeds and/or allocating more cores to bodytrack. Bodytrack then continues to meet the original soft latency goal.

Figure 8 presents a time series for this scenario, including the POET-controlled execution and another that uses a static resource allocation strategy that fixes the resource assignments at the start of the execution. The y-axis is normalized to the latency target, and the vertical line indicates when the second application is launched. For this test, we reduce the window size from 50 to 40 frames which allows for more window periods during the execution but increases volatility. As with the previous experiments, we launch the POET-controlled bodytrack application in the highest-resource setting, configu-

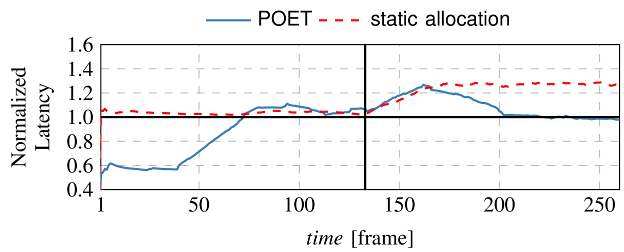

Fig. 8: POET adapting to a background application.

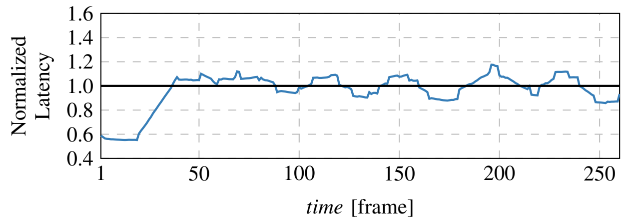

Fig. 9: POET with insufficient window size.

ration $C-1$. During the first window period, POET observes application behavior, then makes its first resource allocation decision at frame 40. By frame 80, the end of the first period of adjustment, the average window job latency is near the target. After the second application is launched, there is a temporary increase in latency. POET detects this change and allocates additional resources so that the latency goal continues to be met. This adaptation results in 5.0% MAPE for the entire execution. In contrast, the static allocation strategy fails to meet job deadlines after the second application begins, resulting in 23.9% MAPE.

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

E. Discussion of Results and Limitations

An important difference from the original POET analysis is the choice of the window period size for applications. For example, the `bodytrack` executions used a window size of 20 on the embedded systems, but we used a size of 50, and later 40, for evaluations on the manycore server-class system for the same application. Faster application performance and larger number of resources on the server-class system increase the relative overhead of changing system resource allocations at runtime, making window size changes necessary. Figure 9 demonstrates the results of using a window size of 20, which is too small, for a latency target of about 50% capacity (the same one used in Section V-D). Although MAPE is still low at 2.95%, the controller fails to converge, causing oscillations.

We measure the overhead of three resource allocation tasks: (1) the POET controller and optimizer, which we call *POET Core*, (2) application core assignment with taskset, which we call *Affinity*, and (3) changing frequency settings with *DVFS* for the 32 virtual cores. The latter two are executed by the platform-specific function defined by `poet_apply_func` (see Section III-B). Compared to a perfect implementation that requires no computation or resource allocation overhead and always meets the latency goal, each POET Core execution adds 0.12 ms average latency overhead, each Affinity change averages 62.24 ms, and each DVFS change averages 65.08 ms. The POET Core overhead is negligible, but the others add 2.36% and 2.47% timing overhead to the example in Figure 9, totaling almost 5%. That cost is like adding a whole additional frame to the window period. Increasing the window size to 50 reduces the Affinity and DVFS overhead to less than 1% each for this bodytrack performance target. Faster applications require longer window periods to reduce the performance impact caused by the fixed overhead of changing resource allocations.

The POET design models overhead as error and lets the control dynamics naturally correct any overhead. This approach works best on small-scale systems like those evaluated in the original POET paper [22], but clearly has drawbacks on the larger system evaluated in this paper. As explained above, we can overcome this drawback by using larger windows to amortize overhead. We could also extend POET to explicitly account for overhead and the cost of switching configurations. Such an approach would force POET to be conservative about switching configurations and likely reduce energy savings. A third approach would be to build hardware and operating system support for rapid configuration changes. We believe supporting this kind of adaptability is key for future multicore systems, as faster configuration changes increase the potential for energy savings.

Our results show that POET provides 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. 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.

The results also demonstrate some limitations of POET's 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 8 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 7 when controlling the high-variance x264 application. This is further evidence that there is a tension between timeliness and energy reduction [5]. Recent work has shown how to augment soft real-time systems with an additional layer to achieve hard real-time constraints [10]. Such an approach uses a system like POET to allocate resources for energy efficiency, but uses an additional mechanism to ensure that the deadlines are still met, even in the case of variability. However, such hard real-time guarantees come at some other costs.

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

Finally, POET currently assumes that only one of the running applications (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 8. 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 a best-effort mode.

VI. Related Work

Multicore processors are becoming increasingly configurable. They expose a variety of configurable resources, which software can adjust to tune the tradeoff between delivered performance and power or energy consumption. Examples include exposing multiple DVFS settings, low-power idle states, cores with aggressive clock-gating that use little energy when idle, and heterogeneous cores of varying capability. This flexibility allows the system to adapt to different circumstances or different application needs, but it comes at the cost of increasing software complexity. The problem is exacerbated when software must achieve portability across a range of different systems, all of which expose different resources to software.

One simple heuristic for minimizing energy is *race-to-idle*, which allocates all resources until a job completes and then idles the system until the next job arrives [2]. This heuristic is portable since it does not require knowledge about the system, but empirical studies show that it is not optimal [2, 7, 39, 40]. A recent study by Kim et al. demonstrates that an optimal solution requires knowledge of how the different configurable resources in a system affect the specific application under control – information which race-to-idle does not use [25]. The same study shows that race-to-idle is dominated by a *pace-to-idle* heuristic, *i.e.,* pace-to-idle is theoretically never worse than race-to-idle and can be much better.

It is not surprising that a number of different frameworks have arisen for intelligently controlling multiple resources to minimize energy. For example, Dubach et al. coordinate several microarchitectural features [8]. Many approaches coordinate various aspects of clockspeed and core usage [1, 4, 28, 32, 40]. METE is a control theoretic approach that simultaneously manages clockspeed, memory bandwidth, and core usage [35]. All of these approaches achieve great energy savings, but do so in a system-specific manner. For example, porting METE to a new system would require retuning the controller. If the new system exposes new resources (*e.g.,* heterogeneous core types), then the controller would have to be redesigned from scratch. Clearly portability across a range of multicore hardware requires a different approach.

Several frameworks have been proposed to meet real-time constraints by managing multiple resources. These approaches are typically implemented as middleware that take a specification of available resources and a performance goal, and then meet that goal [33, 36, 42]. These approaches provide portable real-time guarantees, which is itself a hard problem, but they do not provide energy savings. LEO is a machine learning system that can meet performance constraints with minimal energy consumption [31]. LEO is very accurate and provides high energy savings, even with no prior knowledge of the application currently running. Its approach is extremely portable, but also incurs very high overhead. Interestingly, LEO and POET have complementary weaknesses – POET has low runtime overhead, but requires prior knowledge in the form of a configuration model while LEO has high overhead, but requires no prior knowledge. The prior work most similar to POET is PTRADE, which also uses control theory to manage general collections of resources [14]. PTRADE minimizes power consumption, but not necessarily energy. In addition, PTRADE uses heuristic optimizations, which are not portable, while POET uses a true minimal-energy scheduling algorithm.

Cross-layer approaches coordinate approximate applications and with system resource usage [12, 15, 16, 26, 27, 29, 37]. This approach has shown great benefits for media applications which can switch to reduced accuracy algorithms in response changing system constraints, *e.g.,* reduced resource availability [37]. Such cross-layer approaches can achieve greater energy savings than adapting application or system configurations alone, but they require the application to alter its behavior dynamically. In contrast, POET provides portable, energy-minimal resource usage without application-level changes.

POET is inspired by prior approaches that abstract resource management for portability [14, 33, 36, 42]. It is unique in its energy awareness and the fact that it works across multiple systems without application-level changes (apart from adding POET calls in the first place).

VII. Conclusion

Prior work presented POET, a library and runtime designed to provide portable energy efficiency under performance constraints. Previous evaluations were confined to small-scale embedded and mobile systems, like an ARM big.LITTLE System-on-Chip with only 8W chip peak power dissipation. This paper expands POET's evaluation by running on a larger-scale multicore – an Intel dual-socket server with about 200W total peak power dissipation and an order of magnitude more configurations. The combined evaluations demonstrate that POET provides portable energy efficiency with soft performance guarantees on a large range of systems. POET's ability to adapt its own internal control models (via the Kalman filter) make it an example of a *self-aware* computing system, an emerging class of management systems that helps navigate conflicting requirements (e.g., achieving high performance with low power consumption) [9, 19, 20].

References

[1] N. AbouGhazaleh et al. "Integrated CPU and L2 Cache Voltage Scaling Using Machine Learning". In: *LCTES*. 2007.

[2] L. A. Barroso and U. Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (2007).

[3] C. Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *17th Conference on Parallel Architectures and Compilation Techniques*. 2008.

[4] R. Bitirgen et al. "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach". In: *MICRO*. 2008.

[5] G. C. Buttazzo et al. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer, 2006.

[6] A. Carroll and G. Heiser. "Mobile Multicores: Use Them or Waste Them". In: *HotPower*. 2013.

[7] H. Cheng and S. Goddard. "SYS-EDF: a system-wide energy-efficient scheduling algorithm for hard real-time systems". In: *IJES* 4.2 (2009).

[8] C. Dubach et al. "A Predictive Model for Dynamic Microarchitectural Adaptivity Control". In: *MICRO*. 2010.

[9] N. D. Dutt et al. "Toward Smart Embedded Systems: A Self-aware System-on-Chip (SoC) Perspective". In: *ACM Trans. Embedded Comput. Syst.* 15.2 (2016).

[10] A. Farrell and H. Hoffmann. "MEANTIME: Achieving Both Minimal Energy and Timeliness with Approximate Computing". In: *USENIX ATC*. 2016.

[11] A. Filieri et al. "Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees". In: *ICSE*. 2014.

[12] J. Flinn and M. Satyanarayanan. "Managing Battery Lifetime with Energy-aware Adaptation". In: *ACM Trans. Comput. Syst.* 22.2 (2004).

[13] J. L. Hellerstein et al. *Feedback Control of Computing Systems*. 2004.

[14] H. Hoffmann et al. "A generalized software framework for accurate and efficient management of performance goals". In: *EMSOFT*. 2013.

[15] H. Hoffmann. "CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems". In: *ECRTS*. 2014.

[16] H. Hoffmann. "JouleGuard: energy guarantees for approximate applications". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 2015.

[17] H. Hoffmann et al. "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments". In: *ICAC*. 2010.

[18] H. Hoffmann et al. "Remote Store Programming: A Memory Model for Embedded Multicore". In: *HiPEAC* (2010).

[19] H. Hoffmann et al. *SEEC: A Framework for Self-Aware Computing*. Tech. rep. MIT-CSAIL-TR-2010-049. MIT, 2010.

[20] H. Hoffmann et al. "Self-aware Computing in the Angstrom Processor". In: *DAC*. 2012.

[21] 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.

[22] C. Imes et al. "POET: a portable approach to minimizing energy under soft real-time constraints". In: *RTAS*. 2015.

[23] S. Iqbal et al. "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems". In: *Computer Architecture Letters* 9.2 (2010).

[24] B. Jeff. "Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration". In: *DAC*. 2012.

[25] D. H. K. Kim et al. "Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics". In: *CPSNA*. 2015.

[26] M. Kim et al. "xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems". In: *ACM TECS* 11.4 (2013).

[27] 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).

[28] M. Maggio et al. "Power Optimization in Embedded Systems via Feedback Control of Resource Allocation". In: *IEEE TCST* 21.1 (2013).

[29] M. Maggio et al. "A Game-Theoretic Resource Manager for RT Applications". In: *Conference on Real-Time Systems (ECRTS)*. 2013.

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

[31] N. Mishra et al. "A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints". In: *ASPLOS*. 2015.

[32] V. Petrucci et al. "Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems". In: *HotPower*. 2012.

[33] R. Rajkumar et al. "A Resource Allocation Model for QoS Management". In: *RTSS*. 1997.

[34] E. Rotem et al. "Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge". In: *Hot Chips*. 2011.

[35] A. Sharifi et al. "METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management". In: *SIGMETRICS PER* 39.1 (2011).

[36] 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).

[37] V. Vardhan et al. "GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy". In: *IJES* 4.2 (2009).

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

[39] C.-Y. Yang et al. "System-Level Energy-Efficiency for Real-Time Tasks". In: *SOCRTD*. 2007.

[40] H. Yun et al. "System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks". In: *ECRTS*. 2010.

[41] H. Zhang and H. Hoffmann. "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques". In: *ASPLOS*. 2016.

[42] R. Zhang et al. "ControlWare: a middleware architecture for feedback control of software performance". In: *ICDCS*. 2002.