CoPPer: Soft Real-time Application Performance Using Hardware Power Capping

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

Huazhe Zhang
University of Chicago
huazhe@cs.uchicago.edu

Kevin Zhao
University of Chicago
kzhao@uchicago.edu

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

***Abstract*—Dynamic voltage and frequency scaling (DVFS) has been the cornerstone of innumerable software approaches to meeting application timing requirements with minimal energy. However, recent trends in technology—*e.g.,* moving voltage converters on chip—favor hardware control of DVFS, as hardware can both react faster to external events and perform fine-grained power management across a device. We respond to these trends with CoPPer, which instead uses hardware power capping to meet application performance requirements with high energy efficiency. We find that meeting performance requirements with power capping is more challenging than using DVFS because the relationship between power and performance is non-linear and has diminishing returns at high power values. CoPPer overcomes these difficulties by using adaptive control to approximate non-linearities and a novel *gain limit* to avoid over-allocating power when it is no longer beneficial. We evaluate CoPPer with 20 parallel applications and compare it to both a classic linear DVFS controller and to a sophisticated control-theoretic, model-driven software DVFS manager. CoPPer provides all the functionality of the sophisticated DVFS-based approach, without requiring a user-specified model or time-consuming, exhaustive application/system pre-characterization. Compared to DVFS, CoPPer's gain limit reduces energy by 6% on average and by 12% for memory-bound applications. For high performance requirements, the energy savings are even greater: 8% on average and 18% for memory-bound applications.**

I. Introduction

As energy usage and power dissipation have become key concerns for computer systems, a number of software approaches have arisen to manage the tradeoffs between timing and power/energy. The vast majority of those approaches use dynamic voltage and frequency scaling (DVFS) to set the processor frequency (and by extension the voltage) to trade compute capacity for reduced power consumption. This technique is especially useful for workloads with real-time or quality-of-service demands—the DVFS setting is adjusted so that the timing demands (*i.e.,* job performance or latency requirements) are just met and no additional power is dissipated.

Recent trends, however, indicate that exposing DVFS to software control is being deprecated; instead, future hardware will directly control frequency and voltage. Linux kernel developers acknowledge that manufacturers are moving DVFS management to hardware, beyond software's control [26]. Indeed, current Linux distributions no longer have default support for the `userspace` DVFS governor, which allowed software to explicitly set processor frequency. The Linux kernel documentation notes, "the idea that frequency can be set to a single frequency is fictional for Intel Core processors. Even if the scaling driver selects a single P-State, the actual frequency the processor will run at is selected by the processor itself" [7]. Intel processors have moved away from Speed Step, which allows software complete control of DVFS, to Speed Shift, which gives hardware control over DVFS settings. Intel claims that moving DVFS control from the OS to hardware provides 20-45% improvements in responsiveness for bursty workloads [21] and reduces the latency of DVFS changes to about 1/30th of the time it takes for software to make the same change [5].

There is one major drawback of moving DVFS to hardware: for applications with timing requirements, software has all the knowledge about both the current and desired performance. Existing software-based DVFS approaches select the lowest DVFS setting that meets an application's timing requirement. When DVFS is completely transitioned to hardware, software will need another mechanism to meet timing constraints with minimal energy. Fortunately, emerging interfaces let software set *power caps* on hardware, with hardware free to determine what DVFS settings should be used and when, so long as the average power over some time window is respected. For example, Intel's Running Average Power Limit (RAPL) allows software to set power limits on hardware [6]. The challenge is that meeting timing requirements with DVFS is easy: simple linear models map changes in clockspeed to changes in speedup. *Meeting timing requirements with power capping is harder: power and speedup have a non-linear relationship and most applications exhibit diminishing performance returns with increasing power.*

We prepare for the transition away from software DVFS management by proposing CoPPer (**Co**ntrol **P**erformance with **P**ow**er**), a software system that uses adaptive control theory to meet application performance goals by tuning hardware power caps. CoPPer has three key features. First, it works on applications *without prior knowledge* of their specific performance/power tradeoffs; *i.e.,* it does not require a system or application-specific power cap/performance model based on pre-characterization, making it suitable for general purpose computing workloads composed of repeated jobs. Second, it uses a Kalman filter to adapt control to *non-linearities* in the power cap/performance relationship. Third, it introduces adaptive *gain limits* to prevent power from being over-allocated when applications cannot achieve additional speedup.

That is, if a workload's performance does not improve with expanded power limits, CoPPer will not allocate additional power, whereas standard control-theoretic approaches take no additional power-saving action. Thus, CoPPer saves energy in many cases compared to existing DVFS-based approaches while *maintaining its formal guarantees*.

In summary, this paper makes the following contributions:

- Proposes using software-defined, hardware-enforced power capping instead of software-managed DVFS to more energy-efficiently meet application timing constraints.
- Presents CoPPer, a feedback controller that: meets performance goals by manipulating hardware power caps, handles non-linearity in power cap/performance tradeoffs, and introduces adaptive *gain limits* to further reduce power when it does not increase performance.
- Evaluates CoPPer using Intel RAPL, achieving better energy efficiency than software DVFS control, with similar timing guarantees. Specifically, CoPPer improves energy efficiency by 6% on average with a 12% improvement for memory-bound applications. At the highest performance targets, CoPPer's gain-limit saves even more energy: 8% on average and 18% for memory-bound applications.
- Open-source release of CoPPer reference implementation and benchmark patches.[1]

In short, CoPPer overcomes the difficulties in using software power capping to meet performance goals and improves energy efficiency over software-managed DVFS, which is becoming obsolete.

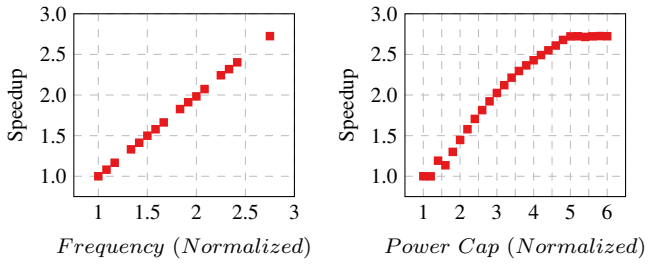
II. Background and Motivation

Many modern computer systems are underutilized, leading to significant portions of time where application performance requirements can be met with less than the full system capacity [3, 22, 23]. This trend has led to flourishing research in energy-aware scheduling that tailors resource usage to meet the performance requirements while minimizing energy. Software DVFS management has been essential in many energy-aware scheduling algorithms [1, 16, 31, 34]. Recent survey papers devote entire sections to the various ways DVFS has been used in scheduling systems [24, 35]. However, there are strong indications that DVFS will not be directly controllable by software in future processors.

A. The Future of Software DVFS

Since SandyBridge, Intel processors take software DVFS settings as suggestions, and hardware has been free to dynamically alter the actual clockspeed and voltage independently from the software-specified setting [7, 26]. With the Skylake architecture, Intel has been actively campaigning to move DVFS management wholly to hardware and instead have software specify power. The hardware is then free to rapidly change DVFS settings to achieve better performance while still respecting those power limits [21]. For example, if software sets power limits requiring any 50 ms time window to average a maximum of 100 W, hardware is free to use turbo mode to speed up the processing of any bursty work within that 50 ms, as long as it compensates by running in a low-power state for some of that time.

[1] Available at: https://github.com/powercap/



(a) DVFS (b) Power cap

Fig. 1: DVFS / Power cap performance impact for `vips`.

Of course, even as DVFS shifts to hardware, it is still the software's responsibility to provide its own notion of either "best" or "good enough" performance. The capability to specify power caps and simultaneously provide some optimization is already provided by interfaces like Intel's RAPL [6]. Recent works show that a combination of RAPL and software resource management can achieve even better performance or energy proportionality while guaranteeing power consumption [15, 32]. What is still needed, however, is a software component that can *guarantee performance constraints* while minimizing energy consumption. We address this need with CoPPer, which provides soft, application-level performance guarantees by manipulating hardware power caps, without the need to pre-characterize application workloads.

Two related RAPL-based works merit a brief comparison. PEGASUS meets timing constraints using a multi-step bang-bang controller, but requires (1) pre-characterization of workloads and (2) empirically-determined, workload-dependent values for both latency headroom thresholds and coarse-grained power cap deltas [18]. HyPPo meets SLAs using an observe-decide-act loop for heterogeneous workloads in cloud environments, but its SLAs are in the form of CPU utilization requests rather than actual application timing goals, and, unlike CoPPer, does not provide formal guarantees [2].

B. The Challenges of Actuating Power

Meeting performance targets with power caps instead of DVFS settings introduces new challenges. Figure 1 demonstrates how the compute-bound `vips` application's performance is affected by DVFS frequencies (Figure 1a) compared to processor power caps (Figure 1b) on our evaluation system. Three challenges are immediately apparent from the figures. First, DVFS produces a linear response in performance, but power capping is *non-linear*. Second, power capping has *diminishing returns*: as power increases, the change in performance becomes smaller. Third, *the range* of DVFS settings is much smaller than power settings: the ratio of the maximum to minimum DVFS setting is 2.75, but power capping has a ratio of over 6 (as can be seen from the x-axes).

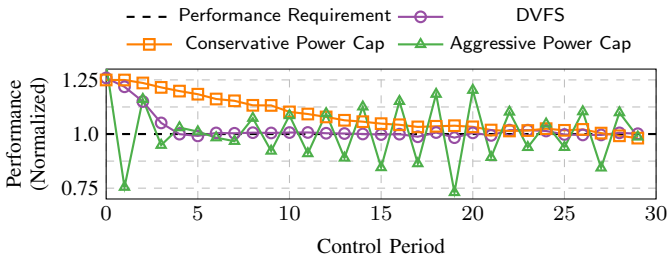


Fig. 2: DVFS and power capping with linear models.

The linear relationship between DVFS and performance makes it easy to apply textbook control-theoretic techniques to build a performance management system based on DVFS, and many examples exist in the literature [8, 9, 14, 17, 27, 29, 33]. With DVFS, control models assume that—for compute-bound applications—a 2× change in frequency produces a 2× change in performance. Applying the same techniques to build a performance management system based on power capping is more complicated. The major issue is that controllers based on time-invariant linear models will have varying error dependent on the current power cap. The simple solution is to build a linear model that never overestimates the relationship between power and speedup [8]. The downsides to this approach are: (1) a developer must know the maximum error for any application the system might run and (2) using such a conservative estimate slows down the controller's reaction to changes in the application or environment.

Figure 2 shows the difference between a controller based on a linear DVFS model extracted from Figure 1a and two (one conservative and one aggressive) based on fitting time-invariant linear models to the power capping data from Figure 1b. All approaches start at the maximum DVFS or power setting and must bring performance down to the required level while minimizing energy. Figure 2 shows the DVFS controller quickly reaches the desired performance, but the conservative power capping controller is much slower to react. The conservative approach never violates the performance requirement, but its slow reaction wastes energy. The aggressive approach overreacts, oscillating around the performance target instead of settling on it. These results demonstrate how sensitive power capping approaches can be to their input models. The next section describes an adaptive control design for meeting performance goals with power capping that overcomes the difficulties highlighted by this example without requiring a user-specified model.

III. A General Power Capping Design

CoPPer's goal is to provide soft performance guarantees, with the competing goal of keeping power as low as possible. To achieve the best energy efficiency, a power capping framework for meeting performance targets must not allocate more power than is actually needed by an application. CoPPer uses an adaptive control-theoretic approach to meet soft real-time performance constraints and employs a *gain limit* to proactively reduce power consumption when it determines that power is over-allocated. For maximum portability, *CoPPer is independent of any particular system, application, and power capping implementation.*

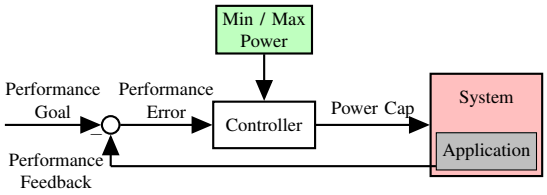


Fig. 3: CoPPer's feedback control design.

A. Adaptive Controller Formulation

Figure 3 presents CoPPer's feedback control design. CoPPer requires three pieces of information at runtime: (1) the soft **performance goal**, (2) **performance feedback**, and (3) the **minimum and maximum power** that the system allows. A user provides CoPPer with the performance goal, P_{ref}, which is just the inverse of a job latency deadline τ, *i.e.,* $P_{ref} = \tau^{-1}$. At runtime, the application measures its own performance, $p_m(t)$, which it provides to CoPPer. The minimum and maximum power caps, U_{min} and U_{max}, are system properties that can often be determined at runtime.

The **controller** first computes the **performance error**, followed by a *speedup* value:

$$e(t) = P_{ref} - p_m(t) \tag{1}$$

$$s(t) = gain(t) \cdot \left(s(t-1) + \frac{e(t)}{b(t)} \right) \tag{2}$$

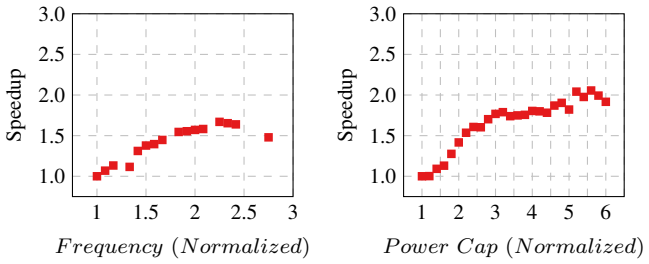
where $s(t-1)$ is the speedup signal generated in the previous iteration, $b(t)$ is the base speed estimate produced by a Kalman filter [30], and $gain(t)$ (where $0 < gain(t) \leq 1$) is a time-varying value that scales the control response. The gain is described in more detail shortly (Section III-B). Finally, the new **power cap** to be applied is computed as:

$$u(t) = U_{min} \cdot s(t) \tag{3}$$

Figure 1b (Section II-B) shows that, unlike with DVFS frequencies, a scalable compute-bound application's speedup is a non-linear function of the power cap. Figure 2 then illustrates how formulating a controller based on a linear model can cause the controller to converge very slowly, or to not converge at all. CoPPer overcomes this limitation by treating the application's base speed, $b(t)$, as a time-varying value and estimating it with a Kalman filter. In practice, this approach is analogous to estimating a non-linear curve with a series of tangent lines, each with slope $b(t)$. Thus, CoPPer's use of the Kalman filter allows it to overcome the problematic non-linear relationship between performance and power caps.

B. The Gain Limit

In many cases, allocating a higher power cap increases power consumption without actually increasing an application's performance. With `vips` in Figure 1 (Section II-B), performance scales linearly as higher DVFS frequencies are



(a) DVFS (b) Power cap

Fig. 4: DVFS / Power cap performance impact for HOP.

applied, but eventually a predefined maximum allowable frequency is reached and TurboBoost is enabled. TurboBoost allows the processor to run at higher frequencies for short periods of time, at the cost of higher power consumption and heat generation. In contrast, *performance increases non-linearly* as higher power caps are applied, until a little beyond the system's thermal design power (TDP), *i.e.,* the power level at which the processor is designed to safely dissipate heat under most workloads. Unfortunately, TDP is not a reliable indicator of the maximum power cap that can be applied efficiently. Figure 4 demonstrates the power cap/performance behavior of the HOP application, where performance levels-off well before the system TDP, and begins exhibiting performance unpredictability before beginning to achieve some small increases in average performance again once the power cap is *greater than* than the system TDP. These behaviors make it difficult for controllers to efficiently meet performance targets. CoPPer uses a *gain limit* to avoid over-allocating power when it is not useful, *e.g.,* for unachievable performance targets in vips or moderate targets in HOP (Figures 1 and 4).

The gain limit is used in computing the $gain(t)$ term in Eqn. 2. This term is initially 1, and will remain so until the controller settles. Based on the performance history, gain may be reduced to lower the speedup when CoPPer detects that the extra speedup is not beneficial (and thus wasting power). In general, the convex properties of performance/power tradeoff spaces ensure that reducing the speedup never increases power consumption, and in most cases reduces it.

Intuitively, if the performance error value computed in Eqn. 1 is low, then the system has converged to the performance target and the speedup signal should remain where it is. However, if error values are high but the *difference* in error values between iterations is low, the controller has settled, but the performance target is not achievable. It may then be beneficial to reduce the speedup, and thus the power. Speedup is reduced by setting gain to:

$$gain(t) = 1 - \alpha_c \cdot e_{ns}(t) \cdot \Delta e_{ns}(t) \tag{4}$$

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

$$e_n(t) = \frac{|e(t)|}{P_{ref}} \tag{5}$$

$$\Delta e_n(t) = |e_n(t-1) - e_n(t)| \tag{6}$$

$$e_{ns}(t) = 1 - \frac{1}{e_n(t)+1} \tag{7}$$

$$\Delta e_{ns}(t) = \frac{1}{\Delta e_n(t)+1} \tag{8}$$

Since P_{ref} is the performance target, $e_n(t)$ is the absolute normalized performance error. $\Delta e_n(t)$ in Eqn. 6 is the absolute change in $e_n(t)$ since the previous iteration. Eqns. 7 and 8 compute values $e_{ns}(t)$ and $\Delta e_{ns}(t)$, which determine how much impact $e_n(t)$ and $\Delta e_n(t)$ have on reducing the speedup. Both $e_{ns}(t)$ and $\Delta e_{ns}(t)$ lay in the unit circle. As normalized performance error $e_n(t)$ approaches 0, $e_{ns}(t)$ also approaches 0, which reduces the impact of the gain limit in Eqn. 4. Conversely, if the error is high, $e_{ns}(t)$ approaches 1 and the gain limit will have a greater impact on the speedup. As the change in normalized performance error $\Delta e_n(t)$ approaches 0, $\Delta e_{ns}(t)$ approaches 1, thus increasing the gain limit's impact in Eqn. 4. Conversely, if the change in error is high, $\Delta e_{ns}(t)$ approaches 0 and the gain limit will have less impact on the speedup. Therefore, Eqn. 4 reduces the speedup signal by a factor of at most α_c, with the greatest change in speedup occurring when the absolute performance error $e_n(t)$ is high and the absolute change in error $\Delta e_n(t)$ is low. Setting $\alpha_c = 0$ disables the gain limit entirely, corresponding to $gain(t) = 1$ in Eqn. 2.

In practice, Eqn. 2 clamps the speedup value at $\frac{U_{max}}{U_{min}}$ prior to applying the gain to prevent slow controller response if P_{ref} was previously unachievable.[2] High performance errors force a speedup value too high for the gain to overcome if the speedup is not clamped first. In cases of high performance error, the gain limit's effectiveness is also constrained by the accuracy of U_{max} and U_{min}. If speedup is not clamped at a reasonable upper value, the gain limit must be quite high to overcome the inaccuracy.

Recall that CoPPer holds $gain(t)$ at 1 until the controller converges. Until then, the controller is an adaptive deadbeat control system that retains the corresponding control-theoretic guarantees [8]. Specifically, it can converge to the desired performance in as little as one control iteration. At that point, CoPPer computes Eqns. 4–8, which may change $gain(t)$. In control-theoretic terms, a non-zero gain is equivalent to adding a zero to the characteristic equation of the closed loop system. Given the definition of $gain(t)$, it is equivalent to a zero greater than 1, which moves the controller in the opposite direction from the feedback signal. Normally, this would be undesirable behavior, but this is exactly the behavior we want when we are no longer seeing performance improvements by increasing the power cap.

[2] In control terminology, this is an *anti-windup* mechanism.

C. Using CoPPer

Application-level feedback provides high-level metrics that are conducive to goal-oriented software and has been shown to provide a more reliable measure of application progress than low-level metrics like performance counters or memory bandwidth [10]. Many applications that are subject to performance constraints already measure performance and integrate with runtime DVFS controllers to meet performance targets. A performance target is any positive real value that makes sense for the application, and can conceivably be configured from any number of sources, *e.g.,* a command line parameter, a configuration file, or dynamically via a software interface. At desired time or work intervals called *window periods* (described further in Section IV-B), the application measures its performance and calls the controller. Developers for this class of applications already perform these tasks, so all that remains is to replace function calls to an existing DVFS controller with those for CoPPer.

CoPPer is designed to be independent of any particular system, application, and power capping implementation. It is initialized with a performance target, the minimum and maximum allowed power values, and the starting power cap.[3] After each window period, the `copper_adapt` function is called with an identifier and the current application performance. This function returns the new power cap, which is then applied to the system. For example:

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

Listing 1: Using CoPPer to compute and apply power caps.

The underlined functions simply replace the existing DVFS-related ones. Furthermore, the `apply_powercap` function is independent of CoPPer—an example is provided in the next section.

IV. Experimental Setup

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

A. Testing Platform

We evaluate CoPPer on an Ubuntu Linux system with kernel 3.13.0, configured to support both software-managed DVFS and Intel RAPL. To record runtime power behavior, we read energy from the Model-Specific Registers [12, 28]. Energy measurements are only used to evaluate CoPPer, they are *not* required in practice. For our experiments, we enable TurboBoost and set power caps for the RAPL *short_term* constraint at the *Package* level. We keep the system's default time window of 7812.5 μs. To apply RAPL power caps, we provide an easy-to-use tool called `RAPLCap`, but stress again that CoPPer is independent of the power capping implementation. For example, the `apply_powercap` function used in Listing 1 might be:

```
raplcap rc;

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

Listing 2: Applying a power cap with RAPLCap

The RAPL interface sets a limit on average power consumption over a time window, with hardware controlling DVFS and power allocation within that window.

B. Evaluation Applications

Our experiments use applications from the PARSEC benchmark suite [4], MineBench [25], `STREAM` [20], and `SWISH++` [19]. PARSEC provides a wide variety of parallel applications that exhibit different ranges of performance and power behavior. MineBench provides a representative set of data mining applications, some of which support parallel execution. `STREAM` is a synthetic benchmark that stresses main memory and represents memory-bound applications. `SWISH++` is a file indexing and search engine. All inputs are delivered with or generated directly from the benchmark sources, with the exception of `dedup` which uses a publicly available disc image, and `raytrace` and `x264` which are from standard test sequences.

Applications contain top-level loops, where each loop iteration completes a *job*. We instrument the applications with the Heartbeats interface to measure job performance, as real applications would [11]. As is common in control systems, CoPPer executes at fixed job intervals called *window periods*. For example, CoPPer will compute a new power cap every 50 video frames in `x264`.

Applications exhibit variability in their performance behavior, with some behaving more predictably than others. Figure 5 demonstrates the behavior of the applications used in this paper when running in an uncontrolled setting (default system power caps). Naturally, better predictability typically results in lower error in meeting performance targets, as will be shown in Section V-A.

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

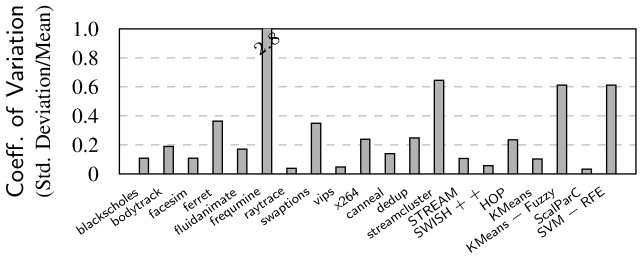


Fig. 5: Application job performance variability.

C. Execution and Analysis

Prior to performing the evaluation, we first characterize the behavior of all applications by running them without any control at each of the evaluation system's DVFS frequencies and measuring their performance and power behavior. These exhaustive characterizations are *only required for our analysis and not in practice*. We use the results to derive an oracle with perfect foreknowledge of job behavior and no computation overhead. The oracle runs at the highest-performance frequency for the application (which is not always the highest frequency or the TurboBoost setting) until a job completes, then sets the most energy-efficient frequency and aggressively places the processor cores in a low-power sleep state, with no delay or transition overhead.

The oracle is thus an ideal *performance* DVFS governor which never misses a performance goal, and is a good baseline for comparison. Modern Linux systems provide a real performance governor, which is not as efficient as our oracle since it does not know the highest-performing frequency for each application and incurs state transition overhead. With the exception of unachievable performance targets, we compare the energy consumption of all the executions in the evaluation against the oracle to determine their relative energy efficiency.

The different analyses compare CoPPer with various *gain limits* against a simple linear DVFS controller and a sophisticated DVFS controller that meets soft performance constraints and schedules for optimal energy consumption [13]. The simple linear DVFS controller estimates the ratio of control change (a primitive application-specific base speed estimate) in the first iteration, whereas a textbook controller requires this value at initialization and is rarely as good as our runtime estimate. It then uses an $O(log(n))$ algorithm to map speedup values to the lowest of n DVFS frequencies that meets the performance target, which is also an improvement over textbook approaches in that limiting the controller to discrete DVFS settings prevents oscillations.

The sophisticated DVFS controller requires a system model that maps DVFS frequencies to speedup and powerup values. It uses this model to divide window periods between two DVFS settings to meet a performance target precisely, where the schedule is computed using an $O(n^2)$ algorithm to find the best energy consumption subject to the performance constraint. This approach results in low error and often higher energy efficiency than the simple approach, as Section V-A will show. We also use a much more efficient DVFS actuation function than the sophisticated DVFS controller comes with, reducing its actuation overhead by two orders of magnitude. We use a lower bound of 20 jobs per window period in our evaluation for the benefit of the sophisticated DVFS controller—a minimum of 20 jobs ensures less than 5% performance error in its scheduling. *CoPPer does not suffer this scheduling limitation*, but we make the accommodation for the DVFS controller anyway in an effort to provide the most challenging comparison possible.

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

V. Experimental Evaluation

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

A. Efficiently Meeting Performance Goals

We begin by quantifying CoPPer's ability to achieve high energy efficiency while meeting soft performance goals. We use *gain limits* of 0.0 (disabled) and 0.5. For this analysis, we consider the steady-state behavior of the controllers. Therefore, each controller is initialized with the same $s(t)$ value for $t = 0$ (see Eqn. 2 in Section III).

For each application, we define and evaluate three different performance goals which specify how much to favor performance over energy consumption: high, medium, and low. We define the high performance goal to mean that the application must maintain at least 90% of top performance. The medium and low goals correspond to maintaining 70% and 50% of top performance, respectively. We note that actual performance values provided to CoPPer are application-specific, as described in Section III-C (*i.e.,* not a percentage), and are chosen by the application designer or user depending on the application deployment context.

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

- *Energy efficiency* is the ratio of the ideal *performance* governor's energy consumption (as computed by the oracle) to the actual energy consumption achieved.

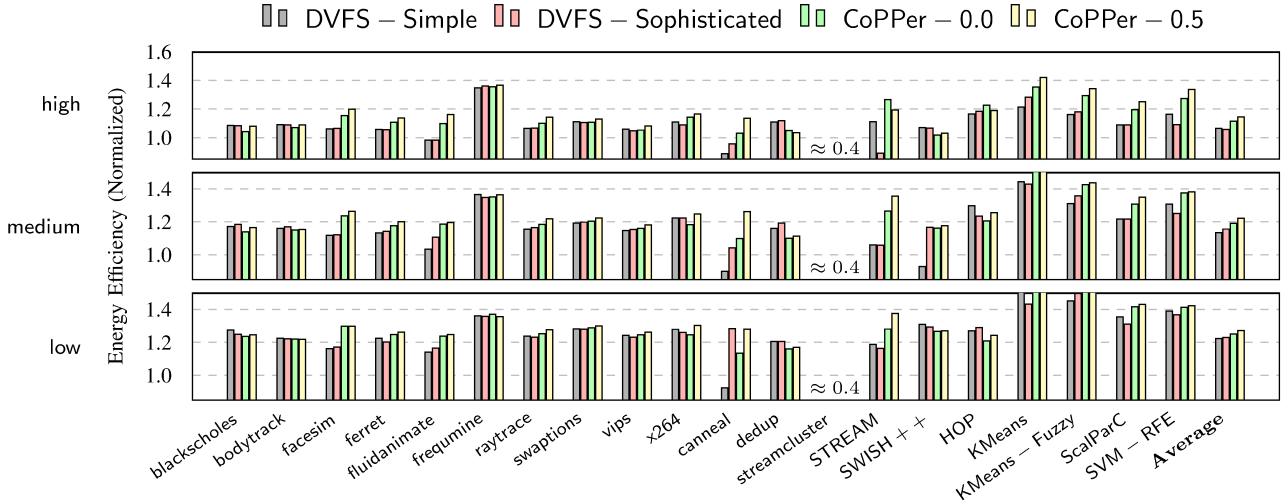


Fig. 6: Application energy efficiency for DVFS controllers and CoPPer, with and without a gain limit, for high, medium, and low performance targets (higher is better). Results are normalized to an ideal *performance* DVFS governor.

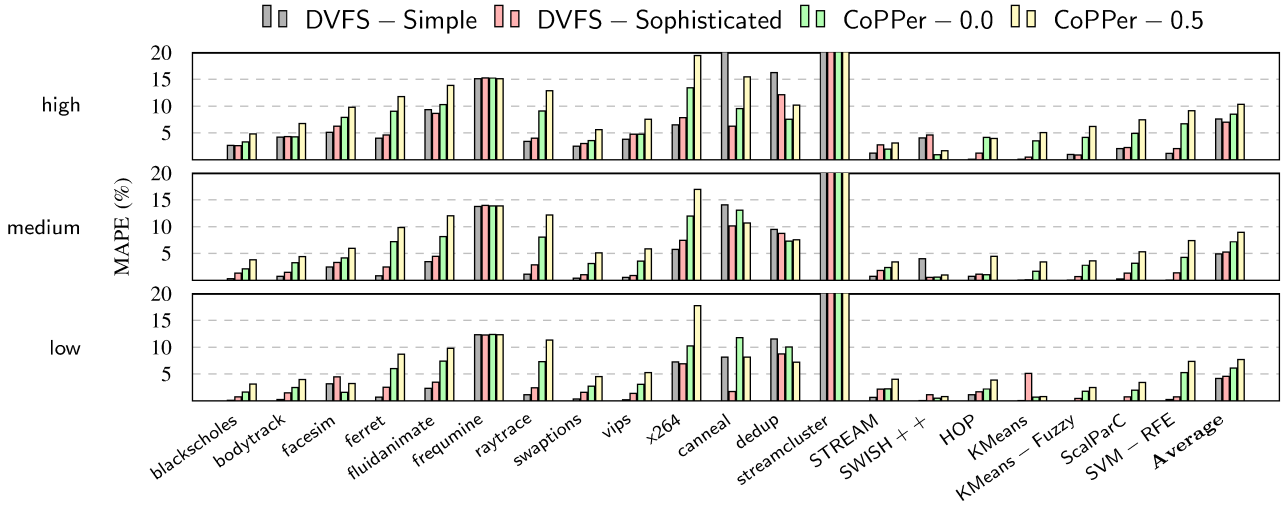


Fig. 7: Application performance error for DVFS controllers and CoPPer, with and without a gain limit, for for high, medium, and low performance targets (lower is better).

- *Mean Absolute Percentage Error* (MAPE) quantifies the error between the desired performance and the achieved performance; it is a standard metric for evaluating control systems [8].

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

$$\text{MAPE} = 100\% \cdot \frac{1}{n} \sum_{i=1}^{n} \begin{cases} p_m(i) < P_{ref}: & \frac{P_{ref} - p_m(i)}{P_{ref}} \\ p_m(i) \geq P_{ref}: & 0 \end{cases} \quad (9)$$

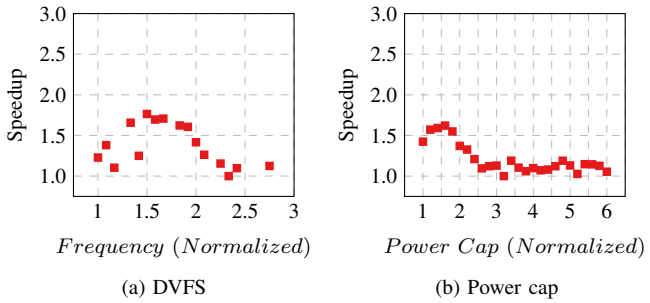
where $p_m(i)$ is the achieved performance for the i-th job. Each failure to achieve the performance target increases MAPE by an amount relative to how badly the target was missed.

Figures 6 and 7 present the energy efficiency and MAPE values for all applications and targets. Despite the challenges described in Section II-B (*e.g.,* non-linearity and larger range in the power cap/performance relationship), CoPPer achieves higher energy efficiency and similar MAPE compared to both the simple and sophisticated DVFS controllers for most applications and performance targets.

TABLE I: CoPPer energy efficiency with gain limits of 0.0 and 0.5 compared to the sophisticated DVFS controller.

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

Table I shows the average energy efficiency gains of CoPPer compared to the sophisticated DVFS controller for different performance goals. CoPPer is 3% more energy-efficient with no gain limit and 6% more efficient with gain limit 0.5. Note that CoPPer's energy efficiency gains increase as the performance goal increases. For high goals, the DVFS approach must



(a) DVFS (b) Power cap

Fig. 8: (a) DVFS and (b) power cap effects on `streamcluster`.

use TurboBoost, which is inefficient. CoPPer however, can set a power cap that allows the performance goal to be met and leave the decision to Turbo or not to hardware, which has more information about whether that choice is appropriate—exactly the motivation to move DVFS control to hardware and allow software to simply cap power.

`Freqmine` and `streamcluster` are outliers for both energy efficiency and MAPE. `Freqmine` is composed of repeated jobs, but its behavior is not predictable with feedback—the application uses a recursive algorithm that causes job performance to continually slow as it progresses. This behavior is quantified by its high job variability as shown in Figure 5 (Section IV-B). `Streamcluster` exhibits a performance/power tradeoff space that does not scale well beyond a fairly low DVFS setting or power cap, as demonstrated in Figure 8. In fact, its performance degrades dramatically as resource allocation increases. Even CoPPer's gain limit cannot adapt since it detects a change in performance when trying to reduce the power cap. The ideal *performance* DVFS governor (the oracle) knows not to allocate higher frequencies since it has access to the application-specific characterizations, but practical runtime controllers do not have this information. Using `streamcluster`-specific power cap ranges for CoPPer would produce results similar to the other applications; the DVFS controllers, however, would need whole new models.

B. Controlling Memory-bound Applications

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

We see that even without a gain limit, CoPPer already improves on the sophisticated DVFS controller's energy efficiency by 10% on average. With a gain limit of 0.5, the improvement rises to 12%. *These are significant energy savings.* CoPPer performs especially well compared to DVFS with these memory-bound applications for the higher performance targets. Again, even without a gain limit, CoPPer improves

TABLE II: CoPPer energy efficiency compared to the sophisticated DVFS controller for memory-bound applications.

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

TABLE III: Energy efficiency for unachievable performance targets, normalized to the sophisticated DVFS controller.

Gain Limit	Energy Efficiency
0.0	1.00
0.2	1.01
0.5	1.10
0.6	1.16
0.8	1.29
0.99	1.46

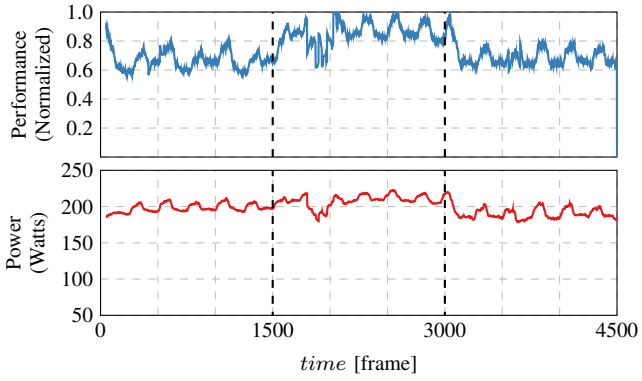
energy efficiency by 15% for the high performance target. DVFS can benefit from TurboBoost at high performance targets for many applications, but the higher DVFS frequencies also result in unnecessarily high energy consumption for memory-bound applications. By setting power caps instead of forcing DVFS frequencies, CoPPer achieves better energy savings by allowing the processor to scale frequencies more quickly between computational and memory-intensive periods. The gain limit provides significant energy savings for memory-bound applications with only a small loss in performance. In general, we advocate the use of 0.5 gain limit in practice since it produces almost no difference in MAPE, but can provide significant energy savings for memory-bound workloads.

C. Reducing Power for Unachievable Goals

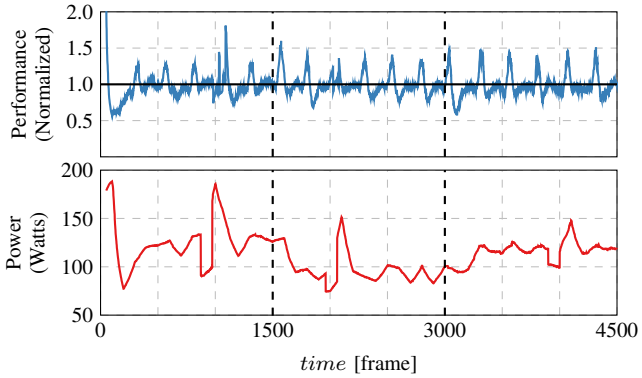
Sometimes performance targets simply are not achievable. This could be due to a user requesting too much from an application given the available processing capability, or the application may just want to run as fast as possible. When a performance target is unachievable, a naive resource controller will continue to increase resource allocations like DVFS frequencies or power caps in an attempt to improve performance, needlessly wasting energy. In this part of the evaluation, we demonstrate that CoPPer's *gain limit* helps avoid this pitfall. In Section III-B, we explained that the gain limit's effectiveness is constrained by the accuracy of the minimum and maximum power values. For this experiment, we use a more reasonable (and safer) maximum power limit—the evaluation system's TDP of 270 W.

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

The DVFS controller runs in the TurboBoost setting for the entirety of each execution. We also verified that the



(a) Uncontrolled behavior.



(b) Meeting a performance target with CoPPer.

Fig. 9: An `x264` input with distinct phases.

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

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

D. Adapting to Runtime Changes

This experiment demonstrates CoPPer's ability to respond to changes in application behavior at runtime. We run `x264` with a video input that exhibits three distinct levels of encoding difficulty. Figure 9a demonstrates the uncontrolled behavior of the input, with performance normalized to the maximum achieved. Dashed vertical lines denote where phase changes occur. The first phase has the lowest average performance and is therefore the most difficult to encode, followed closely by the third phase. The second phase has the highest performance, meaning it is the easiest part of the video to encode. Frames that are easier to encode offer an opportunity to save energy when meeting a performance target, as fewer resources are needed to satisfy the constraint. In the uncontrolled execution, power is consistently high as no changes to resource allocations are being made.

Figure 9b shows the time series for CoPPer with a gain limit of 0.0 for the medium performance target and a window size of 50 frames. Performance is normalized to the target. Now the performance remains mostly fixed (per the constraint), whereas the power consumption fluctuates as the power cap changes. Power values are now inversely proportional to the performance behavior seen in Figure 9a since CoPPer is able to reduce power consumption to save energy during phases of easier encoding. Of course, the actual power consumption recorded for any given frame does not necessarily match the power cap—it may be lower.

The fluctuations around the performance target in each phase are a result of input variability. The uncontrolled execution in Figure 9a testifies to the variability's presence in the input. We see similar behavior in other applications to varying degrees, but this visual clarifies where performance error (MAPE) comes from, and why some applications are difficult to control. Still, CoPPer meets the performance target with high energy efficiency and low error. For this execution, energy efficiency is 1.25 compared to the ideal *performance* governor (the oracle) and MAPE is 6.48%.

E. Multiple Applications

This section evaluates CoPPer's resilience to interference from another application. We begin the experiment by launching each application with a performance target. Roughly halfway through each execution, we launch a second application which was randomly selected from the PARSEC benchmark suite. The second application does not perform any DVFS or power control, but introduces interference into the system by consuming resources.

Figures 10 and 11 present the energy efficiency and MAPE results for each application. As should be expected, MAPE is higher than in previous experiments given that there is significant disturbance to system resources, which also makes the application more difficult to control even when the controller recognizes the disturbance and adapts to it. Some applications (*e.g.,* `facesim`, `canneal`, `dedup`, `streamcluster`, `STREAM`, and `SVM_RFE`) were not able to achieve the performance target after the second application is started, simply because there were not sufficient resources remaining in the system. Instead, the controller makes a best effort. These applications drag down the average energy efficiency, which otherwise remains high like in the single application analysis (*e.g.,* `KMeans and ScalParC`). Still, the average across all applications is nearly as good as the ideal *performance* governor would achieve in the absence of interference.

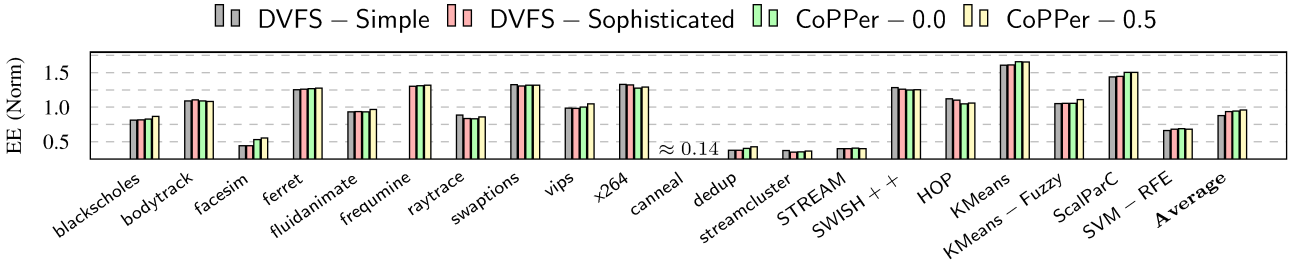


Fig. 10: Application energy efficiency for DVFS controllers and CoPPer, with and without a gain limit, under interference by a second application (higher is better). Results are normalized to an ideal *performance* DVFS governor.

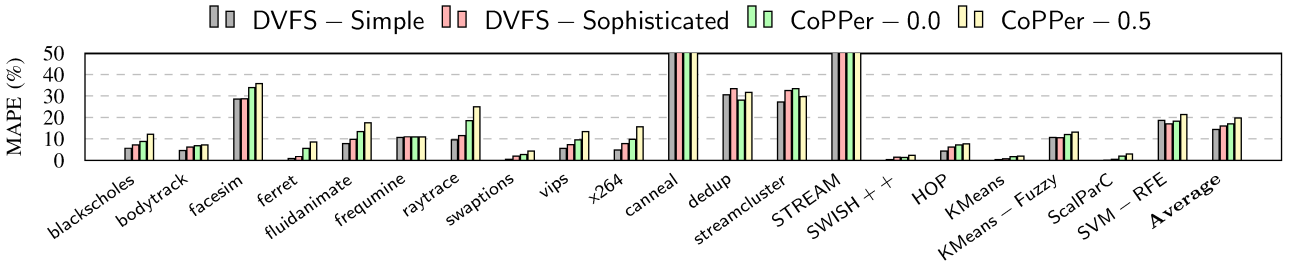


Fig. 11: Application performance error for DVFS controllers and CoPPer, with and without a gain limit, under interference by a second application (lower is better).

VI. Conclusion

Recent trends in processor design have increased the tension between hardware and operating systems designers over who should manage voltage and frequency scaling. Software can make better estimates about future processing requirements, but the hardware can react faster to events that dictate a need for adjustment. The latest Intel hardware is moving in the direction of limiting software access to DVFS, which negatively impacts the huge number of power and energy-aware schedulers that depend on DVFS for managing performance/power tradeoffs. We propose using software-specified, hardware-enforced power capping instead of software-managed DVFS to energy-efficiently meet application timing requirements. We present CoPPer, a control-theoretic approach that meets soft performance goals by manipulating hardware power limits, and evaluate it using Intel RAPL. CoPPer overcomes the challenges of non-linearity and wider control ranges in power cap/performance tradeoff spaces compared with DVFS. This paper demonstrates that controlling power limits provides similar performance guarantees as state-of-the-art DVFS approaches but with better energy efficiency, particularly when using CoPPer's *gain limit* to prevent over-allocation of power when it is not beneficial. Controlling power allows the hardware to have fine-grained control over frequency and voltage but still enables the huge array of power and energy-aware scheduling techniques that currently depend on DVFS.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. This research is supported by NSF (CCF-1439156, CNS-1526304, CCF-1823032, CNS-1764039). Additional support comes from the Proteus project under the DARPA BRASS program and a DOE Early Career award.

References

[1] S. Albers, "Algorithms for dynamic speed scaling," in *STACS*, 2011.

[2] M. Arnaboldi, R. Brondolin, and M. D. Santambrogio, "HyPPO: Hybrid performance-aware power-capping orchestrator," in *ICAC*, 2018.

[3] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, Dec. 2007.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[5] P. Bright, "The many tricks Intel Skylake uses to go faster and use less power," *Ars Technica*, Aug. 2015. [Online]. Available: http://arstechnica.com/information-technology/2015/08/the-many-tricks-intel-skylake-uses-to-go-faster-and-use-less-power/.

[6] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *ISLPED*, 2010.

[7] L. K. Documentation. (2016). Intel p-state driver, [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt.

[8] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *ICSE*, 2014.

[9] A. Goel, D. Steere, C. Pu, and J. Walpole, "SWiFT: A feedback control and dynamic reconfiguration toolkit," in *2nd USENIX Windows NT Symposium*, 1998.

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

[11] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application Heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments," in *ICAC*, 2010.

[12] C. Imes, L. Bergstrom, and H. Hoffmann, "A portable interface for runtime energy monitoring," in *FSE*, 2016.

[13] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "POET: A portable approach to minimizing energy under soft real-time constraints," in *RTAS*, 2015.

[14] C. Karamanolis, M. Karlsson, and X. Zhu, "Designing controllable computer systems," in *HotOS*, 2005.

[15] J. Krzywda, A. Ali-Eldin, E. Wadbro, P. stberg, and E. Elmroth, "ALPACA: Application performance aware server power capping," in *ICAC*, 2018.

[16] P. Kumar and L. Thiele, "p-YDS algorithm: An optimal extension of yds algorithm to minimize expected energy for real-time jobs," in *EMSOFT*, 2014.

[17] C. Lefurgy, X. Wang, and M. Ware, "Power capping: a prelude to power shifting," *Cluster Computing*, vol. 11, no. 2, 2008.

[18] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ISCA*, 2014.

[19] P. J. Lucas. (2014). SWISH++, [Online]. Available: http://swishplusplus.sourceforge.net/.

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

[21] Meghana R., "An overview of the 6th generation Intel Core processor (code-named Skylake)," *Intel Developer Zone*, Mar. 2016. [Online]. Available: https : / / software . intel . com / en - us / articles / an - overview - of - the - 6th - generation - intel - core - processor-code-named-skylake.

[22] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *ISCA*, 2011.

[23] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "CALOREE: Learning control for predictable latency and low energy," in *ASPLOS*, 2018.

[24] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems," *IJCAET*, vol. 6, no. 4, 2014.

[25] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A benchmark suite for data mining workloads," in *IISWC*, 2006.

[26] N. Pitre. (2014). Teaching the scheduler about power management, [Online]. Available: http : / / lwn . net / Articles/602479/.

[27] R. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using multiple input, multiple output formal control to maximize resource efficiency in architectures," in *ISCA*, 2016.

[28] E. Rotem, A. Naveh, D. R. amd Avinash Ananthakrishnan, and E. Weissmann, "Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge," in *Hot Chips*, 2011.

[29] M. H. Santriaji and H. Hoffmann, "GRAPE: Minimizing energy for GPU applications with performance requirements," in *MICRO*, 2016.

[30] G. Welch and G. Bishop, "An introduction to the Kalman filter," UNC Chapel Hill, Department of Computer Science, Tech. Rep. TR 95-041.

[31] F. F. Yao, A. J. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *FOCS*, 1995.

[32] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," in *ASPLOS*, 2016.

[33] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic, "ControlWare: A middleware architecture for feedback control of software performance," in *ICDCS*, 2002.

[34] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *RTAS*, 2012.

[35] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of energy-cognizant scheduling techniques," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 7, 2013.