Handing DVFS to Hardware: Using Power Capping to Control Software Performance

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 performance 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 nonlinearities and a novel *gain limit* to avoid over-allocating power when it is no longer beneficial. We evaluate CoPPer with 20 parallel applications on a dual-socket, 32-core server system and compare it to both a simple linear DVFS controller and to an existing control-theoretic, model-driven software DVFS manager. CoPPer does not require a user-specified model or application pre-characterization, yet provides all the functionality of the sophisticated DVFS-based approach. 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.

1. 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 performance 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 compute-bound workloads with real-time or quality-of-service demands—the DVFS setting is adjusted so that the performance demands 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. Kernel developers acknowledge that manufacturers are moving DVFS management to hardware, beyond software's control [39]. 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" [30]. 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 [35] and reduces the latency of DVFS changes to about 1/30th of the time it takes for software to make the same change [6].

There is one major drawback of moving DVFS to hardware: for applications with performance 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 performance requirement. When DVFS is completely transitioned to hardware, software will need another mechanism to meet performance goals 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 [9]. The challenge is that meeting performance requirements with DVFS is easy: simple linear models map changes in clockspeed to changes in speedup. *Meeting performance 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 manipulating 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-theoretical 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:

- Shows the need for a DVFS alternative that allows software to manage performance/power tradeoffs.
- Proposes software-defined power capping as a replacement for software-managed DVFS control.
- 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.
- Design and evaluation of CoPPer using Intel RAPL on a dual-socket, 32-core server. We find that CoPPer achieves performance guarantees similar to software DVFS control, but with better energy efficiency. Specifically, CoPPer improves energy efficiency by 6% on average with a 12% improvement for memory-bound applications. At the highest performance targets, CoPPer's gain-limit saves even more energy: 8% on average and 18% for memory-bound applications.
- Open-source release of CoPPer reference implementation and benchmark patches.[1]

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

2. Background and Motivation

We review a number of energy-aware schedulers that rely on DVFS. We then discuss the current processor landscape as justification for why DVFS might soon no longer be controllable by software. Finally, we illustrate how the simple linear models that work well for meeting performance requirements by tuning DVFS can produce sub-optimal behavior when directly applied to power capping.

2.1 DVFS and Scheduling for Energy

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, 36]. 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, 49]. Recent survey papers devote entire sections to the various ways DVFS has been used in scheduling systems [37, 52]. Resurveying this work is beyond the scope of this paper, but we highlight several examples to show the broad applicability of DVFS.

At the processor level, DVFS has been used to meet performance requirements [31, 48] and implement power capping [28]. Allowing DVFS to be set separately on different cores provides further benefits [26, 41]. Within a processor, separating the cache into its own voltage domain and scaling its frequency independent of the processor provides additional energy savings [2]. Similarly, managing DRAM DVFS increases energy efficiency [11, 12].

Indeed, with DVFS benefiting so many different components, it is clear that solutions which coordinate DVFS among components provide even better performance and energy benefits [36]. Examples include systems that coordinate DVFS with core usage [8, 33, 42], coordination of processor and DRAM DVFS [7, 10, 15, 29], and DVFS with thread scheduling [41, 47]. Several other approaches coordinate processor-wide DVFS with adaptations to the memory system and processor pipeline [5, 13, 45].

2.2 The Future of Software DVFS

There are strong indications that DVFS will not be directly controllable by software in future processors. Since SandyBridge, Intel processors take software DVFS settings as suggestions, and hardware has been free to dynamically alter

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

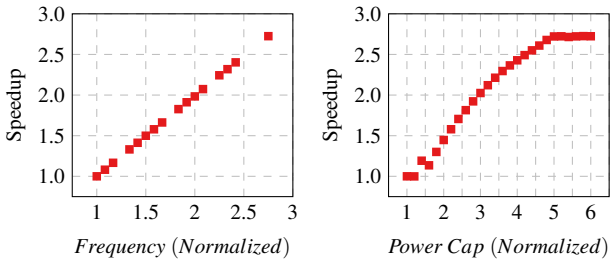
the actual clockspeed and voltage independently from the software-specified setting [30, 39]. With the Skylake architecture, Intel has been actively campaigning to move DVFS management wholly to hardware and instead have software specify power. The hardware is then free to rapidly change DVFS settings to achieve better performance while still respecting those power limits [35]. For example, if software sets power limits requiring any 50ms time window to average 100W, hardware is free to use turbo mode to speed up the processing of any bursty work within that 50ms, as long as it compensates by running in a low-power state for some of that time.

Of course, even as DVFS shifts to hardware, it will still be software's responsibility to provide its own notion of either "best" or "good enough" performance, subject to hardware-imposed constraints like thermal design power and energy consumption costs. The capability to specify power caps and simultaneously provide some optimization is already provided by interfaces like Intel's RAPL [9]. Recent work shows that a combination of RAPL and software resource management can achieve even better performance while guaranteeing power consumption [50]. What is still needed, however, is the software component that guarantees performance without using DVFS. We address this need with CoPPer, which provides soft performance guarantees by manipulating hardware power caps, thus allowing the hardware to perform fine-grained optimizations.

2.3 The Challenges of Actuating Power

DVFS is being replaced with hardware power capping, but meeting performance targets with power caps instead of DVFS settings introduces new challenges. Figure 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 $11/4 = 2.75$, but power capping has a ratio of over 6 (as can be seen from the x-axes).

The linear relationship between DVFS and performance makes it easy to apply textbook control theoretic techniques to build a performance management system based on DVFS, and many examples exist in the literature [16, 17, 27, 28, 40, 44, 51]. With DVFS, control models assume that—for compute-bound applications—a $2\times$ change in frequency produces a $2\times$ 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.



(a) DVFS (b) Power Cap

Figure 1: DVFS / Power Cap performance impact for `vips`.

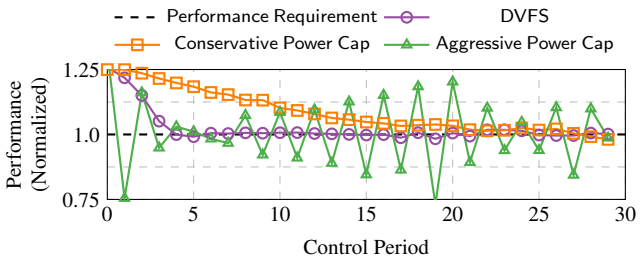


Figure 2: DVFS and power capping with linear models.

The simple solution is to build a linear model that never overestimates the relationship between power and speedup [16]. The downsides to this approach are: (1) a developer must know the maximum error for any application the system might run and (2) the overestimate slows the control reaction.

Figure 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 over-reacts, oscillating around the performance target instead of settling on it. These results demonstrate how sensitive the 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.

3. A General Power Capping Design

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

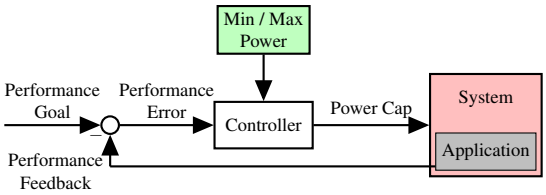


Figure 3: CoPPer's feedback control design.

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

3.1 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}. 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.

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

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

It then computes a *speedup* value as:

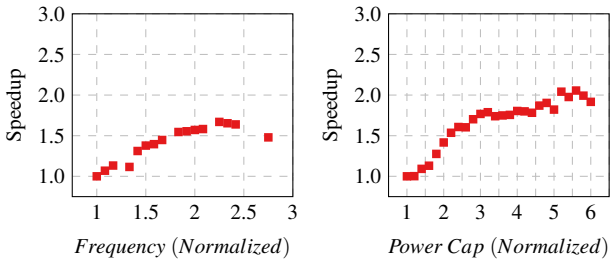
$$s(t) = max\left(1, gain(t) \cdot min\left(s(t-1) + \frac{e(t)}{b(t)}, \frac{U_{max}}{U_{min}}\right)\right) \tag{2}$$

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 [46], 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 3.2). Other feedback controllers use similar formulations, but without the gain [23, 24]. Clamping between 1 and $\frac{U_{max}}{U_{min}}$ prevents slow controller response if P_{ref} is sometimes unachievable.[2] Finally, the new **power cap** to be applied is computed as:

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

In Section 2.3, Figure 1b 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

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



(a) DVFS (b) Power Cap

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

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.

3.2 The Gain Limit

Applications exhibit different performance behaviors when controlling DVFS frequencies or power caps. In many cases, allocating more power to an application does not actually increase its performance. With `vips` in Figure 1, performance scales (linearly) as higher DVFS frequencies are applied, but eventually a predefined maximum allowable frequency is reached. Performance increases (non-linearly) as higher power caps are applied, until a little beyond the system's thermal design power (TDP). Unfortunately, TDP is not a reliable indicator of the maximum power cap that can be applied efficiently. Figure 4 demonstrates the power cap/performance behavior of the `HOP` application, where performance levels-off well before the system TDP, and begins exhibiting performance unpredictability before beginning to achieve some small increases in average performance again once the power cap is *greater than* than the system TDP. These behaviors make it difficult for controllers to efficiently meet performance targets. CoPPer uses a *gain limit* to avoid over-allocating power when it is not useful, *e.g.,* for unachievable performance targets in `vips` or moderate targets in `HOP`.

The gain limit is used in computing the $gain(t)$ term in 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 therefore is 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 therefore may be beneficial to reduce the speedup, and thus the power. Speedup is reduced by setting gain to:

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

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

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

$$\Delta e_n(t) = |e_n(t-1) - e_n(t)| \quad (6)$$

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

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

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 change 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 change in 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 $e_n(t)$ is low. Setting $\alpha_c = 0$ disables the gain limit entirely, corresponding to $gain(t) = 1$ in Eqn. 2.

Note that in Eqn. 2, the upper clamping is performed prior to applying the gain. High performance errors force a speedup value too high for the gain to overcome if the speedup is not clamped first. In cases of high performance error, the gain limit's effectiveness is also constrained by the accuracy of U_{max} and U_{min}. If speedup is not clamped at a reasonable upper value, the gain limit must be quite high to overcome the inaccuracy.

Recall that CoPPer holds $gain(t)$ at 1 until the controller converges. Until then, the controller is an adaptive deadbeat control system that retains the corresponding control theoretic guarantees [16]. 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.

3.3 Using CoPPer

Application-level feedback provides high-level metrics that are conducive to goal-oriented software and has been shown to provide a more reliable measure of application progress than low-level metrics like performance counters or memory bandwidth [20]. 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 make 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 4.2), 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 of our implementation is provided in the next section.

4. Experimental Setup

This section details the platform and applications used to evaluate CoPPer. We then quantify application performance variability, which directly impacts an application's ability to

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

be controlled. Finally, we describe the control approaches CoPPer is evaluated against.

4.1 Testing Platform

We evaluate CoPPer using a server-class system with dual Xeon E5-2690 processors running Ubuntu 14.04 LTS with Linux kernel 3.13.0. Each processor socket has its own memory controller and supports 8 physical cores and 8 Hyperthreads, for a total of 32 virtual cores in the system. The processors support 15 DVFS settings, 1.2–2.9 GHz, plus TurboBoost up to 3.3 GHz. We bind applications to all 32 virtual cores and interleave with both memory controllers using `numactl`. To record runtime power behavior, we read energy from each socket's Model-Specific Register and sum the two values [22, 43]. Energy measurements are only used to evaluate CoPPer, they are *not* required in practice.

RAPL supports a variety of zones, otherwise known as power planes, for controlling the power limits on different hardware components. The Core power plane controls the power cap for all cores in a socket while the Uncore power plane, typically only available on client hardware, controls the power cap for on-board graphics hardware. The DRAM power plane, only available on server-class hardware,[4] controls the power for main memory. The Package power plane encompasses the Core the Uncore power planes, the last-level cache, and memory controllers. Intel Skylake processors support the PSys (or Platform) power plane for managing the entire system-on-chip [14]. The Package and PSys zones both support *long_term* and *short_term* power constraints; other zones only support a single constraint. See the RAPL documentation for a more complete description.

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

To apply RAPL power caps, we provide an easy-to-use tool called `RAPLCap`, but we stress again that power capping implementations are independent of CoPPer. For example, the `apply_powercap` function used in Listing 1 might be:

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

```
raplcap rc;

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

Listing 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.

4.2 Applications and Inputs

Our experiments use applications from the PARSEC benchmark suite [4], MineBench [38], STREAM [34], and SWISH++ [32]. We instrument the applications with a modified Heartbeats interface to measure performance, as real applications would [21]. 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*. For very high performance applications, we batch a fixed number of iterations into a single job. As is common in feedback control systems, CoPPer executes at fixed job intervals called *window periods*. For example, CoPPer will compute a new power cap every 50 video frames in `x264`. We select window periods that are sufficiently long to prevent changing power caps too frequently, but small enough to allow CoPPer to adapt behavior in reasonably responsive times intervals. Table 1 presents our application configurations.

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 5.1.

If a window period is too short, the overhead of changing system settings combined with signal noise resulting from application variability prevents performance controllers from converging. Both actuation overhead and performance predictability can be improved by increasing the

Table 1: Application Inputs and Configurations.

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

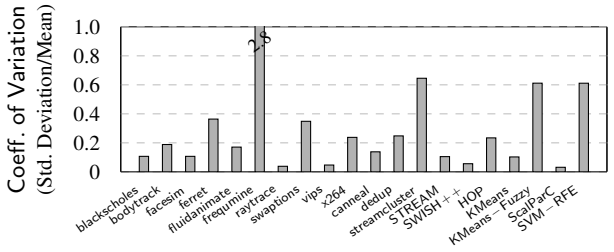


Figure 5: Application job performance variability.

size of window periods. However, longer window periods mean that it takes controllers more time to converge on a target and reduces their responsiveness to changes in application or system behavior. In practice, choosing a window period is dependent on the both the application and system properties, as well as the deployment context.

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

4.3 Execution and Analysis

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

The different analyses compare CoPPer with various *gain limits* against a simple linear DVFS controller (based on modification of an existing controller for meeting power requirements [28]) and a sophisticated DVFS controller that meets soft performance constraints and schedules for optimal energy consumption [24]. 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 5.1 will show. We also use a much more efficient DVFS actuation function than the sophisticated DVFS controller comes with, reducing its actuation overhead by two orders of magnitude.

We provide the DVFS controllers with linear models (*e.g.,* a $2\times$ change in frequency results in a $2\times$ change in performance and power), which works quite well on our

evaluation system. It should be noted, however, that poor models can cause slow, oscillating, non-convergent, or otherwise unpredictable behavior in model-driven controllers. In contrast to the DVFS controllers, CoPPer does not require a model, only the minimum and maximum power values, and therefore can run in constant $O(1)$ time.

5. Experimental Evaluation

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

5.1 Efficiently Meeting Performance Goals

We begin by quantifying CoPPer's ability to achieve high energy efficiency while meeting soft performance goals. We use *gain limits* of 0.0 (disabled) and 0.5. For this analysis, we consider the steady-state behavior of the controllers. Each controller is therefore initialized with the same $s(t)$ value for $t = 0$ (see Eqn. 2 in Section 3).

For each application, we define and evaluate three different performance goals which specify how much to favor performance over energy consumption: high, medium, and low. We define the high performance goal to mean that the application must maintain at least 90% of top performance. The medium and low goals correspond to maintaining 70% and 50% of top performance, respectively. While it is likely that most users would want high performance, we include the others to demonstrate CoPPer's ability to meet a range of different goals. We note that actual performance values provided to CoPPer are application-specific, as described in Section 3.3 (*i.e.,* not a percentage).

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

- *Energy efficiency* is the ratio of the ideal *performance* governor's energy consumption (as computed by the oracle) to the actual energy consumption achieved.
- *Mean Absolute Percentage Error* (MAPE) quantifies the error between the desired performance and the achieved performance; it is a standard metric for evaluating control systems [16].

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

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

Table 2: Energy efficiency ratio of CoPPer with gain limits of 0.0 and 0.5 to the sophisticated DVFS controller.

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

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 2.3 (*e.g.,* non-linearity and larger range in the power cap/performance relationship), CoPPer achieves higher energy efficiency and similar MAPE compared to both the simple and sophisticated DVFS controllers for most applications and performance targets. Compared to the sophisticated DVFS controller, CoPPer is on average 3% more energy-efficient with no gain limit and a 6% more efficient with gain limit 0.5.

Table 2 shows the average energy efficiency gains of CoPPer compared to the sophisticated DVFS controller for different performance goals. Note that CoPPer's energy efficiency gains increase as the performance goal increases. For high goals, the DVFS approach must use TurboBoost, which is inefficient. CoPPer however, can set a power cap that allows the performance goal to be met and leave the decision to Turbo or not to hardware, which has more information about whether that choice is appropriate—exactly the motivation to move DVFS control to hardware and allow software to 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 in Section 4.2. `Streamcluster` exhibits a performance/power tradeoff space that does not scale well beyond a fairly low DVFS setting or power cap, as demonstrated in Figure 8. In fact, its performance degrades dramatically as resource allocation increases. Even CoPPer's gain limit cannot adapt since it detects a change in performance when trying to reduce the power cap. The ideal *performance* DVFS governor (the oracle) knows not to allocate higher frequencies since it has access to the application-specific characterizations, but our runtime controllers do not have this information. If we were to specify `streamcluster`-specific power cap ranges for CoPPer, the results would be similar to the

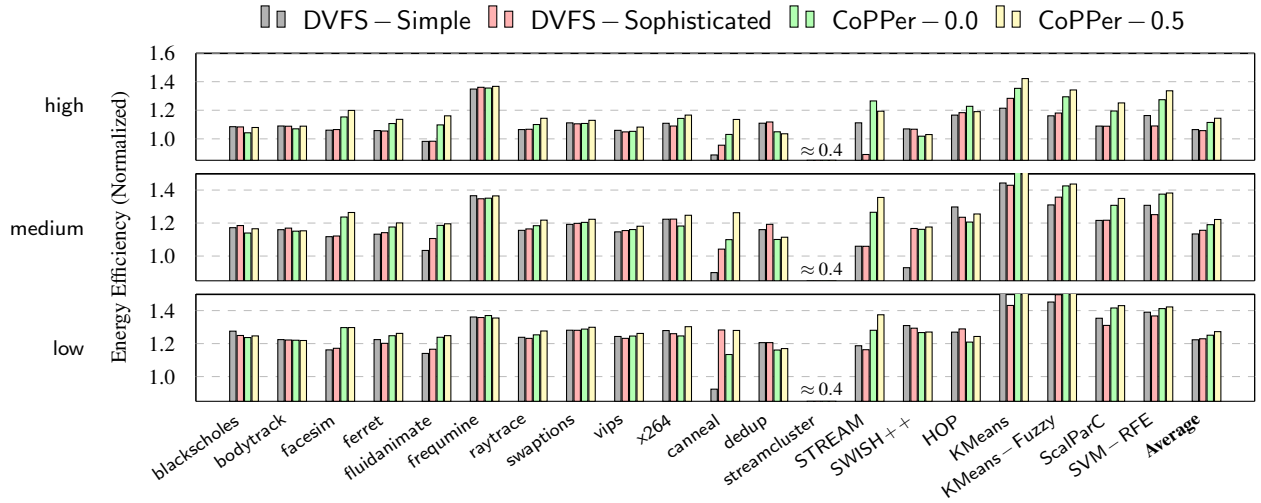


Figure 6: Application energy efficiency for DVFS controllers and CoPPer, with and without a gain limit, for high, medium, and low performance targets (higher is better). Results are normalized to an ideal *performance* DVFS governor.

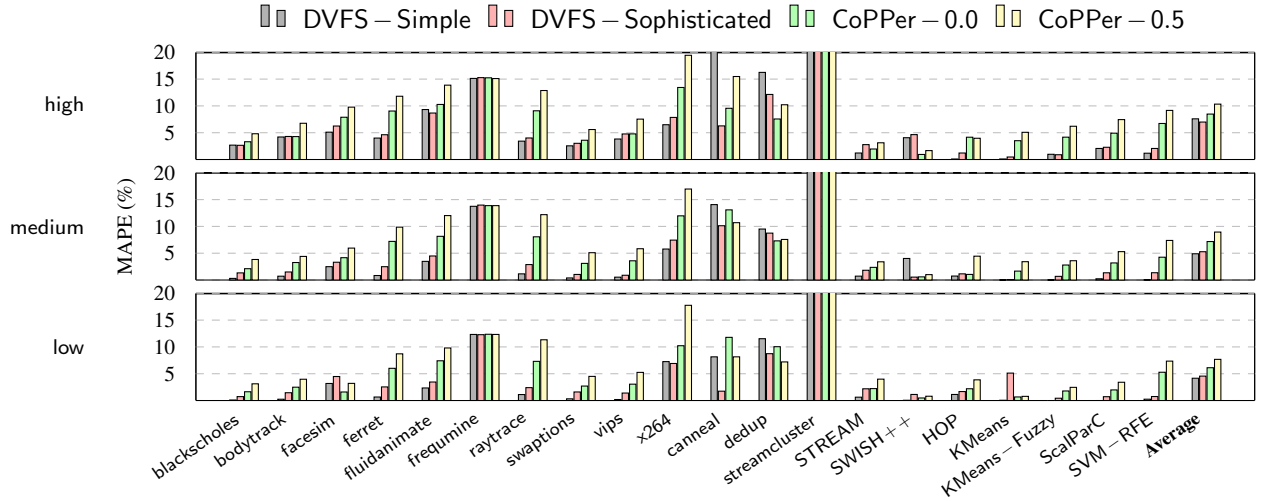


Figure 7: Application performance error for DVFS controllers and CoPPer, with and without a gain limit, for for high, medium, and low performance targets (lower is better).

other applications; the DVFS controllers, however, would need whole new models.

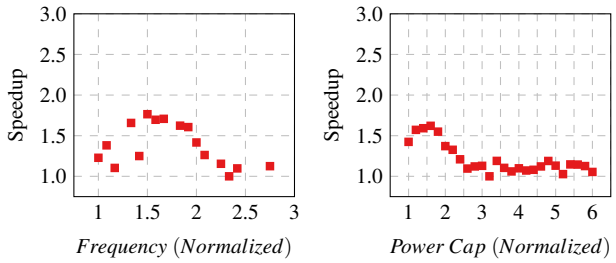
5.2 Controlling Memory-bound Applications

Our benchmark set contains 6 memory-bound applications: `kmeans`, `kmeans-fuzzy`, `ScalParC`, `stream`, `streamcluster`, and `SVM-RFE`. CoPPer achieves noticeably higher energy efficiency for these applications than with DVFS. Table 3 summarizes the average ratio of energy efficiencies across all performance targets for these, comparing CoPPer with and without a gain limit to the sophisticated DVFS controller.

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 sav-

Table 3: Energy efficiency ratio of CoPPer to the sophisticated DVFS controller for memory-bound applications.

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



(a) DVFS (b) Power cap

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

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

5.3 Reducing Power for Unachievable Goals

Sometimes performance targets simply are not achievable. This could be due to a user requesting too much from an application given the available processing capability, or the application may just want to run as fast as possible. When a performance target is unachievable, a naive resource controller will continue to increase resource allocations like DVFS 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 3.2, we explained that the gain limit's effectiveness is constrained by the accuracy of the minimum and maximum power values. For this experiment, we use a more reasonable (and safer) maximum power limit, the evaluation system's TDP of 270 W.

For each application, we set an unrealistically high performance target—1000× greater than what the system can actually achieve. We then execute both the sophisticated DVFS controller and CoPPer with a range of gain limit values. As the performance target is not actually achievable, MAPE is meaningless. Instead, we normalize energy efficiency to the sophisticated DVFS controller. Table 4 presents the results for select gain limits.

Table 4: Average runtime and energy efficiency for unachievable performance targets, normalized to the sophisticated DVFS controller.

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

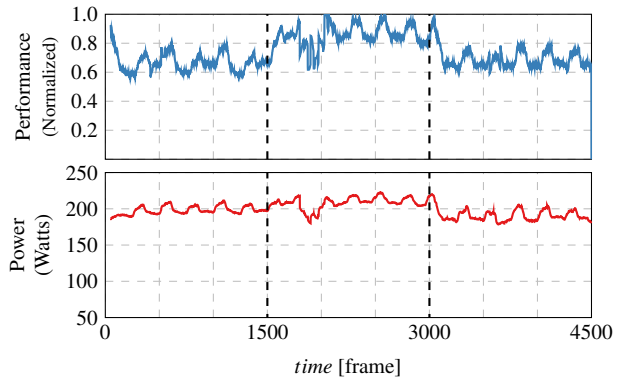
The DVFS controller runs in the TurboBoost setting for the entirety of each execution. We also verified that the simple DVFS controller and the evaluation system's real Linux performance governor achieved nearly identical results as the sophisticated controller. As should be expected, CoPPer without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

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.

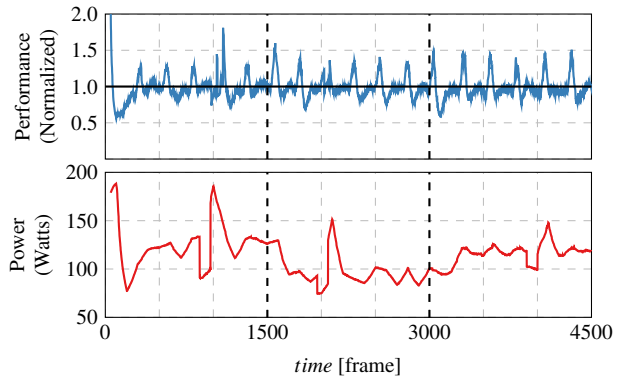
5.4 Adapting to Runtime Changes

This experiment demonstrates CoPPer's ability to respond to changes in application behavior at runtime. We run `x264` with a video input that exhibits three distinct levels of encoding difficulty. Figure 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 propor-



(a) Uncontrolled behavior.



(b) Meeting a performance target with CoPPer.

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

tional 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%.

5.5 Multiple Applications

This section evaluates CoPPer's resilience to interference from another application. Existing work in the literature addresses the problems of co-scheduling resources like cores and memory channels for applications [18, 19], but that is beyond the scope of CoPPer. 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.

5.6 Overhead Analysis

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

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

We compare the overhead of power capping and setting DVFS frequencies on our evaluation system. Each test is run one million times, with the average of 5 runs presented. Power capping alternates between applying the lowest and highest power caps (50 and 430 W, split between the two processor sockets) in each iteration, and achieves an average iteration time of **15.6** μ**s**. DVFS alternates between setting the minimum and maximum frequencies (1.2 GHz and TurboBoost) on all 16 physical cores, achieving an average iteration time of **118.5** μ**s**. As noted in Section 4.3, our experiments use a better actuation function than the sophisti-

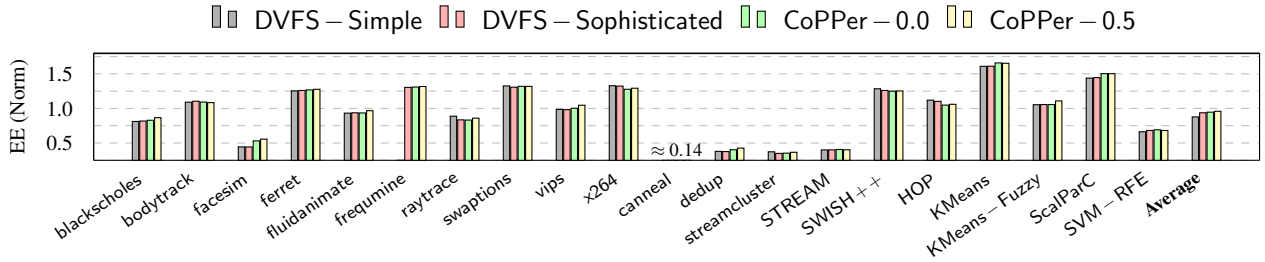


Figure 10: Application energy efficiency for DVFS controllers and CoPPer, with and without a gain limit, under interference by a second application (higher is better). Results are normalized to an ideal *performance* DVFS governor.

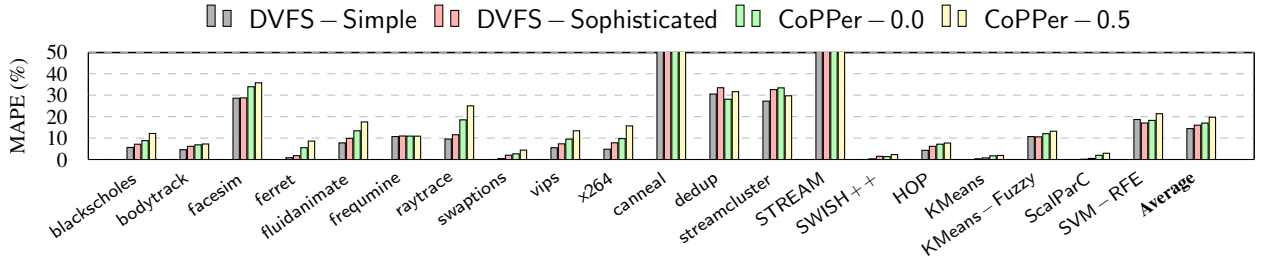


Figure 11: Application performance error for DVFS controllers and CoPPer, with and without a gain limit, under interference by a second application (lower is better).

cated DVFS controller comes with – this average overhead is much improved over the double-digit millisecond overhead the controller previously achieved [25]. Yet, the overhead imposed by power capping is still another order of magnitude lower than the improved DVFS overhead. Limiting software to coarse-grained power management at a socket level provides clear runtime benefits over more fine-grained DVFS management.

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

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

6. 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 conclude that these recent trends establish the need for an alternative to software control of DVFS. In response, we propose software-specified, hardware-enforced power capping as the solution. We present CoPPer, a feedback controller that meets soft performance goals by manipulating hardware power limits. While managing performance with DVFS is relatively simple, CoPPer overcomes the challenges of non-linearity and wider control ranges in power cap/performance tradeoff spaces. We evaluate CoPPer on a dual-socket multicore server, using RAPL as the power capping implementation. This paper demonstrates that controlling power limits provides similar performance guarantees with better energy efficiency than state-of-the-art DVFS approaches, particularly when using CoPPer's *gain limit* to prevent over-allocation of power when it is not beneficial. Controlling power gives the hardware 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.

References

[1] S. Albers. "Algorithms for Dynamic Speed Scaling". In: *STACS*. 2011.

[2] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures". In: *MICRO*. 2000.

[3] L. A. Barroso and U. Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (2007). ISSN: 0018-9162. DOI: 10.1109/MC.2007.443. URL: http://dx.doi.org/10.1109/MC.2007.443.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *PACT*. 2008.

[5] R. Bitirgen, E. Ipek, and J. F. Martinez. "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach". In: *MICRO*. 2008.

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

[7] J. Chen and L. K. John. "Predictive coordination of multiple on-chip resources for chip multiprocessors". In: *ICS*. 2011.

[8] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. "Pack & Cap: adaptive DVFS and thread packing under power caps". In: *MICRO*. 2011.

[9] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. "RAPL: Memory Power Estimation and Capping". In: *ISLPED*. 2010.

[10] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. "Coscale: Coordinating cpu and memory system dvfs in server systems". In: *MICRO*. 2012.

[11] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. "MemScale: Active Low-power Modes for Main Memory". In: *SIGPLAN Not.* 47.4 (Mar. 2011).

[12] B. Diniz, D. Guedes, W. Meira Jr., and R. Bianchini. "Limiting the power consumption of main memory". In: *ISCA*. 2007.

[13] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle. "A Predictive Model for Dynamic Microarchitectural Adaptivity Control". In: *MICRO*. 2010.

[14] E. Fayneh, M. Yuffe, E. Knoll, M. Zelikson, M. Abozaed, Y. Talker, Z. Shmuely, and S. A. Rahme. "4.1 14nm 6th-generation Core processor SoC with low power consumption and improved performance". In: *ISSCC*. 2016.

[15] W. Felter, K. Rajamani, T. Keller, and C. Rusu. "A performance-conserving approach for reducing peak power consumption in server systems". In: *ICS*. 2005.

[16] A. Filieri, H. Hoffmann, and M. Maggio. "Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees". In: *ICSE*. 2014.

[17] A. Goel, D. Steere, C. Pu, and J. Walpole. "SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit". In: *2nd USENIX Windows NT Symposium*. 1998.

[18] D. Goodman, G. Varisteas, and T. Harris. "Pandia: Comprehensive Contention-sensitive Thread Placement". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. 2017.

[19] T. Harris, M. Maas, and V. J. Marathe. "Callisto: Co-scheduling Parallel Runtime Systems". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. 2014.

[20] 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.

[21] 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.

[22] C. Imes, L. Bergstrom, and H. Hoffmann. "A Portable Interface for Runtime Energy Monitoring". In: *FSE*. 2016.

[23] C. Imes and H. Hoffmann. "Bard: A Unified Framework for Managing Soft Timing and Power Constraints". In: *SAMOS*. 2016.

[24] 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.

[25] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. "Portable Multicore Resource Management for Applications with Performance Constraints". In: *MCSoC*. 2016.

[26] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget". In: *MICRO*. 2006.

[27] C. Karamanolis, M. Karlsson, and X. Zhu. "Designing controllable computer systems". In: *HotOS*. 2005.

[28] C. Lefurgy, X. Wang, and M. Ware. “Power capping: a prelude to power shifting”. In: *Cluster Computing* 11.2 (2008).

[29] X. Li, R. Gupta, S. V. Adve, and Y. Zhou. “Cross-component Energy Management: Joint Adaptation of Processor and Memory”. In: *ACM TACO* 4.3 (2007).

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

[31] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. “Control-theoretic Dynamic Frequency and Voltage Scaling for Multimedia Workloads”. In: *CASES*. 2002.

[32] P. J. Lucas. *SWISH++*. `http://swishplusplus.sourceforge.net/`. 2014.

[33] M. Maggio, H. Hoffmann, M. Santambrogio, A. Agarwal, and A. Leva. “Power Optimization in Embedded Systems via Feedback Control of Resource Allocation”. In: *IEEE TCST* 21.1 (2013).

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

[35] Meghana R. “An overview of the 6th generation Intel Core processor (code-named Skylake)”. In: *Intel Developer Zone* (Mar. 2016). URL: `https://software.intel.com/en-us/articles/an-overview-of-the-6th-generation-intel-core-processor-code-named-skylake`.

[36] 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.

[37] S. Mittal. “A survey of techniques for improving energy efficiency in embedded computing systems”. In: *IJCAET* 6.4 (2014).

[38] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. “MineBench: A Benchmark Suite for Data Mining Workloads”. In: *IISWC*. 2006.

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

[40] 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.

[41] K. K. Rangan, G.-Y. Wei, and D. Brooks. “Thread motion: fine-grained power management for multi-core systems”. In: *ISCA*. 2009.

[42] S. Reda, R. Cochran, and A. K. Coskun. “Adaptive Power Capping for Servers with Multithreaded Workloads”. In: *IEEE Micro* 32.5 (2012).

[43] 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.

[44] M. H. Santriaji and H. Hoffmann. “GRAPE: Minimizing energy for GPU applications with performance requirements”. In: *MICRO*. 2016.

[45] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. “METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management”. In: *SIGMETRICS PER* 39.1 (2011).

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

[47] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker. “Scalable thread scheduling and global power management for heterogeneous many-core architectures”. In: *PACT*. 2010.

[48] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. “Formal online methods for voltage/frequency control in multiple clock domain microprocessors”. In: *ASPLOS*. 2004.

[49] F. F. Yao, A. J. Demers, and S. Shenker. “A Scheduling Model for Reduced CPU Energy”. In: *FOCS*. 1995.

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

[51] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. “ControlWare: a middleware architecture for feedback control of software performance”. In: *ICDCS*. 2002.

[52] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. “Survey of Energy-Cognizant Scheduling Techniques”. In: *IEEE Trans. Parallel Distrib. Syst.* 24.7 (2013).