Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques

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

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

Abstract

Power and thermal dissipation constrain multicore performance scaling. Modern processors are built such that they could sustain damaging levels of power dissipation, creating a need for systems that can implement processor *power caps*. A particular challenge is developing systems that can maximize performance within a power cap, and approaches have been proposed in both software and hardware. Software approaches are flexible, allowing multiple hardware resources to be coordinated for maximum performance, but software is slow, requiring a long time to converge to the power target. In contrast, hardware power capping quickly converges to the the power cap, but only manages voltage and frequency, limiting its potential performance.

In this work we propose PUPiL, a hybrid software/hardware power capping system. Unlike previous approaches, PUPiL combines hardware's fast reaction time with software's flexibility. We implement PUPiL on real Linux/x86 platform and compare it to Intel's commercial hardware power capping system for both single and multi-application workloads. We find PUPiL provides the same reaction time as Intel's hardware with significantly higher performance. On average, PUPiL outperforms hardware by from 1.18–2.4× depending on workload and power target. Thus, PUPiL provides a promising way to enforce power caps with greater performance than current state-of-the-art hardware-only approaches.

Categories and Subject Descriptors C.1.3 [*Other Architectural Styles*]: Adaptable architectures; I.2.8 [*Problem Solving, Control Methods, and Search*]: Heuristic methods; D.4.8 [*Performance*]: Measurements

General Terms Performance, Design, Experimentation

ASPLOS '16, April 02-06, 2016, Atlanta, GA, USA

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2872362.2872375

Keywords Adaptive Systems, Power Management, Decision-tree

1. Introduction

Modern processors are constrained by *dark silicon* – their abundance of transistors enables them to draw more power than they can safely sustain [11, 57]. For example, the Exynos 5 processor (in the Samsung Galaxy S4 phone) has a 5.5W peak power – nearly 2× its sustainable heat dissipation, limiting peak speed to less than 1 second [49]. At the other end of the spectrum, future exascale supercomputers have a predicted operating budget of 20 MW [2], making power management a central challenge of supercomputer operating systems [54].

These physical constraints create a need for *power control systems* which guarantee the processor operates within a strict *power cap*. Research power capping systems have been implemented in software [5, 6, 14, 31, 42–44, 62]. The need for power capping has become so great, however, that Intel processors now support power capping in hardware with their Running Average Power Limit (RAPL) interface [7].

Whether implemented in hardware or software, there are two essential properties for a power capping system. The first is **timeliness** – the speed with which a new cap can be enforced. The second is **efficiency** – the performance delivered under the cap. Without timeliness, critical operating bounds can be violated, damaging the hardware. Without efficiency, application performance suffers unnecessarily. It is, of course, trivial to implement a power cap while ignoring performance – simply turn the machine off.

In general, hardware approaches provide superior timeliness – hardware reacts much faster than software – while software approaches have superior efficiency – they find the highest performance set of resources to activate within the power cap. Hardware's timeliness is due to the relatively simple circuits that control key power indicators like processor voltage and frequency. Software's efficiency derives from its ability to consider the complex interactions between multiple resources, allowing it to solve the constrained optimization problem of scheduling the highest performance resource configuration which obeys the power cap.

This paper explores the tradeoffs between timeliness and efficiency in power capping approaches. Specifically, we advocate a *hybrid* approach that includes both software and hardware components, using each to address the challenge to which it is best suited. We instantiate this hybrid approach in PUPiL – for Performance Under Power Limits – a power capping system based on a novel *decision framework*. To ensure a power cap, PUPiL navigates nodes in a decision framework. Each node represents a choice about how to use a particular resource. For example, one node will select how many cores to use in a multicore. After making a decision, PUPiL measures power and performance and uses that feedback to drive the decision at the next node.

We implement PUPiL and test it on a Linux/x86 server with 20 different multithreaded benchmarks under 5 different power caps. We compare PUPiL to both RAPL (Intel's state-of-the-art hardware power capping system [7]) and two software-only approaches. We evaluate the timeliness and efficiency of all approaches for both single and multi-application workloads. Our results show:

- **Efficiency:** For single application workloads, a software-only approach can achieve higher performance than RAPL, but PUPiL achieves the highest performance. Specifically, PUPiL outperforms RAPL by 1.32–1.18× depending on the power cap. (Section 5.2.)
- **Timeliness:** RAPL's speed enforcing the power caps greatly exceeds software-only approaches by orders of magnitude. PUPiL is equivalent to RAPL. (Section 5.3.)
- **Multi-application Efficiency:** We test two types of multi-application workloads: 1) *cooperative* loads where each application requests a subset of available resources and 2) *oblivious* loads where each application requests all resources. For cooperative loads, PUPiL outperforms RAPL by 1.43–1.18× on average depending on the power cap. In the oblivious case, PUPiL outperforms RAPL by 2.56–2.43× depending on the power cap.
- **Energy Efficiency:** While PUPiL is designed to meet power caps, we find that by increasing performance, it also improves energy efficiency compared to RAPL, with 1.05-1.4× average improvements.

These results indicate that PUPiL's hybrid approach provides the timeliness of hardware with significantly greater efficiency. The performance gains are particularly high when enforcing power caps in the oblivious multi-application scenario. The large number of threads and resulting contention in the oblivious multi-application scenario creates a situation where the applications destructively interfere with each other. RAPL's only mechanism for power enforcement is processor voltage and frequency, which does nothing to limit contention. PUPiL, in contrast, manipulates DVFS as well as core allocation, socket usage, memory usage, and hyperthreading. This diversity allows PUPiL to throttle back multiple resources and reduce overall contention, resulting in large performance gains for the same power cap.

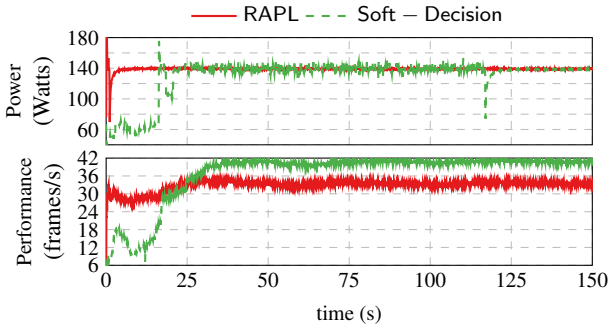

Figure 1. Tradeoff between timeliness and efficiency from hardware and software power capping, running x264.

This paper makes the following contributions:

- Develops a decision framework to maximize performance under a power cap.
- Evaluates this implementation on a real system in multiple usage scenarios.
- Identifies workload properties where Intel's RAPL power capping system fails to deliver best performance.
- Makes all scripts, code, and data collection tools from this evaluation available as open source, so others can test or extend these results[1].

The fundamental contribution of this paper is an empirical demonstration of the need for software and hardware to work together to maximize performance under power caps. The combined software/hardware approach proposed in this paper demonstrates it is possible to achieve significant performance gains over Intel's state-of-the-art, commercial hardware approach – especially for multi-application workloads.

2. Motivational Example

This example highlights the different tradeoffs in hardware and software power capping approaches and motivates the need for a hybrid design. We run the **x264** video encoder on an Intel Linux/x86 system. We compare the timeliness and efficiency of both Intel's RAPL hardware and a software approach that can adjust many settings (presented in Section 3).

Our test system is a dual-socket server with two Intel SandyBridge Xeon E5-2690 processors and 64GB of RAM. These processors support RAPL, but also have a number of configurable resources which affect power and performance tradeoffs, listed in Table 1. Each processor supports 15 frequency settings plus TurboBoost. Each is 8 cores, with hyperthreading, giving a total of 32 virtual cores across both sockets. These processors have a thermal design power (TDP) of 135 Watts, but experimentally we find it extremely rare for any workload to sustain that power consumption.

To illustrate the difference between hardware and software power capping, we set a 140 Watt power cap total for both sockets. RAPL must achieve this power consumption

[1] All source code, scripts, inputs, and patches are available at: https://github.com/PUPiL2015/PUPIL.git.

Table 1. Server resources.

Processor	Cores	Sockets	Speeds (GHz)	TurboBoost	HyperThreads	Memory Controllers	Socket TDP (W)	Configurations
Xeon E5-2690	8	2	1.2–2.9	yes	yes	2	135	1024

by driving each socket to 70 Watts (this is the optimal solution without thread migration, over which RAPL has no control). In contrast, the software approach configures a range of parameters: 1) how many sockets to use, 2) how many cores to use on each socket, 3) whether to use hyperthreads or not, 4) how many memory controllers to use, and 5) the frequency of each socket. For both the hardware and software approaches we measure power and performance (in frames encoded per second) as a function of time.

Fig. 1 illustrates the results, with power shown in the top chart and performance shown on the bottom. Each chart shows time on the x-axis. The hardware approach is represented by the solid line, and the dashed line represents the software approach. Clearly, both approaches meet the power cap – RAPL hits the cap quickly while the software approach operates below the cap for approximately 20 seconds, briefly exceeds it, and finally settles at 140 Watts.

The performance results, however, show that once the software approach converges, it delivers 20% more performance than RAPL. Specifically, after convergence, the software approach averages approximately 41 frames per second while RAPL averages approximately 33.5 frames per second. Software outperforms hardware because it recognizes that hyperthreads do not help this application on this system. Using hyperthreads results in greater power consumption and a small performance loss. The software approach recognizes that it should not make use of hyperthreads and instead it increases the speed of the cores it is using without hyperthreads. Of course, it takes software a long time to recognize and adjust.

These results demonstrate the need for a hybrid approach that enforces power caps with hardware's speed, but has software's flexibility to adapt resource usage to the particular application (or applications) running on the system.

3. Power Capping Methodologies

This section introduces the different power capping approaches we explore in this paper. It first discusses our software approach. It then describes RAPL, a state-of-the-art hardware power capping system. Finally, it introduces PUPiL, a hybrid of software and hardware approaches.

We assume that a computer system is *configurable*; *i.e.,* it has resources or other parameters whose usage can be tuned to navigate performance/power tradeoffs. For each approach, the goal is to configure these resources to meet a power cap in a timely and efficient manner. Timeliness means the cap is quickly enforced. Efficiency means the system delivers maximum performance under the cap.

All three power capping approaches (software, hardware, and PUPiL) operate based on *feedback*. These approaches *observe* their environment, *decide* on a response, and *act* to implement their decisions. This feedback loop is repeated continually, allowing the power capping system to react to application phase changes or other environmental fluctuations. We use this *observe-decide-act* framework as a basis for understanding the methodologies of the three different power capping approaches addressed in this paper.

3.1 Software Power Capping

This section discusses how the software system implements observation, decision, and action.

3.1.1 Observe

In the observation phase, the software collects power and performance feedback.

Power feedback can come from any number of power monitoring mechanisms. For example, external power meters such as a WattsUp device can be used. Other alternatives include on-board power monitoring devices, such as the INA231 [27], or on-chip power monitoring, which is available commercially from Intel [7] and through research prototypes [50].

Performance feedback can also come from a number of sources. High-level performance feedback can come directly from appropriately instrumented applications [21]. It could also come from any number of other sources, including hardware counters that measure floating point computation rate or simply instructions per second [52, 55]. While the methodologies in this paper will work with any metric, the authors personally advocate the use of high-level application-specific feedback, if available as such allows a power capping system to ensure efficiency in terms of real application progress.

One issue with feedback is that real systems are noisy. To meet the efficiency challenge, a power capping system should ensure that it is reacting to persistent phenomena and not some transient effect that momentarily disturbs performance. That is, the system should distinguish between a fundamental change in application workload and a temporary timing fluctuation (*e.g.,* due to a page fault). The power capper should adjust in the first case, but ignore the second case.

To address noise and ensure that the system acts on meaningful feedback, the software approach employs a deviation based filter to remove outliers. Specifically, the software approach measures performance over a window, filters any data that falls more than 3-standard deviations from the mean, and averages the rest. Assuming, X is the list of performance measurements collected, μ is the average of unfiltered X, σ is the standard deviation of unfiltered X, then $X_{feedback}$ is the performance feedback used by the system to make deci-

Algorithm 1 Walking the decision framework.

Require: Set of ordered resources R
Require: Power cap P
  Put system in minimal resource configuration
  $U \leftarrow R$ ▷ the set of untested resources
  while $U \neq \emptyset$ **do** ▷ While untested resources
    $\langle perf_{old}, pow_{old} \rangle \leftarrow$ GetFeedback()
    $r \leftarrow$ RemoveNext(U) ▷ next resource in order
    set r to highest setting
    wait $r.d$ time units ▷ Account for resource delay
    $\langle perf_{cur}, pow_{cur} \rangle \leftarrow$ GetFeedback()
    if $perf_{cur} < perf_{old}$ **then**
      return r to lowest setting
    else
      if $pow_{cur} > P$ **then**
        $s \leftarrow$ BinarySearchResourceSettings(r)
        set r to s
        ▷ This may return the resource to its lowest setting.

sions:

$$\mu = \frac{\sum_i X_i}{N} \quad (1)$$

$$\sigma = \sqrt{\frac{\sum_i (X^i - \mu^2)}{N}} \quad (2)$$

$$X_{feedback} = \frac{\sum_{j \in A} X_j}{size(A)} \quad (3)$$

$$A = \{j \mid |X_j - \mu| < 3\sigma\} \quad (4)$$

3.1.2 Decide

In the decide phase, the software selects a resource configuration. One way to select the best configuration would be to simply walk through all configurations until we find the highest performance configuration that respects the power cap. This approach has the twin drawbacks that it fails to meet the timeliness challenge and it may fail to respect the power cap. In general, the number of possible resource configurations will grow exponentially as we add more resources. Thus exhaustive search is simply not feasible.

Any software approach must find a more intelligent way to explore the configuration space. In this paper, we propose a novel *decision framework*. To begin, the system orders the available resources (the ordering process is described below). It then starts in the lowest resource configuration. Proceeding through resources in order, the approach puts the next resource into its highest setting. Feedback is measured in this new configuration. The software compares the performance feedback of the current configuration to that of last configuration to decide whether 1) performance has improved by using this new resource and 2) the resource usage respects the power cap. Algorithm 1 specifies the decision making process.

Algorithm 1 requires an ordered set of resources. The order is determined by Order() (detailed in Algorithm 2). The algorithm first sets the system to the smallest resource configuration. It then puts the resources into a set of untested resources. While this ordered set of untested resources is non-empty, the algorithm measures power and performance (using the helper function GetFeedback()). It then takes the next resource in order and sets it to its highest configuration setting (using the Set() helper function), waits a resource-specific amount of time, and then measures the feedback again. If this resource provided higher performance, then the algorithm fine tunes the resource setting, otherwise it returns to the lowest setting for this resource. The fine tuning process involves performing a binary search on resource settings to find the highest performance setting that is under the power cap (the BinarySearchResourceSettings() helper function).

We use binary search on a resource-by-resource basis to avoid exhaustive search's overhead. This is an engineering tradeoff. Component-wise binary search is fast, but can get stuck in local extrema and miss the global optimal solution. In exchange, however, it scales well even as the number of configurable resources grows. In practice, this approach works well because resources tend to have a single peak. For example, not all applications can use all cores, but there tends to be a single best core count with no local extrema.

There are four helper functions for this approach. Three are straightforward and their detailed descriptions are omitted for space. We provide a brief overview here. The GetFeedback() function simply measures and returns power and performance data. The $Set()$ function is used to configure the resource. The BinarySearchResourceSettings() function simply does a binary search on the available configurations for a resource. Its goal is to find the highest performance setting that respects the power cap. The ordering function is the fourth helper and it is described below.

The ordering function is essential to Algorithm 1. The software approach establishes the ordering based on the potential impact of each resource. Higher impact resources have precedence over lower impact resources. Algorithm 2 shows the algorithm used for establishing this order. The intuition is to allocate power first to higher impact resources so that we can tune the performance from coarse-grained knobs to fine-grained knobs. We evaluate impact of a resource by the performance improvement that it delivers when activated individually. The one exception is DVFS, which is used at the end to fine-tune power within the cap. To determine impact, we calibrate the system using a well-understood, embarrassingly parallel application. Based on our results, the ordering is insensitive to different applications; *i.e.,* the decision tree finds a near-optimal configuration using the same calibrated ordering for all applications. The detailed process for establishing the order is shown in Algorithm 2.

3.1.3 Act

In the act phase, the software implements the resource allocation proposed by the decision phase. For example, if the decision phase decides to test a resource, the act phase is responsible for actually assigning that resource to the active applications. To implement the act phase, the software requires two pieces of external information. The first is a tim-

Algorithm 2 Ordering Resources in Calibration.

Require: Set of resources R excluding DVFS
Require: a calibration benchmark without inter-thread communication
Put system in minimal resource configuration
$U \leftarrow R$ ▷ the set of disordered resources
while $U \neq \emptyset$ **do** ▷ While disordered resources
$r \leftarrow \mathsf{RemoveNext}(U)$ ▷ next resource in random order
set r to highest setting
wait $r.d$ time units ▷ Account for resource delay
$perf_r \leftarrow \mathsf{GetFeedback}()$
return r to lowest setting
add r to O
Sort r in O by $perf_r$
Add DVFS to the last in O **return** O ▷ The set of ordered resources

ing information about how long to expect from when the resource is allocated to when its effects can be observed. This information is required so that the software does not take a new observation before the resources have actually had an effect. The second piece of information is a function that implements the resource allocation. As most resources are allocated in system-specific ways, this function is necessary to maintain the generality of the approach and let it work on multiple systems.

Given this information, the action phase simply consists of setting the resource configuration to that specified by the decision phase and then putting the decision framework to sleep for the time it will take to see the resource effects. To increase efficiency, the software keeps track of the previous resource allocation and only changes those resource settings which changed since the last decision.

3.2 Hardware Power Capping

We briefly outline the approach taken by Intel's RAPL system [7], in terms of observation, decision, and action. RAPL receives a power cap and a time interval through a machine specific register (MSR). RAPL observes various low-level hardware events and estimates power consumption from those event counts. RAPL determines an energy budget that would meet the desired power cap during the specified time interval. For example, if the time interval is 0.5 seconds and the power cap is 100 Watts, the energy budget is 50 Joules.

RAPL sub-divides the user-specified time interval into a set of smaller intervals. For each of these fine-grained intervals, RAPL calculates the remaining energy budget for the remaining time in the user-specified interval and decides the best possible processor speed and voltage. Given this decision, RAPL sets DVFS to the decided state and waits for the next fine-grained interval. More detail on RAPL operation is available in the literature [7].

It is instructive at this point to compare the hardware and software approaches. Software is clearly flexible, the approach in Algorithm 1 will work with any set of available resources – the only requirement is that we must be able to establish an order on these resources. The drawback of software is that configuring the system requires executing Algorithm 1, which can be costly (as shown in Fig. 1). In

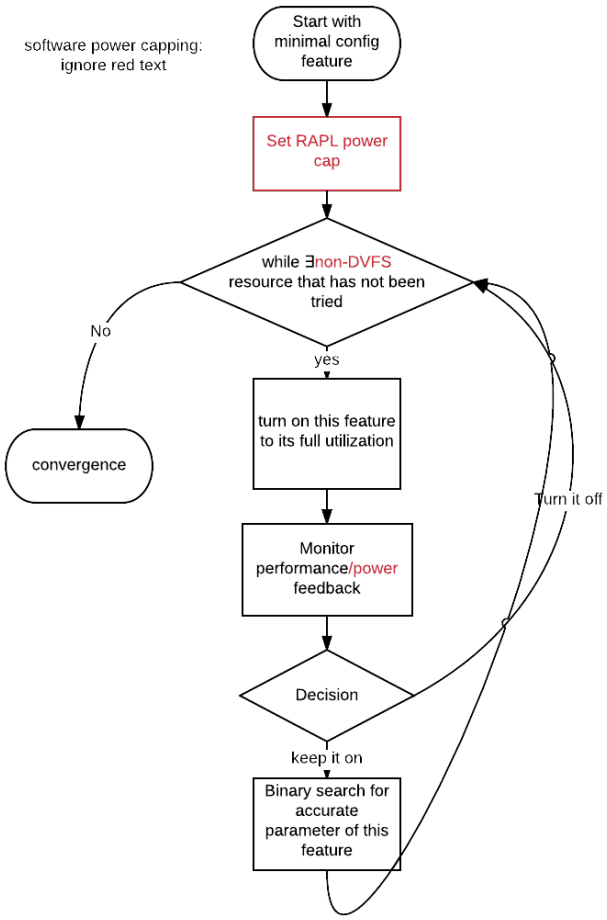

Figure 2. PUPiL's approach to hybrid hardware/software power capping.

contrast, RAPL observes only power feedback (not performance), makes decisions by solving a linear equation, and acts by only tuning voltage and frequency only. All three steps can be done within milliseconds and this ensures the timeliness of hardware approach. However, because RAPL lacks performance feedback and considers only DVFS, this hardware approach cannot deliver the highest performance for many applications.

3.3 PUPiL's Hybrid Power Capping

Our goal is to obtain the efficiency of the software approach and the timeliness of hardware approach. Thus, we propose PUPiL, a hybrid power capping system that incorporates software and hardware to achieve the benefits of both.

3.3.1 Timeliness

We need the system to respect the power cap as soon as the cap is set. To achieve this timeliness, hardware power capping approach has to be in charge of capping the power instead of the much slower control loop of software approach. Thus, we set the power cap in hardware first, before explor-

ing other resources. Meanwhile, to avoid interference with the hardware approach, we remove processor speed and voltage from the set of resources controlled by software. Leaving hardware in charge of voltage and speed ensures timeliness and reduces the configuration space software much search.

Fig. 2 illustrates PUPiL's hybrid decision framework. The major difference between the software-only approach and Fig. 2's is that the hybrid approach explicitly sets RAPL before exploring the configuration space determined by the non-DVFS resources. To achieve this in practice, we modify Algorithm 1 so that it first sets the RAPL power cap.

3.3.2 Efficiency

We need to find the optimal configuration for the running application. This requires two modifications to the decision algorithm shown in Algorithm 1.

First, the power cap is now met by hardware so PUPiL need only manage performance. Thus, the hybrid approach excludes all the power condition checks in Algorithm 1 – PUPiL assumes RAPL ensures the power cap.

Second, power distribution among different chips in a multi-socket environment has to be reconsidered. Hardware power capping caps power on a per-socket manner. However, when we consider thread migration as a tunable parameter, the optimal configuration for an application or workload is often asymmetric, so it is necessary to distribute power accordingly instead of using a default even distribution. PUPiL, therefore, uses a core-number based power distribution across different chips. More specifically, PUPiL distributes the dynamic power (power cap minus static power) proportional to the core number being used by each chip. PUPiL achieves this by setting corresponding hardware power cap to each chip. Thus, whenever there is core number configuration adjustment, power distribution adjusts with it.

4. Experimental Setup

This section describes benchmarks, system, metrics, and points of comparison we use to evaluate PUPiL.

4.1 Benchmarks

We use 20 benchmark applications from three different suites including PARSEC (`x264`, `swaptions`, `vips`, `fluidanimate`, `blackscholes`, `bodytrack`) [3], Minebench (`ScalParC`, `kmeans`, `HOP`, `PLSA`, `svmfe`, `btree`, `kmeans_fuzzy`) [40], and Rodinia (`cfd`, `nn`, `lud`, `particlefilter`)[4]. We also use a partial differential equation solver (`jacobi`) and the `swish++` search webserver [24] and `dijkstra` [28]. These benchmarks test a range of important modern applications, both compute-intensive and memory-intensive. All applications run with up to 32 threads (the maximum supported in hardware on our test machine). In addition, all workloads are long running, taking at least 10 seconds to complete. This duration gives us plenty of time to take measurements of system performance and power.

Table 2. System configurations.

Configuration	Settings	Max Speedup	Max Powerup
cores per socket	8	7.9	2.1
sockets	2	2.0	1.7
hyperthreading	2	1.9	1.2
mem controllers	2	1.8	1.1
clock speeds	16	3.2	3.4

4.2 Platform

We use a dual-socket Intel/Linux system with a SuperMICRO X9DRL-iF motherboard and two Xeon E5-2690 processors (see Table 1). This motherboard supports setting RAPL's power capping feature. The system runs Linux 3.2.0. We use the `msr` module, to access the model specific registers that implement RAPL. We use the `cpufrequtils` package to set the processor's clock speed. These processors have eight cores, fifteen DVFS settings (from 1.2 – 2.9 GHz), hyper-threading, and TurboBoost. In addition, each chip has its own memory controller, and we use the `numactl` library to manage memory controller use. In total, the system supports 1024 user-accessible configurations, each with its own power/performance tradeoffs[2]. The thermal design power for these processors is 135 Watts.

Given those specifications, the following resources are configurable: the clock speed of each socket, core use per socket, hyperthreading, the number of sockets in use, and the number of memory controllers in use. Manipulating thread affinities allows us to change the cores per socket, the active sockets and the use of hyperthreading.

As described in Section 3, implementing the software decision system requires ordering the set of resources under consideration. Table 2 lists these resources in the order established by Algorithm 2. For each resource in the table, it lists the speedup and power up (increase in power, analogous to speedup) measured during the ordering process.

4.3 Evaluation Metrics

Our goal is to evaluate the timeliness and efficiency of various power capping approaches. To compare approaches, we must quantify these properties. We evaluate timeliness by measuring settling time. We evaluate efficiency by measuring the performance achieved by a workload under a power cap.

4.3.1 Timeliness

Settling time is a standard metric for a control system [18]. Given a power cap, it may take some amount of time for the controller to stabilize the system at that power. We call the period after which the system stabilizes the *steady state* and we denote the time at which the system enters steady state as

[2] 16 cores, 2 hyperthreads, 2 memory controllers, and 16 speed settings (15 DVFS settings plus TurboBoost)

t_{ss}. If the controller begins work at time t_0, then the settling time is simply:

$$settle = t_{ss} - t_0 \quad (5)$$

4.3.2 Efficiency

Efficiency is the performance delivered under a power cap. We evaluate efficiency using *weighted speedup*. This is a standard metric for multi-application workloads that weights the performance each application achieves in a multi-application scenario by the performance it would achieve in isolation. This metric has been demonstrated to be both consistent and fair [13].

4.4 Points of Comparison

To evaluate PUPiL, we compare it to several other techniques:

- **RAPL:** The primary approach with which we compare.
- **Soft-DVFS:** This is a software approach that sets the DVFS settings using the `cpufrequtils` package. Our implementation is modeled on a prior approach proposing a software-based DVFS control system [31].
- **Soft-Modeling:** This is a software approach that models the power for different configurations in an offline manner. That is, it uses multiple regression to estimate the power and performance of an application as a function of assigned resources (in this case, clockspeed, memory controllers, sockets, cores per socket and hyperthreads). This approach is an extreme case of a predictive model that needs no feedback information at runtime.
- **Soft-Decision:** This is the software-only decision framework described in Section 3.1.
- **Optimal:** This is determined by running each application in every possible system configuration and measuring its performance. The optimal configuration achieves the best speed for a given power cap.

5. Experimental Evaluation

This section evaluates PUPiL's timeliness and efficiency. To enable others to perform similar evaluations, we have made the software and scripts used to perform this evaluation available online. We begin by evaluating single application workloads and then address multi-application workloads.

5.1 Single Application

To evaluate power control methods for single application workloads, we launch each application under a power cap and measure both its performance and settling time. We evaluate 5 different processor power caps: 60, 100, 140, 180, and 220 Watts. When setting the caps for both RAPL and Soft-DVFS, we split the power budget between both sockets evenly as this is the optimal allocation when no other resource is considered. Soft-Decision and PUPiL are free to divide the power cap among the sockets as they see fit when they migrate threads.

Table 3. Comparison of Harmonic Mean Performance.

Power Cap	RAPL	Soft-DVFS	Soft-Modeling	Soft-Decision	PUPiL
60W	.54	-	-	.70	.71
100W	.68	.66	.66	.80	.85
140W	.74	.71	.65	.87	.89
180W	.78	.74	.76	.88	.92
220W	.79	.75	.85	.91	.94

There are no Soft-DVFS or Soft-Modeling data for the 60W cap. For Soft-DVFS, even the lowest p-state exceeds the 60W power cap when using all cores and hyperthreads. For Soft-Modeling, the errors for this cap are extremely large; approximately 70% of the data points for this technique exceed the power cap. This demonstrates a disadvantage of a system that uses no online feedback to correct its models. It has no ability to recover when the models have high error. PUPiL, in contrast, uses a very simple model but the feedback constantly corrects.

5.2 Performance

Fig. 3 shows the performance delivered under each cap for each application. This figure contains one chart for each power cap. The x-axis shows the benchmark, the y-axis shows performance normalized to optimal (1 is the best possible performance). The charts show one bar for each of RAPL, Soft-DVFS, Soft-Modeling, Soft-Decision, and PUPiL.

While results vary per application and power cap, the general trends show that Soft-Decision provides higher performance than RAPL with Soft-DVFS and Soft-Modeling comparable to RAPL. Furthermore, the hybrid approach generally provides the highest performance. The harmonic mean performance for each power cap and power controller is summarized in Table 3. This table shows PUPiL consistently outperforms RAPL and Soft-DVFS across all power caps by at least 18% (at the 180W cap) and at most 32% (at the 60W cap).

Soft-Modeling takes the advantage historical power data and configures the machine based on the predicted power. It, however, has no guarantee of respecting power cap because it has no feedback mechanism. For some applications and power caps (*e.g.,* HOP, swish++ at 100W), it outperforms all other approaches by exceeding the power cap. The average performance of Soft-Modeling is still not good compared to Soft-Decision and PUPiL, despite the fact that it sometimes exceeds the caps. Furthermore, Soft-Decision is very close to PUPiL. These results confirm that multi-resource approaches out perform systems that only manipulate DVFS, whether in software or hardware.

Clearly RAPL performs well on some applications (*e.g.,* `btree` and `svmfe`) and poorly on others (*e.g.,* `dijkstra` and `kmeans`). Fig. 5 shows the computation (in instructions per second) and memory bandwidth (in GB/s) for each benchmark. Blue dots represent applications for which RAPL does well (is within 10% of optimal for the 140 W cap) and red dots show applications for which RAPL achieves poor effi-

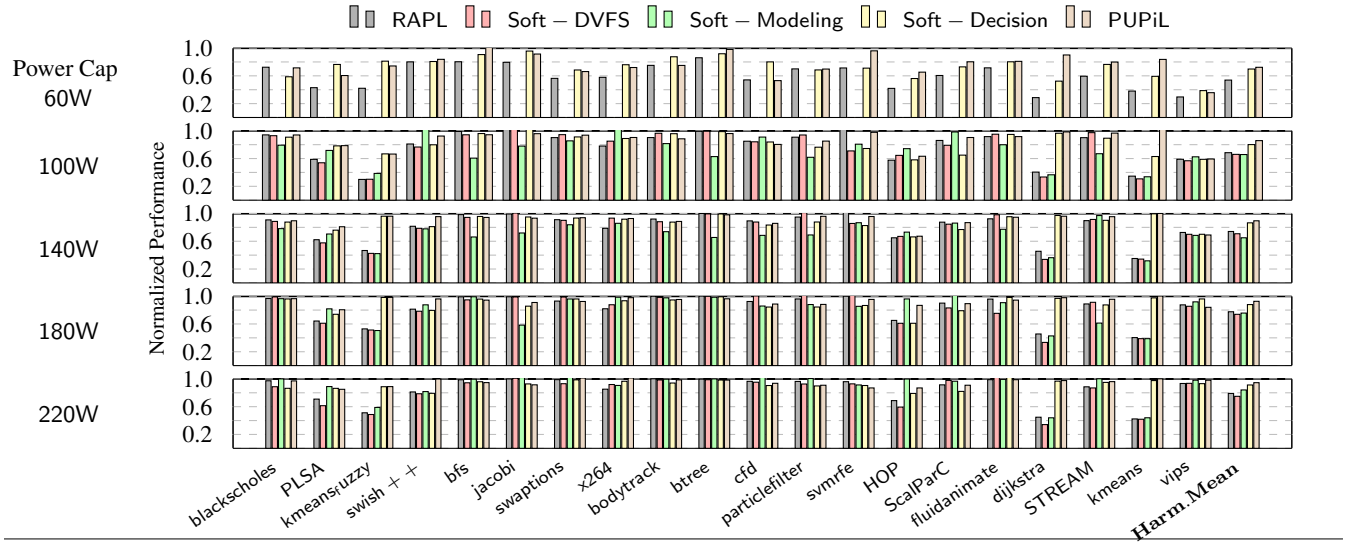

Figure 3. Performance of several power control techniques normalized to optimal.

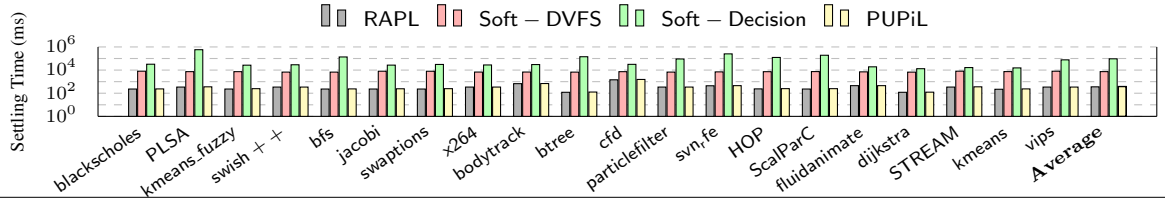

Figure 4. Settling times for several power control techniques.

ciency (greater than 10% from optimal). Clearly simple notions like memory-bound or compute bound are not good predictors of RAPL efficiency. For example, RAPL performs poorly on `STREAM` (which has the highest memory bandwidth), yet does well with `jacobi` (which has the second highest memory bandwidth). RAPL generally performs well for applications that have ample parallelism and scale well to use all 32 virtual cores. RAPL generally performs poorly on applications with scaling issues or limited parallelism. For such applications, it is better to restrict the resources they are using and increase the speed of this small subset.

For example, `kmeans` scales well with more cores on a socket. When `kmeans` is allocated cores on both sockets, however, inter-socket communication becomes a bottleneck, so `kmeans` continues to issue instructions and burn power but without increasing speed. RAPL and Soft-DVFS must reduce clock speed to meet the power cap. In contrast, both Soft-Decision and PUPiL recognize that the second socket decreased performance, and they restrict `kmeans` to a single socket but increase its speed, resulting in higher performance.

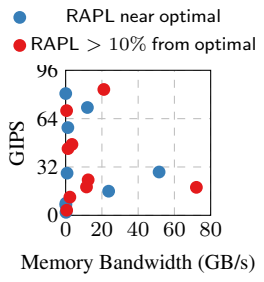

Figure 5. Benchmark characteristics.

5.3 Settling Time

For each application and power cap we measure settling time. Soft-Modeling is omitted as there is no settling time for this offline approach. Fig. 4 shows the settling times for all approaches and applications under the 140 Watt cap. Results for other caps are similar (only 1-2% different) and are omitted for space. Each application is shown on the x-axis and settling time (measured in milliseconds) is shown on the y-axis (in a logarithmic scale).

The data in Fig. 4 demonstrates the tremendous advantages in timeliness that RAPL has over Soft-Decision. On average, across all benchmarks, RAPL's settling time is 356 ms. In contrast, Software-Decision averages 95,000 ms, a difference of approximately 260 $\times$ and soft-DVFS averages 7,300ms, a difference of approximately 13 $\times$. These results demonstrate the claims of timeliness made in the introduction to the paper. RAPL has significant timeliness advantages over software approaches. PUPiL, however, is able to maintain RAPL's timeliness advantages, averaging 365 ms. The small increase in overhead is due to the fact that the power cap is now set through PUPiL's software interface rather than directly setting the register in hardware.

Table 4. Multi-application Workloads.

Name	Benchmarks
mix1	jacobi, swaptions, bfs, particlefilter
mix2	cfd, bfs, fluidanimate, jacobi
mix3	blackscholes, cfd, jacobi, fluidanimate
mix4	particlefilter, blackscholes, swaptions, btree
mix5	x264, dijkstra, vips, HOP
mix6	STREAM, fuzzy-kmeans, HOP, dijkstra
mix7	STREAM, kmeans, vips, HOP
mix8	kmeans, dijkstra, x264, STREAM
mix9	jacobi, swaptions, fussy-kmeans, vips
mix10	cfd, bfs, x264, HOP
mix11	jacobi, blackscholes, dijkstra, fuzzy-kmeans
mix12	btree, particlefilter, kmeans, STREAM

These results demonstrate the main claims in the introduction. Specifically, RAPL's hardware approach addresses the timeliness challenge. The software approach achieves efficiency gains compared to hardware. The mean performance advantage is at least 18%, while for specific applications (*e.g.,* `kmeans`, `dijkstra`) the gains can be over $2\times$. Finally, PUPiL's hybrid approach meets both the timeliness and efficiency challenges, combining hardware's low settling time with software's high performance.

5.4 Multi-Application Workloads

We evaluate RAPL and PUPiL on multi-application workloads. We begin by dividing our benchmarks into two sets: ones for which RAPL delivers near-optimal performance (blue dots from Fig. 5), and ones for which RAPL is more than 10% from optimal (red dots in Fig. 5). We create multi-application workloads by randomly selecting applications from the two sets. Specifically we create 12 separate mixes, each consisting of four applications. For the first four mixes (1–4), all applications are drawn from the set for which RAPL is near optimal. The mixes 5–8 are all taken from applications for which RAPL performs poorly. The applications in mixes 9–12 include two applications from each set. Table 4 summarizes the workloads: each is given a name – `mixN` – and we list the applications used in that workload. We evaluate the multi-application workload by launching all applications at the same time. We use the weighted speedup for efficiency metrics as described in Section 4.3.2.

We evaluate two separate multi-application scenarios: *cooperative* and *oblivious*. In the cooperative scenario, we assume all applications know that they are running with other applications; each is launched with only 8 threads, so that the total number of active threads is equal to the number of virtual cores. In the oblivious scenario, we assume that each application is launched without regard to the other applications in the system and each requests 32 threads, for a total of 128 alive in the system. We compare the performance achieved by RAPL and PUPiL in these two scenarios.

5.4.1 Cooperative Performance

The performance for the cooperative multi-application scenario is shown in the left column of Fig. 6. There is a chart for each power cap. The y-axes show the ratio of PUPiL to RAPL weighted speedup (higher means PUPiL outperforms RAPL) for each application mix (shown on the x-axes).

Table 5. Ratio of PUPiL to RAPL Performance.

Power Cap	Cooperative	Oblivious
60W	1.43	2.53
100W	1.21	2.56
140W	1.18	2.44
180W	1.18	2.46
220W	1.21	2.43

The performance comparison for the cooperative scenario reveals similar trends to the single-application scenarios. There are several mixes for which PUPiL and RAPL achieve similar performance and others where PUPiL far outperforms RAPL. Table 5 shows the ratios of PUPiL to RAPL performance across all mixes for each power cap. In the cooperative scenario, PUPiL outperforms RAPL by at least 18% across all power budgets.

Single-application performance is not necessarily a good indicator of multi-application performance. For each power cap there are examples where PUPiL far outperforms RAPL. For example, across all power caps PUPiL achieves much higher performance for mix2. This happens despite the fact that all applications in mix2 are drawn from the set for which RAPL provides good individual performance. This result shows that multi-application workloads can have complicated behavior and it justifies the need for an adaptive approach, like PUPiL, that can accommodate the unexpected.

5.4.2 Oblivious Performance

Fig. 6's left column shows the performance for the oblivious multiapp scenario. Recall that in the oblivious scenario, each application requests 32 threads. The performance results show that PUPiL provides significantly better performance than RAPL in the oblivious multi-application case. The summary results across all performance caps are shown in Table 5, which indicates that PUPiL achieves at least $2.4\times$ better aggregate performance than RAPL. Furthermore, this advantage can jump up to as much as $6\times$ for some application mixes.

These results demonstrate that in a system that reflects the oblivious multi-application workload – where every application is trying to claim as many resources as possible – RAPL by itself is simply not sufficient to provide high performance under the power cap. Instead, the flexibility of a system like PUPiL is needed to carefully manage resource usage and deliver high performance. The reason for PUPiL's higher performance is that these oblivious workloads typically bottleneck on some non-computational resource. This bottleneck is usually either intersocket communication bandwidth or memory bandwidth. This bottlenecking in the multi-application scenario is similar to what we have seen in the single application case, but now the consequences are more dire. We explore the reasons for this more in the next section.

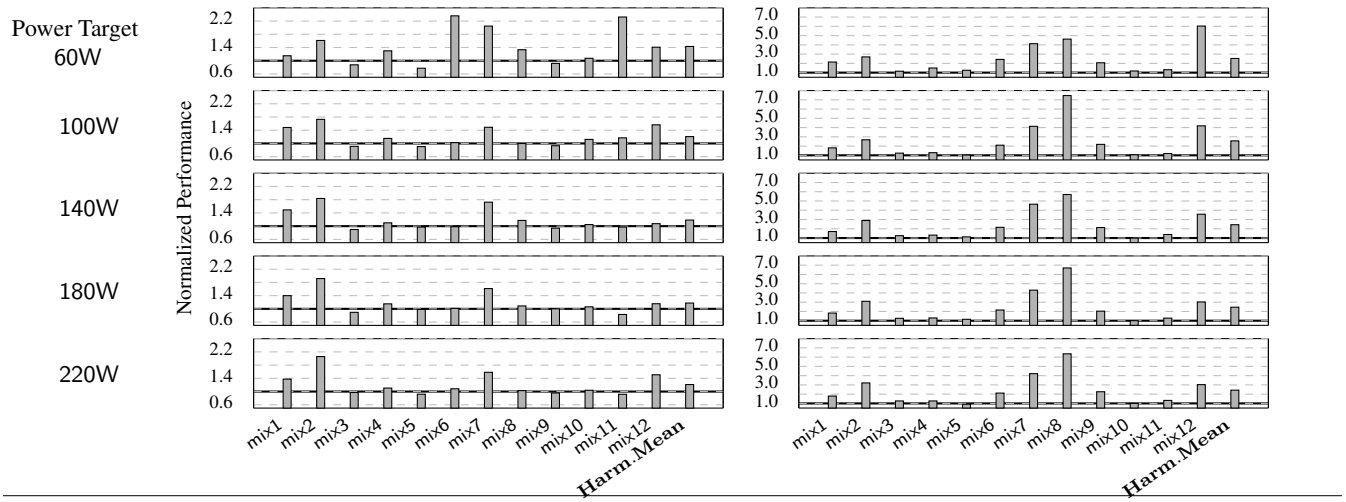

Figure 6. Ratio of PUPiL to RAPL performance in cooperative (left) and oblivious (right) multiapp scenarios.

Table 6. PUPiL and RAPL Multiapp Performance.

Workload	Spin Cycles (%)		Memory Bandwidth (GB/s)	
	RAPL	PUPiL	RAPL	PUPiL
mix7	15	0.23	14.6	23.8
mix8	54	.48	17.5	30.3
mix12	33	.40	14.3	27.0

5.4.3 Detailed Multiapp Data

This section presents some low-level metrics collected to explain the performance difference between PUPiL and RAPL in the oblivious multiapp case. To look for major differences between RAPL and PUPiL we use Intel's VTune tool to collect low-level metrics for the application mixes under both RAPL and PUPiL control.

VTune collects a tremendous amount of data on applications, but when looking at the metrics, two things stood out: *spin cycles* and *memory bandwidth*. This data is shown in Table 6 for the three mixes where PUPiL outperforms RAPL by the greatest amount. For each mix, the table shows the percentage of time spent executing *spin cycles*, cycles for which the processor is retiring instructions, but no forward progress is being made (*e.g.,* test-and-set instructions which fail the test). The table also shows the achieved memory bandwidth in MB/s for these three mixes.

Table 6 shows that under RAPL control these mixes spend significantly larger portions of their time spinning and achieve a significantly smaller memory bandwidth. We believe the problem is that one of the applications in these mixes uses polling synchronization during a fairly long serial portion of operation. The other applications appear to be largely memory limited and are either embarrassingly parallel (no or limited synchronization) or use condition variables to synchronize. Therefore, these other applications need memory bandwidth and yield the CPU when they cannot make progress. The one application that does polling synchronization, however, ruins the behavior of the entire system, as when it gets the CPU it holds it for its entire scheduling quantum while making minimal forward progress. This behavior limits the ability of the other applications to make progress as well. When the mix is scheduled on fewer cores, however, its overall performance increases dramatically. In this case, the polling benchmark (1) has much less contention, (2) finishes its work faster, and (3) yields the cores to other applications more often, boosting the overall performance.

5.5 Energy Efficiency

We compare RAPL's and PUPiL's energy efficiency. We report performance divided by power, which shows how much work can be done per joule. Single application workloads results are shown in Fig. 7. As before, we normalize efficiency of all approaches to the optimal. Soft-decision and PUPiL produce 1.15-1.3$\times$ energy efficiency compared to RAPL or Soft-DVFS. Fig. 8 shows the multi-application workload results. PUPiL has a 5–40% improvement of energy efficiency compared to RAPL across different power caps. These results show PUPiL produces good energy efficiency even though saving energy is not PUPiL's primary purpose.

5.6 Sensitivity and Overhead Analysis

Throughout this section we investigate several factors which affect the results. Our results examine sensitivity to various power caps. Performance under very low power caps is difficult for any power management system. In addition, PUPiL provides consistent performance improvements in both single and multiapp scenarios. Further, the use of diverse workloads demonstrates that some applications achieve high performance with RAPL alone, while others need the greater flexibility of PUPiL's hybrid approach.

In a feedback based system, overhead can take two forms: 1) the number of measurements that need to be taken before the system converges and 2) the impact on the converged system. Our results account for both forms of overhead. All

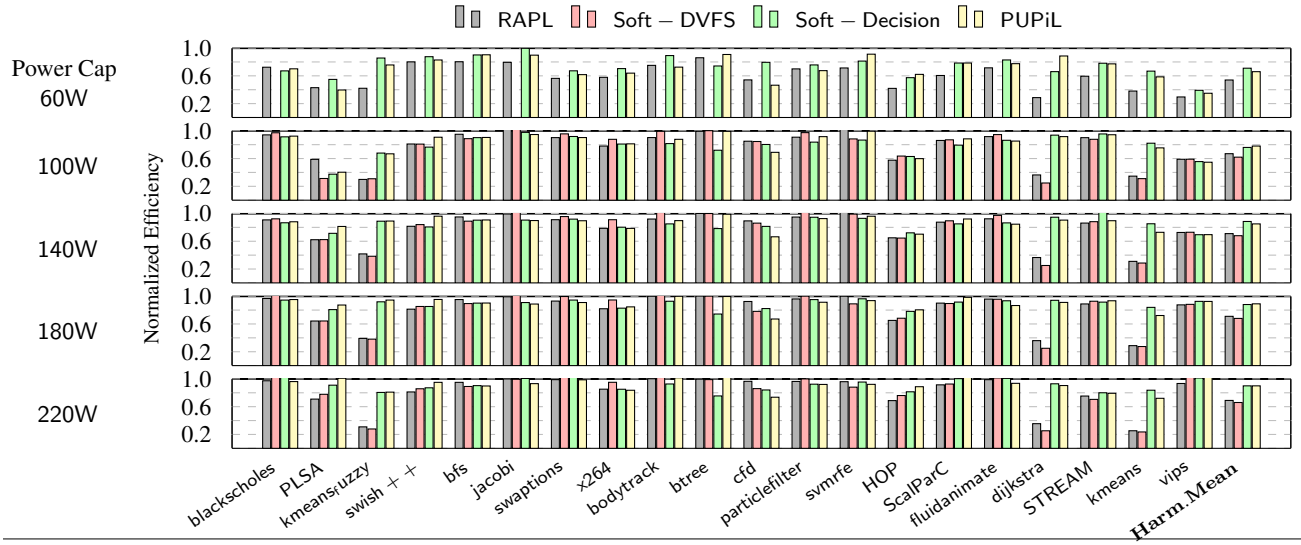

Figure 7. Energy efficiency of several power control techniques normalized to optimal.

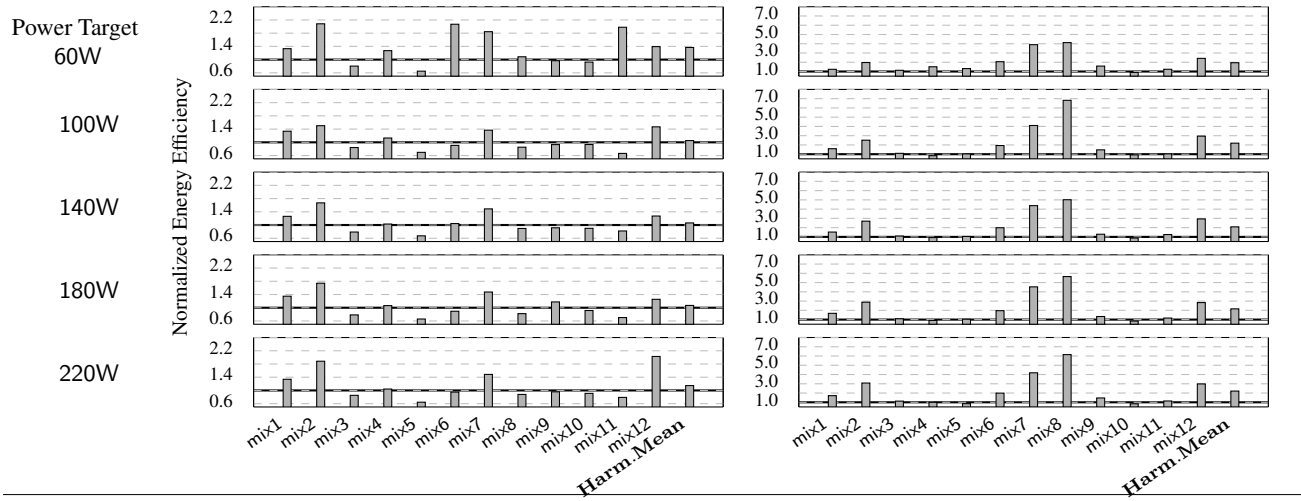

Figure 8. Ratio of PUPiL to RAPL energy efficiency in cooperative (left) and oblivious (right) multiapp scenarios.

reported results include the power and performance impact of the power capping systems themselves. The first type of overhead is measured directly in terms of settling times shown in Fig. 4. Both software approaches have very high, likely unusably high, overhead by this metric. The second type of overhead is accounted for by the comparison to optimal in Fig. 3. This figure shows that the performance impact of the PUPiL runtime system is acceptable in that PUPiL produces the closest to optimal performance.

6. Related Work

As power and energy become first order concerns of computing systems, a number of approaches have been proposed for managing these critical issues. Some approaches focus on minimizing energy, which can reduce costs in data centers and servers [25, 35, 47, 58, 63] or increase battery life in mobile and embedded platforms [15, 20, 26, 30, 39, 46, 56, 64]. These techniques provide performance guarantees (*e.g.,* for meeting quality-of-service or real-time requirements) and minimize power consumption or energy, but they do not provide power guarantees and cannot implement power caps.

To help facilitate energy management, several OS projects have added operating system support for monitoring and allocated energy. The Quanto project facilitates tracking energy usage in networked embedded devices [16]. The Cinder OS allows energy usage to be tracked and allocated across multiple applications in a system [45]. The Koala project also allows energy to be tracked and allocated while supporting several different policies for optimizing energy and performance [51]. Similarly, *power containers* support fine-grain tailoring of heterogeneous resources to varying workloads [48]. LEO is a hierarchical Bayesian learning

framework that produces extremely accurate estimates of an application's performance and power consumption [38]. The Coop-I/O project allows applications to coordinate with the operating system to schedule I/O operations in the most energy efficient manner possible [61]. The GRACE OS meets performance requirements for media while minimizing energy [56, 64]. None of these projects, however, explicitly support maximizing performance under a power constraint, which is the subject of this paper. JouleGuard provides energy guarantees (but not power) by coordinating application behavior with system resource usage [20].

While energy reduction can decrease costs and increase battery life, it is a separate concern from meeting power limits. Operating within power limits has become essential as multicore scalability is increasingly limited by power and thermal management [11, 57]. The physical realities of power dissipation in modern processors have led to hardware designs characterized by *dark silicon*. That is, modern processors cannot physically power all transistors at their maximum speed without damage. Thus, some of those transistors are kept dark (meaning they are not powered at all) or dim (meaning they are powered at less than full speed) [53].

These physical realities create a need to limit processor power dissipation. This concern is important enough that Intel's SandyBridge and later processors support power management in hardware [7]. A number software systems have also been proposed to perform power control or *capping*.

Cluster level solutions which guarantee power consumption include those proposed by Wang et al. [59] and Raghavendra et al. [42]. These approaches require some node-level power capper and node-level systems have been developed to manage different individual components including DVFS for a processor (the Soft-DVFS approach in our evaluation) [31], per-core DVFS in a multicore [29], processor idle-time [17, 65], and DRAM [10].

Several researchers have noted that coordinating multiple components provides greater performance under a power cap than management of a single component in isolation [1, 19, 22, 34, 35, 41]. Thus, approaches have been proposed which provide power guarantees while increasing performance through coordinated management of multiple components, including processor and DRAM [5, 8, 9, 14, 32, 46], processors speed and core allocation [6, 44], combining DVFS and scheduling [43, 62], memory and disk speed [33] and combining DVFS and process placement [36]. The VirtualPower project coordinates power management, virtual machine placement, and server consolidation to meet power constraints in a virtualized data center [41]. Despite differences in mechanisms, these techniques all solve a common problem: select the highest performance set of resources that respect a given power limit. All of these projects found higher performance is available through the coordination of multiple resources. With these results, it is not surprising that a hardware solution alone would not achieve high efficiency for some applications.

We take the position that power management should not solely be the domain of hardware, but must be supported by both hardware and software coordinated through the operating system.. The different resources required by different application workloads are simply too complicated for hardware to handle alone [12]. Hardware should be used to quickly enforce power limits, as hardware can simply act faster than software. Software techniques, however, should be used to determine the set of resources to activate that achieve the best performance under the power limit, considering the current workload. This paper has presented a general, decision-based approach for performing this coordination.

PUPiL complements other approaches which schedule applications to minimize energy [23, 37, 60, 66]. PUPiL determines what set of resources to activate, but it does not explicitly assign those resources to applications. Instead, it lets the underlying operating system scheduler perform that work. In this paper, that scheduler was simply the default Linux scheduler. It is likely that further performance gains could be achieved by coupling PUPiL with advanced energy-aware schedulers.

7. Conclusion

This paper investigates hardware and software power capping techniques. We find that hardware techniques provide significantly faster response time – quickly enforcing power limits – while software can provide much greater flexibility – by tailoring resource usage to the current application workload. We have used these observations to formulate and evaluate a hybrid hardware/software power capping system called PUPiL. We evaluate PUPiL and compared it to a pure software approach and to Intel's state-of-the-art hardware approach. Across a number of power targets and workloads, we find that PUPiL achieves nearly the same response time as the hardware approach and the flexibility of the software approach. In both single and cooperative multi-application workloads, PUPiL provides at least 18% greater mean performance than RAPL. In oblivious multi-application workloads, PUPiL provides at least $2.4\times$ the mean performance. We conclude that delivering performance under a power cap cannot be left to hardware alone, but requires the cooperation of both hardware and software. We have developed one such cooperative approach and released the code and test cases so that others can use it, compare against it, or extend it.

Acknowledgments We are grateful to the anonymous reviewers whose suggestions improved the paper. The effort on this project is funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award.

References

[1] V. Anagnostopoulou, S. Biswas, H. Saadeldeen, R. Bianchini, T. Yang, D. Franklin, and F. Chong. "Power-Aware Resource Allocation for CPU- and Memory-Intense Internet Services". In: *E2DC*. 2012.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead*. 2008.

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

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *IISWC*. 2009.

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

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

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

[8] 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.

[9] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. "MultiScale: memory system DVFS with multiple memory controllers". In: *ISLPED*. 2012.

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

[11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. "Dark silicon and the end of multicore scaling". In: *ISCA*. 2011.

[12] H. Esmaeilzadeh, T. Cao, X. Yang, S. M. Blackburn, and K. S. McKinley. "Looking Back and Looking Forward: Power, Performance, and Upheaval". In: *Commun. ACM* 55.7 (July 2012), pp. 105–114.

[13] S. Eyerman and L. Eeckhout. "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance". In: *Computer Architecture Letters* 13.2 (2014), pp. 93–96. ISSN: 1556-6056. DOI: 10.1109/L-CA.2013.9.

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

[15] J. Flinn and M. Satyanarayanan. "Energy-aware adaptation for mobile applications". In: *SOSP*. 1999.

[16] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. "Quanto: Tracking Energy in Networked Embedded Systems". In: *OSDI*. 2008.

[17] A. Gandhi, M. Harchol-Balter, R. Das, C. Lefurgy, and J. Kephart. "Power capping via forced idleness". In: *Workshop on Energy-Efficient Design*. Austin, TX, 2009.

[18] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[19] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. Morgan and Claypool Publishers, 2009.

[20] H. Hoffmann. "JouleGuard: Energy Guarantees for Approximate Applications". In: *SOSP*. 2015.

[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] H. Hoffmann and M. Maggio. "PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management". In: *ICAC*. 2014.

[23] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. "A Generalized Software Framework for Accurate and Efficient Managment of Performance Goals". In: *EMSOFT*. 2013.

[24] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. "Dynamic Knobs for Responsive Power-Aware Computing". In: *ASPLOS*. 2011.

[25] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. "Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control". In: *Computers, IEEE Transactions on* 56.4 (2007).

[26] 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.

[27] T. Instruments. `http://www.ti.com/product/ina231`.

[28] S. Iqbal, Y. Liang, and H. Grahn. "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems". In: *Computer Architecture Letters* 9.2 (2010). ISSN: 1556-6056. DOI: 10.1109/L-CA.2010.14.

[29] 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.

[30] M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian. “xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems”. In: *ACM Trans. Embed. Comput. Syst.* 11.4 (Jan. 2013).

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

[32] X. Li, R. Gupta, S. V. Adve, and Y. Zhou. “Cross-component energy management: Joint adaptation of processor and memory”. In: *ACM Trans. Archit. Code Optim.* 4.3 (2007).

[33] X. Li, Z. Li, Y. Zhou, and S. Adve. “Performance directed energy management for main memory and disks”. In: *Trans. Storage* 1.3 (2005).

[34] M. Maggio, H. Hoffmann, M. D. S. an d Anant Agarwal, and A. Leva. “Power optimization in embedded systems via feedback control of resource allocation”. In: *IEEE Transactions on Control Systems Technology (to appear)* ().

[35] 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).

[36] A. Merkel and F. Bellosa. “Balancing power consumption in multiprocessor systems”. In: *EuroSys*. 2006.

[37] A. Merkel, J. Stoess, and F. Bellosa. “Resource-conscious scheduling for energy efficiency on multicore processors”. In: *EuroSys*. 2010.

[38] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann. “A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints”. In: *ASPLOS*. 2015.

[39] S. Mohapatra, R. Cornea, H. Oh, K. Lee, M. Kim, N. Dutt, R. Gupta, A. Nicolau, S. Shukla, and N. Venkatasubramanian. “A cross-layer approach for power-performance optimization in distributed mobile systems”. In: *IPDPS*. 2005.

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

[41] R. Nathuji and K. Schwan. “VirtualPower: coordinated power management in virtualized enterprise systems”. In: *SOSP*. 2007.

[42] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. “No ”power” struggles: coordinated multi-level power management for the data center”. In: *ASPLOS*. 2008.

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

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

[45] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. “Energy Management in Mobile Devices with the Cinder Operating System”. In: *EuroSys*. 2011.

[46] R. Sasanka, C. J. Hughes, and S. V. Adve. “Joint Local and Global Hardware Adaptations for Energy”. In: *ASPLOS*. 2002.

[47] 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*. 2011.

[48] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. “Power Containers: An OS Facility for Fine-grained Power and Energy Management on Multicore Servers”. In: *ASPLOS*. 2013.

[49] Y. Shin, K. Shin, P. Kenkare, R. Kashyap, H.-J. Lee, D. Seo, B. Millar, Y. Kwon, R. Iyengar, M.-S. Kim, A. Chowdhury, S.-I. Bae, I. Hon, W. Jeong, A. Lindner, U. Cho, K. Hawkins, J. Son, and S. Hwang. “28nm High- Metal-Gate Heterogeneous Quad-Core CPUs for High-Performance and Energy-Efficient Mobile Application Processor”. In: *ISSCC*. 2013.

[50] Y. Sinangil, S. M. Neuman, M. E. Sinangi, N. Ickes, G. Bezerra, E. Lau, J. E. Miller, H. Hoffmann, S. Devadas, and A. P. Chandraksan. “A Self-Aware Processor SoC using Energy Monitors Integrated into Power Converters for Self-Adaptation”. In: *VLSI Symposium*. 2014.

[51] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. “Koala: A Platform for OS-level Power Management”. In: *EuroSys*. 2009.

[52] B. Sprunt. “The basics of performance-monitoring hardware”. In: *IEEE Micro* 22.4 (2002).

[53] M. B. Taylor. “Is Dark Silicon Useful? Harnessing the Four Horesemen of the Coming Dark Silicon Apocalypse”. In: *Design Automation Conference*. 2012.

[54] E. Team. *Key Challenges for Exascale OS/R*. Online document, `https://collab.mcs.anl.gov/display/exaosr/Challengesl`.

[55] P. Team. Online document, `http://icl.cs.utk.edu/papi/`.

[56] V. Vardhan, W. Yuan, A. F. H. III, S. V. Adve, R. Kravets, K. Nahrstedt, D. G. Sachs, and D. L. Jones. “GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy”. In: *IJES* 4.2 (2009).

[57] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. “Conservation cores: reducing the energy of mature computations”. In: *ASPLOS*. 2010.

[58] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. “Server workload analysis for power minimization using consolidation”. In: *USENIX Annual technical conference*. 2009.

[59] X. Wang, M. Chen, and X. Fu. “MIMO Power Control for High-Density Servers in an Enclosure”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.10 (2010).

[60] M. Weiser, B. B. Welch, A. J. Demers, and S. Shenker. “Scheduling for Reduced CPU Energy”. In: *OSDI*. 1994.

[61] A. Weissel, B. Beutel, and F. Bellosa. “Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications”. In: *OSDI*. 2002.

[62] 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.

[63] 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.

[64] W. Yuan and K. Nahrstedt. “Energy-efficient soft real-time CPU scheduling for mobile multimedia systems”. In: *SOSP*. 2003.

[65] X. Zhang, R. Zhong, S. Dwarkadas, and K. Shen. “A Flexible Framework for Throttling-Enabled Multicore Management (TEMM)”. In: *ICPP*. 2012.

[66] 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), pp. 1447–1464. DOI: 10.1109/TPDS.2012.20. URL: http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.20.