THE UNIVERSITY OF CHICAGO

MAXIMIZING PERFORMANCE UNDER A POWER CAP

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE THE PHYSICIAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER'S

DEPARTMENT OF COMPUTER SCIENCE

BY
HUAZHE ZHANG

CHICAGO, ILLINOIS
2014

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to thank my advisor Hank Hoffmann, without whom this work cannot be done. I have learned almost everything of how to do research from Hank. Whenever I had a problem with my research or even with my personal stuffs, he is there to help. I have been having a great time working with Hank and really looking forward to many times ahead of us.

Second, I would like to thank my family. My father and mother are always there giving me the biggest support. Whenever life treats me not so well, they are always there to talk to, give me wise advice and comfort me with the warmth of family.

Third, I would like to thank my girlfriend. My decision to study in US makes us more than 10000km away and 10 hours different from each other. Not only had she never blamed me on this, but unconditionally supported my study and shared happiness in life through countless phone calls and video chats.

Last but not the least, I would like to thank Andrew Chien and Haryadi Gunawi to be on my defense committee. I have learnt many valuable things from both of you from courses, system seminars and so much more. It's an honor to have you on my committee.

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 1.10–2.4$\times$ 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 approaches.

CHAPTER 1

INTRODUCTION

Modern processors are constrained by *dark silicon* – their abundance of transistors enables them to draw more power than they can safely sustain [8, 42]. For example, the Exynos 5 processor (in the Samsung Galaxy S4 phone) has a 5.5W peak power that is nearly 2× its sustainable heat dissipation, limiting peak speed to less than 1 second [36]. At the other end of the spectrum, the next generation of exascale supercomputers is predicted to be constrained by an operating budget of approximately 20 MW [1]. Managing power constraints has thus been identified as one of the central challenges facing the design of next-generation supercomputers [40].

These physical constraints create a need for *power control systems* which guarantee the processor operates within a strict *power cap*. Power capping systems have been implemented in software [4, 5, 10, 24, 30–32, 47]. In addition, the need for power capping has become so great that Intel processors now implement power capping in hardware with their Running Average Power Limit (RAPL) interface [6].

Whether implemented in hardware or software, there are two essential properties for a successful power capping system. The first is **timeliness** – the speed with which a new cap can be implemented. The second is **efficiency** – the performance delivered under the power cap. Without timeliness, critical operating bounds can be violated, damaging the hardware. Without efficiency, application performance suffers unnecessarily. It is, in fact, trivial to implement a power cap if we do not care about 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 comes from the fact that relatively simple circuits can be used to control key power indicators like processor voltage and frequency, providing a very efficient mechanism for enforcing the power limit. Software's efficiency comes from the fact that it can consider the complexity of 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 tree approach*. To ensure a power cap, PUPiL navigates nodes in a decision tree. 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 in the tree.

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) and a software-only approach. We evaluate the timeliness and efficiency of all three approaches for both single and multi-application workloads. Our results show:

- **Efficiency:** For single application workloads, the software only approach achieves higher performance than RAPL, and PUPiL achieves the highest performance. Specifically, PUPiL outperforms RAPL by 1.25–1.10$\times$ on average depending on the power cap. (See Section 5.2.)
- **Timeliness:** RAPL's speed enforcing the power caps greatly exceeds the software only approach by orders of magnitude. PUPiL is equivalent to RAPL. (See Section 5.3.)
- **Multi-application Efficiency:** We test two types of multi-application workloads: 1) *cooperative* workloads where each application requests a subset of available resources and 2) *oblivious* workloads where each application requests all resources. In the cooperative case, PUPiL outperforms RAPL by 1.43–1.18$\times$ on average depending on the power cap. In the oblivious case, PUPiL outperforms RAPL by 2.56–2.43$\times$ on average depending on the

power cap.

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, sockets 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.

This paper makes the following contributions:

- Develops a decision tree based approach to maximize performance under a power cap.
- Releases an open source implementation of this approach for Linux/x86 servers.
- Evaluates this implementation on a real system in multiple usage scenarios.
- Makes all scripts and data collection tools from this evaluation available as open source, making it easy for other to test these results on different platforms or with different benchmarks[1].
- Identifies workload characteristics where Intel's state-of-the-art RAPL power capping system will fail to deliver best performance.

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.

The rest of this paper is organized as follows. Section 2 presents a motivational example high-

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

lighting the tradeoffs between software and hardware power control systems. Section 3 describes the generalized decision tree approach. Section 4 details the benchmarks and hardware used to evaluate PUPiL. Section 5 presents the results of this empirical evaluation. Section 6 describes related work in power capping and energy management. The paper concludes in Section 8.

CHAPTER 2

MOTIVATIONAL EXAMPLE

Table 2.1: Server resources.

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

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-only power capper.

Our 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 2.1. 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 is extremely rare for any application to sustain that power consumption.

To illustrate the difference between hardware and software power capping approaches, we set a power cap of 140 Watts for both sockets. RAPL must achieve this power consumption by driving each socket to 70 Watts. The software approach selects 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. 2.1 illustrates the results of this experiment, 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

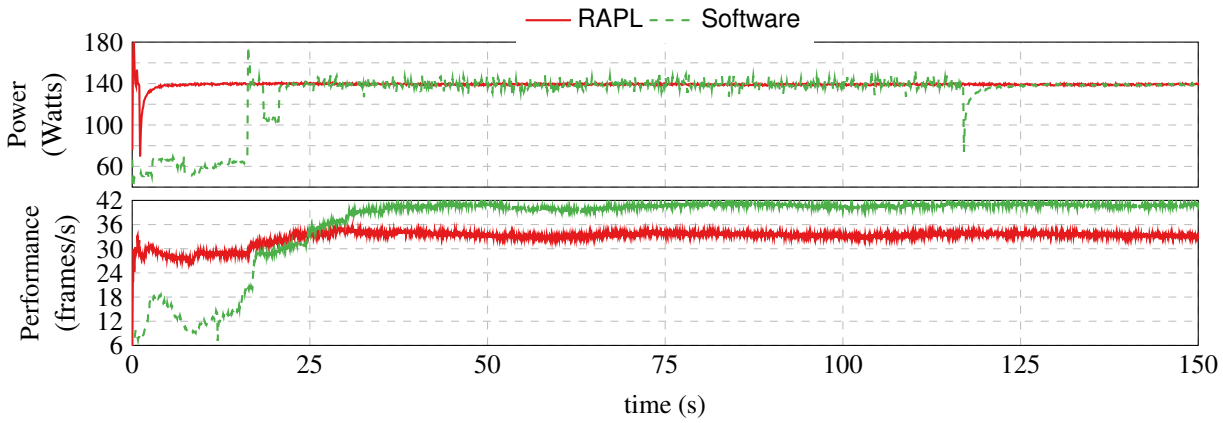

Figure 2.1: Example of the tradeoff between timeliness and efficiency from hardware and software power capping.

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 exceeds the performance delivered by RAPL. Specifically, after convergence, the software approach averages approximately 41 frames per second while RAPL averages approximately 33.5 frames per second. Once the system converges, the software-based approach achieves over 20% better performance.

Software outperforms hardware because it is able to recognize that using hyperthreads does 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 can slightly increase the speed of the cores it is using without hyperthreads. Of course, it takes software a long time to recognize this and adjust.

These results clearly indicate the tradeoffs between hardware and software based power capping. Hardware is fast – it reaches the power cap very quickly, whereas the software approach does not. Software is flexible – it recognizes that x264 cannot use hyperthreading and adjusts, whereas hardware cannot. 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.

The remainder of the paper presents a decision tree based approach for combining hardware and software power control. We then describe how to implement this decision tree for Linux/x86 servers and present the results of our empirical evaluation for both single and multi-application workloads.

CHAPTER 3

POWER CAPPING METHODOLOGIES

This section introduces the different power capping approaches we explore in this paper. It first discusses a software power approach based on decision trees. 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 then *act* to implement these decisions. This feedback loop is then repeated continually, allowing the power capping system to react to application phase changes or other environmental fluctuations. In this section, 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

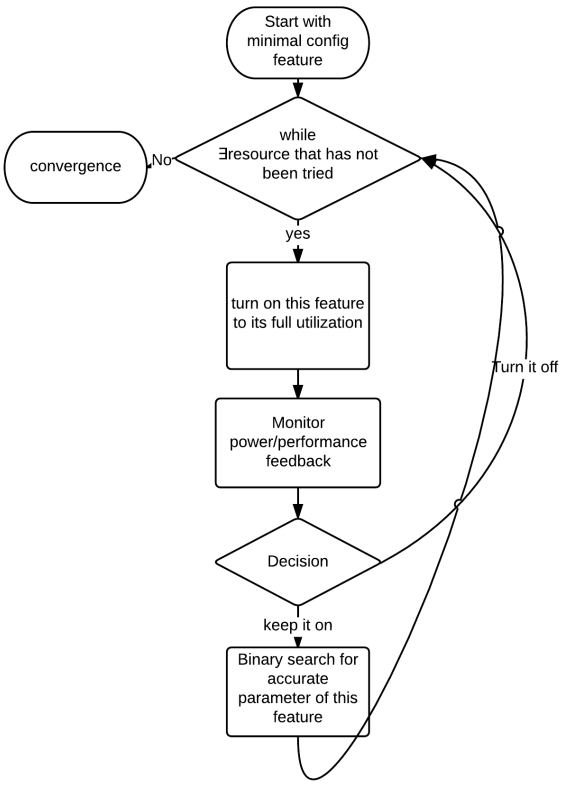

Figure 3.1: Generic decision tree for software power capping.

power monitoring devices, such as the INA231 [21], or on-chip power monitoring, which is available commercially from Intel [6] and through research prototypes [37].

Performance feedback can also come from a number of sources. For example, high-level performance feedback can come directly from appropriately instrumented applications [16]. It could also come from any number of other sources, including hardware counters that measure floating point computation rate or simply instructions per second [39, 41]. 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 disturbed performance. For example, the system should distinguish

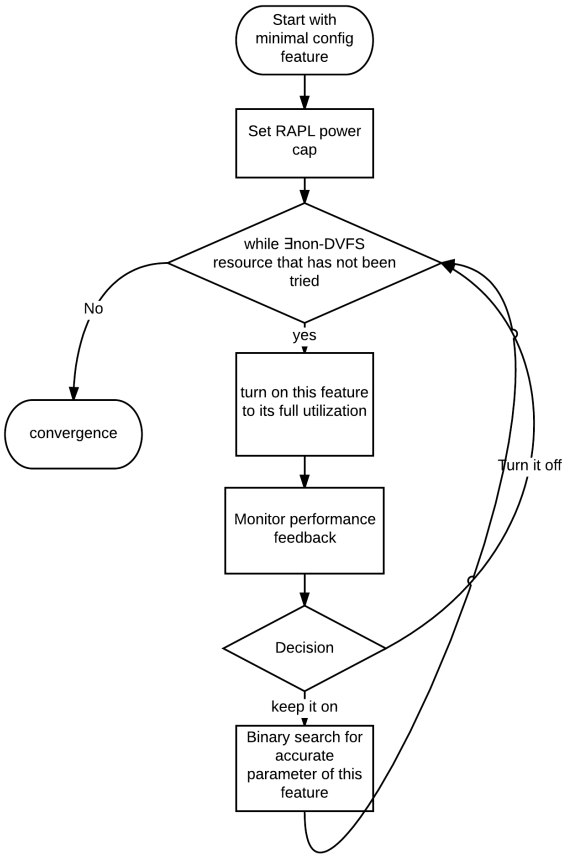

Figure 3.2: PUPiL's approach to hybrid hardware/software power capping.

between a fundamental change in application workload and a temporary timing fluctuation that occurs due to a page fault. We want the system to 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 decisions:

$$\mu = \frac{\sum_i X_i}{N} \tag{3.1}$$

$$\sigma = \sqrt{\frac{\sum_i \left(X^i - \mu^2\right)}{N}} \tag{3.2}$$

$$X_{feedback} = \frac{\sum_{j \in A} X_j}{size(A)} \tag{3.3}$$

$$A = \left\{ j \mid |X_j - \mu| < 3\sigma \right\} \tag{3.4}$$

3.1.2 Decide

In the decide phase, the software controller uses the filtered feedback to select 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.

Instead, the software approach must find a more intelligent way to explore the configuration space. In this paper, we propose the use of *decision trees*. Fig. 3.1 illustrates a generic decision tree based approach. 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 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

Algorithm 1 Walking the decision tree.

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.
      end if
    end if
  end while

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).

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 appear earlier than lower impact resources in the order. Algorithm 2 shows the algorithm used for establishing this

order. The general idea is to allocate power first to higher impact resources. We evaluate impact of each resource by the performance improvement that each resource can deliver 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. The detailed process for establishing the order is shown in Algorithm 2.

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
  end while
  Sort r in O by $perf_r$
  Add DVFS to the last in O **return** O ▷ The set of ordered resources

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 timing 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 tree 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 setting which changed since the last decision.

3.2 Hardware Power Capping

We briefly outline the approach taken by Intel's RAPL system [6], 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 uses a model to estimate 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, then sub-divides the user-specified time interval into a set of finer-granularity intervals. For each of the fine-grained intervals, RAPL calculates the remaining energy budget for the rest of the time in the user-specified time interval and decides the best possible state of processor speed and voltage. Given this decision, RAPL acts to tune DVFS to the decided state and wait for the next fine-grained interval. More detail on RAPL operation is available in the literature [6].

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. 2.1). In 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. In the following sections, we describe how we combine them to achieve both timeliness and efficiency.

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 exploring 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 over which software much search.

The hybrid decision tree is illustrated in Fig. 3.2. The major difference between Fig. 3.1's software approach and Fig. 3.2's hybrid approach 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 its first line sets the RAPL power cap.

3.3.2 Efficiency

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

First, the power cap has now been taken care of by the hardware approach and PUPiL need only focus on 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 re-considered. Hardware power capping caps power on a per-socket manner. However, 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.

CHAPTER 4

EXPERIMENTAL SETUP

This section describes benchmarks, system, and metrics 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`) [2], Minebench (`ScalParC`, `kmeans`, `HOP`, `PLSA`, `svmfe`, `btree`, `kmeans_fuzzy`) [28], and Rodinia (`cfd`, `nn`, `lud`, `particlefilter`)[3]. We also use a partial differential equation solver (`jacobi`) and the `swish++` search webserver [18] and `dijkstra` [22]. These benchmarks test a range of important modern multicore applications with both compute-intensive and data-intensive workloads. 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 consumption.

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 2.1). This motherboard supports setting RAPL's power capping feature. The system runs Linux 3.2.0. We make use of the `msr` module, allowing access to the model specific registers (MSR) used to set RAPL power caps and read energy consumption. 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 control access to memory controllers. In total, the system supports 1024 user-accessible configurations, each with

Table 4.1: 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

its own power/performance tradeoffs[1]. According to Intel's documentation, the thermal design power for these processors is 135 Watts.

Given those specifications, the following resources are configurable on our system: the clock speed of each socket, the cores in 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. Clock speeds are adjusted with the `cpufrequtils` and `numactl` controls access to memory.

As described in Section 3, implementing the software decision tree requires establishing an order on the set of resources under consideration. Table 4.1 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.

1. 16 cores, 2 hyperthreads, 2 memory controllers, and 16 speed settings (15 DVFS settings plus TurboBoost)

4.3.1 Timeliness

Settling time is a standard metric for a control system [14]. 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 t_{ss}. If the controller begins work at time t_0, then the settling time is simply:

$$settle = t_{ss} - t_0 \tag{4.1}$$

Low settling times indicate the desired power is reached quickly. High settling times indicate that convergence is a long process.

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 [9].

CHAPTER 5

EXPERIMENTAL EVALUATION

This section evaluates PUPiL's timeliness and efficiency and compares it to both the software approach and Intel's RAPL. 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 in 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 in RAPL we split the power budget between both sockets evenly. The software and hybrid approaches are free to divide the power cap among the sockets as they see fit.

5.2 Performance

We first evaluate the performance delivered under each power cap. These results are shown in Fig. 5.1. 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, software-only, and PUPiL. We determine optimal speed by running each application in every possible system configuration and measuring its performance. The optimal speed is then the best speed obtainable for that power cap using available system configurations.

While results vary per application and power cap, the general trends show that the software approach provides higher performance, on average, than RAPL. Furthermore, the hybrid approach generally provides the highest performance. The average performance for each power cap and

Table 5.1: Comparison of Average Performance.

Power Cap	**RAPL**	**Software**	**PUPiL**
60W	0.60	0.73	0.75
100W	0.78	0.83	0.87
140W	0.81	0.88	0.90
180W	0.84	0.89	0.92
220W	0.85	0.92	0.94

power controller is summarized in Table 5.1. From this table we see that PUPiL consistently outperforms RAPL across all power caps by at least 10% (at the 180W cap) and at most 25% (at the 60W cap). Furthermore, the software approach is very close to PUPiL. These results confirm that software has an advantage in efficiency over hardware, in general.

Clearly RAPL performs well on some applications (*e.g.,* `btree` and `svmfe`) and poorly on others (*e.g.,* `dijkstra` and `kmeans`). 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 as it is provided more cores on a socket. When `kmeans` is allocated cores on both sockets, however, inter-socket communication becomes a bottleneck. In that case, `kmeans` continues to issue instructions and burn power, but without increasing speed. RAPL must reduce its clock speed to meet the power cap. In contrast, both the software-approach and PUPiL recognize that using the second socket made performance worse, and they restrict `kmeans` to a single socket, but increase the speed of that socket, resulting in higher performance.

5.3 Settling Time

For each application and power cap we measure settling time. Fig. 5.2 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. 5.2 demonstrates the tremendous advantages in timeliness that RAPL has over

the software-based approach. On average, across all benchmarks, RAPL's settling time is 356 ms. In contrast, the software approach averages 95,000 ms, a difference of approximately 260 $\times$. These results demonstrate the claims of timeliness made in the introduction to the paper. RAPL has significant timeliness advantages over the software-based approach. 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.

These results demonstrate the main claims in the introduction. Specifically, RAPL's hardware approach addresses the timeliness challenge, quickly converging to the power cap. The software approach achieves efficiency gains compared to hardware. The average performance advantage is at least 16%, while for specific applications (*e.g.,* `kmeans`, `dijkstra`) the gains can be over 2$\times$. Finally, PUPiL's hybrid approach is able to meet both the timeliness and efficiency challenges, combining the low settling time of the hardware approach with the high performance of the software approach. In the next section, we look at timeliness and efficiency for multi-application workloads.

5.4 Multi-Application Workloads

In this section we evaluate the power capping techniques for multi-application workloads. We begin by dividing our benchmark applications into two sets: ones for which RAPL delivers near-optimal performance, and ones for which RAPL falls short of optimal. Specifically, we consider RAPL to be near-optimal for an application if it achieves greater than 90% of optimal performance under the 180W power cap. Applications above this threshold fall into the set of application for which we consider RAPL to perform well, all others fall into the set where we consider RAPL to perform poorly.

We then 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

Table 5.2: 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, fussy-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, fussy-kmeans
mix12	btree, particlefilter, kmeans, STREAM

four mixes (1–4), all applications are drawn from the set for which RAPL achieves near optimal performance. 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. These multi-application workloads are summarized in Table 5.3: each workload is given a name – `mixN` – and we list the applications used in that workload.

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. 5.3. 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).

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 per-

Table 5.3: Ratio of PUPiL to RAPL Average 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

formance, but there are some where PUPiL's hybrid approach far outperforms RAPL. Table 5.3 shows the average ratio of PUPiL to RAPL performance across all mixes for each power cap. In the cooperative scenario, PUPiL consistently outperforms RAPL by at least 18% on average.

It interesting to note that the 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

The performance for the oblivious multiapp scenario is shown in the right column of Fig. 5.3. 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 average results across all performance caps are shown in Table 5.3, which indicates that PUPiL achieves at least 2.4$\times$ better average 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.

Table 5.4: Ratio of PUPiL to RAPL Average 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

The reason for PUPiL's higher performance is that these oblivious workloads typically bottleneck on some resource early. 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.

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 5.4 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.

From the table, it is obvious that under RAPL control these mixes spend significantly larger portions of their time spinning and achieve a significantly smaller memory bandwidth. We believe that 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 to make

progress and yield the CPU when they cannot make progress. The one application that does polling synchronization, however, apparently 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. Somewhat surprisingly, the use of fewer cores appears to reduce the chance that one application's threads get scheduled and just do polling synchronization.

RAPL cannot detect this, but PUPiL, which dynamically monitors performance can. PUPiL realizes that moving to a state with fewer cores results in an overall performance increase. Thus, this study with multiple applications further demonstrates the efficiency of software-based approaches and validates the design of PUPiL which incorporates dynamic feedback.

5.5 Sensitivity and Overhead Analysis

Throughout this section we investigate several factors which affect the results. Our results examine sensitivity to various power caps. We have seen that performance under very low power caps is difficult for any power management system. We have also seen that in both cooperative and oblivious multiapp scenarios PUPiL outperforms RAPL across a wide range of power caps. Further, the use of diverse workloads has demonstrated 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. First, it can arise from the number of measurements that need to be taken before the system converges. Second, it can arise as an impact on the converged system. Our results account for both forms of overhead. We emphasize that all results reported in this section 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. 5.2. We see that the software only approach has very high, likely unusably high, overhead by this metric. The second type of overhead is accounted for by the comparison to

optimal in Fig. 5.1. This figure shows that the performance impact of the PUPiL runtime system is acceptable in that PUPiL produces the closest to optimal performance.

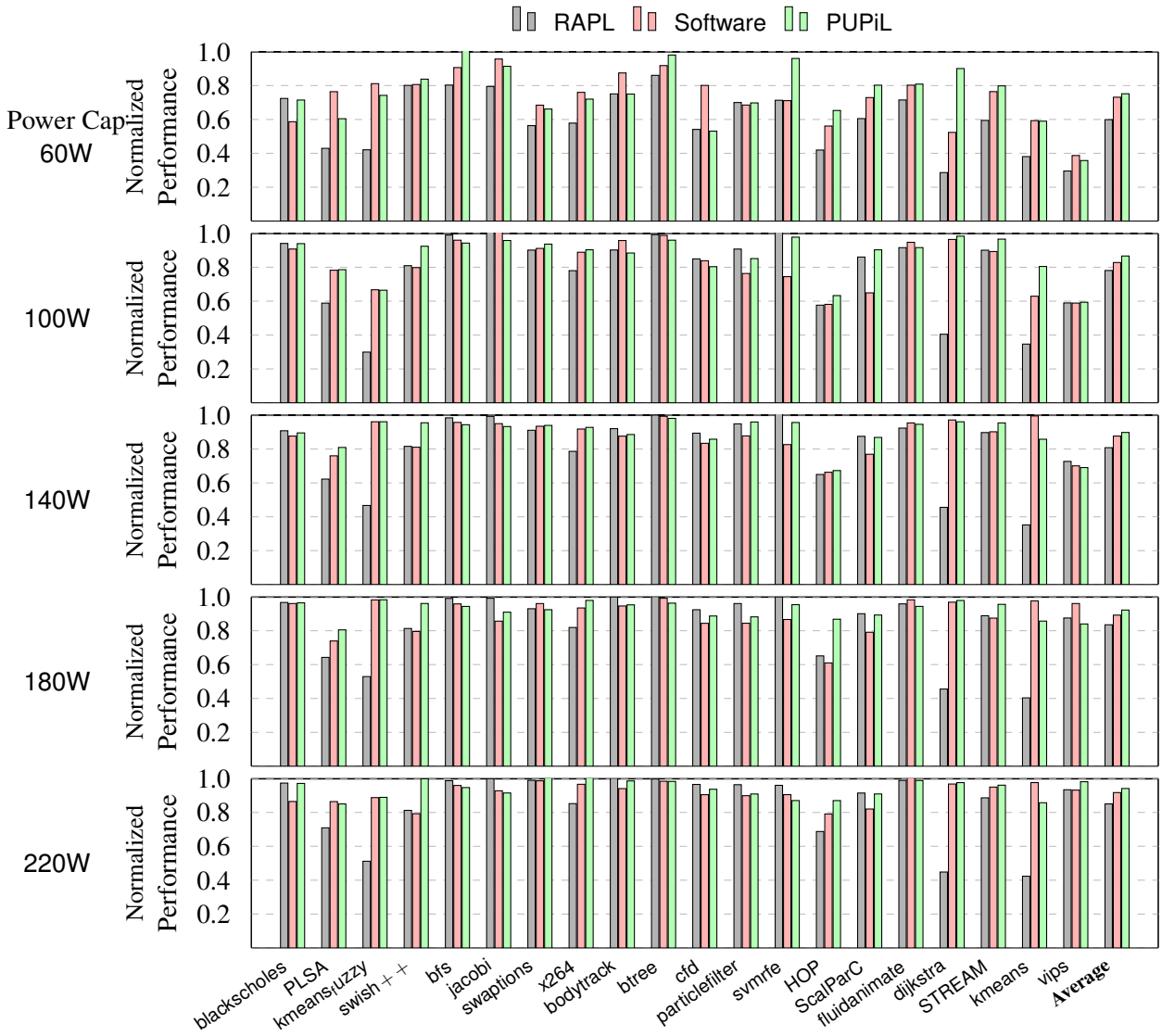

Figure 5.1: Performance of several power control techniques normalized to optimal.

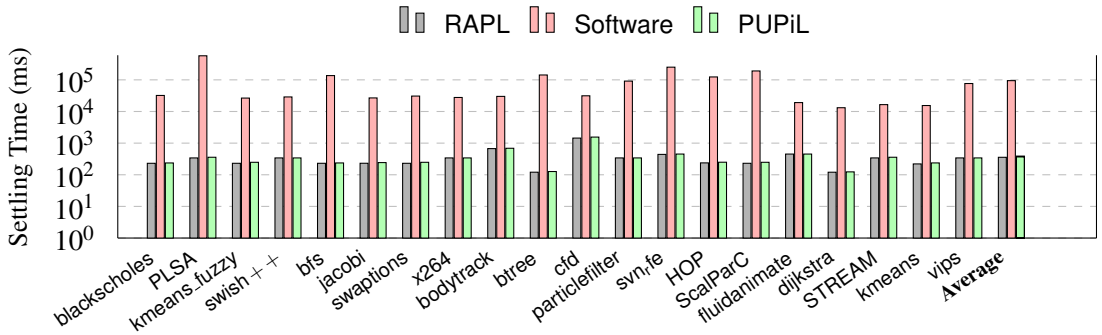

Figure 5.2: Settling times for several power control techniques.

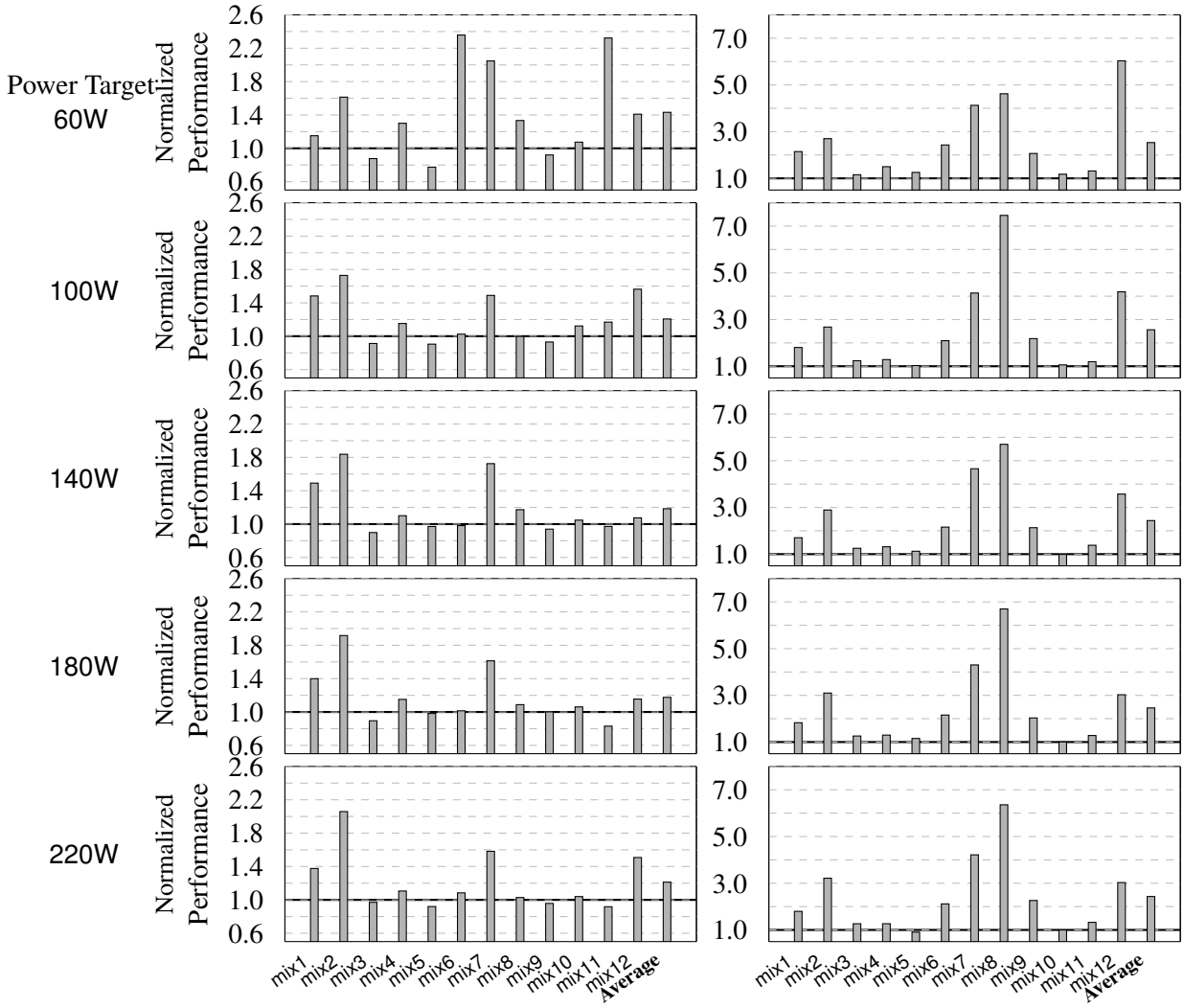

Figure 5.3: Ratio of PUPiL to RAPL performance in cooperative (left) and oblivious (right) multiapp scenarios.

CHAPTER 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. Power management approaches have been proposed in diferent computing fields and level. Approaches focusing on meeting performance target while minimizing energy cost appeared in both cluster computing [19, 34, 43, 48] and mobile computing [11, 20, 49]. However, none of these approaches implement power cap or have power limit guarantee. In system level, several OS projects have added operating system support for monitoring and allocated energy. The Quanto project facilitates tracking energy usage in networked embedded devices [12]. The Cinder OS allows energy usage to be tracked and allocated across multiple applications in a system [33]. The Koala project also allows energy to be tracked and allocated while supporting several different policies for optimizing energy and performance [38]. Similarly, *power containers* support fine-grain tailoring of heterogeneous resources to varying workloads [35]. 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 [46]. None of these projects, however, explicitly support maximizing performance under a power constraint, which is the subject of this paper.

while minimizing energy either in cluster to cut down the electricity bill or in mobile devices to increase batter life is an important problem, power capping is a different concern. Firstly, operating within power limits has become essential as multicore scalability is increasingly limited by power and thermal management [8, 42]. The physical realities of power dissipation in modern processors have led to hardware designs characterized by *dark silicon*. Secondly, uncontrollable power usage has led to great over-provisioning of power delivering and cooling devices in datacenters. Reliable power capping can eliminate the over-provisioning and further increase the computing density of datacenter, which motivates Intel's SandyBridge and later processors to support power management in hardware [6] . However, it only considers tuning the DVFS rather than coordinating all

configurable resources as we proposed, it cannot deliver optimal efficiency.

Cluster level solutions which guarantee power consumption include those proposed by Wang et al. [44] and Raghavendra et al. [30]. These cluster-level solutions require some node-level power management scheme. Node-level systems for guaranteeing power consumption have been developed to manage different individual components including DVFS for a processor [24], per-core DVFS in a multicore [23], processor idle-time [13, 50], DRAM [7].

Several researchers have noted that coordinating multiple components provides greater performance under a power cap than management of a single component in isolation [15, 17, 25, 29]. Thus, approaches have been proposed which provide power guarantees while increasing performance through coordinated management of multiple components, including processor and DRAM [4, 10], processors speed and core allocation [5, 32], combining DVFS and scheduling [31, 47], and combining DVFS and process placement [26]. The VirtualPower project coordinates power management, virtual machine placement, and server consolidation to meet power constraints in a virtualized data center [29]. 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 believe 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. 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 tree-based approach for performing this coordination.

We note that this approach is complimentary to other approaches which schedule applications to minimize energy [27, 45, 51]. 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.

CHAPTER 7

FUTURE WORK

This section introduces several promising research project as extensions to our work.

Firstly, further coordinating memory and processor power is potentially another big win. As a considerable proportion of workloads in server system are memory intensive, this gives memory power a increasingly large weight of the whole system power. Some data from Intel shows that memory can contribute as much as 25 percent of whole system power. Therefore, to efficiently cap the system power at a larger power range, we need to consider both processor and memory power. In this work, the only knob for adjusting memory power is memory controller number and we don't have a choice of on which power state memory nodes are running. As more memory power management techniques appear, such as RAPL memory power limiting, it will be interesting to see how much influence memory power management contributes to the overall power and if it's considerable, how should we coordinate both? We believe that feedback based control loop can find the answer but then how to efficiently find the balance point will be critical.

Second, multi-node power capping is a natural extension of this work. Power capping techniques are a promising approach to solve the over-provisioning of power supply and cooling devices in datacenters. Our approach, however, is currently system-level power capping and cluster-level power capping will be requested for such needs. Therefore, the future project would be extending current work from single node to multiple nodes. Trivially using single node power capping to evenly cap power of multiple nodes may solve the problem of power overload but will generate serious performance issue. To coordinate different nodes to achieve better performance, we have to allocate power to the most needed component based on the information of workload, *e.g.,* data dependency. For example, in a simple pipelined computation model, we need to allocate more power to the slowest stage to boost up overall performance.

Thirdly, Coupling PUPiL and energy-aware scheduler would have further performance gain. In this paper, we determine the set of resources to be activated for best performance, but we don't

directly assign the resources to application, instead, it is done by the underlying operating system. For example, in the oblivious multiple application case, we assign the activated resources to all the four benchmarks and OS will be deciding how many threads of which benchmark is deployed on which core. Given the fact that, different application favors different configuration, a energy-aware scheduler which can isolate each of the application and assign the optimal resources to each of the applications would achieve higher performance, *e.g.,* in our multiple application experiment, when kmeans application competes resources with other 3 applications on all 32 cores, it spins for most of the time and keeps holding the core resource from being used by other application and this behavior destroys the total performance of all four applications. A 'smart' scheduler would isolate kmeans from other applications and maybe keeps it on its optimal configuration(single socket,16 core). In such way, neither will kmeans spin that much, nor it will compromise other application's performance.

Lastly, hardware counter based performance modeling may benefit our management system. Our current performance feedback technique needs code segment to be inserted into the application. This is a constraint for existing software even the code modification is minimal. Also, performance feedback time varies a lot depending on the application, which sometimes greatly increase the converging time of finding the optimal configuration. As we have seen, there are some correlation between the hardware counter such as IPC, cache miss, memory access, spin cycle, context switching, etc. If we can derive performance within acceptable accuracy using only those hardware counters, it will benefit not only our work but almost all of the feedback based approach will have a faster control loop and no more software code modification will be needed. Therefore, this would be a challenging and meaningful thing to study.

CHAPTER 8

CONCLUSION

This paper has investigated hardware and software power capping techniques. We have found that hardware techniques provide significantly faster response time – they quickly enforce power limits – while software techniques provide much greater flexibility – they can tailor 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 evaluated PUPiL and compared it to a pure software approach and to Intel's state-of-the-art hardware based power capping approach. Across a number of power targets and workloads, we find that PUPiL achieves the same response time as the hardware approach and the flexibility of the software approach. In single application workloads, PUPiL provides at least 10% greater average performance than RAPL. In cooperative multi-application workloads PUPiL provides at least 18% greater average performance. In oblivious multi-application workloads, PUPiL provides at least 2.4$\times$ the 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 based approaches. We have developed one such approach and released the code and test cases for this approach so that others can benefit from it, compare against it, or extend it in future work.

BIBLIOGRAPHY

[1] 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.

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

[3] 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.

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

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

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

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

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

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. E. Miller, S. M. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. P. Chandrakasan, and S. Devadas. “Self-aware computing in the Angstrom processor”. In: *DAC*. 2012.

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

[19] 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).

[20] 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.

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

[22] 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`.

[23] 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.

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

[25] D. Meisner et al. “Power management of online data-intensive services”. In: *ISCA* (2011).

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

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

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

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

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

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

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

[42] 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.

[43] 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.

[44] 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).

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

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

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

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

[51] 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.