THE UNIVERSITY OF CHICAGO

# MAXIMIZING PERFORMANCE IN POWER-CONSTRAINED COMPUTING SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
HUAZHE ZHANG

CHICAGO, ILLINOIS
JUNE 2019

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

It has been a wonderful journey exploring the unknown world for last 6 years! I am extremely grateful to all those who have helped to make it special to me, without any of you, this thesis would never be the same or even finished. I can only hope to list all of you here.

The world's BEST advisor, Professor Henry(Hank) Hoffmann. From years of studying science, I learned that an extreme statement is rarely true, but I believe "BEST" here is one of the rare cases. Thank you for patiently teaching all the basics of research and communication in English when I just started in Chicago, for always pointing out the right way to go, for motivating and cheering while I am down, for all the care along the way for not only my research but my life... And this list goes on. You have been way more than a research advisor to me, but role model and great friend!

My wife, Jiayi Liu. Your unconditional love and support mean the world to me. Time has been very long before you came to Chicago and it flies afterwards. You gave up job opportunities for supporting me, you closed the distance of tens of thousands of miles, you made everything, every day easier for me, you are the true hero.

My parents and parents-in-law, Kaiwang Zhang, Yanjun Wang, Xiulin Liu and YongQing Yu. Your unwavering support and love is always with me before and during grad school. You made every effort that parents can make to help and support Jiayi and me. We appreciate for everything you have given us.

Professor Zhihui Du, who aided me getting to Chicago. You gave the initial opportunity to me to get into research and that opened up the amazing world to me and all these followed. I cannot thank you enough for giving me the chance to do research with you when I am an undergrad. I would not be where I am without your kind help.

My master thesis committee members, Andrew Chien and Haryadi Gunawi. My doctoral dissertation committee members, Ian Foster and Haryadi Gunawi. Thank you for dropping wisdom to me and this work. This thesis can not be as good without your great advice.

My great friends in University of Chicago. Zhao Zhang, for so much help on getting me settling down and blending in the life here and for your great companionship. Fan Yang, for your kind support on helping me to configure RIVER machines time after time. Your kindness will keep inspiring me, even you are not around. Jason Lui, one of my best buddies, for everything unrelated to work we have done here, I miss all the lifting, balling, fishing, eating, gaming. Connor Imes and Nikita Mishra, for the strong collaboration and share of life in school and conferences. All my lab mates and peers, Saeid Barati, Zhixuan Zhou, Roselyne Tchoua, Will Kong, Anne Farrell, Bernard Dickens, Ivana Marinčić Tong Hu, Yuanwei(Kevin) Fang, Liwen Zhang, Jialei Wang, for creating this great atmosphere to study and live, for all the valuable inputs into my life.

The department technical and administrative staff, especially Bob Bartlett, Margaret Jaffey. You have done a great job to all students, thank you for make our life easier.

My mentor/managers during internships. Leonardo Piga, for your kind mentoring at AMD research. Qingyuan Deng, for your outstanding management and help at Facebook and for helping me get return offer which saves me tons of time job hunting. Look forward to working with you in the near future.

Special thanks to great research platforms for supporting my research: RIVER – Research Infrastructure to explore Volatility, Energy-efficiency, and Resilience at University of Chicago, administried by Andrew Chien and Fan Yang. Chameleon, a configurable experimental environment for large-scale cloud research.

Finally, for those not mentioned here, I apologize if I miss your name. I appreciate the knowledge, help, and laughters you brought to my life. I appreciate each and every interaction with you and that I believe takes me where I am today. So thank you! I am definitely looking forward to our path crossed in the future.

# ABSTRACT

Power constraints have become arguably the biggest obstacle for the performance scaling of computing machines. No matter what scale of computing system – A mobile phone or supercomputer – they are all power restricted in one way or another to ensure normal operation. While various computing systems may require different power management techniques, the goal of such systems is invariant and contains two folds of requirement: (1) guarantee computing system operates under a certain power budget/cap, and (2) efficiently make use of the limited power to deliver high performance. Thus, the challenge can be formalized to a classic constrained optimization problem – given power consumption constraints, maximize the performance of computing systems. In this dissertation, we focus on solving this problem for server systems from single-node level to large-scale. More specifically, this dissertation contains 3 projects addressing power capping challenge at different scales.

First, we propose *PUPiL*, a hardware/software hybrid power control system to address the power challenge at the node level. It makes the key observations of tradeoffs between existing software-based and hardware-based approaches: (1) hardware techniques provide significantly faster response time – quickly enforcing power limits and, (2) software provides much greater flexibility – by tailoring resource usage to the current application workload – leading to high performance efficiency. *PUPiL* combines the best of software and hardware approaches, achieving significantly higher performance with nearly same response time as hardware approach.

Second, we propose *PowerShift*, a distributed power management system to address the emerging challenge of power capping dependent applications in large-scale systems. *PowerShift*, to our knowledge, is the first work to identify the unique challenge of dependent distributed workloads and presents a family of three techniques for this scenario, demonstrating improved performance, reduced energy, and dynamic adjustment to tail behavior and system noise.

Last, *PoDD*, a hierarchical distributed power control system inspired by both *PUPiL* and *PowerShift*, is proposed to further overcome major limitations in power capping dependent applications. It incorporates learning/hardware hybrid node-level power capping with system-level power shifting to deliver significantly higher performance than prior works and no longer requires offline application profiles because of building power models online, greatly improving practicality and performance efficiency.

The 3 power management frameworks systematically study the problem of maximizing performance in power constrained systems. The key ideas and insights are highly general to guide the design of real world power control systems for a wide range of workloads and platforms. All implemented systems are open-source and evaluated to be practical, scalable, reliable and also not limited to particular applications and systems, which hopefully will serve as a base model/system to future research on power capping.

# CHAPTER 1

# INTRODUCTION

## 1.1 Thesis Statement

This dissertation address the indispensable need for power budgeting/capping techniques in computing systems focusing on server systems from single-node scale (a server) to large-scale computing system (datacenter or supercomputer consists of tens of thousands of servers). The challenge of the power constraint problem are two folds: (1) how to enforce the system respecting power limit, and (2) more challengingly, how to achieve optimal performance under power caps. At node level, we propose using adaptive feedback control system that coordinates software and hardware power capping technique for high performance efficiency and timely system response. At large-scale level, we introduce a hierarchical power management framework to coordinate node-level power optimization and system-level power shifting. The node-level power optimization involves learning optimal resource allocation from hardware performance counter, which requires no code instrumentation nor application profiles. The power shifting involves an original 3-group power shifting mechanism to address the unique challenge of dependent distributed applications.

## 1.2 Challenges

Power constraints have become first-class concerns in computing systems. From a single processor on our phone to gigantic million-node supercomputers, they are all restricted with power budgets. Thus, there is an increasing need for power control systems to help computing systems running within budget while delivering high performance. Since power constraint problems are different for systems of different size, following paragraphs break the problem down into single node power capping and distributed power capping.

### *1.2.1 Node-level Power Capping Challenge*

At node-level, modern processors are constrained by *dark silicon* – their abundance of transistors enables them to draw more power than they can safely sustain [25, 98]. 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 [87]. At the other end of the spectrum, future exascale supercomputers have a predicted operating budget of 20 MW [6], making power management a central challenge of supercomputer operating systems [95].

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 [13, 15, 29, 54, 76, 77, 79, 103]. 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 [16].

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.

However, none of the software-only or hardware-only approaches achieves both satisfying efficiency and timeliness.

### *1.2.2 Distributed Power Capping Challenge*

At the distributed level, next generation *exascale* supercomputers are predicted to have a strict operating budget of approximately 20 MW, but the total power dissipation at full utilization would far exceed this budget [6]. These systems require sophisticated, distributed *power-capping* mechanisms to assure their power budget will not be exceeded; the United States Department of Energy (DoE) has therefore declared power management a key challenge for exascale [95].

While power concerns create new problems, exascale's increased capacity creates new opportunities. Specifically, instead of sequentially running dependent jobs that communicate through disk, the size of exascale supercomputers allows these jobs to be *coupled* [2, 6, 12, 51, 95]. That is, two formerly independent jobs can now be run simultaneously and communicate at runtime. For example, scientific simulations can now be run with *in situ* data analysis or visualization providing scientists the insight needed to alter the simulation as it runs [2, 12]. Additionally, separately developed physics simulations can now be run together, sharing their results to provide much greater fidelity [51]. In fact, the DoE has declared resource management for coupled application workloads an additional concern for exascale [6, 95]. There are three challenges to maximizing performance for coupled applications under a power cap:

1. The coupled applications have distinct power and performance tradeoffs. Thus, optimal performance requires imbalanced power allocation and finding the highest performance power allocation is not trivial. Specifically, the couple's performance is dependent on the slowest job, so a power allocation that provides good performance for an application in one couple may be sub-optimal when that application is part of a different couple.

2. Such coupled applications almost invariably go through distinct phases of communication and computation and a power manager should adapt to these phases.
3. For some couples and power budgets, performance improvement is not possible due to diminishing returns in one application's power/performance trade-off space. The challenge here is tp recognize when performance cannot be improved and reallocating power to save energy.

Unfortunately, existing work on enforcing power caps across large scale systems does not account for coupled applications. For example, several approaches increase overall system performance with unbalanced power allocations [45, 49, 89]. Other approaches shift power as independent applications transition through compute and IO phases [84]. While unbalanced allocation and phase-adaptation are important for coupled applications they are not sufficient. In brief, coupled applications obey the basic principles of pipeline parallelism: the couple's speed is determined by the slowest application. Thus, *optimizing coupled application performance under a power cap often requires slowing down the faster application to shift more power to the slower.* This requirement to slowdown a fast application is unique to coupled applications and is not supported by prior power shifting approaches, which consider only one application at a time [45, 82], or optimize for multiple, independent applications [49, 84, 89]. Furthermore, due to the couple's interaction, sometimes it is not possible to increase performance through any amount of power shifting. Here, the couple scheduler should reduce total power usage to save energy.

## 1.3 Contributions

This dissertation studies two major problems in the spectrum of optimizing performance under power caps: (1) the efficiency and timeliness tradeoffs in power capping single node server system, (2) maximizing performance of dependent workloads in power-constrained distributed systems. For each problem, we propose and implement original power management

systems to address its unique challenges. All systems are designed to work for various real world applications and platforms. Evaluations on real computing systems with a number of important applications show great advantages of performance efficiency and other properties of our systems. The key idea and insights in the thesis, hopefully, can offer help to or be built upon by future research in this area. More specifically, we tackle these two problems with three distinct, yet continuous projects:

- *PUPiL*, maximizes performance under a power cap for single node server. By observing the tradeoffs between software-only and hardware-only power capping techniques in terms of performance efficiency and timeliness, it proposes a hybrid software/hardware approach to take advantage of the benefits from both worlds.
- *PowerShift*, optimizes performance for dependent distributed applications under system-wide power caps. To the best of our knowledge, *PowerShift* is the first work to consider coupled applications in distributed power capping. We point out the fundamental principle to optimize for coupled applications: to keep each application running at even speed. Following this idea, *PowerShift* offers 3 different power capping frameworks to handle dependent workloads.
- *PoDD*, further optimizes for dependent distributed applications for performance and practicality in two major aspects: (1) hierachical power capping system to coordinate advanced node-level power capping (inspired by *PUPiL*) with system-level power shifting (inspired by *PowerShift*), (2) requires no prior knowledge of application profiles by online model building.

This following sections detail each of the 3 contributions.

### *1.3.1* PUPiL

In this project, we explore the node-level power capping techniques. A key observation is: 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. We explore the tradeoff 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. *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. In both single and cooperative multi-application workloads, PUPiL provides at least 18% greater mean performance compared to RAPL (the state-of-the art hardware power capping technique). In oblivious multi-application workloads, PUPiL provides at least 2.4x the mean performance.

The fundamental contribution of *PUPiL* 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 project demonstrates it is possible to achieve significant performance gains over Intel's state-of-the-art, commercial hardware approach – especially for multi-application workloads.

Besides contribution mentioned above, this project 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 the 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].

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

### *1.3.2* PowerShift

This project addresses the emerging challenge of coupled workloads running under a system-wide power cap in distributed computing scenario. The unique performance characteristics of coupled applications – that the couple's speed is determined by the slowest application – throws off the prior works in this area, which either optimize for single application or independent applications [4, 9, 24, 33, 45, 49, 82, 84, 89]. *PowerShift* is the first work, to our knowledge, to address the unique challenges of coupled applications with *PowerShift*, a family of three techniques for shifting power between dependent applications in a distributed system. All three respect the system-wide power cap and maximize the couple's performance by shifting power from the faster application to the slower one, until both run at the same rate. All techniques rely on existing node-local power cap enforcement (*e.g.,* [52, 54, 79, 109]) *PowerShift-S* is a static technique that sets power caps based on individual application profiles. *PowerShift-C* is a *centralized dynamic approach* in which a single decision maker dynamically shifts power. *PowerShift-D* is a *distributed dynamic approach* where nodes put surplus power into a shared pool, and nodes that need more power take from the pool. These techniques provide different tradeoffs in overhead, flexibility, and adaptability. We implement *PowerShift* on a real 26 node system. We compare SLURM, a state-of-the-art, power-aware job scheduler [89] and *PowerShift* to a *Fair* approach that evenly divides power among all nodes. Results show that *PowerShift* improves mean performance up to 17% over *Fair* approach and up to 14% over *SLURM* . Additional results demonstrate that *PowerShift* detects when no performance improvement is possible and instead reduces energy. Furthermore, *PowerShift D & C* adapt to system noise and tail behavior, automatically shifting power to nodes that have unexpected extra load or long tails, improving performance by 36% and 30%, respectively. Finally, we show that *PowerShift* is *topology-oblivious* and works equally well if the coupled applications are physically separate or mapped to different power domains on the same node. In summary, *PowerShift* makes the following

contributions:

- First work to identify the unique challenge of coupled applications.
- Proposes *PowerShift*, a family of power capping frameworks for coupled applications.
- Evaluaton shows *PowerShift* achieves 7-14% performance gain over SLURM, 18% energy saving for 5% performance loss, 30-36% performance gain in noisy environment
- Evaluation and comparison between 3 power capping frameworks offer insights of fundamental design tradeoffs on overhead, flexibility, and adaptability.
- Open-source all scripts, code, and data collection tools for future research to be built on [2].

### *1.3.3* PoDD

Coupled applications are predicted to be one of the most important workloads in exscale supercomputer [95], and the system power budget is predicted to be 20MW [6]. While *PowerShift* has started to address this problem, *PoDD* further addresses two major challenges to significantly improve performance efficiency and practicality. Specifically, 2 major limitations in *PowerShift* are:

- Dependence on offline application profile greatly makes it less practical in real life systems.
- Hardware-only power capping at node-level results in sub-optimal performance.

We propose *PoDD*, a dynamic power management system addressing the challenge of coupled applications. It delivers high performance by incorporating advanced original node-level power capping technique to coupled-workloads-aware system-level dynamic shifting. It no longer requires prior knowledge of application profiles by building power performance model online. Furthermore, it does not need any code instrumentation. Finally, it greatly mitigates tail effect in distributed environment, is resilient to system noise and scales well

2. All source code, scripts, inputs, and patches are available at: https://github.com/huazhe/powershift.git.

large number of nodes. We implement *PoDD* on a 49 node distributed system and evaluate it against 4 widely-used/state-of-the-art power control systems: *Fair*, *SLURM*, *PowerShift-S* and *PowerShift-D*. The evaluation shows *PoDD* improves mean performance over emphFair by 28%, which outperforms *SLURM* by 21% , outperform *PowerShift-S* by 19%, and outperform *PowerShift-D* by 14%. Our evaluation on noisy environment and scalability also shows *PoDD* is resilient to system noise and predicted to have a 20X scalability over current 49-node system. Evaluations show great flexibility of *PoDD*, that it is topology-oblivious and works well whether coupled applications are physically separate or co-located.

To sum up, *PoDD* makes following contributions:

- Proposes an original machine learning classifier and hardware hybrid node-level power capping technique.
- Deploys a hierarchical power control system that coordinates advanced node-level power capping with system-level power shifting.
- Builds performance power model online to derive optimal power distribution for coupled applications, releasing the dependence on prior application profiles.
- Evaluation shows an average 14% speedup over the state-of-the-art approach *PowerShift*, flexible – topology-oblivious, scalable – at least 20x scale potential, reliable – resilient to system noise.
- Open-source [3].

3. All source code, scripts, inputs, and patches are available at: https://github.com/podd2019/podd.git.

# CHAPTER 2

# *PUPIL*: MAXIMIZING PERFORMANCE UNDER A POWER CAP AT NODE-LEVEL

This chapter introduces *PUPiL*, a software and hardware hybrid power capping approach for single node systems.

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 power cap, but only manages voltage and frequency, limiting its potential performance.

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.

The rest of this chapter are organized as follows: First, Chapter 2.1 discusses the related works in this domain. Next, Chapter 2.2 introduces motivational examples. Then, Chapter 2.3 demonstrate the system design and algorithms. After that, Chapter 2.4 and Chapter 2.5 shows the experimental setup, results, and discuss its reason and insight. Fi-

nally, Chapter 2.6 concludes this work.

## 2.1 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 [42, 63, 85, 99, 104] or increase battery life in mobile and embedded platforms [30, 37, 44, 53, 70, 83, 97, 106]. 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 [31]. The Cinder OS allows energy usage to be tracked and allocated across multiple applications in a system [81]. The Koala project also allows energy to be tracked and allocated while supporting several different policies for optimizing energy and performance [90]. Similarly, *power containers* support fine-grain tailoring of heterogeneous resources to varying workloads [86]. LEO is a hierarchical Bayesian learning framework that produces extremely accurate estimates of an application's performance and power consumption [69]. 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 [102]. The GRACE OS meets performance requirements for media while minimizing energy [97, 106]. 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 [37].

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 [25, 98]. 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) [94].

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 [16, 0]. 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. [100] and Raghavendra et al. [76]. 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) [54], per-core DVFS in a multicore [48], processor idle-time [32, 111], and DRAM [23].

Several researchers have noted that coordinating multiple components provides greater performance under a power cap than management of a single component in isolation [3, 36, 39, 59, 63, 72]. Thus, approaches have been proposed which provide power guarantees while increasing performance through coordinated management of multiple components, including processor and DRAM [13, 21, 22, 29, 57, 83], processors speed and core allocation [15, 79], combining DVFS and scheduling [77, 103], memory and disk speed [58] and combining DVFS and process placement [64]. The VirtualPower project coordinates power management, virtual machine placement, and server consolidation to meet power constraints in a virtualized data center [72]. 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 [26]. 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 [40, 65, 101, 112]. *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.

## 2.2 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 approach that can adjust many settings (presented in Chapter 2.3).

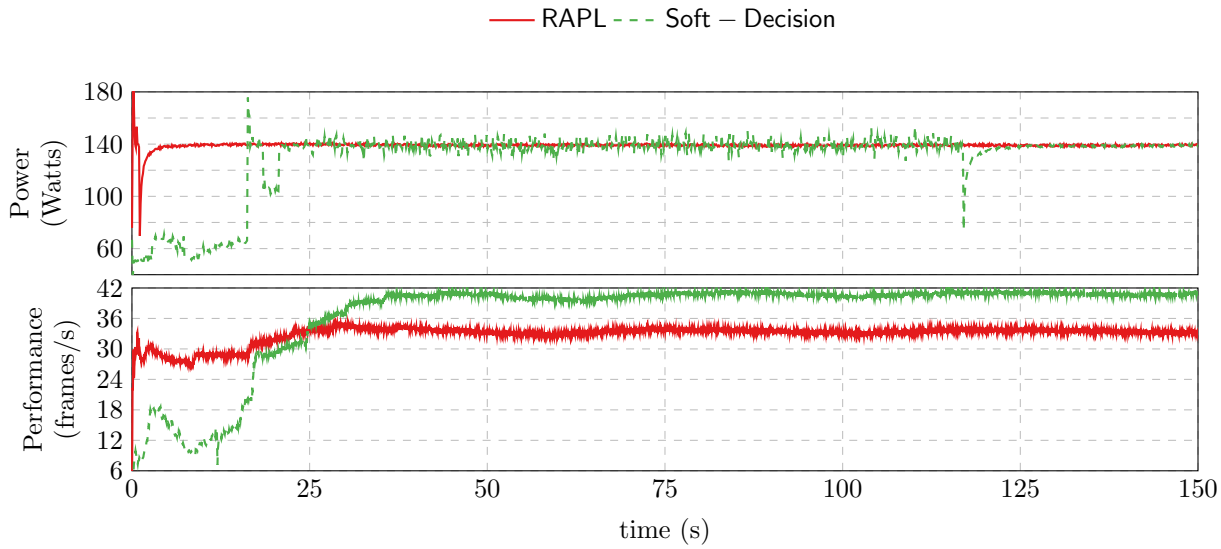


Figure 2.1: Tradeoff between timeliness and efficiency from hardware and software power capping, running x264.

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 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 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 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.

Figure 2.1 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.

## 2.3 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 man-

ner. 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.

### *2.3.1 Software Power Capping*

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

#### 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 [46], or on-chip power monitoring, which is available commercially from Intel [16] and through research prototypes [88].

Performance feedback can also come from a number of sources. High-level performance feedback can come directly from appropriately instrumented applications [38]. It could also come from any number of other sources, including hardware counters that measure floating point computation rate or simply instructions per second [92, 96]. 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, $\mu$ is the average of unfiltered $X$, $\sigma$ is the standard deviation of unfiltered $X$, then $X_{feedback}$ is the performance feedback used by the system to make decisions:

$$\mu = \frac{\sum_i X_i}{N} \tag{2.1}$$

$$\sigma = \sqrt{\frac{\sum_i \left(X^i - \mu^2\right)}{N}} \tag{2.2}$$

$$X_{feedback} = \frac{\sum_{j \in A} X_j}{size(A)} \tag{2.3}$$

$$A = \left\{ j \mid |X_j - \mu| < 3\sigma \right\} \tag{2.4}$$

## 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.

**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.
      **end if**
    **end if**
  **end while**

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

**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

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.

## 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 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.

### *2.3.2 Hardware Power Capping*

We briefly outline the approach taken by Intel's RAPL system [16], 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 [16].

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 Figure 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.

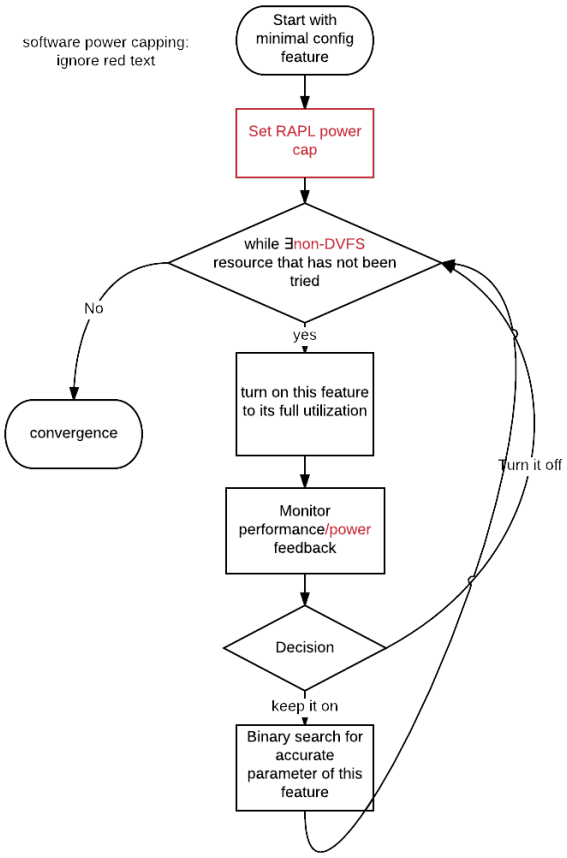


Figure 2.2: *PUPiL*'s approach to hybrid hardware/software power capping.

### *2.3.3* 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.

#### 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 software much search.

Figure 2.2 illustrates *PUPiL*'s hybrid decision framework. The major difference between the software-only approach and Figure 2.2'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.

## 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.

## 2.4 Experimental Setup

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

### *2.4.1 Benchmarks*

We use 20 benchmark applications from three different suites including PARSEC (`x264`, `swaptions`, `vips`, `fluidanimate`, `blackscholes`, `bodytrack`) [7], Minebench (`ScalParC`, `kmeans`, `HOP`, `PLSA`, `svmfe`, `btree`, `kmeans_fuzzy`) [71], and Rodinia (`cfd`, `nn`, `lud`, `particle-filter`)[11]. We also use a partial differential equation solver (`jacobi`) and the `swish++` search web-server [41] and `dijkstra` [47]. 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.

### *2.4.2 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 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

Table 2.2: 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 |

tradeoffs[1]. 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 Chapter 2.3, implementing the software decision system requires ordering the set of resources under consideration. Table 2.2 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.

### *2.4.3 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 Turbo-Boost)

### Timeliness

Settling time is a standard metric for a control system [35]. 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:

$$settle \quad = \quad t_{ss} - t_0 \tag{2.5}$$

### 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 [27].

## *2.4.4 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 [54].
- **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 Chapter 2.3.1.
- **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.

## 2.5 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.

### *2.5.1 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.

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

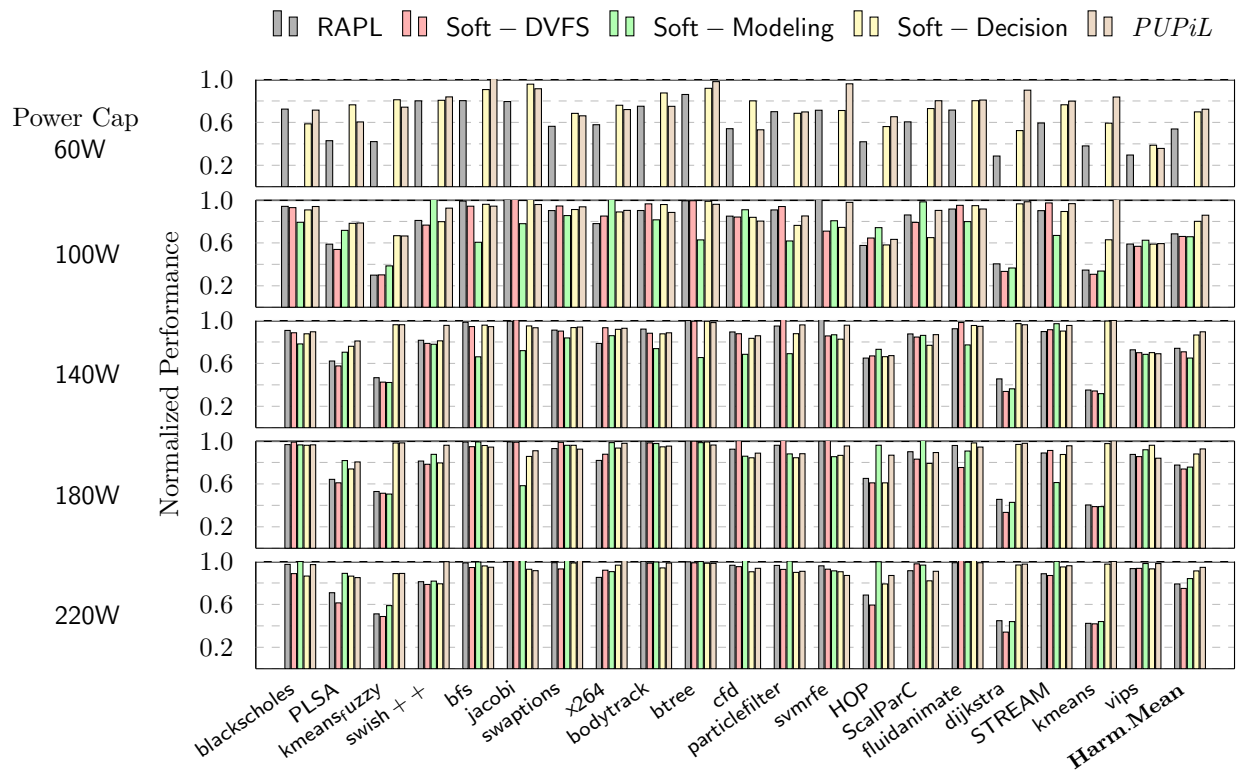


Figure 2.3: Performance of several power control techniques normalized to optimal.

constantly corrects.

### *2.5.2 Performance*

Figure 2.3 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 2.3. This table shows *PUPiL* consistently outperforms RAPL and

Table 2.3: 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 |

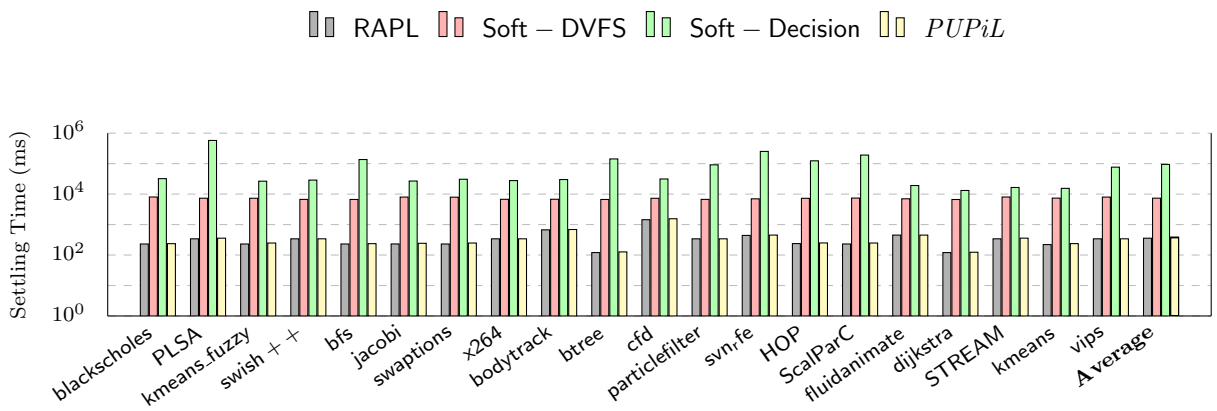


Figure 2.4: Settling times for several power control techniques.

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`). Figure 2.5 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 efficiency (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.

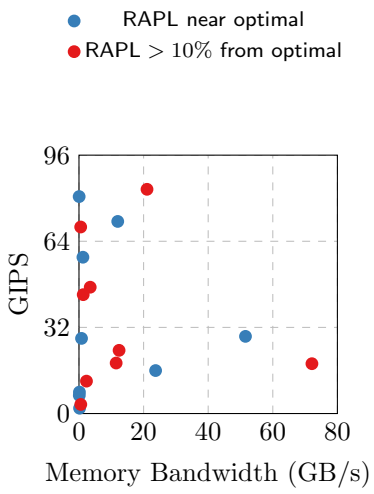


Figure 2.5: Benchmark characteristics.

### *2.5.3 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. Figure 2.4 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).

Table 2.4: 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 |

The data in Figure 2.4 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.

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.

### *2.5.4 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 Figure 2.5), and ones for which RAPL is more than 10% from optimal (red dots in Figure 2.5). 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 2.4 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 Chapter 2.4.3.

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.

#### Cooperative Performance

The performance for the cooperative multi-application scenario is shown in the left column of Figure 2.6. 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).

The performance comparison for the cooperative scenario reveals similar trends to the

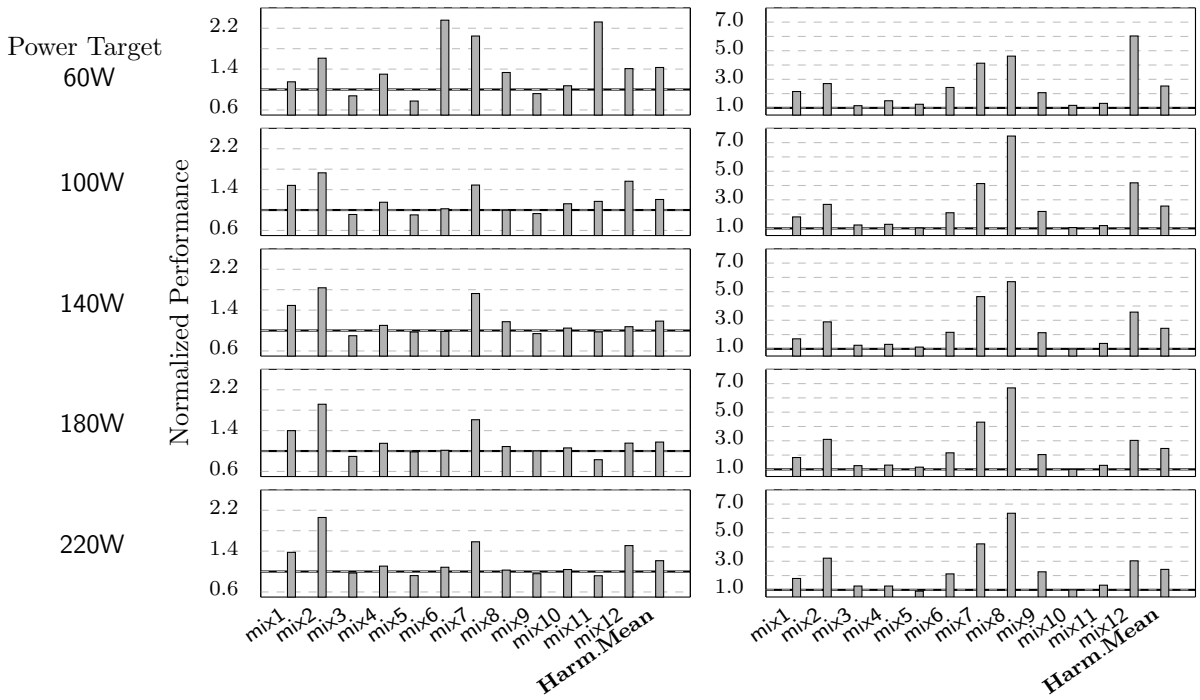


Figure 2.6: Ratio of *PUPiL* to RAPL performance in cooperative (left) and oblivious (right) multiapp scenarios.

Table 2.5: 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 |

single-application scenarios. There are several mixes for which *PUPiL* and RAPL achieve similar performance and others where *PUPiL* far outperforms RAPL. Table 2.5 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.

## Oblivious Performance

Figure 2.6'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 2.5, which indicates that *PUPiL* achieves at least 2.4× better aggregate performance than RAPL. Furthermore, this advantage can jump up to as much as 6× 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.

## 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

Table 2.6: *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 |

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 2.6 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 2.6 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.

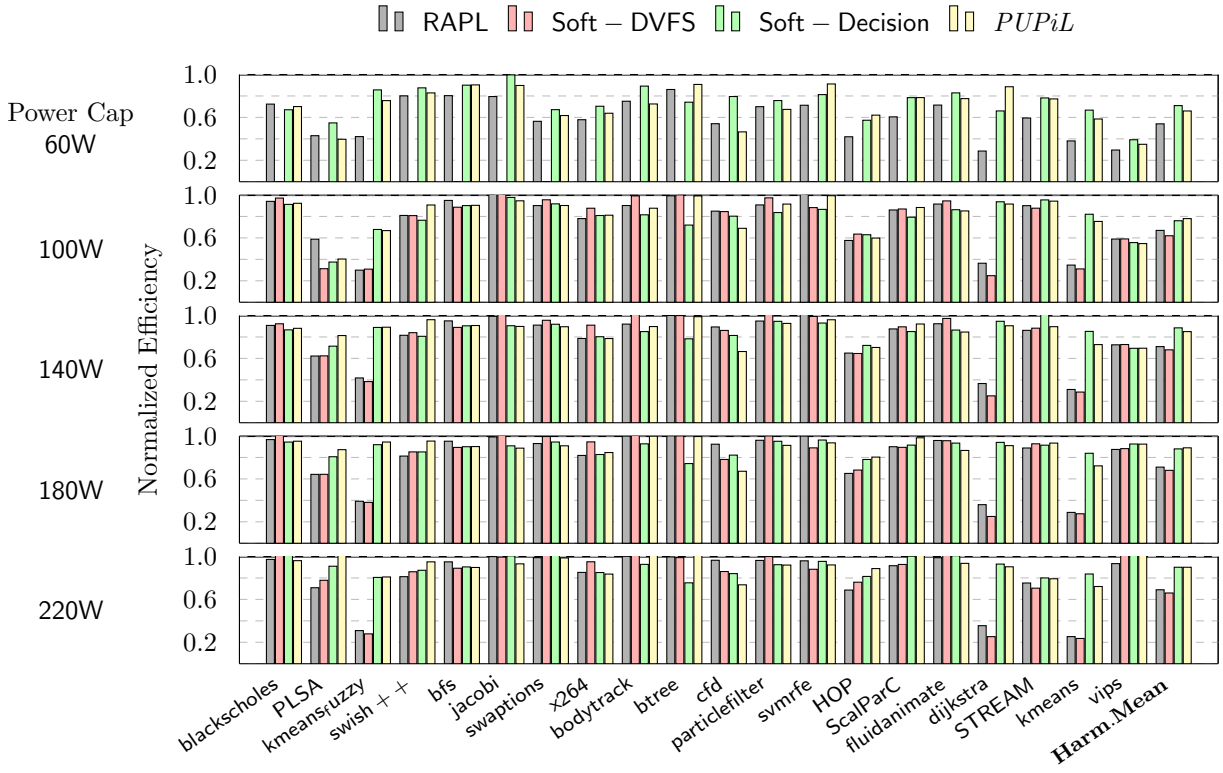


Figure 2.7: Energy efficiency of several power control techniques normalized to optimal.

### *2.5.5 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 Figure 2.7. As before, we normalize efficiency of all approaches to the optimal. Soft-decision and *PUPiL* produce 1.15-1.3× energy efficiency compared to RAPL or Soft-DVFS. Figure 2.8 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.

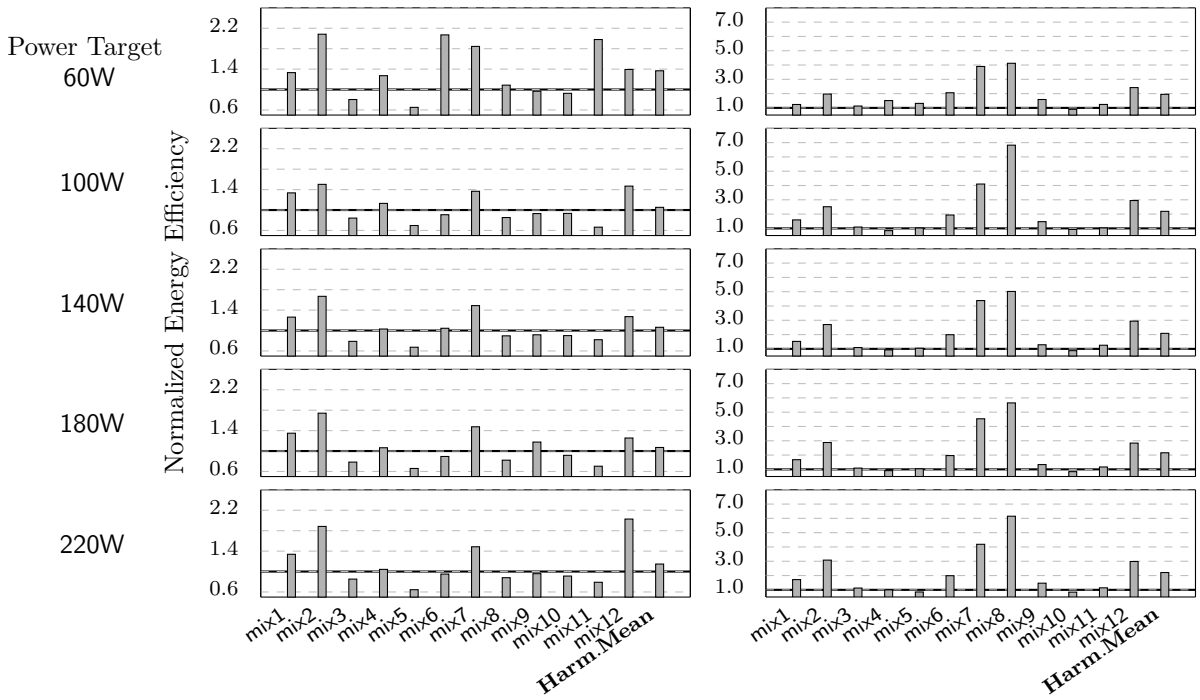


Figure 2.8: Ratio of *PUPiL* to RAPL energy efficiency in cooperative (left) and oblivious (right) multiapp scenarios.

### *2.5.6 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 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 Figure 2.4. 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 Figure 2.3. This figure shows that the performance impact of the *PUPiL* runtime system is acceptable in that *PUPiL* produces the closest to optimal performance.

## 2.6 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× 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.

# CHAPTER 3

# *POWERSHIFT*: PERFORMANCE & ENERGY TRADEOFFS FOR DEPENDENT DISTRIBUTED APPLICATIONS UNDER SYSTEM-WIDE POWER CAPS

This chapter presents *PowerShift*, a family of three techniques for shifting power between dependent applications in a distributed system.

Large scale parallel machines are subject to system-wide power budgets (or caps). As these machines grow in capacity, they can concurrently execute dependent applications that were previously processed serially. Such application *coupling* saves IO and time as the applications now communicate at runtime instead of through disk. Such coupled applications are predicted to be a major workload for future *exascale* supercomputers; *e.g.,* scientific simulations will execute concurrently with *in situ* analysis. While support for system-wide power caps has been widely studied, prior work does not consider the impact on coupled applications.

We study techniques for maximizing coupled application performance under a system-wide power cap and implement them on a 26-node cluster. We compare to SLURM, a state-of-the-art job scheduler that considers power, but not coupling. The proposed techniques increase mean performance over SLURM by 7–14%. Unlike existing approaches, the proposed techniques also recognize when it is not possible to increase performance and, instead, reduce energy, achieving 18% energy reduction for a 5% performance loss. Finally, the dynamic techniques are resilient to tail behavior and system noise, improving performance in noisy environments by 30–36%.

The rest of this chapter are organized as follows: First, Chapter 3.1 discusses the related works in this domain. Next, Chapter 3.2 introduces motivational examples. Then, Chapter 3.3 demonstrate the 3 system framework and algorithms. After that, Chapter 3.4

presents the experimental setup, results, and discuss its reason and insight. Finally, Chapter 2.6 concludes this work.

## 3.1 Related Work

Large-scalde computings systems are limited by power consumption. Power and energy management is extremely important for data centers [18, 28, 36, 69, 100] and it has long been important for mobile and embedded systems [30, 31, 37, 44, 55, 68, 81, 90, 107].

For HPC and data center power management, hooks exist at multiple levels; *e.g.,* the processor [16], load balancer [10], enclosure [78, 100], and the installation itself [28]. Raghavendra et al. present a method for ensuring that power management systems at multiple levels coordinate to stabilize at the desired power [76].

Apart from simply respecting a power cap, though, it is important to deliver performance. Several approaches examine the problem of maximizing performance subject to a power cap. Again, this work can be done at the processor-level [39, 43, 79, 109], DRAM [16], storage [52], and across a data-center [61, 72]. At the data-center or cluster-level, power can be saved by consolidating workloads to use fewer physical machines [19, 49, 62, 66, 67, 105], coordinating co-existing applications [80], and scheduling with green power [34] Other approaches shift power from different components within the data center (*e.g.,* from processor to memory) to ensure that those components that provide the biggest speedup get sufficient power [63]. These approaches do not consider dependent jobs, but simply work to maximize independent applications' throughput.

Scheduling jobs under a power cap has recently become a major concern for HPC operating systems [9, 24] and job schedulers [4, 33]. Recent work suggests that HPC workloads can actually achieve higher performance by over-provisioning large-scale installations—such that using all nodes at full capacity would drastically violate the power budget—and severely power capping the individual nodes [82].

Next generation *exascale* supercomputers are predicted to have a strict operating budget of approximately 20 MW, but the total power dissipation at full utilization would far exceed this budget [6]. These systems require sophisticated, distributed *power-capping* mechanisms to assure their power budget will not be exceeded; the United States Department of Energy (DoE) has therefore declared power management a key challenge for exascale [95].

While power concerns create new problems, exascale's increased capacity creates new opportunities. Specifically, instead of sequentially running dependent jobs that communicate through disk, the size of exascale supercomputers allows these jobs to be *coupled* [2, 6, 12, 51, 95]. That is, two formerly independent jobs can now be run simultaneously and communicate at runtime. For example, scientific simulations can now be run with *in situ* data analysis or visualization providing scientists the insight needed to alter the simulation as it runs [2, 12]. Additionally, separately developed physics simulations can now be run together, sharing their results to provide much greater fidelity [51]. In fact, the DoE has declared resource management for coupled application workloads an additional challenge for exascale [6, 95]. Unfortunately, existing work on enforcing power caps across large scale systems does not account for coupled applications.

Closest to *PowerShift* is a collection of prior work that shifts power unevenly among jobs in HPC systems. *Power Routing* shifts power to maintain throughput and minimize power infrastructure; it does not support dependent applications [75]. SLURM will shift power from nodes operating below their budget to those at the budget [89]. Another approach is aware of application-level semantics and will shift power from processes executing IO (and thus not using their full power allocation) to those doing computation (i.e., at their power limit) [84]. Inadomi et al. propose variation-aware power shifting that shifts power within a single parallel job to compensate for manufacturing variations [45].

While these approaches and *PowerShift* all move power from one process to another, *PowerShift* is unique in its ability to speedup dependent, coupled applications. *The key*

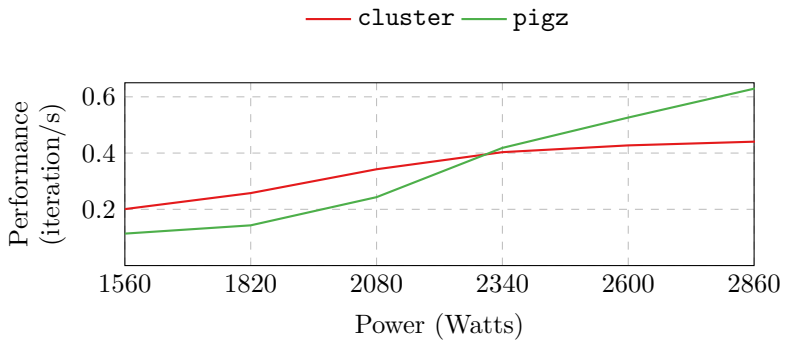


Figure 3.1: Performance/power for `cluster` and `pigz`.

*difference is that PowerShift will move power from one process operating near its cap to another that is also near its cap.* This ability is essential for speeding up coupled applications whose performance is dependent on the slower application. To the best of our knowledge no other power shifting system will shift from one high-power process to another to speed up dependent applications.

## 3.2 Motivational Example

To highlight the challenges of maximizing coupled jobs' performance under a power cap we run the `cluster` scientific simulation (from the Gadget 2.0 suite) as a *frontend* job coupled with `pigz`, a parallel version of gzip, as a *backend* job. We compare 3 approaches: *Fair*, *PowerShift-S*, *PowerShift-C*. The *Fair* approach allocates equal power to all nodes so that the cluster power cap is respected.

We test on a 26-node cluster. Each node is a dual-socket with Xeon E5-2670 v3 processors, 256GB of RAM and 10 GigE Ethernet NIC, supporting Intel RAPL technology [16]. Each processor is 12 cores, with hyperthreading, for a total of 48 virtual cores across both sockets. These nodes are connected with a 32-port software-defined 40 GigE switch.

We set a 1820W cluster power budget. The offline profiles shown in Figure 3.1 illustrate the different performance and power tradeoffs for `cluster` and `pigz`. There is one power cap (when the lines cross) where the front and backend performance is equal, and this case

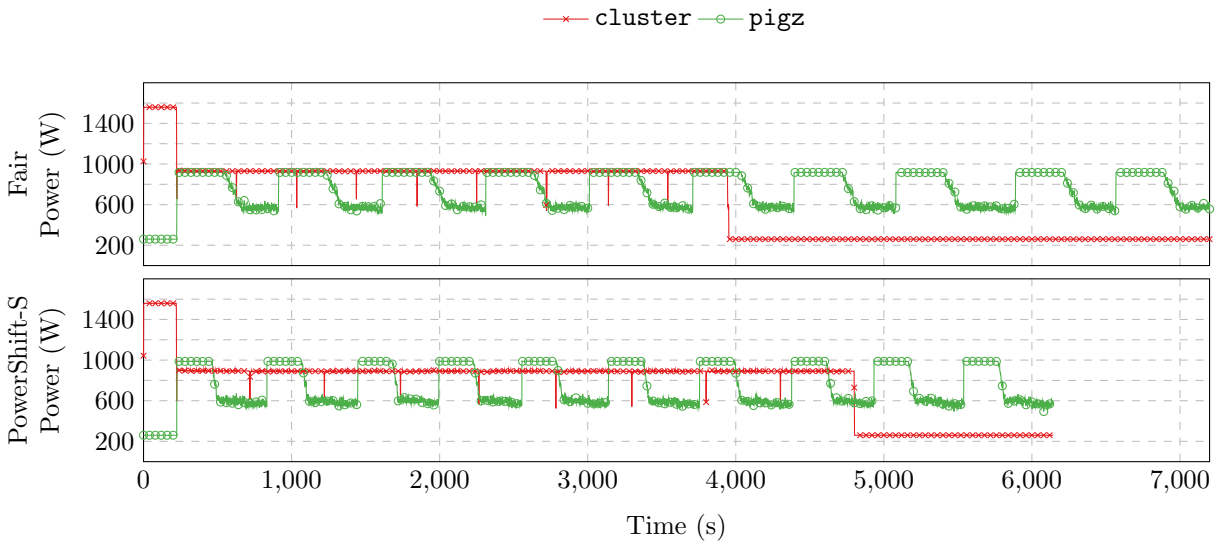


Figure 3.2: *Fair* (top) vs. *PowerShift-S* (bottom).

would require no work. To the left of that cap, we can increase performance by shifting power from `cluster` to `pigz`. To the right of that cap, we can decrease energy by shifting power away from `pigz`, as `cluster` has reached the point of diminishing returns.

`cluster` iteratively produces output. To reduce storage space, `pigz` compresses each output. To produce the charts, we report performance as *iteration/s*. *PowerShift* itself does not measure performance. As shown, when the global power cap is 1820W `cluster` reaches 0.258 iteration/s, whereas `pigz` achieves only 0.143 iteration/s; *i.e.,* `pigz` is about half `cluster`'s speed.

### *3.2.1 Static Power Shifting*

Figure 3.2 compares *Fair* and *PowerShift-S*, highlighting the importance of shifting power from the fast application to the slow one so that they run at the same speed. The top figure shows *Fair* and the bottom shows *PowerShift-S*. We run the coupled workload for 10 iterations, the backend beginning after the first frontend iteration finishes. *Fair* divides the power budget equally among all nodes. At the beginning of the couple's execution, *PowerShift-S* uses the individual profiles to determine an unbalanced power distribution

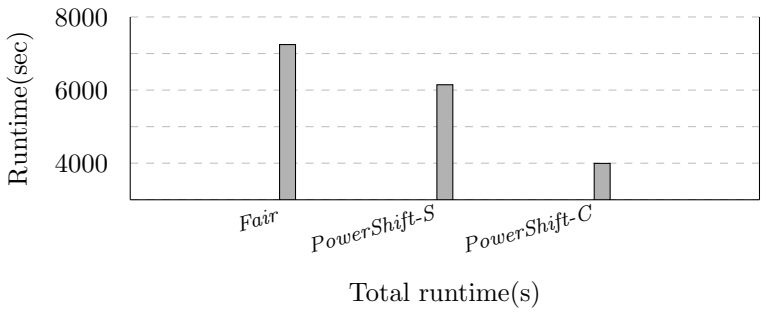


Figure 3.3: Runtimes for different approaches.

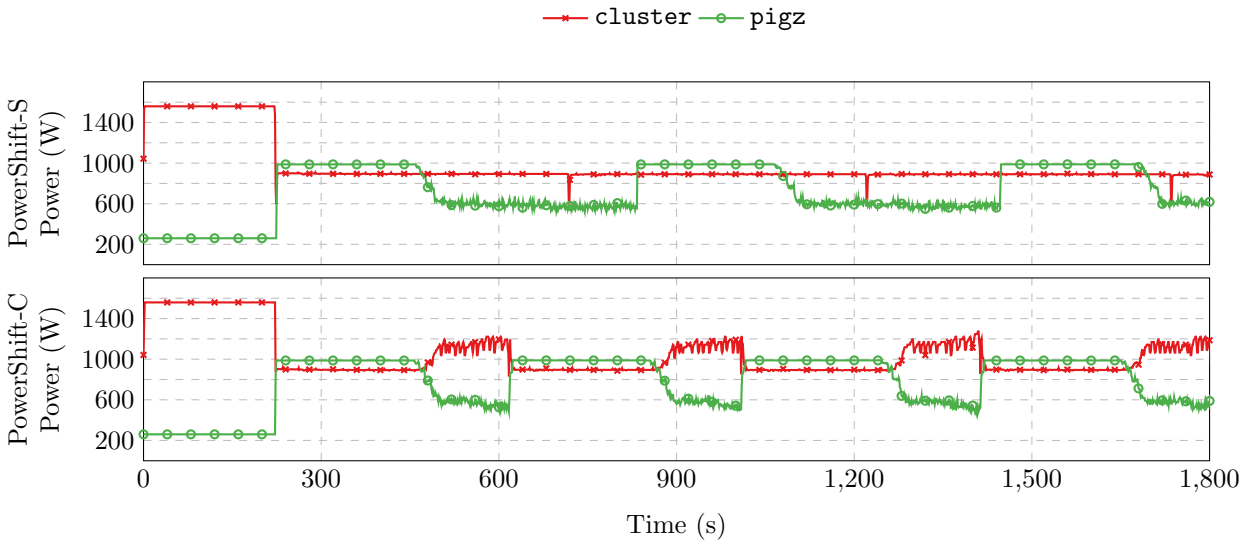


Figure 3.4: *PowerShift-S* (top) vs. *PowerShift-C* (bottom).

such that the jobs finish close together.

With *Fair*, `cluster` is much faster than `pigz`, requiring 4000 seconds. The couple, however, only finishes when `pigz` is done—at around 7000s. *PowerShift-S* allocates `pigz` more power to balance performance, resulting in around 6000s elapsed time, as shown in Figure 3.3. Here, `pigz` still runs fairly slower than `cluster`, because there is a physical limit—imposed by the hardware—on the power we can shift from `cluster` to `pigz`.

### 3.2.2 Dynamically Shifting Unused Power

Figure 3.3 shows that *PowerShift-S* is much faster than *Fair*, but *PowerShift-C* is faster still. To illuminate this performance gain, Figure 3.4 compares *PowerShift-S* and *PowerShift-C*. For visibility, this figure shows just the first 1800 seconds of execution, highlighting the importance of dynamic power shifting to take advantage of application phases. The upper portion of Figure 3.4 highlights the two distinct phases within one `pigz` iteration: (1) a power-hungry phase that compresses data and (2) a low-power phase that waits for more data. *PowerShift-S* does not make runtime adjustment and cannot take advantage of `pigz`'s low-power phase. *PowerShift-C*, however, detects unutilized power and shifts it to `cluster` for a dramatic speedup. Under *PowerShift-C*, `pigz`'s second phase also finishes much faster and the reason will be introduced in next section. Therefore, *PowerShift-C* completes the couple much faster (in about 4000s, see Figure 3.3).

This figure exemplifies power shifting that is not supported by current solutions. When `pigz` enters its low-power phase, power can be shifted away from those nodes without performance loss. Current schedulers can recognize this. The issue is that when `pigz` transitions back to a power-hungry phase, *PowerShift* has to reallocate power from `cluster` to `pigz` despite the fact that both are operating at or near their assigned budgets. *Power shifting between nodes operating at their caps is a unique challenge of supporting coupled application performance, not supported by prior work.*

### 3.2.3 Dynamically Shifting Power to Tail Nodes

Dynamic power monitoring also allows shifting power to tail nodes. In coupled applications, tail nodes ruin the performance of not just their own application, but the entire couple. Figure 3.5 shows a histogram of completion times for one `pigz` iteration under both *PowerShift-S* and *PowerShift-C*. Static power distribution has a longer tail, with one node completing in the 600-700s range while the maximum completion time for dynamic scheduling is 500s.

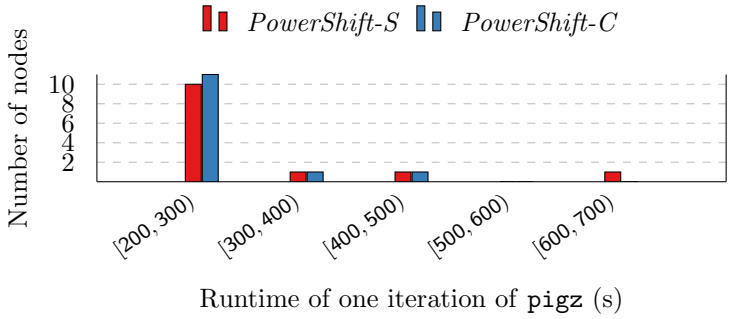


Figure 3.5: Tail distribution under static and dynamic power shifting. The static approach has a long tail, but the dynamic approach has a much shorter tail.

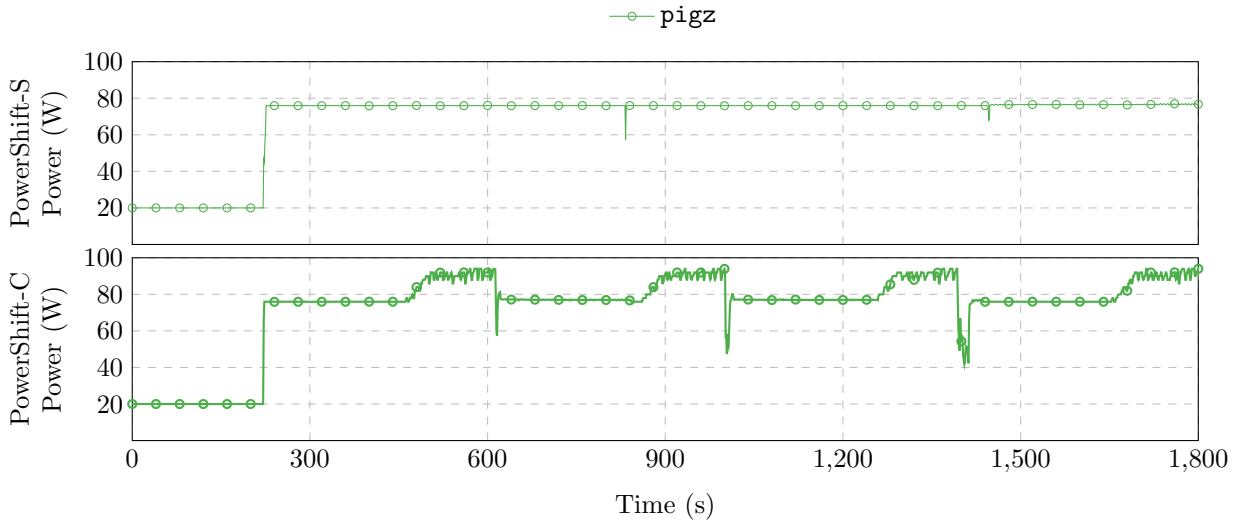


Figure 3.6: Tail node power time series comparing PowerShift-S (top), to PowerShift-C (bottom).

Thus, dynamic power shifting in coupled applications both shifts power from one application to another and to tail nodes in either application. Figure 3.6 shows the time series of one tail node in the `pigz` application. As noted, `pigz` has two phases. In the second, most of the backend nodes finish and wait for a few tail nodes. The tail nodes under *PowerShift-S* continue running at the same speed and slow down the entire couple. *PowerShift-C* detects that a few nodes do not go to idle in this phase, so it shifts power from the idle nodes to both the frontend application and the tail nodes. As the bottom figure shows, the tail node gets more power during the second phase and therefore finishes much faster.

## 3.3 Power Management Approaches

We discuss the 5 distributed power capping approaches we evaluate. We briefly review *Fair* and SLURM [89]. We then describe our couple-specific approaches: a static power manager using an offline power profile and two dynamic managers, one using a centralized decision mechanism, the other making distributed decisions.

All approaches assume there is a node-level power capping mechanism. Many exist in the literature [54, 79, 109] and Intel RAPL (Runtime Average Power Limiting) is a commercial example available on all current Intel processors [16]. The *Fair* and the *Static* approach do not involve any runtime operation, while other approaches (SLURM, centralized, and distributed) make runtime power measurements. The dynamic approaches follow a standard three step feedback process: (1) observe the current power, (2) decide on a response, and (3) act to implement the decisions.

### *3.3.1 Fair Power Allocation and SLURM*

In the *Fair* approach, the system has no prior knowledge of the applications running on top of it. Therefore, it allocates power evenly to all nodes. Furthermore, at runtime, *Fair* has no mechanism to make changes adaptively. SLURM is an open-source job scheduler used by many supercomputers and clusters. SLURM provides an integrated system for power capping. SLURM starts with the same initial power distribution as *Fair*, then monitors actual power consumption, and redistributes power from nodes operating below their fair budget to those that are at their budget. This heuristic improves the throughput of independent applications, but is often sub-optimal for coupled applications as the couple's performance is defined by the slowest application.

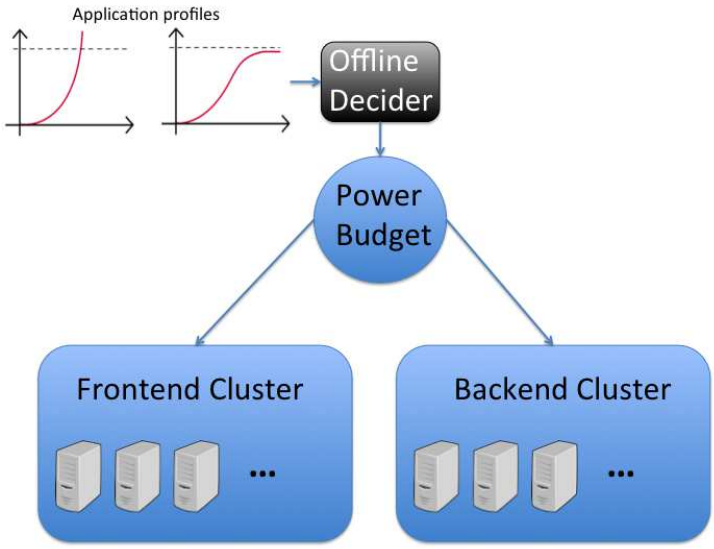


Figure 3.7: *PowerShift-S* overview.

### *3.3.2* PowerShift-S*: Static Power Allocation*

*PowerShift-S*, the static approach, takes advantage of offline application profiles based on two key concepts:

1. The couple's performance is determined by the slower of the frontend and backend applications.
2. All applications have monotonically non-decreasing power/performance tradeoffs; *i.e.,* performance may level off with increasing power, but will never get worse.

We assume that we have profiled each application in isolation in the cluster to obtain power performance profiles, but we have never seen the coupled applications together. The profile only requires performance and power information. Scalability is not an issue, as the profile just captures performance as a function of node-level power cap, which does not change with the number of nodes. For this paper, we sample performance at 15 evenly spaced power caps, 3 evenly spaced input sizes, and apply quadratic regression to build the model.

Figure 3.7 overviews the high level design. Given the individual application profiles and a total system power budget, *PowerShift-S* maximizes performance by choosing the optimal

**Algorithm 3** *PowerShift-S* Decision Algorithm

**Require:** Power budget $P$, Max shiftable power $P_m$
**Require:** Accuracy threshold $A$
**Require:** Performance improvement threshold $T$
**Require:** $S_f = \{(power_i, perf_i) | i = 1, 2, ..., n\}$ for frontend app
**Require:** $S_b = \{(power_i, perf_i) | i = 1, 2, ..., n\}$ for backend app
  Fit $S_f$,$S_b$ into functions $F_f(power) = perf$, $F_b(power) = perf$
  $pow_f = pow_b = P/2$
  $perf_f = F_f(pow_f)$, $perf_b = F_b(pow_b)$
  $\delta = perf_f - perf_b$ ▷ assume $perf_f >= perf_b$
  **if** $F_f(pow_f - pow_s) >= F_b(pow_b + pow_s)$ **then**
    $pow_f = pow_f - pow_s$, $pow_b = pow_b + pow_s$
  **else**
    $pow_f = pow_f - pow_s$, $pow_b = pow_b + pow_s$
    **while** $|\delta| > A$ **do** ▷ binary search for optimal power
      $pow_s = pow_s/2$
      **if** $F_f(pow_f) > F_b(pow_b)$ **then**
        $pow_f = pow_f - pow_s$, $pow_b = pow_b + pow_s$
      **else**
        $pow_f = pow_f + pow_s$, $pow_b = pow_b - pow_s$
        $\delta = F_f(pow_f) - F_b(pow_b)$
      **end if**
    **end while**
  **end if**
  Power distribution: $\langle pow_f, pow_b \rangle$
  Final performance: $perf_{final} = min(F_f(pow_f), F_b(pow_b))$
  Fair performance $perf_{fair} = min(F_f(P/2), F_b(P/2))$
  **if** $perf_{final} > (1 + T) * perf_{fair}$ **then**
    ▷ decide to enter performance or energy mode
    Return $(pow_f, pow_b)$ ▷ Enter performance mode
  **else**
    search for $S_f$,$S_b$, find the highest energy efficiency power state, $pow_f$ and $pow_b$, that satisfy $F_f(pow_f) >= (1 - T) * perf_{fair}$ and $F_b(pow_b) >= (1 - T) * perf_{fair}$
    ▷ Enter energy mode
  **end if**

static power distribution between the frontend and backend applications. In some cases, *PowerShift-S* recognizes that there is no additional performance to be gained, so it enters an *energy savings mode* that maintains a user-defined performance, while reducing energy. Algorithm 3 shows the basic logic of *PowerShift-S*. It requires the power budget $P$ from the user and the maximum power that can be shifted from one application to the other based on the system $P_m$. It also requires two parameters: (1) the power accuracy $A$—the tolerance for error as application performance can usually not be perfectly matched—and (2) the performance improvement threshold $T$—the minimum performance improvement over the

*Fair* approach required to avoid energy savings mode.

The algorithm first does a binary search for the highest performance power distribution between the front and back end applications. The power distribution returned is $\langle pow_f, pow_b \rangle$, which are distributed evenly among the front and backend nodes. The performance achieved here is called $perf_{final}$, the performance under *Fair* is called $perf_{fair}$. When $perf_{final} > (1+T) * perf_{fair}$—*i.e.,* there is acceptable performance gain—it enters performance mode and returns power distribution $\langle pow_f, pow_b \rangle$. Otherwise, it enters energy mode, where it searches all profile data points for each application to find the highest energy efficiency such that $F_f(pow_f) >= (1-T) * perf_{fair}$ and $F_b(pow_b) >= (1-T) * perf_{fair}$. It returns the most energy efficient power state within at least $(1-T)$ of max performance for both applications. In our implementation, we choose $A = 0.5$, because such a small difference has little effect on performance. We choose performance improvement threshold $T = 3\%$. The larger $T$ is, the more *PowerShift-S* prefers to optimize for energy, and the smaller $T$ is, the more it prefers to optimize for performance.

A simple proof of the optimality is as follows: By concept 2, above, both applications have monotonically non-decreasing performance as a function of power. Then, at the optimal state, if we reallocate power from one end to the other, one application slows down, and the other speeds up. But by concept 1, performance is determined by the time when both applications finish; *i.e.,* the slower application is the performance limiter. Therefore, any power shifting from the optimal state will increase the overall runtime.

### *3.3.3 Dynamic Power Shifting*

We propose 2 dynamic power management approaches: *PowerShift-C* and *PowerShift-D*. Critically for coupled applications, both approaches can take power from high-performance nodes to shift to slower nodes with the result of speeding up the entire couple. *PowerShift-C* has a single centralized decision maker. In *PowerShift-D* decisions are distributed and

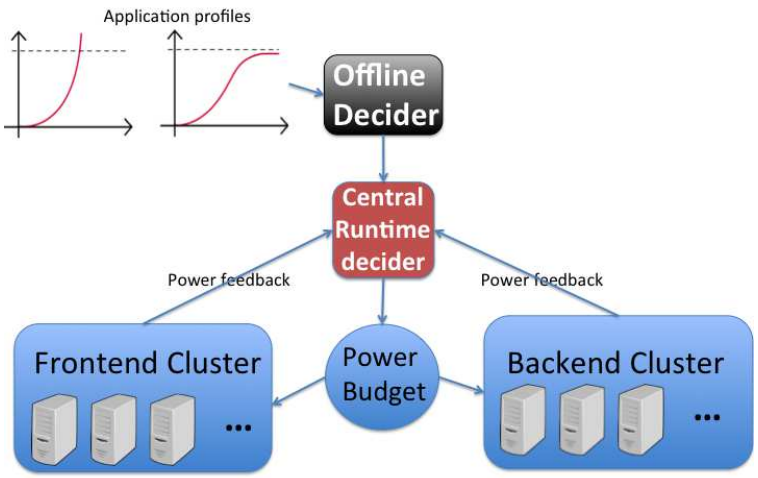


Figure 3.8: *PowerShift-C* overview.

each node has its own local decision maker. Both approaches take advantage of the offline profiles and their initial power distribution state is the same as *PowerShift-S*. In energy mode, runtime power shift is disabled, because any power state change will cause the coupled applications to move away from the most energy-efficient power state.

## **PowerShift-C**: Centralized Power Shifting

Figure 3.8 overviews the system design. *PowerShift-C* starts with the optimal static power from *PowerShift-S* and operates on a typical control loop: observing its environment, deciding on a response, and acting to implement its decisions. During observation, it collects each nodes power usage. To decide a new power distribution, it first groups all nodes into 3 power priority groups (described below) and then it shifts power from those nodes that either (1) have lower priority or (2) simply have unused power to the nodes that are hungry for power with a higher priority. Finally, it acts by sending new power capping information to each node.

Algorithm 4 details *PowerShift-C*. The algorithm requires a *power margin* $M$; any node

**Algorithm 4** *PowerShift-C* decision algorithm.

**Require:** Node number $N$, power margin $M$
**Require:** Optimal static power budget $B = \{b_i | i = 1, 2, ...N\}$
  **while** application couples not finished **do**
    $\{pow_i | i = 1, 2, ...N\} \leftarrow GetAveragePower()$
    Power budget of last iteration $\{bl_i | i = 1, 2, ...N\}$ ▷ the average power of each node since last iteration
    Group all nodes into 3 priority groups
    Group 1 $\leftarrow \{Node_i | pow_i < b_i - M\}$
                                                ▷ Nodes with extra power
    Group 2 $\leftarrow \{Node_i | pow_i >= b_i - M, bl_i >= b_i\}$
                                                ▷ Nodes at power limit
    Group 3 $\leftarrow \{Node_i | pow_i >= b_i - M, bl_i < b_i\}$
                                                ▷ Nodes at power limit needing more power
    **if** Group 1's extra power > Power needed by Group 3 **then**
      Shift all needed power from Group 1 to Group 3
      Any extra power from Group 1 goes to Group 2
    **else**
      Shift power exceeding B from Groups 1 & 2 to Group 3
      Any extra power is divided between Group 2 and 3
      Sleep(t)                                  ▷ make power shifting every t time
    **end if**
  **end while**

that consumes power within $M$ of its local power cap is defined to be *intensively* using power. Otherwise, the node can reduce power. The central decider first groups each node into 1 of 3 priority classes by its distance to its own power cap and whether its current power cap is higher than the optimal static power cap. Higher group numbers mean the node needs power more urgently. Group 1 has power to spare. Group 2 is operating at or near their power cap, but will not increase the couple's performance with extra power. Group 3 is operating at or near the power cap and increased power will increase the couple's performance. *A key difference between PowerShift and prior work, is that PowerShift will move power from Group 2—which has high power utilization—to Group 3—which also has high utilization. Existing schedulers, like SLURM, will only shift from low utilization nodes to high utilization nodes, making them unsuitable for coupled applications.*

After assigning nodes to priority groups, the algorithm checks the shiftable power of each group to make sure that on the next iteration, all nodes in Group 3 will at least get power equivalent to their optimal static power budget. If necessary, the algorithm will force Group

2 to reduce power in favor of Group 3; otherwise, extra power will go to Group 2. Finally, the decision engine idles for a time interval $t$. This idling both reduces overhead and allows the nodes to settle into their new power consumption.

Front and backend nodes get different amounts based on their performance derivative. $F()$ is the performance power function generated by offline data, $p_f$ is the power acquired by frontend node and $p_b$ is the power acquired by backend node.

$$\frac{\delta F_f(pow_f)}{\delta pow_f} : \frac{\delta F_b(pow_b)}{\delta pow_b} \quad = \quad p_b : p_f \tag{3.1}$$

This shifting power ratio ensures that all nodes in one end (front or back) get extra power proportional to the performance growth of the other end. Thus, *PowerShift* allocates less power to the end where performance scales faster and more power to the end that scales worse so that both ends work at the same rate.

For power margin $M$ and time interval $t$ in Algorithm 4, we use 2 Watts and 1 second. Choosing $M$ decides how aggressively we want to shift power. Lower $M$ results in higher aggressiveness, at the risk of possibly removing power from nodes that need it. Higher $M$ means lower aggressiveness, at the risk of not being able to fully utilize the unused power. For the time interval, larger intervals make the system reaction time longer and it may miss short duration power shifting opportunities. A shorter time interval exaggerates the system noise but provides more opportunities to react.

## **PowerShift-D**: Distributed Power Shifting

*PowerShift-D* distributes power shifting. Figure 3.9 overviews the system architecture. The key data structure is a *power pool* that contains unused power. In contrast to *PowerShift-C*, *PowerShift-D* does not have any centralized component. Instead, each node has its own local decision maker to shift power. All local power deciders talk to the power pool asynchronously

**Algorithm 5** *PowerShift-D* algorithm.

**Require:** Power margin $M$
**Require:** Optimal static power budget of itself $B$
  **while** not finished **do**
    Current power $p_c \leftarrow GetAveragePower()$
    Power budget of last iteration $b$
    Define power state of itself
    Group 1: $p_c < b - M$ ▷ have extra power
    Group 2: $p_c >= b - M$, $b > B$ ▷ at power limit
    Group 3: $p_c >= b - M$, $b < B$ ▷ at power limit, need power
    Read $p_{pool}$ and urgency flag $F$
    **if** Group 1 **then**
      Post extra power to power pool
    **else**
      **if** Group 3 **then**
        Grab unused power from power pool
        When $p_{pool} < B - b$, increment $F$
        After needed power is satisfied, decrement $F$
      **else**
        When $F = 0$, take from power pool
        When $F \neq 0$, post $b - B$ to power pool
      **end if**
    **end if**
    Sleep(t)
  **end while**

and make decisions based on their own power state and the power pool.

The power pool structure serves as a bulletin board to all the local deciders. It has a floating point variable $F$ to post how much power is in the pool and one integer variable $I$ to post how many nodes are in Group 3 (from above) if any. When $I = 0$, nodes Groups 2 & 3 take power from the power pool. Otherwise, only nodes in Group 3 can take power from the pool, and nodes in Group 1 or 2 have to give up power exceeding their initial power budget if any. Group 3 nodes will always get the power they need. If there is not enough power in the pool, Group 2 nodes will have to release power (even if they are at their budget).

We maintain the invariant that any node adding power to the pool must first reduce its local power usage and any node taking power must decrement unused power in the pool before raising its local power usage. This simple invariant ensures the global power budget is respected with local decision making. Our implementation uses sockets to communicate between local nodes and the power pool.

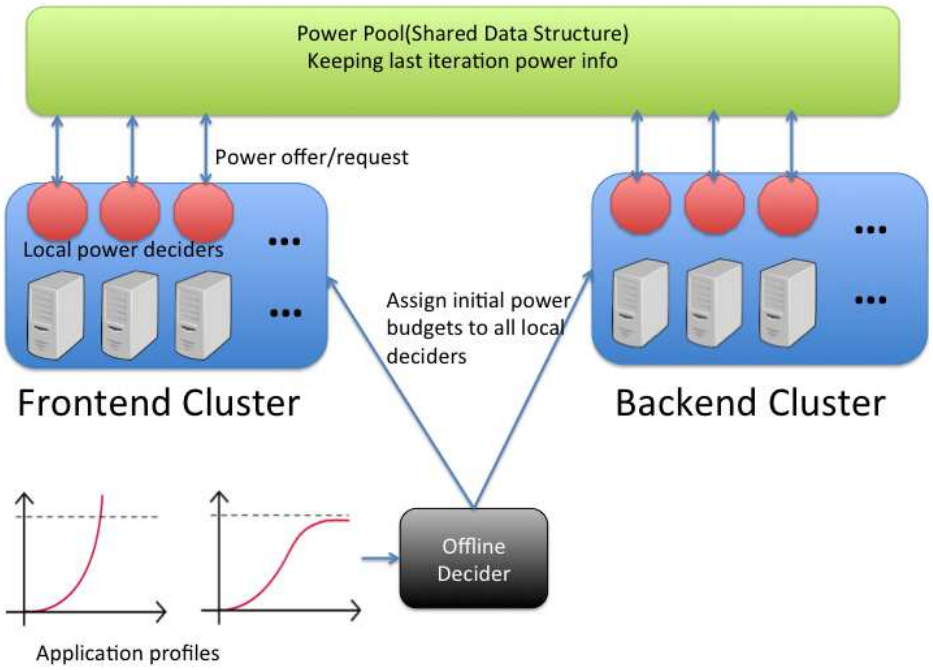


Figure 3.9: *PowerShift-D* overview.

Algorithm 5 shows the local power decision systems. The rule to choose the power margin $M$ is same as in *PowerShift-C*. The time interval $t$ is different here. Since every local decider can have different values, we use 2 seconds for nodes in Groups 1 and 2, and 1 second for nodes in Group 3, to allow the node to get out of an urgent state faster. *PowerShift-D* can easily integrate with local power managers by coordinating its interaction with the pool with the decisions of the local power manager. Other approaches are not compatible with independent, local power managers.

### *3.3.4 Extension Beyond Two Applications*

A simple modification version of *PowerShift-S* can work on any arbitrary number of dependent applications. In each loop iteration, we shift power from the fastest application to the slowest, with the goal that all applications run at the same speed. In this way, the binary search for the optimal performance extends across all applications and get the power

distribution of each.

*PowerShift-C* and *-D* are also trivially extended to any number of dependent applications: the global or local decider follows the same algorithm regardless of the number of applications. This scalability to multiple applications for the dynamic cases is easy because the algorithms do not reason about front and backend applications. Instead they reason about priority levels which are set based entirely on a node's behavior relative to its assigned power cap. This flexibility does rely on the stated assumption that performance is monotonically non-decreasing with increasing power consumption.

### *3.3.5 Complexity, Guarantees, and Discussion*

**Algorithmic Complexity**

1. *Fair* is $O(1)$.
2. *PowerShift-S* is $O(n * \log P_m)$, given $n$ profile data points and $P_m$ maximum shiftable power. It uses binary search to find the optimal power distribution.
3. *PowerShift-C*: Executes the *PowerShift-S* algorithm once at launch. At each iteration, it executes Algorithm 4, which is $O(N)$, where $N$ is the number of nodes.
4. *PowerShift-D*: Executes the *PowerShift-S* algorithm once at launch. The communication with the power pool is $O(1)$.

The complexity of *PowerShift-S* grows with larger $n$ (more profile data points). We do not consider this a scaling problem, because only a small $n$ is needed to have a good picture of power and performance tradeoffs for each application. In our implementation, we use $n = 15$. Additionally, this algorithm is only executed once before launch. *PowerShift-C*'s complexity, however, grows with $N$, the number of nodes. In a larger cluster, $N$ might be thousands or tens of thousands. The centralized approach may suffer from overload. In that case, we have to provide more computing power and hardware to the central decider or have a multi-tier central decider to solve the scaling problem. This linear scaling does not occur

with *PowerShift-D*, which essentially parallelizes the power distribution process, leading to an approach that naturally scales with the number of nodes.

**Guarantees**

1. *Fair*: equal power distribution between frontend and backend; equal power distribution between each node; and assurance that the global power budget is met.
2. *PowerShift-S*: the maximum performance static power distribution with at least $T$ speedup over *Fair*. When $T$ speedup is not achievable, the most energy efficient power distribution within $(1 - T)$ of *Fair*'s performance.
3. *PowerShift-C* and *PowerShift-D*, both provide speedup when the applications have unbalanced power needs over time or across different nodes. In energy-saving mode, these approaches have the same guarantees as *PowerShift-S*. Due to the simple invariants that nodes first lower their power before contributing to the global power budget and first take from the global power budget before raising their power, these approaches also ensure that the global power cap is respected.

**Topology-obliviousness** These guarantees are *topology-oblivious*. While we have assumed—for simplicity of discussion—that the front and backend applications are on physically separate nodes, they can actually be scheduled in any manner such that their power control is independent. For example, front and backend processes could be scheduled on separate sockets in a node, or separate cores in a chip, if the processor supports independent core-level power caps. Our empirical analysis shows that performance is the same under different mappings (see Chapter 3.4.4).

## 3.4 Experimental Evaluation

This section evaluates the performance efficiency of *PowerShift*. It also examines the energy efficiency when the offline decider chooses to enter energy-saving mode and how the dynamic versions of *PowerShift* can improve performance in the presence of tail-nodes and system

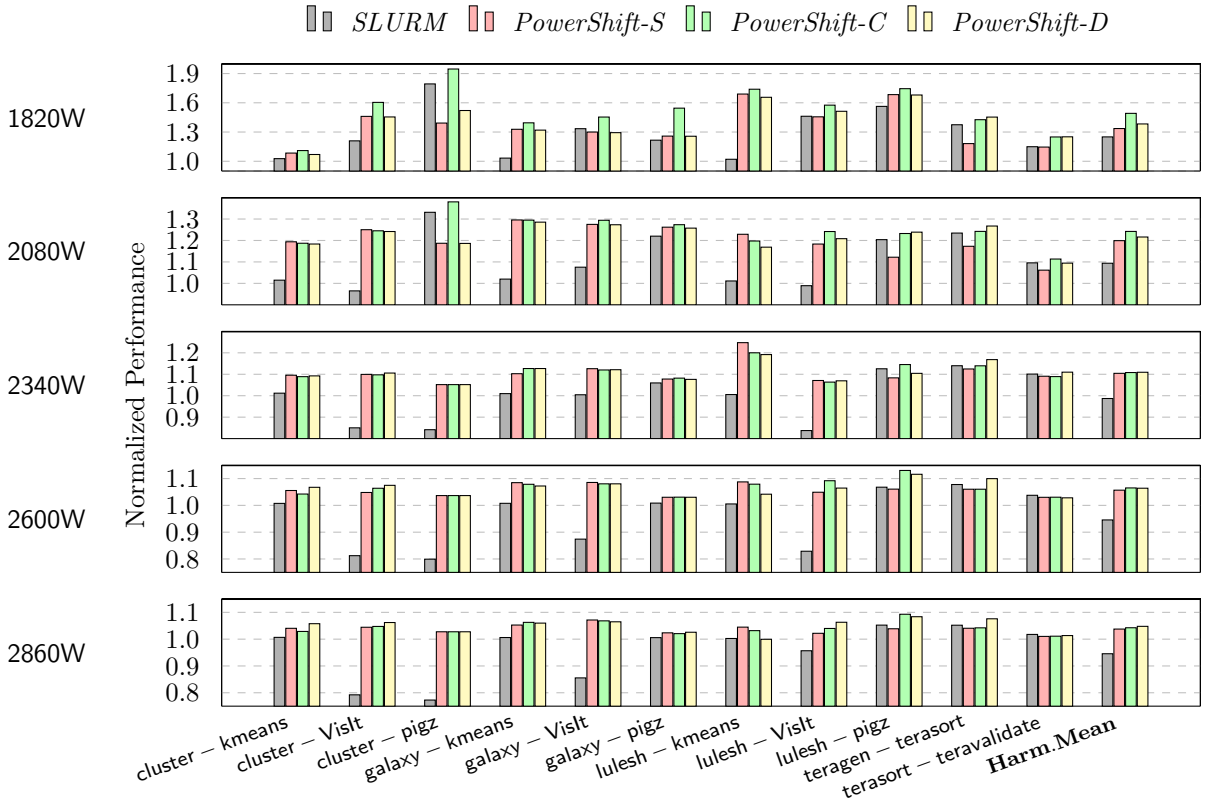


Figure 3.10: Performance for different PowerShift strategies under different power caps.

noise.

### *3.4.1 Experimental Setup*

**Benchmarks** We use 9 individual applications including two cosmological simulation benchmarks, `cluster` and `galaxy` from Gadget 2.0 [91], a hydrodynamic simulation benchmark `lulesh` [50], a visualization application `VisIt` [14], a data mining application `kmeans` [8], a data compression application `pigz` [1], and three MapReduce [17] applications: `teragen`, `terasort`, and `teravalidate` [74]. These benchmarks test a range of important cluster applications, both compute-intensive and memory-intensive. All applications run with up to 48 threads per node (the maximum supported on our test machine). All workloads are long running, taking at least a few minutes to complete.

We pair applications from this pool. The frontend applications are: `cluster`, `galaxy`,

`lulesh`. The backend applications are: `VisIt`, `kmeans`, `pigz`. These nine pairs are representative of the emerging HPC trend of coupling scientific simulation benchmarks with *in situ* analysis or visualization [2, 12]. The 3 MapReduce applications pair among themselves. In each pair, the output of the frontend application is sent to the backend application as input. In total, we have 11 pairs of coupled applications.

**Platform and Metrics** We use the 26-node cluster described in Chapter 3.2. All nodes run Linux 3.13.0. We use RAPL [16] for both power monitoring and capping. The bulk of our results use an execution model where frontend and backend applications are deployed to physically separate nodes. We reiterate, however, that *PowerShift* is topology-oblivious. We explore a different mapping—where front and backend applications are mapped to different sockets in the same physical nodes in Section 3.4.4. We use $1/runtime$ as our performance metric, and $1/energy$ as our energy efficiency metric. All comparisons, normalize to the *Fair* approach.

### *3.4.2 Performance*

We evaluate our eleven coupled applications with five different cluster power budgets ranging from 1820W to 2860W. In all approaches, the sum of the node-level power caps is less than or equal to the cluster power budget at all times.

Table 3.1 shows harmonic mean performance under each power cap for SLURM and *PowerShift*. SLURM performs better than *Fair* at lower power caps and worse at higher power caps, but *PowerShift* uniformly outperforms both SLURM and *Fair*. SLURM's performance suffers for two reasons. First, moving power from nodes with low-power phases to power-hungry nodes, violates the principal that shifts power from the faster application to the slower one, because the power-hungry one is not necessarily the slower one. Second, SLURM does not have a mechanism to return the power back to the node that gave it up when that node shifts to a phase that requires more power. For example, when running

Table 3.1: Comparison of performance under power caps.

| Power Cap | SLURM | PowerShift-S | PowerShift-C | PowerShift-D |
|---|---|---|---|---|
| 1820W | 1.25 | 1.28 | 1.49 | 1.38 |
| 2080W | 1.09 | 1.15 | 1.24 | 1.22 |
| 2340W | 0.99 | 1.06 | 1.11 | 1.11 |
| 2600W | 0.95 | 1.01 | 1.07 | 1.06 |
| 2860W | 0.95 | 1.04 | 1.04 | 1.05 |
| **Har. Mean** | **1.03** | **1.10** | **1.17** | **1.15** |

the `cluster-pigz` couple, `cluster` is always power hungry and `pigz` alternates between power-hungry phase and low-power phase. Once `pigz` enters the low-power phase the first time, all the unutilized power is shifted to `cluster` and never returns. As a result, when `pigz` re-enters its power-hungry phase, it suffers great performance loss. This effect happens at both low and high power budgets, however, when the power budget is low, it turns out that focusing all power budget on one application has fairly good performance. But at higher power budgets, the speedup of focusing all power on one application at a time is much smaller and often worse than *fair*.

The key to *PowerShift's* performance is that it understands coupling. In the static case, this understanding allows some performance gain over SLURM. In the dynamic case, *PowerShift* is designed so that it will shift power from one high-power application to another. This design manifest itself in *PowerShift's* three priority groups (Chapter 3.3.3). This unique feature of *PowerShift* allows it to shift power back from `cluster` to `pigz` in the above example and achieve much greater performance than SLURM.

*PowerShift-S* consistently outperforms *Fair* by 1–28%, with higher speedup under lower power budgets. *PowerShift-C* outperforms *Fair* by 4–49%, while *PowerShift-D* outperforms *Fair* by 5–38%. When the power budget is relatively low, the performance scales better with power, so *PowerShift* has more potential to boost the slower application's performance. For higher power budgets, it is more likely that application performance stops scaling even with more power. As an extreme example when the power budget is set to the total of each node's maximum power, the overall performance cannot grow.

Figure 3.10 shows the performance delivered under each power budget for each coupled application pair. This figure contains one chart for each power cap, the x-axis shows the couple, the y-axis shows performance normalized to *Fair*. The charts show one bar for each of *PowerShift-S*,*PowerShift-C*, and *PowerShift-D*. While results vary per application and power cap, the general trends show that:

1. All *PowerShift* approaches outperform *Fair*.
2. *PowerShift-C* and *PowerShift-D* provide higher performance than *PowerShift-S* across all power budgets.
3. For lower power budgets, *PowerShift-C* out-performs *PowerShift-D* on average. At higher power budgets, *PowerShift-D* can out-perform *PowerShift-C*.

As shown in Figure 3.10, *PowerShift-C* and *PowerShift-D* consistently outperform *PowerShift-S*—by as much as 20%. Applications with long tails receive the biggest benefits from runtime power shifting. In our case, applications having significant I/O phases—*e.g.,* MapReduce, VisIt, and Lulesh—or applications having both phases and long tails—*e.g.,* pigz—all get big speedups with dynamic, runtime power shifting compared to *PowerShift-S*'s static allocation.

The tradeoffs between the dynamic approaches *PowerShift-C* and *PowerShift-D* are not as obvious but they do show:

1. At lower power budgets, *PowerShift-C* outperforms *PowerShift-D*. In *PowerShift-D*, the power pool may not ever be fully drained, but *PowerShift-C* uses global knowledge to allocate all power. At low power budgets, a small amount of unallocated power left in *PowerShift-D*'s pool can have a large effect on performance.
2. If an application has distinct phases—where each node frequently goes from a power-saturated state to a power-hungry state or vice versa—*PowerShift-D* reacts faster than *PowerShift-C*. First, each local decider in *PowerShift-D* acts asynchronously, providing faster reaction to local power state changes. Second, local decisions have their own

control frequency. In our tests, the local decider runs at double the frequency in the power-hungry state compared to the power-saturated state.

3. As the cluster grows, the central decider in *PowerShift-C* may have too much work and generate workload imbalance if co-located with frontend or backend nodes. Eventually, it will need a separate node.

### *3.4.3 Coupling's Effects on Optimal Power*

The prior sections show that*PowerShift* boosts performance compared to fair power distribution. We now demonstrate that power shifting is a hard problem by showing that power allocations that work well for an application in one couple are poor when that application is coupled with a different application; *i.e.,* optimal power allocation is both application- and couple-dependent.

Figure 3.11 shows the optimal power distribution when the three scientific frontend applications are paired with the three different backend applications. There are three charts, one for each of the front-end applications. The x-axis shows power and the y-axis shows performance. There is a labeled mark for each backend application showing its optimal static power assignment when coupled with that frontend. As the figure shows, the different frontends place different requirements on the backends. For example, when coupled with `lulesh`, `VisIt` needs the least power of the three backends. When coupled with `cluster`, however, `VisIt` needs the most power of all backends. *These results show that the front and backends cannot be optimized in isolation, but require consideration of the application couple as a whole.*

### *3.4.4 Results with Co-located Front and Backends*

This section evaluates *PowerShift* in a different topology: when frontend and backend application processes are mapped to separate sockets on the same physical nodes. Other than

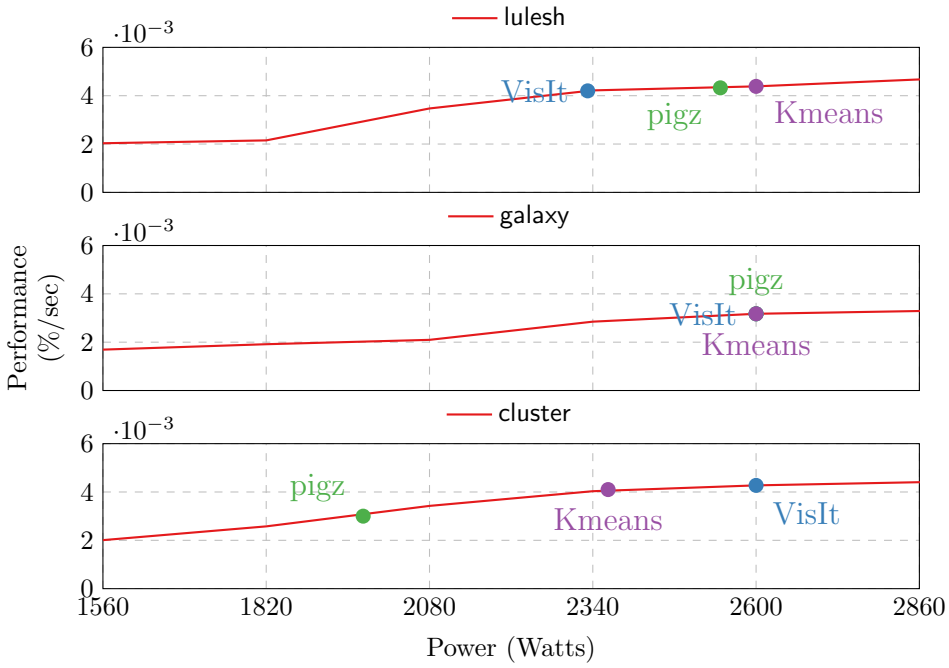


Figure 3.11: Power distribution of different backends paired with the three scientific frontends.

Table 3.2: Performance with co-located couples.

| Power Cap | PowerShift-S | PowerShift-C | PowerShift-D |
|---|---|---|---|
| 1820W | 1.26 | 1.49 | 1.46 |
| 2080W | 1.18 | 1.36 | 1.32 |
| 2340W | 1.04 | 1.11 | 1.10 |
| 2600W | 1.01 | 1.10 | 1.10 |
| 2860W | 1.01 | 1.06 | 1.07 |
| **Har. Mean** | **1.09** | **1.20** | **1.19** |

this process-to-node allocation, the experiments are the same as in Section 3.4.2. Table 3.2 summarizes the results, showing that with a different mapping topology, *PowerShift* has very similar results and properties. The mean speedups compared to *Fair* are 1.09, 1.20, and 1.19 for *PowerShift-S*,*PowerShift-C*, and *PowerShift-D*, respectively.

### *3.4.5 Considering Three Dependent Applications*

We illustrate how *PowerShift* can scale beyond two dependent applications. In this example, `cluster` generates scientific simulation data, `VisIt` turns this data into a movie, and `pigz`

Table 3.3: Performance of three dependent applications.

| | Performance | | Energy Efficiency | |
|---|---|---|---|---|
| Power Cap | **PowerShift-S** | **PowerShift-D** | **PowerShift-S** | **PowerShift-D** |
| 1820W | 1.55 | 1.65 | - | - |
| 2080W | 1.15 | 1.21 | - | - |
| 2340W | 1.07 | 1.09 | - | - |
| 2600W | 1.03 | 1.01 | - | - |
| 2860W | .99 | .99 | 1.02 | 1.02 |
| **Har. Mean** | **1.13** | **1.15** | - | - |

compresses the movie for storage. Table 3.3 shows the performance and energy efficiency of *PowerShift-S* and *PowerShift-D* normalized to *Fair*. *PowerShift-C* is omitted in this experiment, because its overhead grows significantly beyond two applications. For power caps 1820W, 2080W, 2340W, 2600W, *PowerShift* enters performance mode. At the 2860W cap it enters energy-saving mode. The results are consistent with those for coupled applications. Under lower power caps, *PowerShift* tends to enter performance mode boosting performance significantly. Under higher power caps, the potential performance gain is less and it tends to favor energy saving. By harmonic mean, *PowerShift-S* improves performance of *Fair* by 13%, while *PowerShift-D* achieves a 15% improvement. *These results indicate that* PowerShift *still achieves significant performance gains when working with more than two applications.*

### *3.4.6 Energy Savings with High Power Budgets*

This section evaluates the energy efficiency when *PowerShift* enters energy-saving mode. The performance improvement threshold $T$, discussed in Chapter 3.3.2 is a key impact factor for energy savings. The results in Figure 3.10 show $T = 3\%$; *i.e.,* only very small performance loss is allowed. We now evaluate the energy savings when $T = 10\%$.

Table 3.4 shows the performance loss and energy efficiency for all pairs of coupled applications under the maximum cluster power budget. *PowerShift* achieves 18% higher energy efficiency with only 5% performance loss based on harmonic mean. The reason for this savings is that with higher threshold $T$, a wider range of power states can be searched for the

Table 3.4: Energy efficiency with larger threshold T.

| Benchmark | **performance** | **energy efficiency** |
|---|---|---|
| cluster-kmeans | 1.02 | 1.31 |
| cluster-VisIt | 0.94 | 1.21 |
| cluster-pigz | 0.93 | 1.23 |
| galaxy-kmeans | 0.92 | 1.33 |
| galaxy-VisIt | 0.94 | 1.41 |
| galaxy-pigz | 0.92 | 1.42 |
| lulesh-kmeans | 0.99 | 1.07 |
| lulesh-VisIt | 0.96 | 1.06 |
| lulesh-pigz | 0.96 | 1.07 |
| teragen-terasort | 1.01 | 1.04 |
| terasort-teravalidate | 1.01 | 1.03 |
| **Har. Mean** | **0.95** | **1.18** |

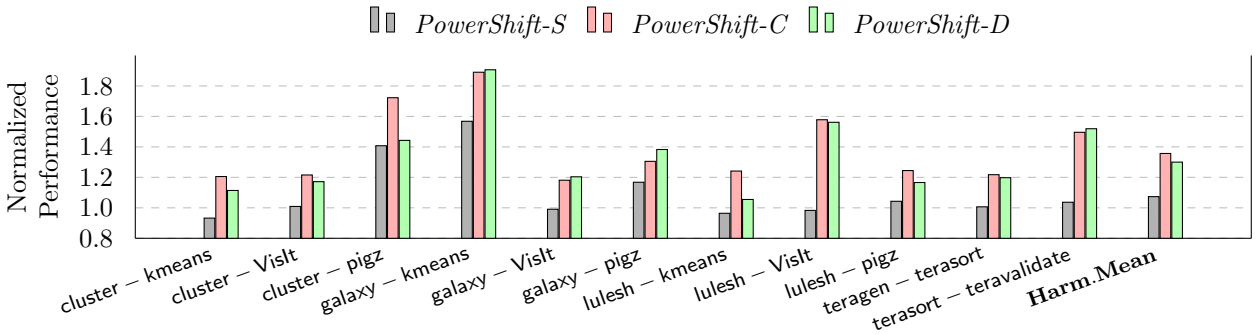


Figure 3.12: Performance under system noise.

most energy-efficient one.

We note that our earlier results show diminishing performance returns for high power budgets (see Table 3.1). The results in this section, however, indicate that *PowerShift* still provides a benefit at high power budgets if users are willing to sacrifice some performance to save energy. In this case, *PowerShift* improves energy efficiency by 18% on average and by as much as 42% for individual couples. *These results indicate that* PowerShift *provides performance gains at low power budgets or energy savings with high power budgets.*

### *3.4.7* PowerShift*'s Resilience to System Noise*

System noise is inevitable in any computing system and prevalent at the large scale of clusters and supercomputers [5]. System noise comes from many kinds of physical and human sources, but its defining characteristic is unpredictability. It often results in great performance fluctuation or slow down on random nodes. It is one of the biggest reasons that large scale systems have severe tail problems. In coupled scientific computations, this system noise slows down not just a single node, but the entire couple.

Figure 3.12 shows the performance results of running all eleven couples in a noisy environment. We create the noise by randomly picking a node and running an additional 8-thread program, stressing the computing resources on one node of the cluster but leaving all other nodes in their usual state. This method of generating noise is general, *PowerShift* does not detect the new application, it simply detects increased power usage on one node. Thus this method could be a stand-in for many other noise sources, including temperature fluctuations or manufacturing variation [45].

All experiments are under the 1820W system-wide power budget and everything is normalized to *Fair*. *PowerShift-C* and *PowerShift-D* consistently outperform *Fair* and average 1.36 and 1.30 speedup, respectively. *PowerShift-S*, however, only has 1.07 speedup and in some cases, is beaten by *Fair*. *Fair* can outperform *PowerShift-S* because the static power distribution derived by the offline decider is not optimal once the second application starts. In *Fair*, however, the stressed node may belong to the slower application, and therefore, may not necessarily slow down the process. Both *PowerShift-C* and *PowerShift-D* deal with tail nodes by shifting power from other finished or light-workload nodes. *These results indicate that dynamic power shifting makes a distributed system significantly more resilient to system noise without requiring any application-level changes.*

Although omitted for space, we have tested many different noise levels. The included data shows that *PowerShift* produces a big benefit for even a small noise level. Our omitted

Table 3.5: Comparison of overhead.

| workload | **Power** | **Core Utilization** |
|---|---|---|
| idle | 24.52W | 0 |
| central decider | 25.75W | 4.725% |
| local decider | 24.58W | 0.323% |

experiments show that the higher the noise, the better the dynamic approaches do compared to static allocation.

### *3.4.8 Overhead and Scalability analysis*

We show the runtime overhead of the central decider from *PowerShift-C*, and the local decider *PowerShift-D*. We have discussed the likelihood of scaling problems in Chapter 3.3.5. At our experimental scale, the overhead of both approaches are negligible. Table 3.5 shows the power and core utilization overhead of both approaches running on power traces. The central decider has $1.23W$ and 4.725% core utilization overhead. The local decider has $0.06W$ and 0.323% core utilization overhead. As the node number grows, local deciders' workload is constant, whereas the central decider's workload grows linearly with node number, shown in Chapter 3.3.5. All our results shown in the Chapter 3.4 include *PowerShift*'s overhead; *i.e.,* the *PowerShift* software is running with the jobs it manages on the same machines. *These results show that the centralized decision mechanism is practical at small scale, while the distributed mechanism is lower overhead and more likely to scale to large systems.* As the system scale grows, interconnection latency is critical to ensure strong scaling. In our design, it is very easy to account for the latency overhead to the frontend. Increased network overhead will result in longer IO phases for the frontend application. This trend will likely make the static scheme worse (since it uses single-application profiles), but *PowerShift-C and -S* should dynamically adjust to find the optimal. Therefore, we believe in a higher latency interconnection environment, there are more opportunities for dynamic shifting.

## 3.5 Conclusion

This paper presents *PowerShift*, a family of techniques for shifting power among nodes running coupled, distributed applications under a system-wide power budget. Specifically, we propose a static scheme, a centralized dynamic scheme, and a distributed dynamic scheme. The static scheme is low overhead, but is not able to adjust to tail behavior or application phases. Both dynamic schemes do adjust, with the centralized scheme generally achieving higher performance—with higher overhead—while the distributed dynamic scheme trades some performance for reduced overhead and increased flexibility. Compared to prior approaches, *PowerShift* provides two advances: (1) instead of just shifting from low-power to high-power nodes, *PowerShift* will shift power away from high utilization nodes to other high utilization nodes that need power more; and (2) *PowerShift* can recognize when it is not helpful to shift power and instead it will reduce energy. Our results confirm that *PowerShift* has practical benefits, demonstrating improved performance, reduced energy, and dynamic adjustment to tail behavior and system noise. We believe the coupled workloads addressed in this paper will become increasingly important in both data centers and supercomputers because they reduce IO burden and take advantage of system scale. *PowerShift* represents one way of improving the performance, energy efficiency, and tail tolerance of this class of application.

# CHAPTER 4

# *PODD*: POWER-CAPPING DEPENDENT DISTRIBUTED APPLICATIONS

This chapter describes *PoDD*, a distributed power management system to maximize performance for dependent applications.

Large scale computer systems are constrained by system-wide power budgets (or caps). As these systems scale out, they are able to concurrently execute dependent applications that were previously processed serially. Such application *coupling* reduces IO traffic and overall runtime as the applications now communicate at runtime instead of through disk. Coupled applications are predicted to be a major workload for future *exascale* supercomputers; *e.g.,* scientific simulations will execute concurrently with *in situ* analysis. As researchers just began to study system-wide power caps for coupled applications, existing work has major limitations: (1) depending on prior application profiles, (2) not coordinating/optimizing node-level power capping technique with system-level power management.

We propose *PoDD* to address the unsolved challenges for coupled applications. *PoDD* enforces a system-wide power cap for coupled applications and maximizes performance. We implement it on a 49-node cluster and compare it to SLURM, a state-of-the-art job scheduler that considers power, but not coupling, and *PowerShift* , from last chapter. *PoDD* increase mean performance over SLURM by 14–22%, over *PowerShift* by 11–13%. Finally, *PoDD* is resilient to tail behavior and system noise, improving performance in noisy environments by up to 44%.

The rest of this chapter are organized as follows: First, Chapter 4.1 briefly overviews the background and the contribution of this work. Second, Chapter 4.2 introduces some closely related works that we use to compare against and discusses the major limitations within them that motivates this work. Then, Chapter 4.3 demonstrates the system and algorithm design. After that, Chapter 4.4 presents the experimental setup, results, and

discuss its reason and insight. Next, Chapter 4.5 summarizes related works in this area. Finally, Chapter 4.6 concludes this work.

## 4.1 Introduction

Exascale computing systems are predicted to provide new opportunities, which will be realized by addressing new challenges. One of the most exciting opportunities is that abundant computing resources will allow a promising new execution model for previously sequential dependent jobs. Specifically, applications can be *coupled* so that they run concurrently and communicate through the network rather than running them sequentially and communicating through the file system [2, 6, 12, 51, 95]. For example, scientific simulations now could be coupled with *in situ* data analysis or visualization [2, 12]. The US Department of Energy (DoE) has declared resource management for coupled applications a key challenge for exascale computing [6, 95].

Among the challenges raised by exascale are the related concerns of power constraints and node-level complexity [95]. At the processor level, computing density is out-pacing cooling capacity, so if all transistors on a chip were used simultaneously, it would generate more heat than can be dissipated and damage itself [25, 98]. At the same time, large-scale computer systems—e.g. datacenters, supercomputers—are constrained by facility-level power budgets. Experts predict that exascale supercomputers will need to operate in a 20-80 MW power budget [6].

To address these power concerns, hardware manufactures have made individual nodes increasingly *configurable*. Almost all processors now support power management either through exposing voltage and frequency settings to software or exposing an interface that allows software to explicitly set power limits [16]. Additionally, aggressive power gating means that performance and power tradeoffs can be further tuned by idling (or choosing not to use) node-level resources like the number of active sockets, the number of active memory con-

trollers, the number of cores per socket and whether hyperthreads are enabled or disabled [109]. For exascale systems, the challenge is determining the node-level configurations that respect the system-wide power budgets while delivering the maximum possible performance. In this paper we explore the specific challenges of coordinating node and system level power management to maximize the performance of coupled applications.

While a great deal of research explores power control for large-scale systems, most does not account for dependent or coupled workloads. For instance, some research aims to increase overall system throughput by better utilizing hardware resources [49], reallocating unused power [89], balancing compute and IO power [84], or mitigating the impact of manufacturing variability [45]. While these studies resolve important issues (unbalanced allocation and phase-adaptation) for power capping distributed systems, they all assume independent workloads. Dependent/coupled workloads, however, follow a fundamentally different performance principal from independent ones. Specifically, for coupled applications the overall speed is determined by the slowest application. Therefore, under a global resource constraint—like a system-level power budget—coupling-aware power management must speedup up the slowest application until each is running at the same speed [20]. None of the works mentioned above is aware of the relative speed between coupled applications nor able to shift power from the faster application to the slower one.

The only work addressing power management for coupled applications that we are aware of is *PowerShift* [110]. While PowerShift represents a family of related algorithms, its distributed, dynamic approach delivers higher performance under power caps than approaches that are not coupling-aware, while offering promising scalability. Unfortunately, PowerShift has two major drawbacks that limit both usability and the maximum delivered performance. First, PowerShift requires users to provide offline profiles of the power and performance tradeoffs for both applications in the couple. This requirement means that both applications will have to be run many times before coupled together under the power cap. Second, Power-

Shift only manages a single node-level resource: processor voltage and frequency. Focusing on a single node-level resource makes PowerShift's implementation simple, but it limits the potential performance gains. For example, several prior works have shown that coordinating DVFS with other node-level resources (like socket, core, and memory usage) leads to much higher performance for the same power budget [13, 15, 21, 22, 29, 57, 77, 79, 83, 103, 109].

To overcome the limitations of prior approaches and achieve higher performance for coupled applications with system-wide power caps, we propose *PoDD*, a dynamic, hierarchical power management system. *PoDD* requires no offline profiling data or code instrumentation. *PoDD* operates in three phases. It first classifies applications based on their optimal resource usage. It then performs a binary search to determine optimal power/performance tradeoffs for the coupled applications. In the final phase, it uses an efficient, distributed power allocation scheme to dynamically adjust power usage and account for application phases, system noise, and tail behavior. *PoDD* amortizes the cost of classification and online profiling by dedicating one node in the system to learning; all application nodes send performance counter information to the dedicated node, which coalesces the data and returns classification results for the applications in the couple.

We implement *PoDD* on a 49 node system and evaluate it against 4 widely-used/state-of-the-art power control systems: *Fair*, *SLURM* [89], *Optimal Static Power Allocation* [110] and *PowerShift* [110]. We normalize all results to *Fair* power allocation, which simply divides the power budget equally among all nodes. We find that:

- *SLURM* improves mean performance by 6%.
- *Optimal Static Allocation* improves mean performance by 8%.
- *PowerShift* improves mean performance by 12%.
- *PoDD* improves mean performance by 28%.
- *PoDD* mitigates tail effects and system noise, improving mean performance by 31% and 44% under these conditions.

- *PoDD* is topology-oblivious—it works well whether coupled applications are physically separate or co-located.

Furthermore, our scalability analysis estimates that *PoDD* could scale to at least 1000 nodes.

*PoDD*'s primary contribution is a design, implementation, and evaluation of a power management system that addresses the three challenges of (1) optimizing node-level performance for a power budget, (2) distributing a system-level power budget, and (3) dynamically adjusting power to achieve high performance for coupled applications. While prior work has addressed some subsets of these challenges, to the best of our knowledge *PoDD* is the first work to address all three holistically. Furthermore, *PoDD* provides a solution that requires minimal input from users.

## 4.2 Background and Motivation

Many distributed power capping approaches have been proposed. In this section, we briefly introduce several prior power management systems, which we will use to evaluate *PoDD*. Then, we point out two major limitations of prior work and describe how *PoDD* overcomes them. Chapter 4.5 contains a full treatment of related work.

### *4.2.1 Prior Power Capping Approaches*

We survey four distributed power capping approaches from the literature. The first two are widely-used, real-world power control systems: *Fair* and *SLURM* [89]. Next, we review two system designs within the *PowerShift* family, the first power capping system designed for coupled applications: *PowerShift-S* is a static approach, *PowerShift-D* is a dynamic approach.

**Fair** power allocation refers to evenly dividing the whole system power cap among each node without knowledge of what applications are running. Moreover, *Fair* allocation has no mechanism to make any runtime power adjustments. *Fair* is widely used as the default power

capping model in data center, supercomputer, or other large-scale systems. This heuristic is simple to implement and works as a one-size-fits-all approach that is workload independent. *Fair* allocation, however, cannot tune power allocation for different workloads. We use *Fair* as our experimental baseline.

**SLURM** is a state-of-art job scheduler for large-scale distributed systems with an intelligent, integrated power capping mechanism. *SLURM* essentially starts the same as *Fair*—with power divided evenly across all nodes—and monitors runtime power consumption to redistribute power using a simple heuristic. Specifically, *SLURM* divides nodes into two groups: those using less power than their assigned cap and those operating near their cap. *SLURM* dynamically reduces the power budget of the nodes in the first group and splits the excess power among nodes in the second group. This heuristic takes advantage of unused power to increase overall throughput of independent applications, but is often sub-optimal in situation of coupled applications for the overall performance is determined by the slowest application, which we demonstrate in detail in the following sections.

**PowerShift** is a family of power management solutions designed for coupled application workloads [110]. All *PowerShift* approaches require offline profiles of each application's performance and power tradeoffs. Furthermore, *PowerShift* only manages node-level DVFS and does not consider any other node-level resources. We are concerned with two variations of *PowerShift*: a static (*PowerShift-S*) and a dynamic (*PowerShift-D*) approach.

*PowerShift-S* takes advantage of offline power and performance profile, to predict the optimal static power allocation between coupled applications. More specifically, the profiles capture performance as a function of node-level power cap and *PowerShift-S* takes in the profiles and derives the power allocation to make coupled applications run as close to the same speed as possible. This approach, however, does not make any runtime power adjustment, so it is not adaptive to runtime changes and often fails to make full use of a given power budget because it cannot correct for any errors.

*PowerShift-D* is a dynamic power management system with decentralized decision makers distributed at each local node. There are two key designs in *PowerShift-D*: a power pool, and power priority groups. The power pool is a data structure that coordinates the dynamic power shifting between nodes. Each local node can reduce its own cap and add power to the pool or request a higher power cap by attempting to extract power from the pool. All nodes are divided into power priority groups, to guide their power intake/output from the pool. Unlike *SLURM* which only shifts power cap from nodes that are not using their full budget, *PowerShift-D* has an additional priority group allowing it to shift power from a fast application operating at its budget to a slow application also operating at its budget.

This ability to shift power from nodes operating at their budget is the key insight for *PowerShift*. Using this approach *PowerShift* offers more than 10% performance gain on average compared to *SLURM* for dependent applications.

### *4.2.2 Major Limitations and Solutions*

*PowerShift*, to our knowledge, is the first work to address the challenge of dependent distributed applications. However, there are two major limitations within *PowerShift*:

- Its dependence on offline application profiles greatly limits the practicality of this approach in real systems.
- Its hardware-only (DVFS only) power capping at the node-level limits opportunities for additional performance through more intelligent node-level management.

In the following sections, we will demonstrate each of the challenges and motivate its solution.

#### Online Power Performance Profiling

Relying on offline profiles has several drawbacks: (1) they may simply not be available, (2) collecting them can be very costly, especially in large scale system and (3) for applications

that dramatically change behavior when given different input and/or arguments, offline profiles are misleading. Thus, there is a need for a practical approach to automatically constructing these profiles online.

Since power capping systems operate in a feedback control loop—constantly monitoring performance and adjusting resources or processor frequency to maintain the cap—it is natural to collect power and performance data online and with enough data points, build a predictive model. The optimization tradeoff is how many data points to collect—fewer data points reduces overhead, but possibly at a cost of higher error in the predictions. Researchers have proposed several techniques for building predictive models of system power and performance. Some collect hardware performance counters and make predictions based on an empirical model [93]. Several approaches applying advanced learning techniques including mutilinear regression [73], probabilistic graphical models [69], and even deep learning [60]. In our case, the nodes running the same application are power capped evenly, so the search space of our problem is very small. For example, in a cluster of nodes with 120 W TDP and 50 W minimum power limit, the most power space we need the model to cover is from 70 W. In practice the space is even smaller as one application will be faster and cover the lower end of the range, while the other application will be slower and want more power. In this space, we find a binary search model works well as it needs only a few observation data points and provides a maximum power error of 2 W (the largest difference between desired and measured power) and has minimal overhead because of the simplicity. We will demonstrate and evaluate this binary search model in detail in later sections.

## Optimizing Node-level Power Capping

We now highlight the benefits of incorporating more complex node-level power management over simply deploying hardware power capping. Because different parallel application may favor different system configurations—e.g. using hyperthreading or not, using dual socket

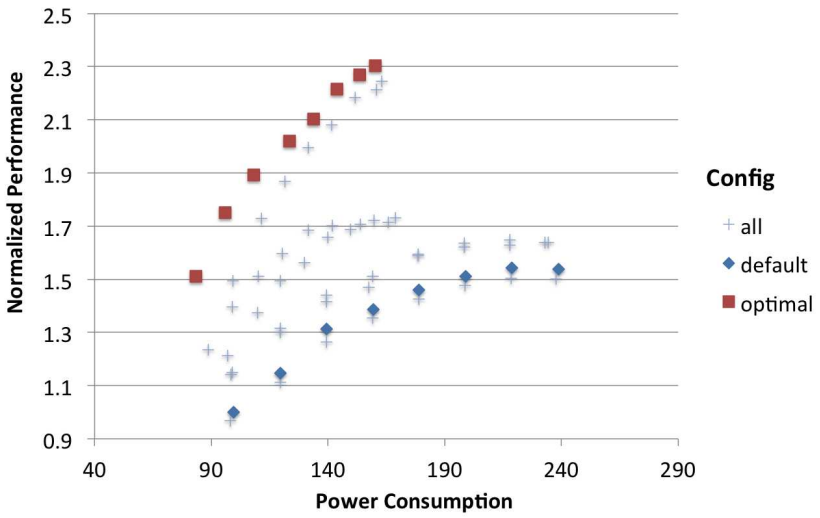


Figure 4.1: Performance of different node configurations.

or not, which memory controllers to use—each application has its unique requirement for computing, memory, and IO resources. Figure 4.1 compares the performance of a scientific simulation, *galaxy*, across different node-level configurations. Each marker in the figure is one run of *galaxy* with different configurations (i.e., different sockets, hyperthreads, memory controllers). For each configuration, we set the power cap from 100 W to 240 W per node. As we can see here, *galaxy* clearly favors a non-default resource allocation (single socket, no hyperthreading, with dual memory controllers). The optimal configuration achieves a 58% performance improvement under the same power cap. In such cases, only setting the hardware power cap would result in sub-optimal configuration and great performance loss.

Achieving better performance, calls for more complex node-level power capping to coordinate multiple resources. The *PUPIL* power management system demonstrates the benefit of this coordinated approach over common approaches that only consider processor voltage and frequency [109]. One key constraint in the PUPiL system design, however, is that it requires

users to add instrumentation to the application code so that the power management can measure performance. Inspired by the performance potential demonstrated both in PUPiL and in Figure 4.1, we choose to develop a system that can coordinate multiple node-level resources without requiring code instrumentation. Specifically, we combine a machine learning classifier with hardware capping to enforce the power cap. The classifier makes decision about which configuration is more suitable for current application based on online monitored hardware performance counters. In such a way, we not only eliminate code instrumentation, but also keep the classifier's overhead low and suitable for online decision-making. Again, more detailed introduction and evaluation of the machine learning classifier follows in the next sections.

## 4.3 Framework

*PoDD* is a hierarchical, distributed, dynamic power management system that handles the unique challenges of maximizing performance for dependent application workloads. *PoDD* addresses these challenges through (1) classifying application response to node level-resources, (2) building models of power/performance tradeoffs online, and (3) coordinating node- and system-level power capping. Furthermore, *PoDD* requires neither prior knowledge of application behavior nor code instrumentation. This section describes the key techniques used in *PoDD*.

For naming simplicity we refer to the two parts of an application couple as the front- and back-ends. The front-end produces data that is consumed by the back-end, so the couple represents a short pipeline. The couple's overall performance will thus be determined by the slowest end, and an ideal resource management system would apportion resources such that each end takes the same time. Under a power cap, then, the power management system should move power from the fastest application to the slowest so that the front and back-ends take the same time. Several works have demonstrated that such a power distribution

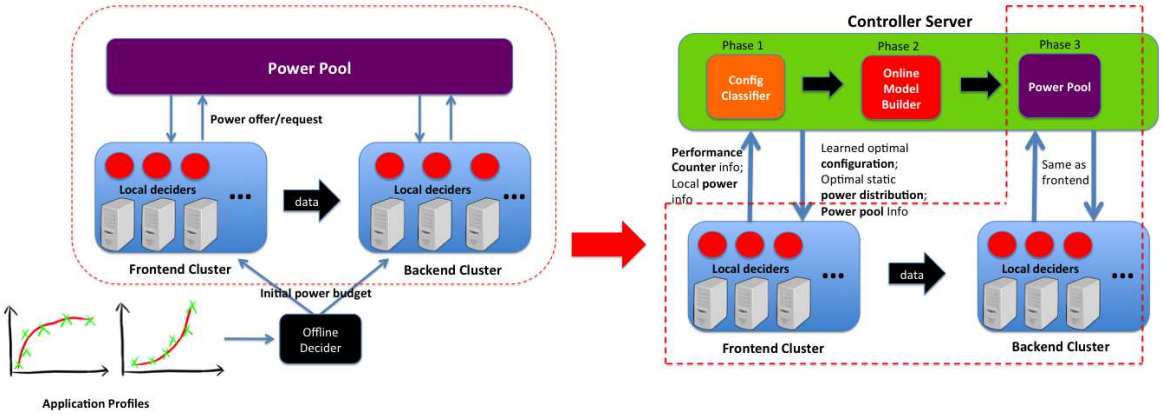


Figure 4.2: Overview and comparison of *PowerShift-D* (left) and *PoDD* (right).

is optimal [20, 110].

*PoDD* uses three phases to achieve a near-optimal power allocation. In the first phase, *PoDD* collects hardware performance counter data from each node and passes that data to a *configuration classifier* to predict the optimal node-level *configuration* for the front- and back-ends, respectively. Here a configuration represents a usage of node-level resources (other than power, which will be managed separately); *e.g.,* a configuration might be the use of two memory controllers, 1 socket and hyperthreads. In the second phase, *PoDD* builds models of the power and performance tradeoffs for the front- and back-ends. This second phase determines an optimal static assignment of power to the front- and back-ends, and thus augments the optimal configurations from the first phase with power information. At the end of this phase, the front- and back-end nodes may have different power allocations and different node-level resources in use. The third and final phase performs dynamic power shifting by coordinating node- with system-level power management. In this final phase, power will dynamically shift from fast to slow nodes with the goal of having the front- and back-end complete at the same time. *PoDD* is designed so that the first and second phase take relatively little time; most of the couple's execution will be in the third phase, which performs dynamic power shifting.

Table 4.1: Overview of performance counters monitored.

| Performance Counter | Description |
|---|---|
| EXEC | Instructions per nominal CPU cycle |
| IPC | Instructions per cycle |
| FREQ | Frequency relative to nominal CPU frequency |
| AFREQ | FREQ excluding when CPU is sleeping |
| L3MISS | L3 cache line miss |
| L2MISS | L2 cache line miss |
| L3MPI | L3 cache line miss per instruction |
| L2MPI | L2 cache line miss per instruction |
| READ | Memory read traffic |
| WRITE | Memory write traffic |
| INST | Instruction retired |
| PhysIPC% | IPC relative to maximum IPC |
| INSTnom% | Relative instructions per nominal cycle |
| TotalQPIout | QPI data traffic estimation |
| QPItoMC | Ratio of QPI traffic to memory traffic |

Each phase has a key enabling technique. The first phase uses multi-level classification to determine optimal resource usage. The second phase uses binary search over possible power distributions to determine optimal static power allocations. The third phase uses power priorities and a power pool to distribute dynamic power management among the constituent nodes. The classification, search, and integration of these components are unique contributions of this paper. The power priorities and power pool used here were first proposed in the prior PowerShift system [110]. Figure 4.2 shows the overview design of *PoDD* and compares it to *PowerShift*. As shown in the figure, the key difference is that *PoDD* automatically determines optimal node-level configurations and static power-performance tradeoffs so that users do not need to perform any profiling or code instrumentation. In other words, *PoDD* automatically collects the data and constructs the models required to use *PowerShift's* dynamic power management scheme. We now discuss each phase and its key component in detail.

### *4.3.1 Phase 1: Configuration Classification*

To achieve better overall performance, *PoDD* incorporates more complex node-level power capping than prior approaches, which simply rely on hardware to manage processor voltage

Table 4.2: Classifier models explored.

| Classifier model | Description |
|---|---|
| SVM-linear | Support vector machine with linear kernel |
| SVM-poly | Support vector machine with polynomial kernel |
| SVM-rbf | Support vector machine with rbf kernel |
| KNN | K-nearest neighbor classifier |
| RF | Random forest, using a bunch of decision trees for classification |
| ET | Extra-tree classifier, a variation of RF, using extreme randomized decision trees |
| AB | AdaBoost, boosted with decision trees |
| LR | Logistic regression classifier |

and frequency. Our two goals for this phase are: (1) classifying application performance without code instrumentation and (2) minimizing overhead.

To achieve these goals, *PoDD* monitors hardware performance counters at runtime and uses a machine learning classifier to predict the most suitable configuration for the current workload. This approach is particularly promising for our use case, because (1) the performance counter info can be easily collected without code instrumentation and (2) the application can be classified after a brief period, reducing overhead. We use Intel's Performance Counter Monitor (PCM) to collect performance counter data at runtime. Table 4.1 shows the overview of system-level performance counters that we monitor. The counter information is normalized to produce per instruction rates (if needed). For example, the READ and WRITE counts are translated into memory traffic per instruction by dividing the measured data by the total instructions retired during the monitor phase.

Each hardware counter information serves as one feature for the classifier. They further go through a series of pre-learning procedures: noise filtering, data standardization, PCA (principal component analysis), feature selection. At that point they are fed into the classifier to predict suitable configuration for current application.

**Two-level classifier:** A key design choice is to have a two-level classifier that predicts computing and memory resources separately instead of treating the problem as a multi-class classification. This choice of two-level classification means that, instead of predicting all configuration preferences (socket allocation, use of hyperthreads, memory controller allocation)

at once, we break predication into two parts: (1) predicting computing resource preferences (socket allocation and hyperthread usage) and (2) predicting memory resource preferences.

We choose this two-level classification because classifying compute and memory needs simultaneously makes the problem harder. The difficulty arises in part from the fact that memory and compute resources are not completely independent. For example, low IPC may result from either poor compute resource allocation or from an application that constantly misses in the cache. In the first case, IPC will improve by allocating more compute resources. In the second case, more computing resources will consume more power, but without increasing performance.

For completeness we did test a multi-class classifier that would predict both memory and compute resource needs. That approach, however, has much worse accuracy than the approach we use. The multi-class classifier achieves 80% recall with less than 70% precision. In comparison, the multi-level classifier we use in *PoDD* achieves 95% recall and 74% precision. We further discuss the choice of classifier in the next paragraph.

**Model selection:** We have compared eight popular classifier models shown in Table 4.2. In all cases, we perform classification after pre-processing the hardware PCM data by performing noise filtering, data standardization, PCA (principal component analysis), and feature selection. All models are tuned by hyper-parameter search. The key evaluation criteria to be noted here is that *high recall rate is a must, and the higher precision the better.* In other words, our system can tolerate false positives but is more sensitive to false negatives. The intuition is that the system will use dynamic feedback, so if it selects a sub-optimal configuration (a false positive) the low performance will be detected and corrected (at the penalty of briefly running slower than necessary). However, if we have false negatives, there will be an optimal configuration which the system will never use. limiting the potential performance gains.

Thus, we tune all models with the goal of 95% recall rate if possible, and then we

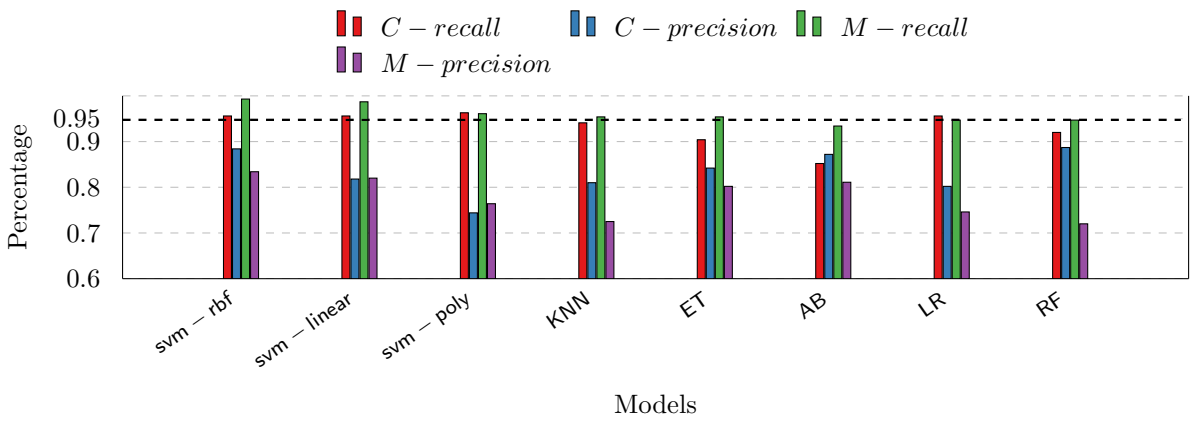


Figure 4.3: Recall and precision of 8 learners for classifying computing and memory resources.

compare the precision between them. Figure 4.3 shows the recall and precision for both predicting computing and memory resources using 8 different learning models. "C" stands for computing resources and "M" stands for memory resources. As the chart shows, svm-rbf delivers above 95% recall with the highest precision for both resources.

**Feature transformation and selection:** Figure 4.4 shows the result of the Principal Component Analysis (PCA) procedure. PCA is a common orthogonal transformation approach to eliminate correlation among raw feature data. This process is important when we use hardware performance counters as input, because there are quite a few correlated counters, e.g. EXEC vs. FREQ vs.AFREQ, L3Miss vs. READ/WRITE, etc. Another benefit of PCA is, with feature selection, computation overhead is minimized and predication rate is optimized by filtering out noisy features. In general, the higher variance carried by the top few components, the better it is. In this case, the top 8 components found by PCA carry more than 99.8% variance, and the top 5 components carry more than 98.1%. Table 4.3 shows the cross-validated averaged recall and precision using different number of top X components as feature input. The performance increases until the component number reaches 5. This data implies that components beyond the 5th are very likely to add noisy or redundant features to the model. Therefore, in *PoDD*, we use the top 5 components as input features to minimize overhead and increase accuracy performance.

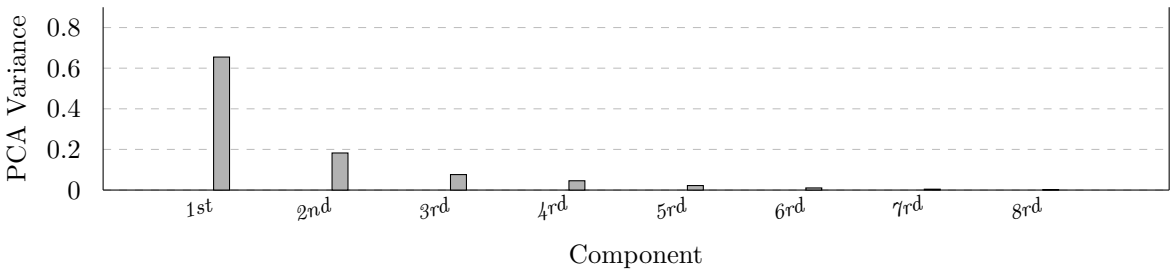


Figure 4.4: Eight features explain more than 99.8% of variance.

Table 4.3: Accuracy comparison of using top X components.

| Top X components | **Recall** | **Precision** |
|---|---|---|
| 1 | 0.912 | 0.861 |
| 2 | 0.941 | 0.727 |
| 3 | 0.963 | 0.824 |
| 4 | 0.963 | 0.852 |
| 5 | 0.963 | 0.885 |
| 6 | 0.948 | 0.860 |
| 7 | 0.941 | 0.848 |
| 8 | 0.941 | 0.848 |

In summary, during the classification process, hardware performance counter data is collected and normalized. This data is sent to the learning node, which collects all node-level data for both front and back end applications. The data is coalesced and used to classify each application in terms of the optimal configuration of compute and memory resources. These predicted optimal configurations are then sent to all individual nodes. After this phase, each node should have an optimal resource configuration, and *PoDD* transitions to the next phase: building power/performance tradeoff models.

### *4.3.2 Phase 2: Online Model Building*

One of *PoDD*'s key design features is finding the optimal power distribution between front- and back-ends online. This process requires modeling how both the front- and back-ends performance changes with changing power allocations. These models are then used to determine a power distribution between front- and back-ends such that each completes its work at close to the same time.

Two key factors make online model building natural with *PoDD*:

- It is easy to collect the data points needed for building.
- In coupled applications it is easy to detect the synchronization points between front- and back-end applications and the time taken to arrive at those points is a good performance indicator.

Similar to the classifier, we want the power/performance model builder to have high accuracy and low overhead. For this component, however, accuracy means the predicted performance under some power cap should be fairly close to the true value. Low overhead means both fast convergence and low computational overhead to construct the model. We compared three different methods of online model building and tested their accuracy and overhead (the details are in Chapter 4.4.6). Of these three the binary search approach had the highest accuracy and close to the lowest overhead because the range of power distributions *PoDD* has to explore online is quite small. There is an underlying assumption here, however, which is that after classification determines optimal node-level resource usage, the power/performance tradeoff space is convex. In practice, we find that classification's high recall and dynamic feedback does a good job eliminating local optima (*e.g.,* the non-optimal points in Figure 4.1) and the convexity assumption holds across all the applications we tested.

Algorithm 6 details the binary search algorithm as implemented for this specific problem. All the power caps mentioned are per-node power caps, and the two parameters to be set by users are the power resolution and performance resolution. Power resolution is a threshold for how close the system should get to the desired power budget. Performance resolution is a threshold for determining when the front- and back-end applications are considered to be running at the "same" speed. Due to system noise and variance we never expect the front- and back-ends to actually complete at exactly the same time. In our case, we use 2 W as the power resolution, since lower numbers do not impact performance; for performance resolution, we use 2%, meaning that if the coupled applications reach a synchronization point

**Algorithm 6** Binary Search Model Algorithm

**Require:** Power cap $P$, maximum power limit $P_{max}$, minimum power limit $P_{min}$
**Require:** Power resolution $R_{pwr}$, performance resolution $R_{perf}$
**Require:** Power cap for frontend and backend $P_{front}$, $P_{back}$
**Require:** Current performance and power for frontend and backend applications: $(power_f, perf_f)$,$(power_b, perf_b)$
Feedback $(power_f, perf_f)$,$(power_b, perf_b)$, when $P_{front} = P_{end} = P$
$\delta_{power} = MIN(P_{max} - P, P - P_{min})/2$ ▷ max shiftable power
$r = perf_f > perf_b ? 1 : -1$ ▷ $r$ reverses power shifting
**while** $\delta_{power} > R_{pwr}$ and $\delta_{perf} > R_{perf}$ **do**
$P_{front} = P_{front} - r * delta_{power}$
$P_{back} = P_{back} + r * delta_{power}$
Apply power caps $P_{front}$ and $P_{back}$.
Get feedback: $(power_f, perf_f)$, $(power_b, perf_b)$.
$\delta_{power} = \delta_{power}/2$ ▷ decrease search range by 2
$\delta_{perf} = |perf_f - perf_b|/MAX(perf_f, perf_b)$
$r = perf_f > perf_b ? 1 : -1$
**end while**
Return optimal static power distribution $(P_{front}, P_{back})$

within 2% difference of each other, we consider them to be running at same speed.

### *4.3.3 Phase 3: Dynamic Power Shifting*

In this phase, *PoDD* dynamically tunes the power allocation. At this point, power can be shifted between front-end application nodes and back-end nodes, or between nodes running the same application to account for tail-latency or system noise. While the first two phases of *PoDD* represent original work, this phase builds of the dynamic power shifting builds off *PowerShift's* distributed, dynamic power management infrastructure. Specifically, *PoDD* integrates *PowerShift's*: *power priority grouping* and *power pool*, but gets optimal power distribution from *the online model* of phase 2, instead of *offline* profiles. Furthermore, each local node now executes with a resource configuration learned by Phase 1's classifier instead of the node's default configuration.

**Power priorities:** There are 3 priority groups:

1 Nodes operating below their assigned power cap.

2 Nodes operating near the power cap for which the online profile predicts additional power will *not* improve couple performance.

3 Nodes operating near the power cap for which the online profile predicts that additional power *will* improve couple performance.

Power is always shifted from lower priority groups to higher ones. This grouping mechanism allows *PoDD* to reallocate power between nodes that operating near power limit, however, with different speedup effect on overall performance, which is the key optimization of the dynamic mechanism for dependent applications.

**Power pool:** This shared data structure coordinates power shifting between local nodes. It keeps track of the minimum information needed: how much power is in the pool (unused power) and how many nodes are in *Group 3*. This structure is the key to enforce strict system-wide power limit, as it keeps the invariant that any node giving up power to the pool must first lower its local power cap and any node taking power from the pool can only increase its local power cap after decreasing the power from the shared pool.

Each local node operates in a classic control loop: observing its environment, deciding on a response, and acting to implement its decision. In observation phase, it collects local power consumption. Then in decision phase, it first places itself into one of the three *power priority groups*. It then sends the power and priority group information to the shared power pool. Upon receiving the response from power pool, it decides its local power allocation for the resources learned by the *configuration classifier*. This process of observing power, interacting with the pool and adjusting the local power cap is repeated continually as the coupled application executes. In this manner *PoDD* constantly fine tunes its power allocation and can adapt to phases within an application, application tail latency, and system noise.


Figure 4.5: Performance for different power management systems under different power caps.

## 4.4 Experimental Evaluation

This section introduces our experimental setup and compares the performance efficiency of *PoDD* to 4 prior works: *Fair*, *SLURM*, *PowerShift-S* (which statically allocates power based on offline profiles), and *PowerShift-D* (which dynamically allocates power, but still requires offline profiling). This section also shows how different design choices within *PoDD* affect the system. Finally, we show the advantage of *PoDD* dealing with tail effects.

### *4.4.1 Experimental Setup*

**Benchmarks** There are 8 individual applications we use for evaluation, including 3 scientific simulation applications: `cluster` and `galaxy` from the cosmological simulation suite Gadget 2.0 [91]and the hydrodynamic simulation benchmark `lulesh` [50]. We use 3 different applications as proxies for in situ analysis and visualization: a scalable visualization application `VisIt` [14], a data compression benchmark `pigz` [1], and an unsupervised learning algorithm `kmeans` [8]. We also explore 2 Spark [108] applications from SparkBench [56]: `SparkKmeans` and `SparkSQL`. These benchmarks feature important workloads in distributed systems, with different resource needs, including compute-intensive, memory-intensive, and IO-intensive applications. We then create pairs of these 8 application to represent the emerging workloads of coupled applications. All coupled pairs are in the form of frontend-backend, where the backend application takes the output of frontend application as input. Overall, we have 15 pairs: cluster-VisIt, cluster-kmeans, cluster-pigz, cluster-SparkKmeans, cluster-SparkSQL, galaxy-VisIt, galaxy-kmeans, galaxy-pigz, galaxy-SparkKmeans, galaxy-SparkSQL, lulesh-VisIt, lulesh-kmeans, lulesh-pigz, luelsh-Sparkkmeans, SparkSQL-SparkKmeans. All applications are launched with 48 threads per node, which is the maximum number of virtual cores in our test servers. Additionally, all applications are relatively long-running, taking at least a few minutes up to several hours.

**Platform** Our test system is a 49-node cluster. Each node is a dual-socket server, with 2 Intel Skylake Xeon Gold 6126 CPUs. The nominal clockspeed is 2.60 GHz. Each node has 256 GB of RAM divided between dual memory controllers. All nodes support Intel RAPL technology for enforcing power caps in hardware through DVFS. Each processor has 12 physical cores, with hyperthreading, giving a total of 48 virtual cores across both sockets. These nodes are connected with 32-port software-defined 10 GigE switches.

**Metrics** We use $1/runtime$ as our performance metric. All performance numbers are normalized to *Fair*.

Table 4.4: Comparison of performance under power caps.

| Power Cap | SLURM | PowerShift-S | PowerShift-D | *PoDD* |
|---|---|---|---|---|
| 5760W | 1.08 | 1.11 | 1.16 | 1.33 |
| 6720W | 1.07 | 1.07 | 1.13 | 1.29 |
| 7680W | 1.06 | 1.08 | 1.12 | 1.31 |
| 8640W | 1.05 | 1.09 | 1.12 | 1.27 |
| 9600W | 1.04 | 1.03 | 1.07 | 1.21 |
| **Har. Mean** | **1.06** | **1.08** | **1.12** | **1.28** |

### *4.4.2 Performance*

*PoDD* and 4 other power management systems are evaluated with our couples across 5 different system power budgets from 5760W to 9600W. While the system-level power budget is enforced by all 5 approaches—*i.e.,* the sum of 48 node-level power is always less than or equal to the system-level budget—the performance varies.

Table 4.4 summarizes the overall harmonic mean performance normalized to *Fair* for each of the evaluated power managers under different power caps. All approaches outperform *Fair*. *SLURM* achieves 6% speedup from dynamically reallocating extra power without awareness of application coupling. *PowerShift-S* achieves 8% speedup from distributing optimal static power between frontend and backend clusters. *PowerShift-D* exploits both of these advantages and achieves 12% speedup on average compared to *Fair*. in comparison, *PoDD* outperforms *Fair* by 28% on average. The large speedup of *PoDD* compared to *PowerShift-D* shows how beneficial it is to coordinate system-level power shifting with advanced node-level power capping.

We note that *PowerShift-D* outperforms *SLURM* because *SLURM* always shifts power from nodes with extra power to nodes running near their power cap. *PowerShift-D*, on the other hand, shifts power from nodes running relatively fast to nodes running relative slowly, allowing the power-hungry nodes running unnecessarily fast to release power to the other nodes. *PoDD* improves over *PowerShift* by incorporating node-level resource classification into the framework so that each node is not only allocated optimal power, but also configured into optimal resource settings.

Figure 4.5 shows the performance delivered by each power management system across 5 power budgets for each coupled application pair. All numbers are normalized to that of *Fair*. The 5 charts from top to bottom shows results of system power budgets ranging from 5760W to 9600W. The x-axis shows the couple and the y-axis shows the normalized performance. The grouped bars stand for *SLURM*, *PowerShift-S*, *PowerShift-D*, and *PoDD* from left to right. We note three observations from the figure:

1. Almost all approaches outperform *Fair*, except 2 data points where running `lulesh-visit` with *PoDD*, the performance is slightly worse, but still within 3%.
2. While a specific application pair might favor one power management approach over another, *PoDD* significantly outperforms *PowerShift* and *SLURM* on average.
3. For all power management approaches, the performance speedup is generally higher at relatively lower power caps, and lowest at the highest power cap.

Here we explain the reasoning for each of these observations. First, *Fair* ensures each node is allocated even power throughout the whole execution. Such power distribution almost always leads to sub-optimal performance for coupled applications as different applications have different power/performance tradeoffs so evenly allocated power in fact results in uneven speeds. For the `lulesh-visit` couple, however, at the two highest power caps both applications already run at maximum speed under *Fair*. Therefore, no power shifting would further speedup either of the coupled applications and the overhead of *PoDD* (and other power management approaches) slightly degrades performance.

Second, we consider why some application couples respond better or worse to different power management systems. `galaxy` and `cluster` favor using a single socket per node without hyperthreading. Because *PoDD* is the only approach that adjust these node-level resources, it is capable of (1) achieving much higher performance for the nodes running these applications and (2) freeing up additional power from these nodes to shift to the node running the other application in the couple. `lulesh` and `kmeans` are both compute-intensive

benchmarks; when paired with applications offering extra power, their performance greatly improves. However, when they are coupled together, there is little power shifting opportunity because both want more power. `visit` and `pigz` have significant I/O phases, and during those phases they are able to offer extra power to the other application in the couple. Also, `pigz` favors a single memory controller—where most applications favor two—which offers additional configuration optimization opportunities for *PoDD*. At the same time, `pigz` has a dramatic tail effect due to workload imbalance. This imbalance favors the dynamic runtime power shifting. Lastly, the Spark workloads do not have distinct I/O phases, but they do have decent amounts of I/O time scattered across their whole execution time. Even these small I/O times can be utilized by fine-grained dynamic power shifting system.

Finally, the reason all power capping approaches performs relatively better at lower power caps are twofold. First, the performance of applications tend to scale better with power at lower power cap, as many power performance tradeoff functions are concave. Second, at higher power caps, the shiftable power is limited. For example, at 200 W per node, as the maximum power cap is 240W, there is only 40W to be shifted around.

Overall, *PoDD* outperforms *Fair* by 28%, *SLURM* by 21%, and *PowerShift* by 14%. These results demonstrate the necessity to optimize power allocation between coupled applications, dynamically shift power at runtime, and coordinate advanced node-level power capping with dynamic system-wide power shifting.

### *4.4.3 Running with Offline Profiles*

While offline profiles are not always available, they often exist for a number of applications run repeatedly in large-scale systems. For example, it would be trivial to modify *PoDD* to output its internal power and performance profiles for each application and reload them if the application was run again. In this section, we evaluate *PoDD*'s performance when profiles are provided. Essentially, when offline profiles are provided, *PoDD* would skip phase 1 and

Table 4.5: Comparison of *PoDD* with or without profiles.

| Power Cap | ***PoDD*** | ***PoDD* with profile** |
|---|---|---|
| 5760W | 1.33 | 1.38 |
| 6720W | 1.29 | 1.35 |
| 7680W | 1.31 | 1.33 |
| 8640W | 1.27 | 1.28 |
| 9600W | 1.21 | 1.21 |
| **Har. Mean** | **1.28** | **1.31** |

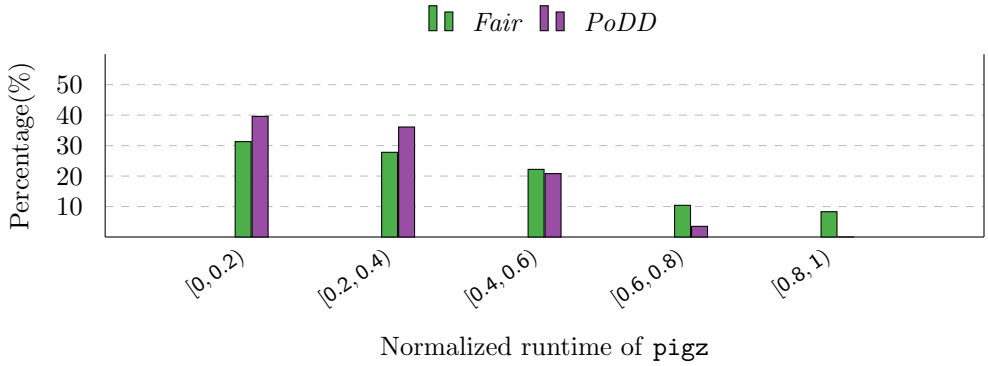


Figure 4.6: Runtime distribution with work imbalance.

phase 2 optimization, and directly enter phase 3 with optimal node-level configuration and power distribution between coupled applications. Table 4.5 shows the benefit if offline profiles are available. All numbers are still normalized to *Fair*. *PoDD* with profiles gives another 3% performance improve on average by eliminating the overhead during the convergence of the learning phase and online model building phases. On the other hand, this also reflects how much of overhead learning classifier and online model builder creates.

### *4.4.4 Resilience to Tail Effects*

Tail effects are a critical issue in distributed workloads, appearing when some small number of nodes has much longer execution time than the majority. These tail nodes drag down overall application performance. In coupled applications, tail nodes slow down both the application to which they belong and the couple.

There are various sources that cause tail effects including: system noise and workload

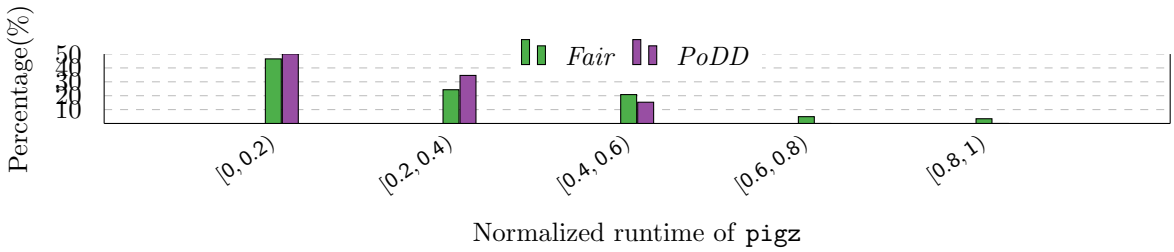


Figure 4.7: Runtime distribution with system noise.

Table 4.6: Comparison with different topology mapping.

| Power Cap | **PowerShift-D** | **PowerShift-D co-located** | ***PoDD*** | ***PoDD* co-located** |
|---|---|---|---|---|
| 5760W | 1.15 | 1.23 | 1.58 | 1.60 |
| 6720W | 1.21 | 1.31 | 1.68 | 1.68 |
| 7680W | 1.15 | 1.25 | 1.76 | 1.70 |
| 8640W | 1.15 | 1.24 | 1.80 | 1.65 |
| 9600W | 1.08 | 1.15 | 1.66 | 1.58 |
| **Har. Mean** | **1.15** | **1.24** | **1.69** | **1.64** |

imbalance. System noise can come from many forms of physical and human sources and it is inevitable in distributed systems [5]. Workload imbalance has always been one of the great challenges in designing distributed applications. Due to the variations of application input and parameters and hardware variety, perfect work balance is hard to achieve. Many prior works mitigate workload imbalance in various ways, *PoDD*'s dynamic power management is naturally resilient to these issues.

Figure 4.6 shows the runtime distribution of the nodes running the `pigz` application, which is known to have tail effect due to workload imbalance. We have collected the execution time of each node for 3 runs of `pigz` in different coupled pairs under 7680W system-wide power budget. We normalize the execution time of each node to the longest one using *Fair*, and accumulate the frequency of execution time, which is reported as percentage over the number of all nodes. As we can see from the chart, comparing *PoDD* to *Fair*, the numbers in range of $[0.6, 1]$ (tail nodes) decrease dramatically from 18.7% to 3.5%. The averaged coupled performance speedup is 31.3%.

Next, we evaluate the performance under system noise. We emulate system noise by randomly picking two nodes in the `pigz` cluster and run a 24-thread program on each of

Table 4.7: Comparison of online profiling techniques.

| Model Name | **Accuracy** | **Convergence Speed** | **overhead** |
|---|---|---|---|
| binary-search | upper bound 2W | 4 | single core, less than 1s |
| quadratic | Average 3W | 3 | single core, less than 2s |
| logarithm | Average 2W | 4 | single core, less than 2s |

Table 4.8: Overhead analysis.

| Resource | **Idle usage** | **Average usage** | **Peak usage** |
|---|---|---|---|
| CPU | 0.10% | 0.65% | 5.25% |
| Memory | 0.70% | 0.75% | 1.24% |
| Network I/O | 0.18% | 0.20% | 0.83% |

those nodes stressing the computing resources. All other nodes are left in their usual state. This method of generating noise is general, similar (though likely less pronounced) effects could come from other sources such as temperature fluctuations or manufacturing variation. While *PoDD* is not explicitly aware of the inserted system noise, it detects the increased power pressure on the 2 nodes. Figure 4.7 reports the runtime distribution under system noise, all other experimental parameters are the same as those in Figure 4.6. Comparing *PoDD* to *Fair*, the numbers in range of $[0.6, 1]$ decrease remarkably from 8.3% to 0%, and the averaged coupled performance speedup is over 44%. *PoDD* achieves these results because it naturally reallocates power from both nodes that finish early and from the other application in the couple to speed up these two tail nodes. This dynamic power management greatly mitigates tail effects in distributed workloads; improving not just the performance of the directly effected but also the overall performance of the coupled pair.

### *4.4.5 Topology-obliviousness*

*PoDD* is *topology-oblivious,* meaning the approach can support multiple mappings of applications to physical hardware as long as the power control of the hardware assigned to front and back end applications is independent. For simplicity of discussion the majority of evaluation assumes different applications are mapped on physically separate nodes, such separation is not a requirement. In this section shows we evaluate when front and backend application

Table 4.9: Power management system capability comparison.

| System | **Distributed** | **Dependent-aware** | **Dynamic** | **No offline profiles** | **No code instrumentation** |
|---|---|---|---|---|---|
| *Fair* | **X** | **X** | **X** | ✓ | ✓ |
| *PUPiL* [109] | **X** | **X** | ✓ | ✓ | **X** |
| *SLURM* [89] | ✓ | **X** | ✓ | ✓ | ✓ |
| *PowerShift-S* | ✓ | ✓ | **X** | **X** | ✓ |
| *PowerShift-D* | ✓ | ✓ | ✓ | **X** | ✓ |
| *PoDD* | ✓ | ✓ | ✓ | ✓ | ✓ |

are scheduled on same nodes but different sockets. Specifically, we evaluate the one application couple for which *PoDD* shows great speedup—`galaxy-visit`—across 5 power caps. Table 4.6 shows the results: that *PoDD* still has great speedup over *PowerShift-D*; however, the relative improvement is less, since the performance of *PowerShift-D* increased in the co-located case. The reason is that in the co-located case, the configurable resources are fewer because there is no option to reduce socket usage. Therefore, *PowerShift-D* happens to be forced to use a better resource allocation than before. Nevertheless, these results show *PoDD* performs well invariant of mapping topology.

### *4.4.6 Online Model Builder*

In this section, we demonstrate a comparison for 3 different approaches to online model building in terms of their accuracy, convergence speed, and computation overhead. Accuracy is quantified by power error (W) from optimal power allocation. Convergence speed is quantified by how many data points the model needs to converge. Computation overhead is evaluated by how much of computation resources the models need. Table 4.7 shows the overall comparison. As discussed in Chapter 4.3.2, our power range of each end is limited to 35 W, binary search is the best fit in this case, because of high accuracy, low overhead and comparable convergence speed. More complex learning models—*e.g.,* [60, 69]–are not suitable in our case, simply due to comparable or even worse convergence speed with much bigger overhead.

### *4.4.7 Scalability analysis*

The controller node is in charge of learning node-level configurations, building online power/performance models, and managing the power pool. For our test system, we have 48-node computing node with one controller node. We evaluate the computing, memory, and network I/O stress on the controller node and the time spent exclusively on the controller node (normalized to the entire execution time). Table 4.8 shows the resource utilization overview. The idle usage serving as a baseline, is the average usage when machine is nominally idle. The average usage is averaged over all experimental runs and over the entire execution time. Peak usage is the average peak usage of each run over all runs. As we can see, none of the resources are even close to being challenged under the stress of supporting a 48-node system. The low utilization is because we chose to use a less costly approach and minimize data exchange. Based on these results, we predict the controller node could handle at least 1000 nodes without severe resource contention. One potential limiting factor, however, is the network latency as the cluster grows larger and we need to make real time adjustment.

## 4.5 Related Work

Power constraints have become one of the biggest concerns in computer systems at all scales.

At node-level, researchers have proposed both software-based approach and hardware-based approaches to control power. Early software approaches manage individual components including DVFS for a processor [54], per-core DVFS in a multicore system [48], processor idle-time [32, 111], DRAM [23], and storage [52]. The coordination of multiple system components consistently out performs approaches that only consider a single component in isolation. Examples include approaches that coordinate processor and DRAM [13, 21, 22, 29, 57, 83] processor speed and core allocation [15, 79], combining DVFS and scheduling [77, 103], memory and disk speed [58] and combining DVFS and process placement [64]. Two recent approaches provide general interfaces to coordinate arbitrary sets of resources to

deliver maximum performance for a given power cap [39, 43]. These approaches, however, require offline profiles of power and performance tradeoffs.

For hardware-based approaches, the most widely used and studied is Intel's RAPL—Runtime Average Power Limiting—system, supported in SandyBridge and later processors [16]. Hardware approaches have the advantage of converging to desired power state faster than software approaches. However, they are not able to achieve optimal performance due to only tuning a single resource: DVFS, or processor frequency and voltage. To get the combined benefits of software and hardware, researchers have developed techniques for coordinating the two to achieve high performance with fast convergence [109].

Early work on cluster-level power capping largely ignored performance concerns and focuses on coordinating different levels of the system (data center, rack, server) to ensure power constraints are respected [0]. Given this foundational work to establish system-wide power budgeting, follow up projects could explore improving performance given those budgets. Examples include consolidating workloads to use fewer physical machines [61, 72], job scheduling to achieve resource efficiency [4, 18, 19, 33, 80], and hardware over-provisioning, such that nodes in the system need to be power capped to avoid power loss [82]. Despite differences in methodologies, these systems all try to deal with single application or independent applications and none address the coupled applications studied in this paper.

The closet work to *PoDD*, is *PowerShift* [110]. To the best of our knowledge, *PowerShift* is the first work to propose a power capping solution that specifically addresses the challenges of coupled application workloads. Earlier sections of the paper describe the differences between *PoDD* and *PowerShift* in great detail and demonstrate the performance improvements that *PoDD* obtains.

Table 4.9 shows the capability comparison of several related power management systems and *PoDD*. Again, *PoDD* offers a practical approach for coupled applications running under system power budget in a large-scale system, delivering high performance efficiency, requiring

no prior application profiles, no code instrumentation with decent scalability.

## 4.6 Conclusion

This paper presents *PoDD*, a hierarchical, distributed, dynamic power management system to address the emerging challenge of power capping for coupled workloads in large-scale system. It makes 3 main breakthroughs: (1) developing a novel node-level power capping technique that monitors hardware performance counter and learns the optimal resources allocation using a classifier, (2) coordinating node-level power capping with system-level power shifting for maximized performance, and (3) requires no application profile by building power performance model online. Under a variety of power caps running a mixture of different coupled workloads, *PoDD* outperforms several state-of-the-art approaches while showing great resilience to tail effects and system noise. The performance improvements available with *PoDD* demonstrate the benefits of (1) developing power management for coupled workloads (such as future multiphysics and in situ analysis), (2) coordinating node-level power optimization with system-level power shifting, and (3) dynamically shifting power. We hope this work inspires future research and further improvements to these critical problems.

# CHAPTER 5

# CONCLUSION

This dissertation systematically studies an essential part of modern computing systems – power capping/budgeting techniques with focus on server-based systems. Power capping gets increasingly important as power has become arguably the number one constraint for performance scaling of computing machines. Meanwhile, it is challenging for 2 reasons: (1) Strictly limiting power consumption is a challenging problem, (2) Enforcing the power cap only is not enough, high performance is required for power capping systems. In the dissertation, we start addressing this problem by looking at the single server system. *PUPiL*, a dynamic hardware/software hybrid power control system, maximized the performance while enforcing the power cap in a timely way. Then, we divert attention to the other end of spectrum – large-scale systems, more specifically, the emerging challenge of power capping coupled workloads in a distributed system. The proposed *PowerShift*, a family of techniques for shifting power among nodes running coupled, distributed applications under a system-wide power budget, is the first work to address the unique challenge of coupled workloads and greatly improves the overall performance over any prior power capping technique. Finally, we propose *PoDD*, a hierarchical distributed dynamic power capping framework, to further address two major challenges for coupled applications. *PoDD* releases the restriction of prior application profiles, optimizes the performance using a hierarchical architecture to coordinate node-level optimization and system-level power shifting, and evaluated to be flexible (topology-oblivious), robust (resilient to system noise and tail effect). The following paragraphs summarize each project in detail.

*PUPiL* 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 – *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× 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.

*PowerShift* presents a family of techniques for shifting power among nodes running coupled, distributed applications under a system-wide power budget. Specifically, we propose a static scheme, a centralized dynamic scheme, and a distributed dynamic scheme. The static scheme is low overhead, but is not able to adjust to tail behavior or application phases. Both dynamic schemes do adjust, with the centralized scheme generally achieving higher performance—with higher overhead—while the distributed dynamic scheme trades some performance for reduced overhead and increased flexibility. Compared to prior approaches, *PowerShift* provides two advances: (1) instead of just shifting from low-power to high-power nodes, *PowerShift* will shift power away from high utilization nodes to other high utilization nodes that need power more; and (2) *PowerShift* can recognize when it is not helpful to shift power and instead it will reduce energy. Our results confirm that *PowerShift* has practical benefits, demonstrating improved performance, reduced energy, and dynamic adjustment to tail behavior and system noise. We believe the coupled workloads addressed in this paper will become increasingly important in both data centers and supercomputers because they reduce IO burden and take advantage of system scale. *PowerShift* represents one way of improving the performance, energy efficiency, and tail tolerance of this class of application.

*PoDD*, inspired by both *PUPiL* and *PowerShift*, demonstrates a hierarchical distributed dynamic power capping framework, that overcomes two major limitations in prior works: (1) dependence on offline profiles, and (2) not exploiting node-level optimization, without any code instrumentation. It delivers high performance by incorporating advanced original node-level power capping technique to coupled-workloads-aware system level dynamic shifting. It no longer requires prior knowledge of application profiles by building power performance model online. Finally, it greatly mitigates tail effects in distributed environments, is resilient to system noise and scales well to large numbers of nodes. We implement *PoDD* on a 49 node distributed system and evaluate it against 4 widely-used/state-of-the-art power control systems: *Fair*, *SLURM*, *PowerShift-S* and *PowerShift-D*. The evaluation shows *PoDD* improves mean performance over *Fair* by 28%, which outperforms *SLURM* by 21%, outperform *PowerShift-S* by 19%, and outperform *PowerShift-D* by 14%. Our evaluation of noisy environments and scalability also shows *PoDD* is resilient to system noise and predicted to have a 20X scalability over the current 49-node system. Evaluations show great flexibility of *PoDD*, that it is topology-oblivious and works well whether coupled applications are physically separate or co-located.

## 5.1 Future Work

This dissertation addresses the challenge of power capping computing systems from a single node machine to a large-scale system. As new technologies emerge, there are new challenges in this area. Here, we introduce 4 major problems.

### *5.1.1 Node-level Challenge*

Different applications have unique requirement and/or preference for system resource or architecture. General-purpose computing systems try to handle all workloads with fair performance, however, deliver sub-optimal performance and energy efficiency compared to custom

accelerators. Thus, on one hand, general-purpose machines are highly configurable (per-core on-chip DVFS, configurable memory, configurable network) to better fit different workloads, on the other hand, many specialized hardwares are built to greatly speed up specific type of workloads, e.g. GPUs for video, machine learning or other high floating-point-need workloads. The incorporation of accelerators offers great opportunities for power capping systems to further efficiency, but also creates a new challenge: the configuration space explodes (for each new custom accelerator, the configuration space grows exponentially), it is extremely difficult to find the right configuration to operate on. At the very high level, the problem to find the configuration delivering the highest performance under a certain power cap is NP-hard. Many heuristics have been proposed to target different system, workload, etc. So the challenges are, with the emerging hardwares, architectures, workloads, the power management system have to keep up with new heuristics.

### *5.1.2 Large-scale Challenge*

datacenters or supercomputers consist of hundreds of thousands of servers, so a centralized control strategy is not feasible, because a single server would not be able to handle the control work for all the servers. A hierarchical (tree structure) control strategy could be one of the solutions. So in a tree-structured hierarchical control system, the root is the global controller and the leaves are each server. For example, In *PoDD*, the controller system is predicted to be able to handle 1k nodes, which can be the controller talking to the leaves. And they will further talk to their parent controller node, and eventually, all information is gathered at the global controller. Each layer of the tree should have a different power capping strategy since they are dealing with different entities and with different constraints. Thus, building such hierarchical power control system is not easy. And one particular challenge is the necessity to mitigate network latency. As the tree scales out, it takes several hops for the information from servers to reach the global controller. This latency could be 10s up to

minutes depending on the scale. It means the global picture is greatly delayed, which makes real time decision-making infeasible. Researchers need to figure out a way to allow each layer of controllers to make real time decisions without having the real time view of the resources it is controlling.

### *5.1.3 Large-scale Reliability Challenge*

The power management system is responsible for keeping machines in normal operating state, that we have to consider rare cases where controller node itself goes down due to unexpected reasons. In order to make sure servers in the fleet running within the power budget, they need to be controlled all the time. Thus, we need to have backup controllers running with the master controller. Basically, a distributed controller pool will be needed in this case, when some controller goes offline, other can take its place. Another challenge comes with this mechanism, is classic synchronization between master controller and backup ones.

### *5.1.4 Performance Model*

Performance monitoring is very important for optimizing system efficiency. In power management system, it almost always needs some kinds of performance monitoring. While most of existing performance monitoring approaches are either requiring software level assistance (e.g. code instrumentation) or have poor accuracy (e.g. use IPS/IPC as performance indicator), to model the performance with hardware performance counter information is very beneficial: (1) it is low overhead and requires no software level knowledge, and (2) more and more hardware counters have been exposed to users, and with suitable learning techniques, the accuracy can be potentially improved.

# BIBLIOGRAPHY


[1] Mark Adler. *A parallel implementation of gzip for modern multi-processor, multi-core machines*.

[2] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, Jim Ahrens, E. Wes Bethel, Hank Childs, Jian Huang, Ken Joy, Quincey Koziol, Gerald Lofstead, Jeremy Meredith, Kenneth Moreland, George Ostrouchov, Michael Papka, Venkatram Vishwanath, Matthew Wolf, Nicholas Wright, and Kesheng Wu. *Scientific discovery at exascale: Report from the doe ascr 2011 workshop on exascale data management, analysis,and visualization*. 2011.

[3] Vlasia Anagnostopoulou, Susmit Biswas, Heba Saadeldeen, Ricardo Bianchini, Tao Yang, Diana Franklin, and FredericT. Chong. "Power-Aware Resource Allocation for CPU- and Memory-Intense Internet Services". In: *E2DC*. 2012.

[4] Peter E Bailey, Aniruddha Marathe, David K Lowenthal, Barry Rountree, and Martin Schulz. "Finding the limits of power-constrained application performance". In: *SC*. 2015.

[5] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. "Operating System Issues for Petascale Systems". In: *SIGOPS Oper. Syst. Rev.* 40.2 (Apr. 2006), pp. 29–33.

[6] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead*. 2008.

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

[8] Halil Bisgin and HN Dalfes. "Parallel clustering algorithms with application to climatology". In: *Geophysical Research Abstracts*. Vol. 10. 2008.

[9] Ron Brightwell, Ron Oldfield, Arthur B. Maccabe, and David E. Bernholdt. "Hobbes: Composition and Virtualization As the Foundations of an Extreme-scale OS/R". In: *ROSS*. 2013.

[10] Jeff Chase and Ronald P. Doyle. "Balance of Power: Energy Management for Server Clusters". In: *HotOS*. 2001.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *IISWC*. 2009.

[12] J. Chen, Alok Choudhary, S. Feldman, B. Hendrickson, C. R. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams. *Synergistic Challenges in data-intensive science and exascale computing*. 2013.

[13] Jian Chen and Lizy Kurian John. "Predictive coordination of multiple on-chip resources for chip multiprocessors". In: *ICS*. 2011.

[14] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data". In: *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. 2012, pp. 357–372.

[15] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. "Pack & Cap: adaptive DVFS and thread packing under power caps". In: *MICRO*. 2011.

[16] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. "RAPL: Memory Power Estimation and Capping". In: *ISLPED*. 2010.

[17] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters". In: *OSDI*. 2004.

[18] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *ASPLOS*. 2013.

[19] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *ASPLOS*. 2014.

[20] Gökalp Demirci, Ivana Marincic, and Henry Hoffmann. "A Divide and Conquer Algorithm for DAG Scheduling Under Power Constraints". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. 2018.

[21] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. "CoScale: Coordinating CPU and Memory System DVFS in Server Systems". In: *MICRO*. 2012.

[22] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. "MultiScale: memory system DVFS with multiple memory controllers". In: *ISLPED*. 2012.

[23] Bruno Diniz, Dorgival Guedes, Wagner Meira Jr., and Ricardo Bianchini. "Limiting the power consumption of main memory". In: *ISCA*. 2007.

[24] Daniel A. Ellsworth, Tapasya Patki, Swann Perarnau, Sangmin Seo, Abdelhalim Amer, Judicael A. Zounmevo, Rinku Gupta, Kazutomo Yoshii, Henry Hoffmann, Allen D. Malony, Martin Schulz, and Peter H. Beckman. "Systemwide Power Management with Argo". In: *IPDPS Workshops*. 2016.

[25] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling". In: *ISCA*. 2011.

[26] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. "Looking Back and Looking Forward: Power, Performance, and Upheaval". In: *Commun. ACM* 55.7 (July 2012), pp. 105–114.

[27] 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.

[28] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andr Barroso. "Power Provisioning for a Warehouse-sized Computer". In: *ISCA*. 2007.

[29] Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. "A performance-conserving approach for reducing peak power consumption in server systems". In: *ICS*. 2005.

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

[31] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. "Quanto: Tracking Energy in Networked Embedded Systems". In: *OSDI*. 2008.

[32] 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.

[33] Neha Gholkar, Frank Mueller, and Barry Rountree. "Power tuning HPC jobs on power-constrained systems". In: *PACT*. 2016.

[34] Md E. Haque, Inigo Goiri, Ricardo Bianchini, and Thu D. Nguyen. "GreenPar: Scheduling Parallel High Performance Applications in Green Datacenters". In: *ICS*. 2015.

[35] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[36] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. Morgan and Claypool Publishers, 2009.

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

[38] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments". In: *ICAC*. 2010.

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

[40] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. "A Generalized Software Framework for Accurate and Efficient Managment of Performance Goals". In: *EMSOFT*. 2013.

[41] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. "Dynamic Knobs for Responsive Power-Aware Computing". In: *ASPLOS*. 2011.

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

[43] C. Imes and H. Hoffmann. "Bard: A unified framework for managing soft timing and power constraints". In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 2016, pp. 31–38.

[44] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. "POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints". In: *RTAS*. 2015.

[45] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, Masaaki Kondo, and Ikuo Miyoshi. "Analyzing and Mitigating the Impact of Manufacturing Variability in Power-constrained Supercomputing". In: *SC*. 2015.

[46] Texas Instruments. `http://www.ti.com/product/ina231`.

[47] S.M.Z. Iqbal, Yuchen Liang, and H. Grahn. "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems". In: *Computer Architecture Letters* 9.2 (2010).

[48] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget". In: *MICRO*. 2006.

[49] M. A. Islam, X. Ren, S. Ren, A. Wierman, and X. Wang. "A market approach for handling power emergencies in multi-tenant data center". In: *HPCA*. 2016.

[50] Ian Karlin, Jeff Keasler, and Rob Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, 2013, pp. 1–9.

[51] D. E. Keyes, Lois Curfman McInnes, C. Woodward, W. D. Gropp, E. Myra, and M. Pernice. "Multiphysics Simulations: Challenges and Opportunities". In: (2012).

[52] Mohammed G. Khatib and Zvonimir Bandić. "PCAP: Performance-aware Power Capping for the Disk Drive in the Cloud". In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 227–240.

[53] Minyoung Kim, Mark-Oliver Stehr, Carolyn Talcott, Nikil Dutt, and Nalini Venkatasubramanian. "xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 11.4 (Jan. 2013).

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

[55] Matthew Lentz, James Litton, and Bobby Bhattacharjee. "Drowsy Power Management". In: *SOSP*. 2015.

[56] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM. 2015, p. 53.

[57] Xiaodong Li, Ritu Gupta, Sarita V. Adve, and Yuanyuan Zhou. "Cross-component energy management: Joint adaptation of processor and memory". In: *ACM Trans. Archit. Code Optim.* 4.3 (2007).

[58] Xiaodong Li, Zhenmin Li, Yuanyuan Zhou, and Sarita Adve. "Performance directed energy management for main memory and disks". In: *Trans. Storage* 1.3 (2005).

[59] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio an d Anant Agarwal, and Alberto Leva. "Power optimization in embedded systems via feedback control of resource allocation". In: *IEEE Transactions on Control Systems Technology (to appear)* ().

[60] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. "Performance modeling under resource constraints using deep transfer learning". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2017, p. 31.

[61] Ivana Marincic, Venkatram Vishwanath, and Henry Hoffmann. "PoLiMEr: An Energy Monitoring and Power Limiting Interface for HPC Applications". In: *E2SC*. 2017.

[62] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations". In: *MICRO*. 2011.

[63] David Meisner, Christopher M. Sadler, Luiz Andr Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. "Power management of online data-intensive services". In: *ISCA* (2011).

[64] Andreas Merkel and Frank Bellosa. "Balancing power consumption in multiprocessor systems". In: *EuroSys*. 2006.

[65] Andreas Merkel, Jan Stoess, and Frank Bellosa. "Resource-conscious scheduling for energy efficiency on multicore processors". In: *EuroSys*. 2010.

[66] Andreas Merkel, Jan Stoess, and Frank Bellosa. "Resource-conscious scheduling for energy efficiency on multicore processors". In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 153–166.

[67] N. Mishra, J. D. Lafferty, and H. Hoffmann. "ESP: A Machine Learning Approach to Predicting Application Interference". In: *ICAC*. 2017.

[68] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. "CALOREE: Learning Control for Predictable Latency and Low Energy". In: *ASPLOS*. 2018.

[69] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. "A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints". In: *ASPLOS*. 2015.

[70] Shivajit Mohapatra, R. Cornea, Hyunok Oh, K. Lee, Minyoung 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.

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

[72] Ripal Nathuji and Karsten Schwan. “VirtualPower: coordinated power management in virtualized enterprise systems”. In: *SOSP*. 2007.

[73] Nishtala, Rajiv, Marc Gonzalez Tallada, and Xavier Martorell. “A methodology to build models and predict performance-power in CMPS”. In: *ICPPW*. 2015.

[74] Owen OMalley. “Terabyte sort on apache hadoop”. In: *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)* (2008), pp. 1–3.

[75] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F. Wenisch, and Jack Underwood. “Power routing: dynamic power provisioning in the data center”. In: *ASPLOS*. 2010.

[76] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. “No ”power” struggles: coordinated multi-level power management for the data center”. In: *ASPLOS*. 2008.

[0] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. “No ”Power” Struggles: Coordinated Multi-level Power Management for the Data Center”. In: *SIGARCH Comput. Archit. News* 36.1 (Mar. 2008), pp. 48–59.

[77] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. “Thread motion: fine-grained power management for multi-core systems”. In: *ISCA*. 2009.

[78] Parthasarathy Ranganathan, Phil Leech, David E. Irwin, and Jeffrey S. Chase. “Ensemble-level Power Management for Dense Blade Servers”. In: *ISCA*. 2006.

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

[80] Haris Ribic and Yu David Liu. “AEQUITAS: Coordinated Energy Management Across Parallel Applications”. In: *ICS*. 2016.

[81] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. “Energy Management in Mobile Devices with the Cinder Operating System”. In: *EuroSys*. 2011.

[82] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kale. “Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget”. In: *SC*. 2014.

[83] Ruchira Sasanka, Christopher J. Hughes, and Sarita V. Adve. “Joint Local and Global Hardware Adaptations for Energy”. In: *ASPLOS*. 2002.

[84] L. Savoie, D. K. Lowenthal, B. R. d. Supinski, T. Islam, K. Mohror, B. Rountree, and M. Schulz. "I/O Aware Power Shifting". In: *IPDPS*. 2016.

[85] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. "METE: meeting end-to-end QoS in multicores through system-wide resource management". In: *SIGMETRICS*. 2011.

[86] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. "Power Containers: An OS Facility for Fine-grained Power and Energy Management on Multicore Servers". In: *ASPLOS*. 2013.

[87] 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.

[88] Yildiz Sinangil, Sabrina M. Neuman, Mahmut E. Sinangi, Nathan Ickes, George Bezerra, Eric Lau, Jason E. Miller, Henry Hoffmann, Srini Devadas, and Anantha P. Chandraksan. "A Self-Aware Processor SoC using Energy Monitors Integrated into Power Converters for Self-Adaptation". In: *VLSI Symposium*. 2014.

[89] SLURM. *The SLURM Workload Manager*. Online document, `https://slurm.schedmd.com/`.

[90] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. "Koala: A Platform for OS-level Power Management". In: *EuroSys*. 2009.

[91] Volker Springel. "The cosmological simulation code GADGET-2". In: *Monthly notices of the royal astronomical society* 364.4 (2005), pp. 1105–1134.

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

[93] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathost, and Zhiying Wang. "PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration". In: *MICRO*. 2015.

[94] Michael B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horesemen of the Coming Dark Silicon Apocalypse". In: *Design Automation Conference*. 2012.

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

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

[97] Vibhore Vardhan, Wanghong Yuan, Albert F. Harris III, Sarita V. Adve, Robin Kravets, Klara Nahrstedt, Daniel Grobe Sachs, and Douglas L. Jones. "GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy". In: *IJES* 4.2 (2009).

[98] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. "Conservation cores: reducing the energy of mature computations". In: *ASPLOS*. 2010.

[99] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. "Server workload analysis for power minimization using consolidation". In: *USENIX Annual technical conference*. 2009.

[100] Xiaorui Wang, Ming Chen, and Xing Fu. "MIMO Power Control for High-Density Servers in an Enclosure". In: *IEEE Transactions on Parallel and Distributed Systems* 21.10 (2010).

[101] Mark Weiser, Brent B. Welch, Alan J. Demers, and Scott Shenker. "Scheduling for Reduced CPU Energy". In: *OSDI*. 1994.

[102] Andreas Weissel, Bjórn Beutel, and Frank Bellosa. "Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications". In: *OSDI*. 2002.

[103] Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. "Scalable thread scheduling and global power management for heterogeneous many-core architectures". In: *PACT*. 2010.

[104] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. "Formal online methods for voltage/frequency control in multiple clock domain microprocessors". In: *ASPLOS*. 2004.

[105] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. "Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 607–618.

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

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

[108] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster computing with working sets." In: ().

[0] Huazhe Zhang. "A quantitative evaluation of the RAPL power control system". In: ().

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

[110] Huazhe Zhang and Henry Hoffmann. "Performance and Energy Tradeoffs for Dependent Distributed Applications Under System-wide Power Caps)". In: *ICPP*. 2018.

[111] Xiao Zhang, Rongrong Zhong, Sandhya Dwarkadas, and Kai Shen. "A Flexible Framework for Throttling-Enabled Multicore Management (TEMM)". In: *ICPP*. 2012.

[112] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. "Survey of Energy-Cognizant Scheduling Techniques". In: *IEEE Trans. Parallel Distrib. Syst.* 24.7 (2013), pp. 1447–1464.