THE UNIVERSITY OF CHICAGO

DPS: ADAPTIVE POWER MANAGEMENT FOR OVERPROVISIONED SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE DIVISION
IN CANDIDACY FOR THE DEGREE OF
MASTER IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
JIANRU DING

CHICAGO, ILLINOIS
NOV. 23TH, 2022

TABLE OF CONTENTS

ABSTRACT

Distributed computing systems are subject to system-wide power limits, which are broken down into limits for each computing node. Constant power caps is a common practice. The runtime power consumption, however, is varying. Because of the different application being executed on each cluster and the different computing loads of each application phase, at a certain time, only a number of nodes are capped and the others are wasting the power budget allocated to them, which creates the problem of how to allocate the system-wide power budget so that the limit is respected and application performance is maximized. State of art works tackle this problem with either stateless systems or model-based approaches. Stateless systems do not need prior data to deploy, but they ignore the power changing speed and sequence of each node, and often fail to assure application performance due to lagging in cap adjustment and lacking ideas of node priorities. Model-based approaches improve application performance, but the need for a lot of hardware or application feedback data to build the models renders a huge deployment overhead.

In this paper, we propose the Dynamic Power Scheduler (DPS) that sustains the low-overhead advantage of model-free approaches and provides performance-boosting power cap allocations by analyzing the dynamics in a short online power history. Compared to constant power allocation, DPS achieves at most 17.5% increase in performance for compute-intense applications and guarantees no decrease for other applications under the same power budget.

CHAPTER 1

INTRODUCTION

As the scale of distributed systems increases, power becomes the bottleneck of the system design. The US Department of Energy (DoE)'s goal for Exascale Computing is to operate in a power envelope of 20–30 MW. Messina [2017] Overprovisioning, limiting the power to nodes, has thus been proposed in distributed systems to increase scale while respecting a power budget. Patki et al. [2013] Modern processors are capable of adjusting power caps in hardware. Since the Sandy Bridge architecture, Intel processors have supported power capping in hardware with the RAPL (Running Average Power Limit) interface, by managing voltage and frequency. David et al. [2010] The ability to set the power cap for each machine allows distributed systems to be overprovisioned. In real-world cloud centers, it is rare that all machines are operating at power cap. Cloud servers operate between 10 to 50% utilization most of the time, Barroso and Hölzle [2007] allowing unused power budget on low-power machines to be migrated to those operating at power caps to increase performance with dynamic overclocking. This creates the need for a power management system that assigns the cluster-level power budget such that application performance is maximized on overprovisioned systems.

Prior work proposes a number of power management systems that broadly fall into three categories: Constant allocation systems, Model-based systems, and Model-free stateless systems.

- **Constant allocation systems** assign an equal power budget to each node. This approach ensures fairness in power allocation for an overprovisioned system and provides cluster-budget assurance. However, a constant allocation is rarely optimal as power can be migrated away from nodes that are not using it and allocated to nodes that can benefit from additional power.

- **Model-based systems** use machine learning or similar approaches to capture the relationship between measureable system behavior and power and performance tradeoffs. They use these models to predict the power and performance of various power allocations and then allocate power accordingly. Various kinds of models have been proposed for power capping, such as characterization Meisner et al. [2011], machine learning Whiteside et al. [2017], and feedback-based models Zhang and Hoffmann [2019]. These approaches adapt to changes in the operating conditions and provide significant performance improvement without violating the cluster-level power limit. But such models require a huge amount of data to train or build up the relationship between power and other metrics, which creates a non-negligible deployment overhead to migrate to new architectures or other types of services. They can also be brittle and subject to failure if the system enters an operating regime that is not captured in the training data Wang et al. [2022].

- **Model-free stateless** systems (later referred as stateless systems) keep no knowledge or reference to past operations and make decisions based solely on current power. Such systems are lightweight and ready to operate once configured. They are also robust to different scenarios as they make decisions based solely on current power usage. However, as they are stateless and model-free, they do not have the data necessary to predict future power states or to make optimal power allocation decisions.

Figure 1.1 compares 4 different power management systems (row 2 to 5) operating on a two-node (depicted in solid bars) overprovisioned system over 5 timesteps (shown on x-axis). The caps assigned at each timestep are depicted in dotted frames. In this example, both nodes could eventually use maximum node power (as shown in the top row with infinite budget); however, Node 0 increases its power needs 2 timesteps before Node 1 (shown on the first row). The constant allocation (row 2) wastes part of the budget at T2 and T3 but ensures fairness at T4. A Perfect Model-based system and a stateless system fully utilize

the budget through time T3. At T4, a perfect model-based system uses its model and optimal scheduler to reassign part of Node 0's budget to Node 1, but a stateless system sees both nodes operating at their power caps and continues with this disproportionate power allocation, starving node 1. This raises the question of whether there is a practically realizable power management system that can arrive at the same place while looking only at power data and not relying on the complicated model.

In fact, this paper argues that there is an important, yet unexplored, middle ground between model-based systems and stateless model-free systems. Specifically, *power dynamics*—changes in node power usage over time—reveal a great deal of information that can be used to allocate power budgets in a distributed system without reliance on complicated models. Consider again the example in Figure 1.1. From time T1 to T3, node 1 is increasing power, based on that power change, it is likely that it will continue to need power in the future. By tracking this power change, it is possible to recognize that power should be redistributed based on the systems power dynamics, without reliance on the complicated model.

We hereby present DPS, a model-free stateful runtime power management system for overprovisioned systems that improves performance while maintaining a cluster-level power limit. Inheriting the low-overhead advantage of the model-free stateless approaches, DPS assumes no a priori knowledge of undergoing applications and only needs to monitor power consumption data. It manages the power cap of each node in a centralized way, leveraging an online monitoring framework. Burdens on computing nodes only include reading and sending power data to the central node, and setting the power cap with RAPL to the value received from the central node, leaving most of the software on the central node. Yet unlike a stateless system, to address the challenge of accounting for *power dynamics*, DPS includes a state module that assigns a priority of either high or low to nodes based on a denoised power history. Besides, DPS has two major control modules, the MIMD (Multiplicative Increase

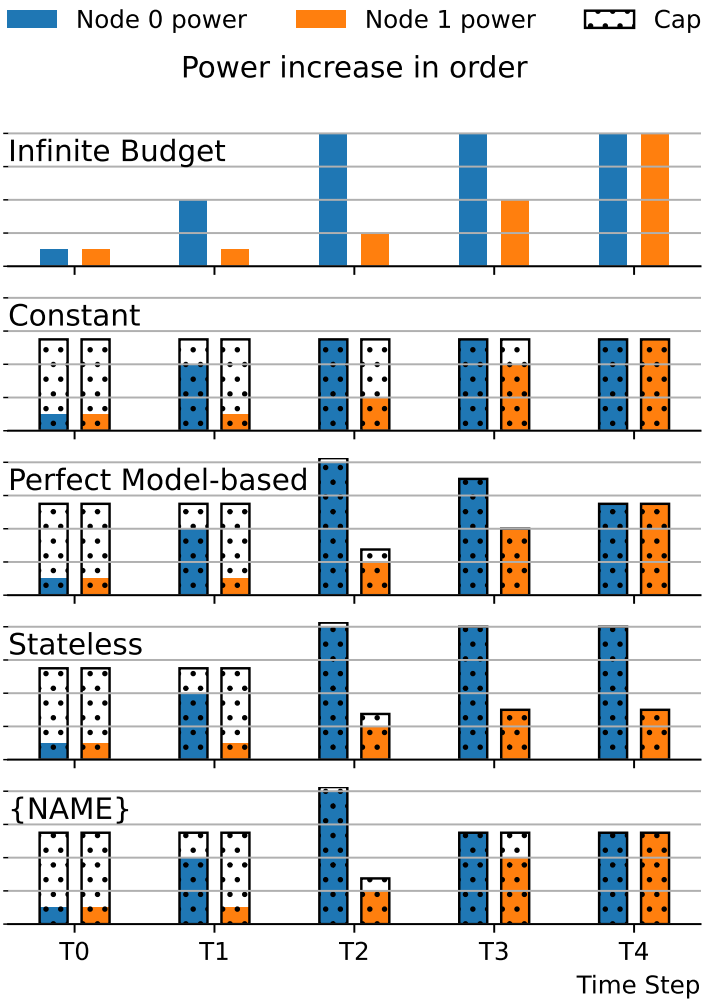

Figure 1.1: Motivational Example: Node 0 increases power sooner than Node 1. Constant allocation does not always fully utilize the budget; A perfect model-based system allocates enough budget to Node 0 till T2 and balances out as Node 1 power increases, returning to the same as constant caps; A stateless system will keep assigning enough budget to Node 0 and underallocating budget to Node 1 after T2; DPS detects the increasing trend Node 1 power at T3 and returns to an equal allocation for both nodes a time step sooner than a perfect model-based system.

Multiplicative Decrease) module, and the resetting module. The MIMD module first makes a stateless decision for power cap adjustments. Meanwhile, DPS updates a smoothed power history with the new data, and the state module updates the priority of each node with the history. Taking the priorities as input, the resetting module takes the *power dynamics* into account, adjusts the stateless decision together with the remaining budget, and makes the final decision.

In a scenario where a stateless system fails to make the right decision, DPS is able to correct that decision and further improve performance. Figure 1.1 includes DPS as the last row. DPS makes the same decision as the stateless system till T2. When the power needs exceed the power budget at T3, different from the stateless system, DPS detects the increasing trend of Node 1 power and resets both nodes to the same cap. Then at T4, it is able to continue assigning balanced caps as a perfect model-based system does.

We implement DPS and test it on an 11-node Linux/x86 server cluster with the machine learning benchmarks in the HiBench benchmark suite. Applications are launched in Apache Spark. We implement a random application scheduler that mimics a real-world cloud service utility at an average 40%. We compare DPS to constant allocation and a state-of-art model-based approach. Our results show that DPS improves compute-intensive application performance by up to 17.5% and remains similar performance for other resource-bound applications. We also compare DPS to SLURM and an oracle representing the optimal model-based system. DPS generates similar performance as SLURM and the oracle when cluster-wide power demands always exceed the budget, and guarantees a lower-bound performance as the constant allocation and out-performs SLURM by up to 22.76% when cluster-wide power demands rarely exceed the budget.

To our best knowledge, this paper proposed the first model-free stateful power management system, covering future power needs with the assumption that power changes with inertia. The major contributions are listed as followed:

- Develops a model-free stateful power management system to maximize performance under a power cap.
- Designs a experiment setup with utility of real world cloud systems.
- Evaluates this implementation on a real system with standard Spark benchmarks addressing multiple scenarios.
- Makes all scripts, code, data, from this evaluation available as open source, so others can test or extend these results.

CHAPTER 2

RELATED WORK

Power and energy have become first-order concerns for computer system design. Therefore, power management has been proposed at both node and cluster-level. We first briefly cover node-level designs before reviewing cluster level solutions (the topic of this paper). Cluster-level solutions, of course,q require some node-level support Bianchini and Rajamony [2004]. Several OS projects add support for node-level power monitoring and energy allocation Fonseca et al. [2008], Roy et al. [2011], Snowdon et al. [2009], Shen et al. [2013], Weissel et al. [2002], Yuan and Nahrstedt [2003], Vardhan et al. [2009]. Several studies profile applications or hardware-level metrics to improve energy efficiency Hoffmann [2015], Mishra et al. [2015]. With power dissipation physically unable to match consumption of modern processors, hardware designs are constrained by *dark silicon*, the part of a chip staying unpowered Esmaeilzadeh et al. [2011], and so operating within power limits becomes essential Venkatesh et al. [2010]. To support setting power limits, Intel's SandyBridge and later processors provide power management in hardware David et al. [2010]. RAPL provides an interface for specifying a power limit and then keeps the processor at or below that limit.

As we enter the exascale era, the United States Department of Energy (DoE) is anticipating a cluster-level power envelope of 10s of Megawatts (MW)Messina [2017]. In this scenario, it is anticipated that large scale systems for supercomputing and datacenters will be *overprovisioned*; that is, it will not be possible to power each server at its TDP (Thermal Design Power) while still respecting the cluster-level budget . For example, prior work proposes improving performance for HPC systems by overprovisioning and using RAPL to set node-level power limits below TDP and guarantee a tight cluster-level power limit Patki et al. [2013]. In industry, Google has deployed overprovisioned systems worldwide for several years Sakalkar et al. [2020]. The overprovisioned system has become a practical and critical approach to addressing tight cluster-level power limits in current and future cloud clusters.

As introduced in section 1, prior works on cluster-level power management systems can be generally separated into three categories and we cover each in the rest of this section.

2.1 Constant allocation systems

Some early works in cluster-level power management explore constant allocation systems with Dynamic Voltage Scaling (DVS) to reduce cluster power consumption in multi-node-scale distributed systems Ge et al. [2005], Rountree et al. [2009]. Patki et al. [2013] first explored application scalability in a strictly overprovisioned system with a constant allocation system using RAPL to set node-level power limits. Sarood et al. [2013] extended this system to include power limits on DRAM. Constant allocation systems assure cluster-level power limits and are easy to implement (given RAPL's hardware support), but using the same peak power limit for all nodes leads to sub-optimal application performance, as nodes with compute-heavy workloads run at the limits, and other nodes run below their limit, making poor use of the available power.

2.2 Model-based systems

Researchers quickly realized the limits of constant allocation schemes and turned to more sophisticated solutions to allocated power. A prominent example is the class of model-based systems that use learned models to make intelligent decisions on how to set node-level caps to meet cluster-wide budgets. For example, Lee et al. [2016] models I/O behavior to dynamically adjust per-node power budgets by shifting power from I/O bound nodes to compute bound ones. Power management with workload characterization models has been studied within various types of services, including online data-intensive services Meisner et al. [2011], parallel workloads Huang and Feng [2009], non-interactive workloads Feng et al. [2005], visualization workloads Brink et al. [2021], and microservices Hou et al. [2020]. Several works

build job performance models to manage an isolated component or coordinate multiple components Sarood et al. [2014], Georgiou et al. [2014], Labasan et al. [2017], Gholkar et al. [2016], Marathe et al. [2015]. Machine learning models are also widely studied for power management. The PowerShift project models coupled applications offline to make power cap decisions based on its model's predictions Zhang and Hoffmann [2018]. The PoDD project upgrades this idea by building machine learning models online Zhang and Hoffmann [2019]. The PANN project uses neural networks to dynamically allocate power in overprovisioned systems Whiteside et al. [2017]. Some other works build feedback-based models to improve power efficiency. Wang et al. [2008] establishes a feedback control framework to improve power efficiency with DVFS. Several feedback-based systems are proposed to adaptively apply power capping and maximize performance Wang and Chen [2008], Zhang and Hoffmann [2016], Gholkar et al. [2018]. Model-based systems generally achieve near-optimal performance under the assumption that their models have sufficient training data. Once trained, their optimality (and sometimes even ability to meet the power budgets) is dependent on the assumption that runtime workloads are drawn from the same distribution as the their training data. If the architecture or underlying workloads change significantly from this training set, the models will no longer maximize performance, and in some cases might fail to maintain the power budget.

2.3 Stateless model-free systems

Stateless model-free systems are proposed to eliminate the dependence on a well-trained model. SLURM is a widely used cluster job and resource management system that incorporates a power management plugin. SLURM's power management system adjusts maintains a cluster-wide power limit by setting node-level power caps using only the current power measurements Yoo et al. [2003]. A power management framework extending SLURM is proposed to provide safe hardware overprovisioning on a 965-node cluster at Kyushu University

Sakamoto et al. [2017]. The Argo project incorporates a conclave-node two-level stateless power management system. Ellsworth et al. [2015b,a], Perarnau et al. [2015], Ellsworth et al. [2016] The Japan supercomputer Fukagu incorporates a core retention mode to turn off idle nodes based on current processes and improve energy efficiency Kodama et al. [2020]. These systems demonstrate the practical benefits that can be derived from approaches without models. However, stateless model-free systems make power assignments only based on current power usage, forcing a greedy optimization strategy. Without models, they lack the ability to predict future power usage and escape local optima, tending to provide sub-optimal performance in an overprovisioned system.

Surprisingly, to the best of our knowledge, only model-free systems are deployed in real clusters. As of June 2022, SLURM is the default resource and job management system of 5 supercomputers in the top 10 of the Top 500 list, including Frontier, LUMI, Sunway TaihuLight, Perlmutter, and Tianhe-2A top [2022]. Fukagu still remains as the second top supercomputer in the world. Therefore this paper takes the position that a power management system that can be applied in real-world systems should not rely on a model that introduces a high deployment cost with training data and limits the operating scope. That creates a need for a model-free system that overcomes the vital drawback of a stateless system—unable to foresee future power needs and escape local optima. This paper explores the power dynamics in the power usage history and presents a model-free power management system that is capable to escape local optima, by consiering the *power dynamics*.

CHAPTER 3

POWER MANAGEMENT WITH POWER DYNAMICS

Overprovisioned systems must divide a cluster-wide power budget among individual nodes. We refer to the maximum power that a node could draw as its *power demand*. Generally, meeting a node's power demand will result in higher performance; compute-bound nodes tend to have high power demands **?**, while IO-bound nodes tend to have lower demands **?**. So, a power manager should meet a node's demands whenever possible. However, sometimes there is not sufficient budget to meet all node's demands simultaneously. In this case we would like to ensure each node is equally penalized so the power budget is fairly distributed according to demands. While conceptually simple, the process of meeting power demands is challenging in practice for two reasons: (1) the true demand cannot be measured directly as the system might be capped and (2) the demands vary dynamically, so even once the demand is known at some point, it is not clear what it will be in the future. This section first discusses the challenges that fluctuating power demands place on a power management system, specifically predicting future power demand; describes relevant power dynamics in computing systems, then shows how these power dynamics can be used to address the challenge of estimating power demand. Because different machines may support different power management scale (cores, sockets, or nodes), in the following text we refer to each part of a machine that supports power capping individually as a unit.

3.1 Observation in power demands

Power demand is continually changing. At any time, a unit's power demand is dependent on its application workload and even different phases within an application. Figure 3.1 includes three distinctive Apache Spark applications executed on a 4-node dual-socket distributed system. The system supports per-socket power cap allocation. The left column

includes three periods of the per-socket power consumption of the three applications, LDA, Bayes, LR, respectively, executed separately without power limit. So we can refer to the power consumption as the *power demand* at any time. From the figures we summarize three observations in the continuously changing power demands.

- **Power phase duration is diverse.** Applications put different computing loads on the system at different time, resulting in phases in power. Such power phases are different based on the undergoing applications. In Figure 3.1.a, LDA has a long phase ranging from second 0 to 125. Yet in Figure 3.1.c, LR has phases shorter than 10 seconds, for example from second 140 to 149. in Figure 3.1.b has phase duration in between, but each phase has different length. One longer duration ranges from second 50 to 75, but a shorter can last only 13 seconds, from second 235 to 248.

- **Peak power at each phase is diverse.** Based on the computing loads, the resources need to execute the tasks can be different, letting the system to consume different peak power throughout a phase. In Figure 3.1.a, LDA's peak power is around 160W but changing lightly. In Figure 3.1.b, Bayes's peak power is different for different phases. For example, for the phases from second 50 to 75, power increases to 165W, but for the phases from second 175 to 192, the peak power is only 110W. In Figure 3.1, the power peaks at 165W.

- **The 1st derivative of power is diverse.** When power changes, its 1st derivative can also be different based on the current computing loads. In Figure 3.1.a, power increases from 20W to 160W in 3 seconds starting from second 3, but decreases slowly from 160W to 70W in 20 seconds starting from second 97. In Figure 3.1.b, changing speeds are different in different phases. For example, for the phase from second 50 to 75, power both increases and decreases fast, but for the phase from second 195 to 225, power both increases and decreases slower.

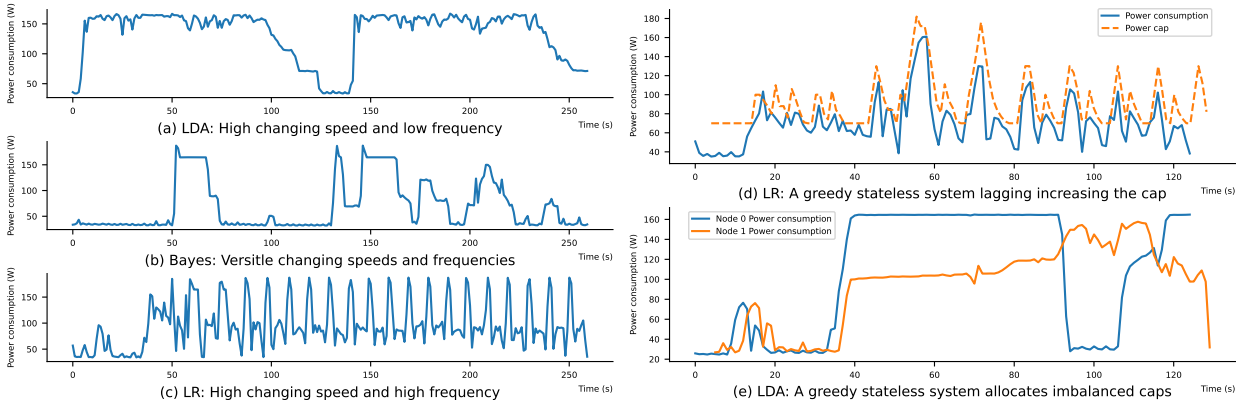

Figure 3.1: Per-socket power demand series data of different applications

3.2 Challenges in power management under a power budget

An overprovisioned system maintains a cluster-wide power budget that is not enough to power each unit at its physical upper bound. Under such a limited power budget, the diverse power demands lead to two challenges for power management.

- **Matching the power demands and the budget.** Under a limited power budget, the excess budget while meeting all units' demands can be small. To respect all units' demands as much as possible, the power management system is required to not over or under allocate the budget to any unit. Under-allocating is limiting the unit's performance, while over-allocating the budget means taking it from other units. Yet given the diverse peak power and 1st derivative of power, this requirement is challenging for a power management system to allocate the budget by matching the demand precisely.

- **Special mechanism when the power management system cannot react fast enough.** A power management system is operating in a fixed time granularity. But the diverse power phase duration can be too short for the system to react. How to react specially in this scenario can be another challenge.

Variable	Description
t	Timestep t
T_t	Time at t
dT	The granularity of timesteps
E_t	Energy consumption till t
P_t	Power consumption at t
$\frac{dP_t}{dT}$	Derivative at t
$E_t(P)$	Estimated power at t
C_t	Power cap at t

Table 3.1: Terminology.

3.3 Power dynamics

The power usage is related to the hardware performance, amounts and types of tasks to finish, and other aspects. The system load increases as new tasks come in and decreases as tasks are finished. Therefore, When we look at the power solely at any instinct, it isn't a random distribution, yet instead, power changes with inertia. That is to say, the changes in power stay with a trend for a while, meaning we can predict future power changes to some extent by only taking measurements of power consumption analyzing the changing trend residing in the most recent power history. Hereby we define the components in the power history that contribute to a more adaptive power management system together as *power dynamics*, which consist of two components, the 1st derivative and the frequency. Both terms are estimation from the measurement of power over a short period. Terminologies explaining the 1st derivative and other measurements in section 4 are included in Table 3.1.

The 1st derivative is as defined in equation 3.1. The 1st derivative indicates that power is increasing over the period if it is positive, and decreasing if negative. If power is increasing, we can at least predict that power demand will stay high at the next time step and it is demanding power budget. Similarly, if power is decreasing, we can at least predict that it will not need more power budget than its current power consumption at the next time step.

The frequency is inferred from the number of power phases during the period. If the

frequency is too high for a power management system to react to, for example, LR in Figure 3.1.c, the system can allocate the power budget differently in this edge case to ensure that whenever a unit's power peaks, it will not be capped.

$$dP_t = \frac{P_t - P_{t-1}}{dT} \tag{3.1}$$

CHAPTER 4

DPS DESIGN

DPS is a model-free stateful power management system that periodically reads power consumption data of each node and adjusts the power cap of each node accordingly while respecting a cluster-level power limit, aiming to maximize undergoing application performance in an overprovisioned system. It's designed on top of a stateless system, differed by taking *power dynamics* into consideration and attaching a state to each power-allocation unit.

4.1 Design principles

Generally, under the requirement of respecting the cluster-level *power demand*, DPS has two fundamental goals in design: overhead minimization and performance maximization. As discussed in Section 3, optimal performance in the context of a power management system is allocating the cluster-wide budget such that either all nodes' demands are met or all nodes are equally penalized when demand exceeds budget. On the other hand, overhead is referring to both operating overhead and deploying overhead. The operating overhead is the cost of making decisions, including data collection, signal transition, power cap setting, etc. A lower operating overhead contributes to a higher performance generally. The deploying overhead is the preparation cost for the system to operate as intended, which, on the contrary, is potentially beneficial to performance. For example, a machine-learning model-based power management system has high deploying overhead because it needs data and time to train the model, but once it's trained, it delivers maximized performance. The goal to minimize operating overhead leads to the first principle and the deploying overhead and performance trade-off leads to the next three principles in designing DPS as followed.

- **Minimum load on executing nodes.** A power management system has a server that received metric readings from and sends cap decisions to the clients on the executing

nodes. While applications are being executed on the executing nodes, it is necessary to keep the power management load on those nodes at minimum, minimizing the operating overhead. Such load includes reading metrics, setting caps, communication with the central node, and the client runtime.

- **No profiling/feedback model is required.** A model of any kind that involves profiling or converging on application feedbacks will increase the deploying overhead to another level compared to a model-free system. Instead of relying on a model to predict the *power demand*, DPS incorporates *power dynamics* to make close-to-perfect decisions on power caps.

- **Only power usage data is involved.** Taking any metric other than power usage requires a profiling model, which brings a huge deploying overhead. DPS needs only the minimum amount of power-related data —- power consumption itself. Such a restriction brought by the goal of minimizing deploying overhead is allowed to be met by *power dynamics* while providing close-to-perfect performance.

- **The system ensures lower-bound performance as the constant allocation** There are several challenges model-free power management systems facing. If not handled well in certain scenarios, a model-free power management system can make poor decisions on power caps such that performance is worse than dividing the cluster-wide power budget equally for each unit. However, DPS ensures a lower-bound performance as the constant allocation. As demonstrated in Section 3, DPS is able to address all the problems with *power dynamics*.

4.2 Hardware support: RAPL in DPS

DPS interacts with the hardware in 2 ways, reading power usage data and setting power caps. Both abilities are supported by Intel's RAPL system David et al. [2010]. RAPL observes

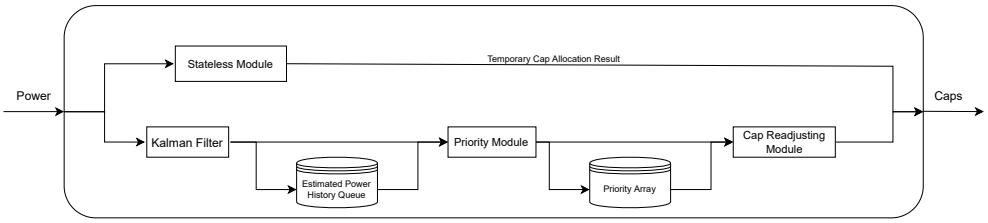

Figure 4.1: DPS control system design

various low-level hardware events and estimates energy consumption based on event counters. Still, the accuracy of RAPL's energy readings is shown to be high and the overhead is low since it is implemented in hardware Khan et al. [2018]. All RAPL counters are stored in the Model Specific Registers (MSR), among which RAPL provides two MSR's for reading the energy counter and setting the power cap. To read the power consumption between time steps, DPS accesses the MSR twice. The exact duration between the two readings is calculated with two timestamps, as shown in equation 4.1. The DPS then calculates the average power consumption throughout the time window, as shown in equation 4.2. To set the power cap, DPS writes the value into a specific location inside the MSR. Reading power usage data involves two readings from the MSR, and setting the power cap involves one write to the MSR, each renders about 5-millisecond overhead.

$$dT = T_t - T_{t-1} \tag{4.1}$$

$$P_t = \frac{E_t - E_{t-1}}{dT} \tag{4.2}$$

4.3 DPS control system

DPS consists of a server on the central node and clients on each computing node. Each Client is responsible for reading power, setting caps, and communicating with the server for

all the power capping units on the node. The control system is excluded from the client so that the only necessary load is added to the computing node, delivering minimum operating overhead. The server keeps a list of all the power capping units, including their current power caps, estimated power histories, and priorities, which we will go into detail about later in this section. The control system on the server, as shown in Figure 4.1, consists of 4 modules and two important global data sets. At each time step, the control system receives current power usage data from all computing nodes. The stateless module takes in current power and outputs a temporary cap allocation decision. The priority module takes in the power history and updates the priorities attached to all units. Due to the variance residing in the power usage data, the priority module can make mistakes easily. Therefore DPS incorperates the Kalman Filter to mitigate the effect power variance brings on the priority module's judgements. The Kalman Filter takes in power usage data and updates the estimated power history which is then fed into the priority module as input. The cap readjusting module finally takes the priorities and the stateless result as input and outputs the cap decisions. The control system then sends the new caps to all computing nodes. This section introduces each individual module and how the control system incorporates *power dynamics* to provide close-to-perfect power cap allocation.

4.3.1 Stateless Module

As indicated by the name, the stateless module alone is a stateless power management system, which takes in the current power and decides the power caps only based on the current power. This cap result serves as a temporary allocation which will be readjusted by the Cap resetting module. As shown in Algorithm 1, the stateless module is a Multiplicative-Increase-Multiplicative-Decrease (MIMD) based controller. It is inspired by SLURM's power management system Yoo et al. [2003] and only changes the power cap with different speeds in some corner cases. It maintains two thresholds for increasing and decreasing the power

cap respectively, which are at certain percentages of the current cap. If the current power of a unit is below the decreasing threshold, its power cap will be decreased by a percentage or to its current power. If the current power of a unit is above the increasing threshold, its power cap will be increased by a percentage or by what's left in the cluster-wide budget. The cap-increasing loop is done in a random manner so that no unit has priority in increasing the cap over others.

Algorithm 1: Stateless module

```
1  Function multp_inc_multp_dec(inc_threshold, dec_threshold, inc_percentile,
     dec_percentile):
       // power:  list of current power consumption of all units
       // cap:  list of current power cap of all units
       // set_flag:  list of flags for all units of whether the cap is changed
2      global power, cap, set_flag;
       // Initialize set_flag to 0
3      for u ∈ units do
4          set_flag[u] ← 0 ;
5      end
       // First loop:  decrease caps
6      for u ∈ units do
7          if power[u] < cap[u] * dec_threshold then
8              cap[u] ← max(power[u], cap[u] * dec_percentile) ;
9              set_flag[u] ← 0;
10         end
11     end
       // Second loop:  increase caps in random order
12     avail_budget ← total_budget - sum(cap);
13     for u ∈ random(units) do
14         if power[u] > cap[u] * inc_threshold then
15             tempt ← min(cap[u] * inc_percentile, avail_budget);
16             cap[u] ← tempt;
17             avail_budget ← avail_budget − tempt;
18         end
19     end
20 return
```

4.3.2 Kalman Filter

The Kalman Filter (KF) produces an estimation of a joint probability distribution over a single measurement for each time frame by taking unknown noise and variance as other variables. We use the version of a 1-d Kalman Filter that renders minimum computing loads while providing reliable estimations. As shown in algorithm 2, the KF module keeps an estimation uncertainty which is updated at the end of each time step. The estimation uncertainty is initialized with an estimated process noise and updated with it every time. At each time step, the KF module calculates the Kalman gain based on the estimation uncertainty and the uncertainty standard deviation of the current time step. Then it produces an estimation of current power which reduces the variance in power. And lastly, it updates the estimation uncertainty.

Algorithm 2: One-dimensional Kalman Filter Module

Function `kf_update`(*current_std, est_uncertainty, process_noise*)**:**
  global *power, power_history*
  for $u \in units$ **do**
    `// Remove oldest history and read the most recent one`
    $power\_history[u]$`.dequeue()`
    $est\_power_{n-1} \leftarrow power\_history[u][-1]$
    `// Calculate the Kalman Gain`
    $KG \leftarrow \frac{est\_uncertainty[u]}{est\_uncertainty[u]+current\_std^2}$
    `// Estimate power and enqueue it`
    $est\_power_n \leftarrow est\_power_{n-1} + KG * (power[u] - est\_power_{n-1})$
    $power\_history[u]$`.enqueue(`est_power_n`)`
    `// Update the history variance`
    $est\_uncertainty[u] \leftarrow est\_uncertainty[u] * (1 - KG) + process\_noise$
  end
return

4.3.3 Priority Module

The priority module estimates the two properties of power dynamics, the 1st derivative and the frequency, from the estimated power history, and attaches a priority of either high or low to each unit correspondingly. A high priority indicates the unit is executing with large loads and needs power, and a low priority indicates the unit is either idle or executing with small loads and not demanding much power. In general, a unit increasing power fast is attached with a high priority, so as a unit changing power with high frequency because it will be under-allocated if no extra power is attached to it.

Since the length of the power history is longer than what a changing trend of power can last, the module is passed with the length of history, which is used to calculate the average 1st derivative, as an argument. As an estimation of the changing frequency, which is hard to extract given the uncertainty in power, it instead calculates the number of prominent peaks in the estimated power history. The module is separated into two parts, the first part analyzing the number of prominent peaks and the second part analyzing the average 1st derivative.

As shown in algorithm 3, the priority module keeps track of whether a unit is identified as of high frequency. At each time step, the module first tries to identify the high-frequency units and attaches high priorities to them. For those who have already been identified as high-frequency units, the module checks both the number of prominent peaks and the standard deviation over the power history. If the two measures are both below their thresholds respectively, they are identified as low-frequency units. The addition check on the standard deviation is because of the uncertainty in the power and the fixed threshold in calculating the number of prominent peaks. Sometimes the number of prominent peaks can fall below the threshold yet power is still changing with high frequency. Bringing in the standard deviation can help identify such scenarios.

After attaching the priority according to the frequency, the module calculates the average

1st derivative of a shorter range in the estimated power history for all units that are identified as low frequency. Two thresholds are involved in classifying the 1st derivative, a positive one and a negative one. If the 1st derivative is above the positive threshold, a high priority is attached. If it is below the negative threshold, a low priority is attached. If it is in between, the unit's current priority is not changed. The positive threshold is used to detect a fast increase in power, and the negative threshold is used to detect a fast decrease in power. The reason behind using two thresholds is that after the power change, the unit's priority should be kept unchanged until the power changes again. For example, if a unit is set as high-priority when its power increases, it should be considered as high-priority till tasks are finished and power decreases.

4.3.4 Cap Readjusting Module

After the Stateless Module makes a temporary cap allocation decision, there could be left some unassigned cluster-wide power budget. The cap readjusting module allocates this unassigned budget to all the high-priority units. If there is no power budget left after the Stateless Module's adjustment, this module instead readjusts the caps of all the high-priority units to ensure all units increasing power in order are not penalized equally. However, if there are no large loads in the whole system at all, and every unit is demanding power lower than the initial cap, this module will ignore all the decisions made by previous modules and restore the cap of each unit to the initial cap if necessary. Such restoration makes sure of enough headroom in the power cap for any unit's incoming tasks. Therefore the Cap Readjusting Module is separated into restoring and readjusting.

As shown in algorithm 4, the restoring part of this module checks if the current power usage of each unit is under a threshold and restores the cap of each unit if so. The boolean flag indicating whether such restoration is made is further passed to the readjusting part as an argument.

Algorithm 3: Priority Module

```
Function set_priority(inc_threshold, dec_threshold, std_threshold,
  pp_threshold, direv_length):
    // duration_history: list of duration of each power reading
    // high_freq_flags: list of flags of whether the unit is demonstrating
       high-frequency power changes
    global power_history, duration_history, high_freq_flags, priority_arr
    for u ∈ units do
        pp_count ←count_prominent_peaks(power_history[u], inc_threshold)
        if not high_freq_flags[u] then
            if pp_count > pp_threshold then
                high_freq_flags[u] ← True
                priority_arr[u] ← True
                continue
            end
        else
            if pp_count < pp_threshold and std(power_history[u])
              < std_threshold then
                high_freq_flags[u] ← False
                priority_arr[u] ← False
                continue
            end
        end
        if not high_freq_flags[u] then
            avg_direv ← (power_history[−1]−power_history[−direv_length]) / sum(duration_history[u][-direv_length:])
            if avg_direv > inc_threshold then
                priority_arr[u] ← True
                continue
            end
            if avg_direv < dec_threshold then
                priority_arr[u] ← False
                continue
            end
    end
return
```

As shown in algorithm 5, the readjusting part first checks the boolean flag passed by the restoring part. If all caps are already restored, this part will be skipped at once. If, not, then it calculates the remaining unassigned budget and readjusts the caps accordingly.

If there is some remaining unassigned budget, the module will assign the budget to all high-priority units in a way that units with lower caps currently will get allocated more additional budget. Such an allocation decision is made considering two aspects. First, units with lower caps will need more budget for them to reach peak power compared to those with higher caps. Second, if these units are increasing power in order, units with lower caps will eventually be penalized harder if they are not allocated more additional budget at this time step.

On the other hand, if there is no remaining unassigned budget, the module will readjust the caps of all high-priority units in case units increasing power in order but capped are not penalized equally. The module leaves low-priority units unchanged and equalizes the caps of all high-priority units. Such a decision not only limits all high-priority units to the same power cap, but also ensures that this power cap is no lower than the initial cap, because low-priority units cannot increase power and get allocated with additional budgets before this time step. Therefore, the Cap Readjusting Module ensures lower-bound performance as the constant allocation.

4.4 State and power dynamics

DPS differs from a model-free stateless power management system by having two states —- priority of either high or low. The additional adjustment to the power budget allocation is made based on the priority of each unit. The priority of a unit is induced by the *power dynamics* in the unit's power history. *Power dynamics* include two components, yielding multiple states —- combinations of different 1st derivatives and frequencies. This section discusses how DPS packages the multiple states in *power dynamics* into the two-state priority,

Algorithm 4: Cap Readjusting Module: Restore

```
1  Function restore(inc_threshold):
2      global power, cap, set_flag
       // Restore all caps to the initial cap if no unit is consuming high power
3      restore_flag ← True
4      for u ∈ units do
5          if power[u] > initial_cap * inc_threshold then
6              restore_flag ← False
7              break
8          end
9      end
10     if restore_flag then
11         for u ∈ units do
12             cap[u] ← initial_cap
13             set_flag[u] ← True
14         end
15     end
16 return restore_flag
```

and further addresses the challenges listed in section 3.1.

The two components of *power dynamics* are used to distinguish different situations. The 1st derivative tells whether a unit is demanding power. The frequency tells whether the power oscillates too fast for a DPS to react to. When a unit is not demanding power, or when its power is not oscillating, no special adjustments are in need for its power cap. When a unit is demanding power, it needs to be allocated with extra power budgets, or at least capped at the same power as others. When a unit's power oscillates too fast, DPS cannot react to the power increases fast enough, so it needs to always allocate extra power budgets to the unit in response to cover the peak power. When there is no extra power budget, it needs to make sure that the unit's power cap stays no lower than others so that it is not penalized worse. In both situations where special adjustments are needed, the budget allocation requirements are the same —- extra budget if possible, otherwise no lower than others. Therefore DPS combines these situations into a binary state. A high priority indicates the unit is demanding power and will be ensured a power cap no lower than others when the budget is used up.

Algorithm 5: Cap Readjusting Module: Readjust

```
1  Function readjust(restore_flag):
2      global power, cap, set_flag, priority
3      if restore_flag then
4          return
5      end
6      avail_budget ← total_budget - sum(cap)
7      if avail_budget > 0 then
           // Assign all the rest budge if any left
8          high_priority_ratios ← Dictionary()
9          for u ∈ units do
10             if priority[u] then
11                 high_priority_ratios[u] ← cap[u]
12             end
13         end
14         budget_high ←sum(high_priority_ratios.values())
15         for u ∈ priority[u].keys() do
16             high_priority_ratios[u] ← budget_high/high_priority_ratios[u]
17         end
18         total ←sum(high_priority_ratios.values())
19         for u ∈ priority[u].keys() do
20             cap[u] ← min(spec_max_cap,
                avail_budget * high_priority_ratios[u]/total)
21             set_flag[u] ← True
22         end
23     else
           // Readjust all high-priority units
24         budget_high ← 0
25         count_high ← 0
26         for u ∈ units do
27             if priority[u] then
28                 budget_high ← budget_high + cap[u]
29                 count_high ← count_high + 1
30             end
31         end
32         readjusted_cap ← budget_high/count_high
33         for u ∈ units do
34             if priority[u] then
35                 cap[u] ← readjusted_cap
36                 set_flag[u] ← True
37             end
38         end
39     end
40 return
```

The Cap Readjusting Module includes all necessary mechanisms to tackle the challenges in power management based on the correctly induced priority for each unit. We discuss each challenge specifically in the following.

- **Matching the power demands and the budget.** Power demands of the units with different-speed power decrease speed and low-speed power increase can be covered well by the Stateless Module. When a unit is increasing power fast, i.e. a large positive 1st derivative, it is identified as high priority. By assigning extra power budget to such units, the Cap Readjusting Module ensures that they always get allocated enough budget for the next time even though the power is fast increasing. When cluster-wide power demand exceeds budget, the Cap Readjusting Module restores all high-priority units' power caps to the same. Even though DPS has no knowledge of the exact *power demand* at this point, it ensures a lower-bound performance as the constant allocation by matching the power demands and the budget for low-priority units and providing caps as low as the initial cap to the high-priority units.

- **Special mechanism when the power management system cannot react fast enough.** DPS identify short-power-duration units as high priority. By assuming they are in need for extra power budgets, DPS assures power caps as low as the initial cap to them and further guarantees a lower-bound performance as the constant allocation.

CHAPTER 5

EXPERIMENTAL SETUP

This section describes benchmarks and the underlying systems we use to evaluate DPS.

5.1 Platform

We experiment on the Chameleon configurable cloud computing platforms Keahey et al. [2020]. Each experiment runs on a Server node and ten client nodes. The ten client nodes include two clusters, each consisting of one master node and four worker nodes. Each node is a dual-socket system running Ubuntu 18.04 (GNU/Linux 5.4) with 2 Intel(R) Xeon(R) Gold 6240 processors, 192 GB of RAM, and hyperthreads. Each socket has 24 cores/48 threads and a 20 MB last-level cache. TurboBoost is turned on and the CPU frequency governor is set in performance mode. All hardware resources are configured as Linux default (see Table 5.1). For all the experiments in this paper, a 66.7% power limit is forced, i.e. each socket has a TDP at 165W, and the power budget is 110W per socket.

Configuration	**Settings**
Sockets	2
Cores per socket	24
Hyperthreading	2
CPU frequency	1.0-4.0 GHz
Uncore frequency	1.0-2.4 GHz
Spec Lowest Power Cap	70W
Thermal Dynamic Power (TDP)	165W

Table 5.1: Hardware resources.

5.2 Benchmarks

We test and evaluate DPS on 7 widely-used data analytic applications from the HiBench Benchmark Suite, in their Apache Spark implementations. Huang et al. [2012] These applications are labeled as power-hungry since their peak power reaches the TDP, and their performance will be affected when the power is capped. In addition, we also use 4 micro applications that are not power-hungry to build up the experimental environment. The workload sizes are in Table 5.2. The durations in Table 4 are the harmonic means of 100 runs under a 110W power cap for each socket.

Application	Data size	Duration	Power-hungry
Kmeans	224.4 GB	1467.1s	Yes
LDA	4.1 GB	1254.1s	Yes
Linear	745.1 GB	928.3s	Yes
LR	52.2 GB	499.3s	Yes
Bayes	70.1 GB	342.1s	Yes
RF	32.8 GB	415.7s	Yes
GMM	8.6 GB	2432.4s	Yes
Wordcount	3.1 GB	44.36s	No
Sort	313.5 MB	38.48s	No
Terasort	3.0 GB	54.53s	No
Repartition	3.0 GB	44.92s	No

Table 5.2: Benchmark applications.

Industry cloud servers operate between 10 to 50% utilization most of the time Barroso and Hölzle [2007]. To mirror real-world computing loads, the experimental enviroment launches

benchmark applications in the two clusters with a utility goal. The system utility is estimated with the resources used by Apache Spark executors. Applications are launched with different executor configurations based on whether they are power-hungry or not. The two executor settings are in Table 5.3. Note that the CPU utility will be always lower than this estimated utility because this utility is the percentage of resources Spark is able to use rather than actually using.

Application type	**Executor number**	**Cores per executor**
Power-hungry	48	8
Not power-hungry	1	8

Table 5.3: Spark executor settings.

CHAPTER 6

RESULTS AND EVALUATIONS

This section evaluates benchmark applications' performance in terms of throughput time with DPS as the power management system. To enable others to perform similar evaluations, we have made the software and scripts used to perform this evaluation available online. The experimental environment is set with a utility goal. As shown in Table 5.3, power-hungry applications take up 40% utility as they use all the cores in 4 worker nodes, and applications that are not power-hungry take up 8.33% utility. Therefore we construct two separate experimental goals, 50% utility, aiming to have at least one cluster executing power-hungry applications, and 80% utility, aiming to have both clusters executing power-hungry applications all the time. We begin with the 50% utility and then the 80% utility.

6.1 50% utility

Aiming for a 50% utility goal, the cluster-wide power demands rarely exceed the budget. We compare DPS with three power management systems The homogeneous allocation sets the power cap for each socket at 110W and never changes. The oracle sets the power cap for each socket at 200W and never changes since we know in advance that the cluster-wide

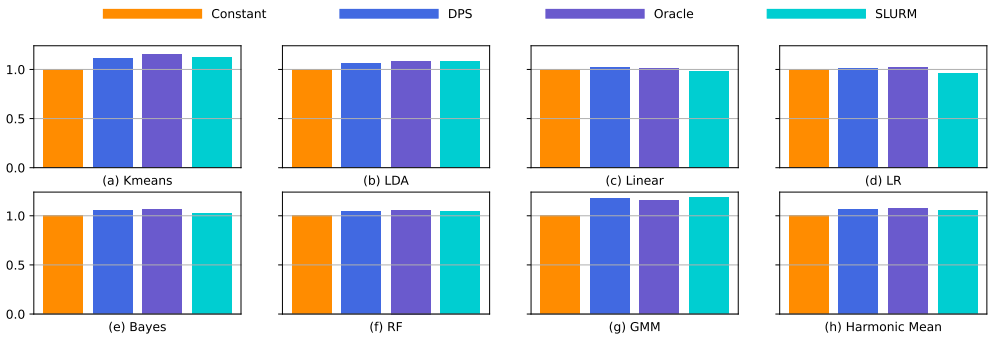

Figure 6.1: Normalized application performance with 50% utility

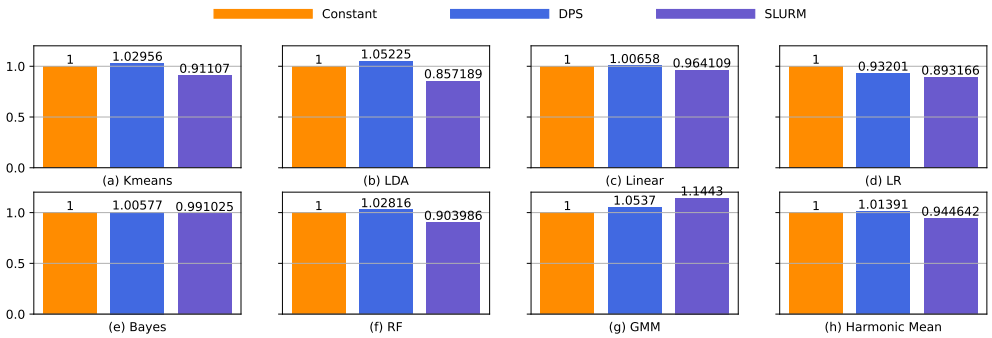

Figure 6.2: Normalized application performance with 80% utility

power budget can almost always cover the demands. We also implement SLURM's power management plugin, referred as SLURM. 7 Power-hungry benchmark applications and 4 other applications that are not power-hungry are launched at random.

The harmonic mean application performance, the reciprocal of the throughput time, normalized to the homogeneous allocation, which serves as the baseline, is reported in Figure 6.1. As shown in Figure 6.1, DPS and the oracle exceeds the baseline for all applications. DPS provides a maximum performance improvement for GMM at 17.6%. SLURM also improves performance for 5 applications at a similar extent as DPS and the oracle, except for Linear and LR, which are both with short power phases. LR's performance is decreased by 4.0% with SLURM. Yet generally, DPS and SLURM, a stateless model-free power management system reaches similar performance improvement as the oracle when the utility is low.

6.2 80% utility

As shown in Tabel 5.2, different applications have different power utilities. For example, while both as clustering applications, Kmeans spent around 50% of the time doing io and only has 50% power utility, but GMM spent around 10% of the time doing io and has 80% power utility. Therefore the pairing of the two power-hungry applications executed on the two clusters can affect the performance of both applications. We pair the most power-hungry

application, GMM with every other applications. Figure 6.2 reports the normalized harmonic mean application performance. Oracle is not included because it is impractical to implement such a power management system given the diverse power dynamics of different applications and the high variety resulted from Spark. We compare the baseline, DPS, and SLURM.

For all applications, DPS delivers either the same performance or improvements up to 5.2%. SLURM penalizes all applications except GMM. For applications with long power phases, Kmeans, LDA, and RF, SLURM slows down them by from 8.9% to 14.3%. The application with short phases, LR, is penalized by 10.7%. Bayes and Linear are not affected much by power capping. On the contrary, GMM's performance is improved by 14.4%, since it's taking the power budget from others as much as possible due to its extremely long power phases. Compared to SLURM, DPS ensures a lower-bound performance as the constant allocation and provides a single-digit performance improvement in the worst-case.

CHAPTER 7

CONCLUSION

This paper presents DPS, a model-free stateful power management system that maximizes application performance. DPS contributes a new methodology for designing a power management system by analyzing the power dynamics. Future work can explore dynamically changing the tunable parameters in DPS or combining DPS and model-based approaches to provide better performance.

CHAPTER 8

FUTURE WORK

Recent specialized cloud clusters have accelerators to handle specific types of workloads. For example, GPU-based clusters provide acceleration to diverse applications based on Deep Learning. While the accelerators take a large part in power consumption in such clusters, there lacks an efficient power management system for distributed accelerators. Power management systems for distributed systems are widely studied for traditional CPUs, and power capping methods for small-scale GPU devices cannot be applied to accelerator-based clusters. An online-training based approach, CD-GPC, is proposed to minimize the training performance degradation by power capping GPUS. Yet similar to model-based power management systems for traditional clusters, it introduces an inevitable deploying overhead. Since the same essential assumption in this paper still stands for accelerators —- performance is maximized when the power demand is respected, we can extend DPS to large-scale accelerator-based clusters with a more sophisticated design. To evaluate the extended DPS, we will compare the training performance and power utility of well-known DNN models such as AlexNet and ResNet152 with no power management system and CD-GPC.

REFERENCES

2022. Top 500 Supercomputing Site. https://top500.org/

Luiz André Barroso and Urs Hölzle. 2007. The Case for Energy-Proportional Computing. *Computer* 40, 12 (2007), 33–37. https://doi.org/10.1109/MC.2007.443

R. Bianchini and R. Rajamony. 2004. Power and energy management for server systems. *Computer* 37, 11 (2004), 68–76. https://doi.org/10.1109/MC.2004.217

Stephanie Brink, Matthew Larsen, Hank Childs, and Barry Rountree. 2021. Evaluating adaptive and predictive power management strategies for optimizing visualization performance on supercomputers. *Parallel Comput.* 104-105 (2021), 102782. https://doi.org/10.1016/j.parco.2021.102782

Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. 189–194. https://doi.org/10.1145/1840845.1840883

Daniel Ellsworth, Tapasya Patki, Swann Perarnau, Sangmin Seo, Abdelhalim Amer, Judicael Zounmevo, Rinku Gupta, Kazutomo Yoshii, Henry Hoffman, Allen Malony, Martin Schulz, and Pete Beckman. 2016. Systemwide Power Management with Argo. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1118–1121. https://doi.org/10.1109/IPDPSW.2016.81

Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. 2015a. Dynamic power sharing for higher job throughput. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/2807591.2807643

Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. 2015b. POW: System-Wide Dynamic Reallocation of Limited Power in HPC. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 145–148. https://doi.org/10.1145/2749246.2749277

Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

Xixhou Feng, Rong Ge, and K.W. Cameron. 2005. Power and energy profiling of scientific applications on distributed systems. In *19th IEEE International Parallel and Distributed Processing Symposium*. 10 pp.–. https://doi.org/10.1109/IPDPS.2005.346

Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. 2008. Quanto: Tracking Energy in Networked Embedded Systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 323–338.

R. Ge, Xizhou Feng, and K.W. Cameron. 2005. Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 34–34. https://doi.org/10.1109/SC.2005.57

Yiannis Georgiou, Thomas Cadeau, David Glesser, Danny Auble, Morris Jette, and Matthieu Hautreux. 2014. Energy Accounting and Control with SLURM Resource and Job Management System. In *Distributed Computing and Networking*, Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 96–118.

Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power tuning HPC jobs on power-constrained systems. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 179–190. https://doi.org/10.1145/2967938.2967961

Neha Gholkar, Frank Mueller, Barry Rountree, and Aniruddha Marathe. 2018. PShifter: Feedback-Based Dynamic Power Shifting within HPC Jobs for Performance. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/3208040.3208047

Henry Hoffmann. 2015. JouleGuard: Energy Guarantees for Approximate Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 198–214. https://doi.org/10.1145/2815400.2815403

Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Yang Hu, and Minyi Guo. 2020. ANT-Man: Towards Agile Power Management in the Microservice Era. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00082

S. Huang and W. Feng. 2009. Energy-Efficient Cluster Computing via Accurate Workload Characterization. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. 68–75. https://doi.org/10.1109/CCGRID.2009.88

Shengsheng Huang, Jie Huang, Yan Liu, and Jinquan Dai. 2012. HiBench : A Representative and Comprehensive Hadoop Benchmark Suite.

Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbah, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the

Chameleon Testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 219–233. https://www.usenix.org/conference/atc20/presentation/keahey

Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (mar 2018), 26 pages. https://doi.org/10.1145/3177754

Yuetsu Kodama, Tetsuya Odajima, Eishi Arima, and Mitsuhisa Sato. 2020. Evaluation of Power Management Control on the Supercomputer Fugaku. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 484–493. https://doi.org/10.1109/CLUSTER49012.2020.00069

S Labasan, M Larsen, B Rountree, and H Childs. 2017. PaViz: A Power-Adaptive Framework for Optimal Power and Performance of Scientific Visualization Algorithms. (3 2017). https://www.osti.gov/biblio/1366964

Savoie Lee, David K. Lowenthal, Bronis R. De Supinski, Tanzima Islam, Kathryn Mohror, Barry Rountree, and Martin Schulz. 2016. I/O Aware Power Shifting. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 740–749. https://doi.org/10.1109/IPDPS.2016.15

Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. 2015. A Run-Time System for Power-Constrained HPC Applications. In *High Performance Computing*, Julian M. Kunkel and Thomas Ludwig (Eds.). Springer International Publishing, Cham, 394–408.

D. Meisner, C.M. Sadler, Luiz Barroso, W. Weber, and Thomas Wenisch. 2011. Power management of Online Data-Intensive services. *Proceedings - International Symposium on Computer Architecture*, 319–330. https://doi.org/10.1145/2000064.2000103

Paul Messina. 2017. The USDOE Exascale Computing Project–Goals and Challenges.

Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. 2015. A Probabilistic Graphical Model-Based Approach for Minimizing Energy Under Performance Constraints. *SIGPLAN Not.* 50, 4 (mar 2015), 267–281. https://doi.org/10.1145/2775054.2694373

Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. 2013. Exploring Hardware Overprovisioning in Power-Constrained, High Performance Computing. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 173–182. https://doi.org/10.1145/2464996.2465009

Swann Perarnau, Rajeev Thakur, Kamil Iskra, Ken Raffenetti, Franck Cappello, Rinku Gupta, Pete Beckman, Marc Snir, Henry Hoffmann, Martin Schulz, and Barry Rountree. 2015. Distributed Monitoring and Management of Exascale Systems in the Argo Project. In *Distributed Applications and Interoperable Systems*, Alysson Bessani and Sara Bouchenak (Eds.). Springer International Publishing, Cham, 173–178.

Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 460–469. https://doi.org/10.1145/1542275.1542340

Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. 2011. Energy Management in Mobile Devices with the Cinder Operating System. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 139–152. https://doi.org/10.1145/1966445.1966459

Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. 2020. Data Center Power Oversubscription with a Medium Voltage Power Plane and Priority-Aware Capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 497–511. https://doi.org/10.1145/3373376.3378533

Ryuichi Sakamoto, Thang Cao, Masaaki Kondo, Koji Inoue, Masatsugu Ueda, Tapasya Patki, Daniel Ellsworth, Barry Rountree, and Martin Schulz. 2017. Production Hardware Overprovisioning: Real-World Performance Optimization Using an Extensible Power-Aware Resource Management Framework. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 957–966. https://doi.org/10.1109/IPDPS.2017.107

Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant Kale. 2014. Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 807–818. https://doi.org/10.1109/SC.2014.71

Osman Sarood, Akhil Langer, Laxmikant Kalé, Barry Rountree, and Bronis de Supinski. 2013. Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–8. https://doi.org/10.1109/CLUSTER.2013.6702684

Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. 2013. Power Containers: An OS Facility for Fine-Grained Power and Energy Management on

Multicore Servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2451116.2451124

David Snowdon, Etienne Sueur, Stefan Petters, and Gernot Heiser. 2009. Koala a platform for OS-level power management. *Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys'09*, 289–302. https://doi.org/10.1145/1519065.1519097

Vibhore Vardhan, Wanghong Yuan, Albert III, Sarita Adve, Robin Kravets, Klara Nahrstedt, Daniel Sachs, and Douglas Jones. 2009. GRACE-2: Integrating fine-grained application adaptation with global adaptation for saving energy. *IJES* 4 (01 2009), 152–169. https://doi.org/10.1504/IJES.2009.027939

Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 205–218. https://doi.org/10.1145/1736020.1736044

Xiaorui Wang and Ming Chen. 2008. Cluster-level feedback power control for performance optimization. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 101–110. https://doi.org/10.1109/HPCA.2008.4658631

Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. 2022. SOL: Safe on-Node Learning in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 622–634. https://doi.org/10.1145/3503222.3507704

Zhikui Wang, Cliff McCarthy, Xiaoyun Zhu, Partha Ranganathan, and Vanish Talwar. 2008. Feedback Control Algorithms for Power Management of Servers. (01 2008).

Andreas Weissel, Bjórn Beutel, and Frank Bellosa. 2002. Cooperative I/O: A Novel I/O Semantics for Energy-Aware Applications. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/osdi-02/cooperative-io-novel-io-semantics-energy-aware-applications

Will Whiteside, Shelby Funk, Aniruddha Marathe, and Barry Rountree. 2017. PANN: Power Allocation via Neural Networks Dynamic Bounded-Power Allocation in High Performance Computing. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing* (Denver, CO, USA) *(E2SC'17)*. Association for Computing Machinery, New York, NY, USA, Article 8, 7 pages. https://doi.org/10.1145/3149412.3149420

Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.

Wanghong Yuan and Klara Nahrstedt. 2003. Energy-Efficient Soft Real-Time CPU Scheduling for Mobile Multimedia Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 149–163. https://doi.org/10.1145/945445.945460

Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. *SIGARCH Comput. Archit. News* 44, 2 (mar 2016), 545–559. https://doi.org/10.1145/2980024.2872375

Huazhe Zhang and Henry Hoffmann. 2018. Performance & Energy Tradeoffs for Dependent Distributed Applications Under System-Wide Power Caps. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 67, 11 pages. https://doi.org/10.1145/3225058.3225098

Huazhe Zhang and Henry Hoffmann. 2019. PoDD: Power-Capping Dependent Distributed Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 28, 23 pages. https://doi.org/10.1145/3295500.3356174