DPS: Adaptive Power Management for Overprovisioned Systems

Jianru Ding
University of Chicago
Chicago, USA
jrding@uchicago.edu

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

ABSTRACT

Maximizing performance under a power budget is essential for HPC systems and has inspired the development of many power management frameworks. These can be broadly characterized into two groups: model-based and stateless. Model-based frameworks use machine learning to achieve good performance under a power budget but are highly dependent on the quality of the learned model and the data used to train it. Stateless frameworks are more robust and require no training, but are generally lower performance. In this paper, we propose a new framework that does not require a model, but does track a small amount of state in the form of recent power dynamics. We implement this idea and test it on a public cloud running both Spark and HPC jobs. We find when total power demand is low, our framework achieves equivalent performance to prior work, but when power demand is high it achieves mean 8% performance improvement (with no reliance on a learned model).

KEYWORDS

power-efficient design and power-management strategies, resource management, job scheduling, system interoperations and energy-aware techniques for large-scale systems

ACM Reference Format:
Jianru Ding and Henry Hoffmann. 2023. DPS: Adaptive Power Management for Overprovisioned Systems. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3581784.3607091

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 [30]. Overprovisioning—building a system which is capable of drawing power over its budget and then limiting individual nodes to respect the budget—has thus been proposed to increase scale while respecting a power budget [33]. To support this goal, modern processors include power-capping hardware. Since the Sandy Bridge architecture, Intel processors have supported the RAPL (Running Average Power Limit) interface, which takes a processor power limit and meets it by automatically managing voltage and frequency [6].

In real systems, it is rare that all machines operate at full power. For example, cloud servers typically operate between 10 to 50% utilization [3]. Thus, unused power budget on low-power machines to be migrated to those operating at their power caps to increase performance. This creates the need for a *power management system* that assigns the cluster-wide power budget dynamically based on power usage such that system performance is maximized.

Prior work proposes three broad categories of power managers:

- **Constant allocation systems** assign an equal power budget to each node. This approach is simple to implement and clearly respects the cluster-wide power budget. However, it is rarely optimal as it cannot shift power dynamically based on demand.
- **Model-based systems** use machine learning or similar approaches [29, 50, 55] to construct models that capture the relationship between measurable system behavior and power-performance tradeoffs. These systems then use their models to find performance-optimal power assignments and can dynamically adapt to changes in the operating conditions or workload. But such models require a huge amount of data to train, which creates a non-negligible deployment overhead to migrate such systems to new architectures. They can also be brittle and subject to failure if the system enters an operating regime that is not captured in the training data [46].
- **Model-free stateless systems** (later referred to as stateless systems) use no model, and in fact, keep no knowledge or reference to past operations. Instead, they make decisions based solely on instantaneous power consumption, shifting unused power from nodes operating below their caps, to nodes operating at their cap. This model-free approach works because it is assumed that nodes operating at their cap can achieve good performance with more power and nodes operating below their cap will not be hurt if their cap is lowered. A well-known example of the stateless system is the power management plugin of SLURM (later referred to solely as SLURM) which is currently world-widely deployed on many supercomputers and distributed systems [51]. 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, lacking a model, they do not have the means to predict future power states and thus have limited ability to optimize power allocations.

To illustrate the differences between these approaches (and ultimately argue for a new approach), Figure 1 compares 4 different power management systems (rows 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

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607091

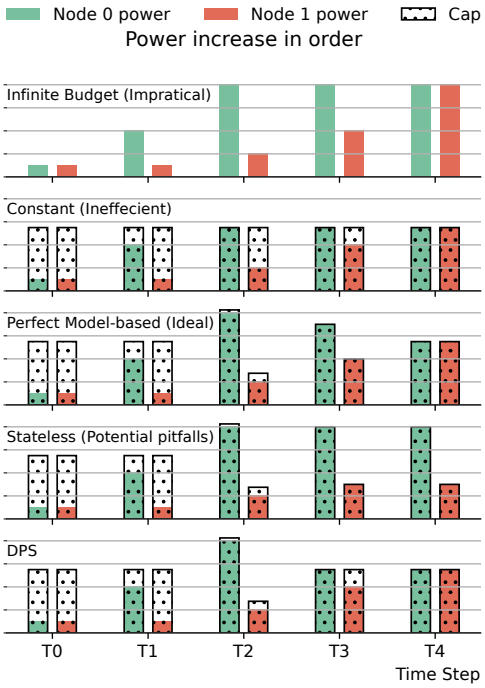

Figure 1: Motivational Example: power management systems

in dotted frames. In this example, both nodes will eventually execute at the maximum power (as shown in the top row with infinite budget); however, Node 0 increases its power 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 evenly distributes power 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 reassigns 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 as a perfect model-based system while looking only at power data.

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 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. From time T1 to T3, node 1's power increases and it is likely that it will need more power in the future. By tracking these changes, power can be redistributed based on the system's dynamics, without reliance on a complicated model.

Based on these observations, we present the Dynamic Power Scheduler (DPS), a model-free stateful power manager 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 application workloads and monitors only power data. DPS decides the power cap for each node by first using a stateless approach and then updates this initial power distribution using *power dynamics*, which include an estimate of each node's first derivative of power and its frequency of power changes. These power dynamics are then converted into a priority that modifies the initial power distribution to ensure that nodes that either (1) need power now or (2) will likely need power in the near future get that power. DPS also uses power dynamics to ensure that when power usage fluctuates too quickly to benefit from active management, it never produces worse performance than a constant allocation.

Figure 1 includes DPS as the last row. DPS makes the same decision as the stateless system until T2. When the power needs exceed the power budget at T3, different from the stateless system, DPS detects the increasing trend of Node 1's power and adjusts the caps to the anticipated power need. Then at T4, it is able to continue assigning balanced caps just 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 workloads in the HiBench benchmark suite [20] and an HPC benchmark suite, the NAS Parallel Benchmarks (NPB) [2]. The Spark applications demonstrate distinct phases in power changes while the HPC applications continuously require high power. Machine learning applications are implemented and executed in Apache Spark, and categorized into two groups based on the application's power usage. We test in three experimental setups where two clusters are run in parallel to reflect a real-world cloud service utility. In the first setup where mid-power Spark applications are paired with low-power Spark applications, our results show that DPS achieves similar mean performance to prior stateless systems, but with a maximum improvement in application performance of up to 17.5%. In the second setup where high-power and mid-power Spark workloads are paired together, DPS outperforms the power-management plugin in the wide-used resource manager, SLURM by a mean 5.4%. In the third setup where Spark applications are paired with NPB applications, DPS out-performs SLURM, by a mean 8.0% and up to 21.3%.

We acknowledge that a highly constrained set of application workloads (and inputs) could benefit from 1-time training in a model-based approach, which, however, bares a non-negligible overhead beyond such a constrained scenario, as we believe HPC (and cloud) workloads are becoming increasingly diverse and dynamic. On the contrary, DPS can be deployed on any cloud system, including commercial cloud systems like elastic clouds. DPS can also be deployed in systems where the applications are black-boxes where Model-based systems are essentially unable to operate.

To our best knowledge, this paper proposes the first model-free stateful power manager and makes the following contributions:

- Developing a model-free stateful power management system to maximize performance under a power cap. The state is simply the recent power usage changes, which we refer to as power dynamics.
- Proposing a new methodology for designing power management systems based only on recent power behavior.
- Making all scripts, code, and data available as open source, so others can test or extend these results.

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, require some node-level support [4]. Several OS projects add support for node-level power monitoring and energy allocation [11, 17, 21, 36, 41, 42, 44, 48, 52]. Several studies profile applications or hardware-level metrics to improve energy efficiency [16, 31]. To support setting power limits, Intel's SandyBridge and later processors provide power management in hardware [6]. 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)[30]. 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 [33]. In industry, Google has deployed overprovisioned systems worldwide for several years [37]. 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 this section covers each.

2.1 Constant Allocation Systems

Some early works in cluster-level power management explore constant allocation systems that set a static power cap for each node in the system [12, 35]. Patki et al. [33] 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. [40] 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 limits, making poor use of the available power.

2.2 Model-based Systems

Researchers quickly realized the limits of constant allocation and turned to solutions that dynamically allocate power. A prominent class of model-based systems uses machine learning to make decisions on how to set node-level caps to meet cluster-wide budgets. For example, Lee et al. [26] models I/O behavior to dynamically adjust per-node power budgets by shifting power from I/O bound nodes to compute bound ones. The idea of power management via a workload characterization model has been studied within various types of services, including online data-intensive services [29], parallel workloads [19], non-interactive workloads [10], visualization workloads [5], and microservices [18]. Several works build job performance models to manage an isolated component or coordinate multiple components [13, 14, 25, 27, 39]. 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 [54]. The PoDD project upgrades this idea by building machine learning models online [55]. The PANN project uses neural networks to dynamically allocate power in overprovisioned systems [50]. Some other works build feedback-based models to improve power efficiency. Wang et al. [47] 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 [15, 28, 45, 53]. 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 their training data. If the architecture or 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 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 is a stateless system that maintains a cluster-wide power limit by setting node-level power caps using only the current power measurements [51]. A framework based on SLURM provides safe hardware overprovisioning on a 965-node cluster at Kyushu University [38]. The Argo project incorporates a conclave-node two-level stateless power management system [7–9, 34]. The Fukagu supercomputer incorporates a core retention mode to turn off idle nodes based on current processes and improve energy efficiency [24]. These systems demonstrate the practical benefits of approaches that eschew models. However, stateless model-free systems make power assignments using only current power usage, forcing a greedy optimization strategy. They lack the ability to predict future power usage and escape local optima, providing sub-optimal performance, especially when workloads transition through phases of high- and low-power needs.

To the best of our knowledge, only model-free systems are deployed in production clusters. As of June 2022, SLURM is the default resource and job manager for 5 supercomputers in the top 10 of the Top 500 list, including Frontier, LUMI, Sunway TaihuLight, Perlmutter, and Tianhe-2A [1]. 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 and a reliance on well-calibrated training data. Thus, this paper explores a model-free power management system that can escape local optima by incorporating *power dynamics*.

3 POWER MANAGEMENT WITH 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 memory- and 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 nodes' 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. We note that this is a novel definition of fairness—instead of evenly distributing power among units, we seek to distribute power proportional to each node's demands. In other words, we define fairness based on a node's power demand rather than its absolute power.

Although it is difficult to quantify how much a single node's power demand is capped at a certain time, we can define a more general term of how a node's power demand is satisfied throughout the lifetime of a workload, as the node's *satisfaction*. Equation 1 shows the definition of *satisfaction* of Node n. Then we can further define the fairness of allocating the power budget to two nodes. As shown in Equation 2, the *fairness* of Node i and j is unity minus the absolute difference between the satisfaction of i and j. The *fairness* is between 0 and 1, and the higher the *fairness* is, the two nodes are getting closer percentages of the power budget they demand. As will be discussed later in section 6, a higher *fairness* will generally result in a higher harmonic mean performance of different workloads.

$$satisfaction(n) = \frac{\text{average power under current cap}}{\text{average power under no cap}} \quad (1)$$

$$fairness(i, j) = 1 - |satisfaction(i) - satisfaction(j)| \quad (2)$$

While conceptually simple, meeting power demands fairly 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. Because different machines may support different power management scales (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 Fluctuating Power Demands

Power demand is continually changing. At any time, a unit's power demand depends on its application workload and even different phases within an application. Figure 2 shows three Apache Spark applications—LDA, Bayes, and LR—and their power over time, executed separately without power limit. We refer to the power consumption that an application would exhibit without a cap as its *power demand*. From the figures we summarize three observations about these applications' power dynamics; i.e., their changes in power demand over time.

- **Power phase duration is diverse.** Applications put different computing loads on the system at different times, resulting in power phases. In Figure 2a, LDA has a long phase ranging from second 0 to 125. Figure 2c shows LR has many phases shorter than 10 seconds, for example from second 140 to 149. Figure 2b shows phase durations in between these extremes, but each phase has a different length. One longer duration ranges from second 50 to 75, but a shorter one lasts only 13 seconds, from second 235 to 248.
- **Peak power at each phase is diverse.** In Figure 2b, Bayes's peak power is different for different phases. For example, from second 50 to 75, power increases to 165W, but from second 175 to 192, the peak power is only 110W.
- **The first derivative of power is diverse.** When power changes, its first derivative also differs based on the current computing loads. In Figure 2a, 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 2b, changing speeds are different in different phases: from second 50 to 75, power both increases and decreases quickly, but from second 195 to 225, power both increases and decreases slowly.

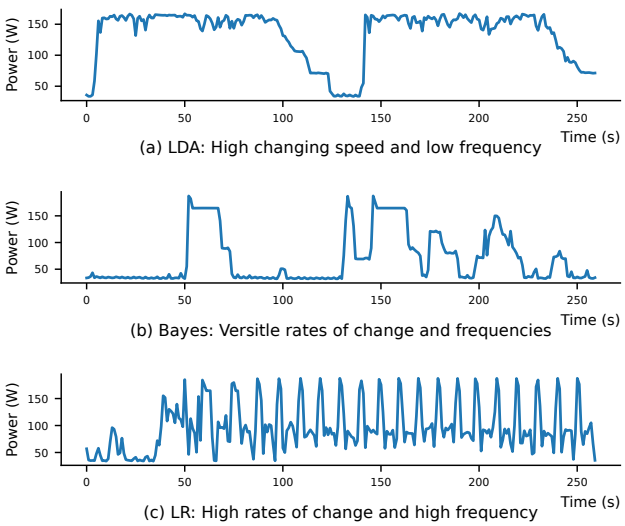

Figure 2: Power phases for different applications.

3.2 Power Management Challenges

Under a system-wide power budget, the power dynamics observed above lead to two power management challenges:

- **Matching the power demands and the budget through application phases.** To respect all units' demands fairly, the power management system should not over or under allocate the budget to any unit. Under-allocating limits that unit's performance, while over-allocating means less budget is available for other units. Yet the diversity in both peak power and power derivatives makes it hard to match the allocation to demands fairly.
- **Handling high-frequency power changes.** Power management decisions take time. But the diverse power phase duration can be too short for the system to react. For example, in Figure 2c the periods of high and low power usage

Table 1: Terminology.

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

are very short and the power manager must determine if it can reallocate power fast enough to improve performance. If it cannot, there is a risk that overall performance is worse as the system reacts too slowly, providing extra power just as the application transitions to a low-power phase and taking power just as it transitions to high power.

3.3 Power Dynamics

A key observation in this work is that power changes with inertia; power changes are locally predictable, meaning we can estimate near-future power changes from recent power history. We define the components in the power history that are useful for this estimation as *power dynamics* and identify two key components: the first derivative and the frequency. Relevant symbols and terminology are included in Table 1.

The estimation of the first derivative is as defined in equation 3. It indicates the direction and magnitude of power changes. If it is positive, power is increasing, and vice versa. The absolute value of the first derivative indicates the speed of change. Though it does not allow us to predict the exact power consumption, it implies the direction that the power is most likely to go. If power is increasing quickly, power will likely either increase or stay high in the near term. Similarly, if power is decreasing quickly, we can predict the unit will not need more power budget in the near term.

$$dP_t = \frac{P_t - P_{t-1}}{dT} \tag{3}$$

But such prediction can be wrong when the frequency change is high. Therefore the frequency is another important element in the *power dynamics*. The frequency is inferred from the number of power peaks in the recent past. The frequency may be too fast for the power management system to keep up with. In this case, it is important to know the reaction speed of the power manager, and have a separate power allocation mechanism that handles the case when the power change frequency is faster than the power management system's reaction.

4 DPS DESIGN

DPS is a model-free stateful power manager. It builds off of prior stateless approaches, but distinguishes itself by taking *power dynamics* into consideration and attaching a priority to each power-allocation unit based on the observed dynamics.

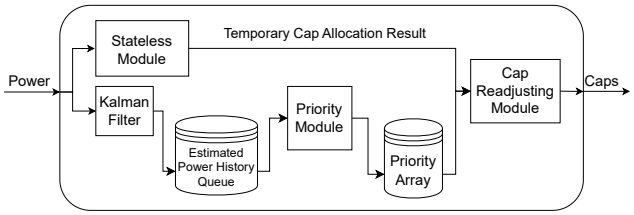

Figure 3: DPS control system design

4.1 Design Principles

DPS has two fundamental design goals: overhead minimization and performance maximization. Overhead includes both operation and deployment. The operating overhead is the cost of making decisions, including data collection, signal transition, power cap setting, etc. The deployment overhead is the preparation cost for the system to operate, for model-based systems, this can be large. In summary, the goals of low overhead (deployment and operational) and high performance lead to the following design principles:

- **Minimum load on applications.** The system should have minimal impact on application performance, including the cost of monitoring and setting power, communication, etc.
- **No initial profiling or data-driven model building.** A model of any kind that involves profiling or training data will dramatically increase overhead compared to a model-free system. Instead of relying on such a pre-trained model to predict the *power demand*, DPS incorporates *power dynamics* to make decisions on power caps.
- **Only power usage data is involved.** Making decisions based on any metric other than power usage requires additional operational and deployment overhead. DPS thus uses only unit-level power consumption data.
- **Ensures the same lower-bound performance as constant allocation.** For any power management system that makes dynamic decisions, there is a risk that it may make worse decisions than simply dividing the cluster-wide power budget equally for each unit, resulting in a lower *fairness* than what constant allocation guarantees. Indeed, prior work can reduce performance compared to constant allocation in certain scenarios (see Section 6). However, DPS ensures the same lower-bound performance as constant allocation because it incorporates *power dynamics*, so it knows when power behavior will be poorly suited to dynamic reallocation and it switches to a constant approach.

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 [6]. RAPL observes 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 [23].

Although DPS uses RAPL to read power and set the power caps, it is not tied to the RAPL interface. DPS only needs to interact with the hardware in these two ways and it can be implemented with any interface with these functionalities.

4.3 DPS Control System

DPS consists of a server on a central node and clients on each computing node (similar to SLURM's power management plugin). 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 to minimize 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 3, 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. Although RAPL has been verified by previous work to deliver reliably high accuracy, noise exists in power usage traces and we further assume pessimistically that RAPL bares certain measurement noise. Therefore we assume the exact power is not known, but is a hidden variable that must be estimated from these noisy measurements. To address such noisy measurements, DPS incorporates a Kalman Filter that takes the (potentially noisy) power measurements and updates the estimated power history, which stores the recent power dynamics for each unit. This power history is then fed into the priority module. The cap readjusting module finally takes the priorities and the caps produced by the stateless module and outputs the cap decisions, internally modifying the caps set by the stateless system based on the priorities. The control system then sends the new caps to all computing units. This section introduces each individual module and how the control system incorporates *power dynamics* to provide improved power cap allocation compared to a stateless system, without requiring any additional measurements or models.

4.3.1 Stateless Module. As indicated by the name, this module 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 basis to be readjusted by the cap resetting module. As shown in Algorithm 1, the stateless module is a Multiplicative-Increase-Multiplicative-Decrease (MIMD) based controller, inspired by SLURM's power management system [51]. It maintains two thresholds for increasing and decreasing the power cap respectively, set at percentages of the current cap. If a unit's current power 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 is 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.

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-dimensional Kalman Filter in the standard formulation that renders minimum computing loads while providing reliable estimations [49]. More specifically in DPS, at each time step, the KF module calculates the Kalman gain based on the estimation uncertainty, which is updated after each estimation, and produces an estimation of current power. The estimation will be pushed into the global estimated power history queue.

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 ;
       // First loop: decrease caps
5      for u ∈ units do
6          if power[u] < cap[u] * dec_threshold then
7              cap[u] ← max(power[u],
                 cap[u] * dec_percentile) ;
8              set_flag[u] ← 0;
       // Second loop: increase caps in random order
9      avail_budget ← total_budget - sum(cap);
10     for u ∈ random(units) do
11         if power[u] > cap[u] * inc_threshold then
12             tempt ← min(cap[u] * inc_percentile,
                 avail_budget);
13             cap[u] ← tempt;
14             avail_budget ← avail_budget − tempt;
```

4.3.3 Priority Module. The priority module estimates the two properties of power dynamics, the frequency and the first derivative, and attaches a priority of either high or low to each unit.

As shown in algorithm 2, the priority module keeps track of the changing frequency of units' power phases. At each time step, the module first identifies 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 [32] and the standard deviation over the power history. If the two measures are both below their thresholds, they are identified as low-frequency units. The additional 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, the standard deviation identifies such scenarios.

After attaching the priority according to the frequency, the module estimates the first derivative for all units that are identified as low frequency. Two thresholds are involved in classifying the first derivative, one positive and one negative. If the first 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 detects fast power increases, and the negative threshold detects fast decreases. The reason behind using two thresholds is

Algorithm 2: 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
        else
            if pp_count < pp_threshold and
              std(power_history[u]) < std_threshold then
                high_freq_flags[u] ← False
                priority_arr[u] ← False
                continue
        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
            if avg_direv < dec_threshold then
                priority_arr[u] ← False
                continue
```

Algorithm 3: Cap Readjusting Module: Restore

```
Function restore(inc_threshold):
    global power, cap, set_flag
    // Restore all caps to the initial cap if no unit is
       consuming high power
    restore_flag ← True
    for u ∈ units do
        if power[u] > initial_cap * inc_threshold then
            restore_flag ← False
            break
    if restore_flag then
        for u ∈ units do
            cap[u] ← initial_cap
            set_flag[u] ← True
```

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 until tasks are finished and power decreases.

4.3.4 Cap Readjusting Module. After the Stateless Module makes a temporary cap allocation decision, some of the cluster-wide power budget could be unassigned. 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 force a relatively high *fairness* between all units and fix any major unfairness due to the Stateless Module's random ordering. However, if there are no large loads in the whole system at all, and every unit's power demand is lower than the constant cap (power budget evenly divided among all units), this module will ignore all previous decisions and set each unit's cap to the constant cap. Such restoration makes sure there is headroom for any unit's incoming tasks. Therefore the Cap Readjusting Module is separated into restoring and readjusting.

Algorithm 3 shows that 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 restoration is made is passed to the readjusting module.

As shown in algorithm 4, 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, high-priority 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 constant cap, because low-priority units cannot increase power and get allocated with additional budgets before this time step. Therefore, the Cap Readjusting Module ensures the same lower-bound performance as the constant allocation.

4.4 Power Dynamics Address Challenges

DPS includes all necessary mechanisms to tackle the two power management challenges identified in Section 3.2:

- **Matching the power demands and the budget through application phases.** When a unit quickly increases power—i.e., a large positive first derivative—it is identified as high priority. By assigning extra power to such units, the Cap Readjusting Module ensures that they always get a fair allocation in anticipation of their future demands. When cluster-wide power demand exceeds budget, the Cap Readjusting Module restores all high-priority units' power caps to the same level. Since it is impossible for a model-free system to

Algorithm 4: Cap Readjusting Module: Readjust

```
Function readjust(restore_flag):
    global power, cap, set_flag, priority
    if restore_flag then return

    avail_budget ← total_budget - sum(cap)
    if avail_budget > 0 then
        // Assign all the rest budge if any left
        high_priority_ratios ← Dictionary()
        for u ∈ units do
            if priority[u] then
                high_priority_ratios[u] ← cap[u]
        budget_high ←sum(high_priority_ratios.values())
        for u ∈ priority[u].keys() do
            high_priority_ratios[u] ←
              budget_high/high_priority_ratios[u]
        total ←sum(high_priority_ratios.values())
        for u ∈ priority[u].keys() do
            cap[u] ← min(spec_max_cap,
              avail_budget * high_priority_ratios[u]/total)
            set_flag[u] ← True
    else
        // Readjust all high-priority units
        budget_high ← 0
        count_high ← 0
        for u ∈ units do
            if priority[u] then
                budget_high ← budget_high + cap[u]
                count_high ← count_high + 1
        readjusted_cap ← budget_high/count_high
        for u ∈ units do
            if priority[u] then
                cap[u] ← readjusted_cap
                set_flag[u] ← True
```

predict the exact *fairness* at any moment, DPS assumes that all high-priority units demand the highest power possible. While it does not guarantee that the maximum *fairness* is achieved, it avoids the situation that a high-priority unit's power budget is aggressively stripped by another one, resulting in low *satisfaction* for a high-priority unit, and forces a relatively high instantaneous *fairness*.

- **Handling high-frequency power changes.** DPS identifies high-frequency units as high priority. By assuming they are in need of extra power, DPS assures their performance does not suffer, even when it cannot react to the speed of their phase changes. This approach guarantees the same lower-bound performance as constant allocation, where prior work (that does not account for these high-frequency changes) can easily produce worse results than constant allocation (as we will see in the evaluation section).

5 EXPERIMENTAL SETUP

5.1 Hardware Evaluation Platform

We use the Chameleon cloud [22]. Each experiment runs on a Server node and ten client nodes. The client nodes include two clusters. 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. The servers support power capping at the granularity of sockets, with a Thermal Dynamic Power (TDP) of 165W per socket. For all the experiments in this paper, a cluster-wise 66.7% power limit is enforced.

5.2 Data Analytics and HPC Benchmarks

We test 7 widely-used machine learning applications (written in Apache Spark) from the HiBench Benchmark Suite [20], and 8 compute-intensive HPC benchmarks from the NAS Parallel Benchmark Suite (NPB) [2]. We also use 4 micro applications from the HiBench Benchmark Suite that consume much less power. Spark workload specs are listed in Table 2 and the NPB workload specs are listed in Table 4. Spark workloads are labeled as mid-power if the time portion above 110W is above 10%, and they are labeled as high-power if more than 2/3 of the time power is above 110W. Low-power workloads are executed with 1 Spark executor with 8 cores. The mid-power and high-power workloads are executed with 48 Spark executors with 8 cores, taking up all cores (including virtual cores) on the worker nodes. All NPB workloads are considered as high-power as over 99% of the time power is above 110W. The thread setting for the NPB workloads is the maximum allowed.

To test the power manager's ability to adapt to a wide range of situations, we create several different workload profiles by co-executing applications with differing power demands. Spark workloads have distinct power phases while NPB workloads consistently consume high power. Therefore, we experiment with the following three benchmark setups:

- **Spark low utility**: mid-power and high-power paired with low-power workloads, representing total utility below 50%.
- **Spark high utility**: A high-power workload paired with mid-power workloads, with average utility above 50%.
- **Spark NPB**: mid-power and high-power Spark workloads paired with NPB workloads, representing high utility and tight power demands.

In the first group we compare DPS to constant allocation, SLURM's power management plugin, and an oracle. In the second and third group no low-power workloads are included, and the cluster-wide power demands often exceed the power budget. Meanwhile, the variation ins Spark makes the instantaneous power demand at a specific time uncertain. Altogether, implementing an oracle in such cases is extremely difficult. Therefore we only compare DPS to constant allocation and the SLURM plugin in this group.

6 RESULTS AND EVALUATIONS

We report results for all scenarios described above. We compare a number of power managers based on their harmonic mean performance improvement over constant allocation. For reference, Tables

Table 2: Spark benchmark workloads.

Workload	Data size	Duration	Power Type	Above 110W
Wordcount	3.1 GB	44.36s	low-power	0.18%
Sort	313.5 MB	38.48s	low-power	0.10%
Terasort	3.0 GB	54.53s	low-power	0.07%
Repartition	3.0 GB	44.92s	low-power	0.20%
Kmeans	224.4 GB	1467.08s	mid-power	47.58%
LDA	4.1 GB	1254.12s	mid-power	51.54%
Linear	745.1 GB	928.36s	mid-power	14.53%
LR	52.2 GB	499.37s	mid-power	16.69%
Bayes	70.1 GB	342.18s	mid-power	33.20%
RF	32.8 GB	415.71s	mid-power	35.78%
GMM	8.6 GB	2432.43s	high-power	68.96%

Table 3: Spark benchmark computing Resources.

Power Type	# Executors	Cores per executor
low-power	1	8
mid-power	48	8
high-power	48	8

Table 4: NAS Parallel Benchmark applications.

Workload	Data size	Duration	Threads
BT (Block Tri-diagonal solver)	247.1 GB	3509.29s	144
CG (Conjugate Gradient)	21.8 GB	1839.00s	128
EP (Embarrassingly Parallel)	4 TB	6019.07s	192
FT (Fourier Transform)	400.0 GB	152.83s	128
IS (Integer Sort)	128.0 GB	416.80s	128
LU (Lower-Upper Gauss-Seidel)	296.5 GB	1895.89s	192
MG (Multi-Grid communication)	400.0 GB	143.82s	128
SP (Scalar Penta-diagonal solver)	494.2 GB	3563.23s	144

2 and 4 report the mean latency of each workload under the constant allocation scheme (each socket gets 110 W power cap). Although experiments with multiple power limits lower than the TDP can provide a more comprehensive evaluation of DPS, multiple runs of each benchmark workload are needed to diminish the performance variance and the benchmark workloads are time-consuming. We repeated at least 10 runs for each workload and spent over 1,000 hours collecting data for experiments with one power limit. We note that in all cases (and for all power managers) the power caps are respected. We therefore focus the evaluation solely on the difference in performance.

6.1 Performance: Spark Low Utility Workloads

When a mid-power workload is paired with a low-power workload, cluster-wide power demands rarely exceed the budget. As shown in Figure 4, both DPS and the oracle improve over constant allocation by 5% to 8% on average. The performance improvements differ between workloads, though. DPS provides a maximum performance improvement for GMM at 17.6%. SLURM also improves performance for 5 workloads to a similar extent as DPS and the oracle, except for Linear and LR, which both have high-frequency power phases. LR's performance is decreased by 4.0% with SLURM. The performance difference between DPS and SLURM mainly results from DPS's ability to identify and handle these high-frequency changes.

For two workloads, LDA and GMM, the oracle doesn't provide maximum performance. In fact, the oracle doesn't provide maximum performance at all times for any workload. The two reasons behind this are first, cluster-wide power demands rarely exceed the budget and DPS and SLURM are able to provide close-to-oracle performance as long as they match power caps to power demands fast enough, and second, the Spark workloads demonstrate such variable performance between different runs under the same execution condition that the average performance of DPS and SLURM may exceed that of the oracle.

6.2 Performance: Spark High Utility Workloads

When pairing mid-power and high-power Spark workloads, the cluster-wide power demands often exceed the power budget. In such cases, the performance reflects not only how fast the power management system reacts to demand changes but how both workloads are penalized when the power budget is exceeded.

Figure 5 (a) shows the harmonic mean of each mid-power workload when it is paired with a high-power workload. DPS delivers either the same performance or improvements up to 5.2% compared to the constant allocation. SLURM penalizes all workloads except GMM. For workloads with long power phases, Kmeans, LDA, and RF, SLURM slows them down by from 8.9% to 14.3%. The workloads with high frequencies, Linear and LR, are penalized by up to 7.7%.

Figure 5 (b) shows the harmonic mean of the performance of each workload and its paired GMM executions. DPS improves the mean performance of both workloads or ensures the lower-bound performance all the time, but the mean performance of SLURM is up to 8.1% below the constant allocation baseline. DPS outperforms SLUM by up to 22.76% at maximum for LDA. While SLURM always prioritizes power allocation to GMM and penalizes the other paired workload, DPS makes sure both workloads are equally penalized. By taking power from GMM and allocating it to the other workloads when necessary, DPS improves performance for both workloads and ensures the same lower-bound performance as constant allocation.

6.3 Performance: Spark & NPB Workloads

Spark workloads have phases but the NPB workloads always consume high power. That is to say, when a Spark workload is paired with an NPB workload, they compete for power budget whenever the Spark workload is not idle. Therefore these performance results reflect how well the managers balance a limited power budget.

The Spark NPB group runs Spark workloads on one cluster and NPB workloads on another, including all exhaustive pairs of mid-power and high-power Spark workloads and the NPB Workloads. Among all the pairs, DPS always outperforms SLURM by a range from 1.7% to 21.3%. Figure 6 (a) shows the result grouped by the Spark workloads. For each workload in the figure, the two bars represent the harmonic mean performance gains of the paired workloads managed by DPS and SLURM respectively, normalized to the constant allocation at 110W per socket. While performance differs between different workloads, DPS improves the performance of all the workloads. On the contrary, SLURM decreases the performance of all workloads but Linear and LR. Since the data in the

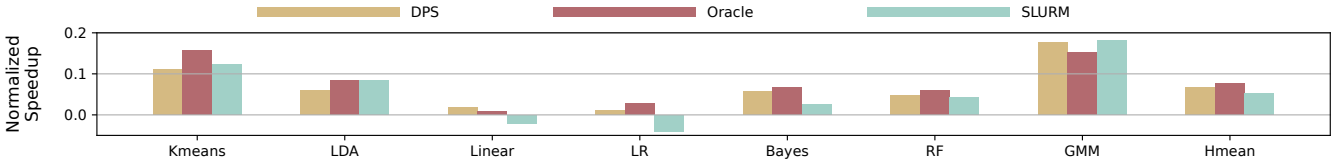

Figure 4: Hmean performance gain of the Spark low utility group normalized to under constant 110W cap.

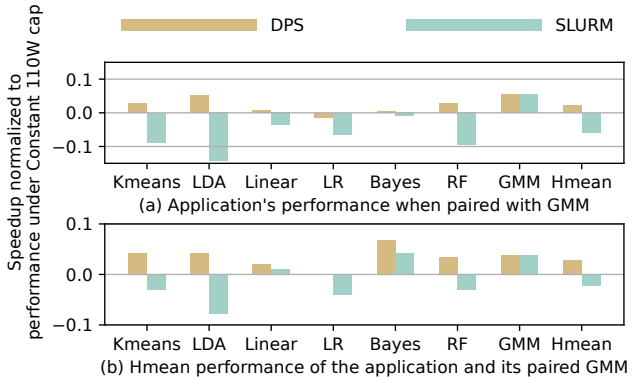

Figure 5: Normalized hmean performance gain of the Spark high utility group.

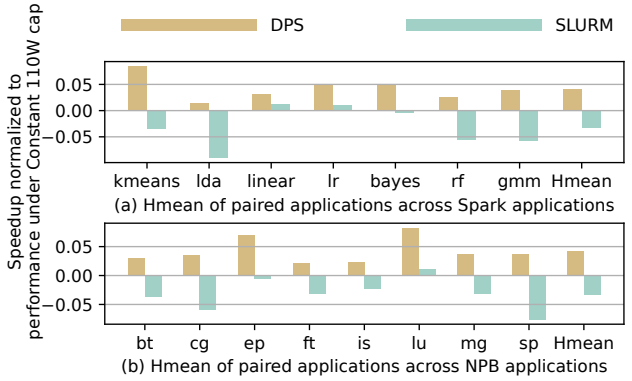

Figure 6: Normalized hmean application performance gain of the Spark NPB group.

figure represents the harmonic mean between the Spark workload and NPB workloads, it shows that the performance improvement of the NPB workloads and the performance decline of the Spark workloads under SLURM leads to an overall performance drop. For example, for the pair of GMM and EP, SLURM results in a 30.3% performance decline for GMM and 27.8% performance increase for EP, altogether resulting a harmonic mean 9.8% performance decline. DPS, on the other hand, yields 3.9% and 5.2% performance increases for GMM and EP respectively, and a harmonic mean 4.6% performance improvement. DPS outperforms SLURM on Kmeans by a max 12.4% and a min 1.9% on Linear, and an overall mean 8.0%.

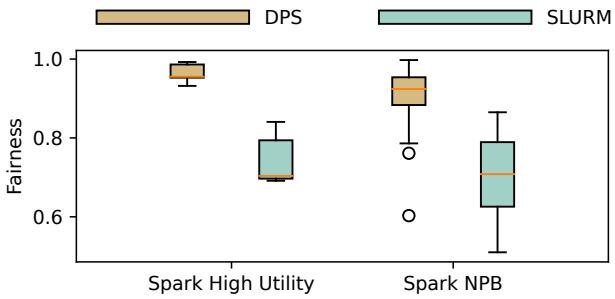

Figure 7: Fairness of the two High-utility workload groups.

Figure 6 (b) shows the result grouped by the NPB workloads. Similar to grouping by Spark workloads, DPS improves performance for all NPB workloads, and SLURM decreases performance for all workloads except LU. The performance speedup also varies among NPB workloads. For short-duration NPB workloads, such as FT and EP, when paired with long-duration Spark workloads, such as Kmeans and GMM, multiple runs are in need to match one run of the Spark workload. Although NPB workloads consistently demand high power, the short period between workloads makes each execution look like a power phase. Given a long range of time, the short-duration NPB workloads do not always demand high power. Therefore SLURM delivers better performance with short-duration NPB workloads. In general, DPS outperforms SLURM by a minimum of 4.9% on IS and a maximum 12.2% on SP.

6.4 Fairness Analysis

For the Spark Low Utility workloads, DPS, SLURM, and the oracle achieve similar *fairness*, close to 1. This is because there is generally sufficient power to cover all power demands. Yet for the Spark High Utility workloads, DPS achieves a mean 0.97 *fairness* while SLURM achieves only 0.75 mean *fairness*. Figure 7 shows the distribution of the workload *fairness* under DPS and SLURM. More specifically for each workload, DPS gets a *fairness* from 10.9% to 43.6% higher than that of SLURM. For the Spark & NPB workloads, where *fairness* has the most influence on the harmonic mean performance, DPS and SLURM achieve a mean 0.96% and 0.71% *fairness* respectively. Workload-wise, DPS always gets a higher *fairness* than SLURM, from 0.2% to 77.5%. We observe a general positive correlation between *fairness* and harmonic mean performance.

6.5 Overhead Analysis

All above reported performance data is real measurements on systems running either DPS or SLURM. The performance data includes the overhead of the power management systems themselves. DPS

and SLURM are implemented using the same Internet communication protocol, while Constant Allocation and Oracle do not have overhead since they do not change the power caps.

As discussed in section 4, DPS is designed with multiple modules among which is a stateless module. The stateless module has theoretically the same overhead as SLURM. Apart from that, the overhead of the other three modules also scales by a constant cost. As DPS keeps a short range of estimated power history of each socket, default 20 time steps, the power history can easily fit in the last-level cache even scaled to tens of thousands of nodes, taking up several megabytes. Therefore IO is not bottle-necking the turnaround time of DPS and we observe that it incurs less than 0.5% average CPU usage on the controller node with 10 client nodes. In fact, we observe that communication using the BSD socket interface takes up most of the turnaround time. The network latency is of tens of microseconds. Scaling to 1,000 nodes would only incur a several millisecond latency, which is little compared to a one-second decision loop time. Since only 3 bytes are exchanged per request with each node, scaling to even 1M nodes, requiring a network traffic size of 3MB, would put a little burden on a network bandwidth in GB/s. Such overhead is shared among all centralized power management systems and consistent with SLURM and prior work [43, 55]. Therefore we claim that DPS has negligibly more operating overhead than SLURM, a stateless system, and the controller could handle tens of thousands of nodes with no bottleneck in computation or network.

Different from model-based systems, DPS is ready to operate once deployed on a cluster. It needs at most the time of the range of estimated power history to make desired decisions, which is defaulted at 20 seconds. SLURM, on the other hand, sets power caps based on instantaneous power, so it starts to operate functionally as soon as deployed on a cluster. However, considering 20 seconds are negligible compared to the lifetime of a cluster, we claim that DPS has negligibly more deployment overhead than SLURM.

6.6 Summary of Key Results

DPS ensures the same lower-bound performance as the Constant Allocation. In low-utility scenarios where cluster-wide power demands rarely exceed the power budget, DPS yields similar performance to SLURM and an oracle. In high-utility scenarios where cluster-wide power demands often, or even always exceed the power budget, DPS outperforms SLURM from 1.7% to 21.3%. Other work on model-based approaches reports a range of 14% to 22% improvement compared to SLURM [55]. DPS yields competitive performance improvement without requiring a model.

7 CONCLUSION

This paper presents DPS, a model-free stateful power management system that improves application performance in an overprovisioned system. DPS contributes a new methodology for designing a power management system by analyzing *power dynamics*. It explores a promising middle ground between prior work on model-based systems, which require an expensive and error-prone model but yields very high performance, and model-free stateless systems, which make decisions based solely on instantaneous power usage. We hope this inspires people to further explore the use of *power dynamics* and the temporal relationships residing in power usage data, and achieve performance even closer to the model-based systems.

8 ACKNOWLEDGEMENT

This work is supported by the the National Science Foundation grant CCF-1822949, CCF-2119184, and CNS-1764039, and based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0082. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. We would like to thank Ahsan Pervaiz, Yi Ding, and the reviewers for their insightful guidance.

REFERENCES

[1] 2022. Top 500 Supercomputing Site. https://top500.org/

[2] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. 2015. 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

[9] Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. 2015. 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

[10] 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

[11] 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.

[12] 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

[13] 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.

[14] 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

[15] 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

[16] 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

[17] Henry Hoffmann and Martina Maggio. 2014. PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014*, Xiaoyun Zhu, Giuliano Casale, and Xiaohui Gu (Eds.). USENIX Association, 241–247. https://www.usenix.org/conference/icac14/technical-sessions/presentation/hoffman

[18] 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

[19] 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

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

[21] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. 2019. CoPPer: Soft Real-Time Application Performance Using Hardware Power Capping. In *2019 IEEE International Conference on Autonomic Computing, ICAC 2019, Umeå, Sweden, June 16-20, 2019*. IEEE, 31–41. https://doi.org/10.1109/ICAC.2019.00015

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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.

[28] Ivana Marincic, Venkatram Vishwanath, and Henry Hoffmann. 2020. SeeSAw: Optimizing Performance of In-Situ Analytics Applications under Power Constraints. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*. IEEE, 789–798. https://doi.org/10.1109/IPDPS47924.2020.00086

[29] 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

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

[31] 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

[32] Girish Palshikar et al. 2009. Simple algorithms for peak detection in time-series. In *Proc. 1st Int. Conf. Advanced Data Analysis, Business Analytics and Intelligence*, Vol. 122.

[33] 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

[34] 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.

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] Tapan Srivastava, Huazhe Zhang, and Henry Hoffmann. 2023. Penelope: Peer-to-Peer Power Management. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 43, 11 pages. https://doi.org/10.1145/3545008.3545047

[44] 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

[45] 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

[46] 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

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

[48] Andreas Weissel, Björn 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

[49] Greg Welch, Gary Bishop, et al. 1995. An introduction to the Kalman filter. (1995).

[50] 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

[51] 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.

[52] 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

[53] 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

[54] 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

[55] 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

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

10.5281/zenodo.8066467

ARTIFACT IDENTIFICATION

This paper proposes a power management system, named DPS, that improves undergoing application performance while respecting a cluster-wide power budget without relying on a model. DPS interacts with the hardware and assumes no knowledge of any application. The artifact includes a Python package of DPS which can be imported to deploy or test on any workload[1]. The package also includes an implementation of the power plugin for the SLURM workload manager, and an oracle as described in section 5.2. The experiments of this paper are conducted with the HiBench and the NAS Parallel Benchmark Suites. The provided artifact also includes a script to automatically reproduce all experiments in this paper.

In addition to the above, we include scripts to install and configure all required software packages, and produce plots for all figures in the paper. A detailed README.md, describing how to run experiments and reproduce each figure, is included in the description.

REPRODUCIBILITY OF EXPERIMENTS

The artifact includes a setup script that needs to be executed at the beginning to establish the connection between nodes and install all the required benchmarks and tools. Next, a script is provided to run experiments on two clusters with any workload pair as described in section 5.2, under any power cap as long as it respects the hardware specification. We also provide a script to automatically run all workload pairs using the configuration mentioned in this paper. The script will run four main experiments.

- **Constant allocation** Each benchmark workload will be tested under a constant 110W power cap per socket. This experiment will provide the baseline throughput time.
- **Low utility** Mid-power and high-power Spark workloads will be executed when paired with low-power Spark workloads. 28 pairs of workloads will be tested. The results of this experiment will be used to generate a similar plot as Figure 4.
- **High utility** Mid-power and high-power Spark workloads will be paired with each other and executed. The results of 49 pairs of workloads in this experiment will be used to generate a similar plot as Figure 5.
- **Spark NPB** Mid-power and high-power Spark workloads will be paired with NPB workloads. The results of this experiment include 56 pairs of workloads and will be used to generate a similar plot as Figure 6.

Once the experiments complete, the results can be plotted using the provided plotting scripts. Given the performance variance in Spark workloads, Spark workload in each pair is repeated at least 10 times. Consequently, the total execution time of the experimental runs is over 1,000 hours.

[1]https://github.com/jerryding95/DPS-Dynamic-Power-Scheduler

Once all pairs are executed, the experimental results can be separated by the power management systems tested, Constant Allocation, Oracle, SLURM, and DPS. The results include the start time, end time, and throughput time of each workload. The *throughput time* is the performance metric we use to plot figures in this paper. The harmonic mean throughput time of each workload in the Constant Allocation group will be the baseline. The speedup of a workload in a pair in another group, for example in Figure 5(a), is calculated as the baseline divided by the workload's harmonic mean throughput time in that group. When several pairs are grouped, for example in Figure 4, the speedup is the baseline divided by the harmonic mean of all the runs of the workload. In Figure 5(b) and Figure 6, the value is the harmonic mean of the speedup of the two paired workloads.

The experimental results also include a log of the average power during every operating cycle, the power cap set, and the priority (if DPS is running) at every operating decision for each socket. One can match the power-related data to each workload using the start and end time and further plot the time-series power-related data. We can also get the average power consumption during the lifetime of a workload. Furthermore, we can compute the $satisfaction$ of each node and the $fairness$ between the two clusters. In the artifact, we include a script that computes the $fairness$ of each pair and generates a figure similar to Figure 7.

ARTIFACT DEPENDENCIES REQUIREMENTS

As stated in section 4.2, although DPS is not tied to any power-management interface, the package is implemented to interact with the hardware with RAPL, therefore the only hardware requirement is Intel processors with RAPL available. The experiments in this paper were conducted on Cascade Lake R processors, yet the requirement is any newer generation than Sandybridge inclusively. A Linux system version newer than 18.04 inclusively is required as we did not test on older versions. To reproduce the experiment results in this paper, the following software libraries are needed:

- Python 3.6
- Java OpenJDK 8u362-b9
- Hadoop 2.7.7
- Spark 2.4.8
- OpenMPI 4.1

The experiments in this paper use two public datasets as follows:

- HiBench 7.1
- NPB 3.4.2

These two benchmark suites represent two distinct workload sets on a distributed system. The Spark workloads in the HiBench benchmark suite have distinct *power phases* while the HPC workloads in the NPB benchmark suite always consume the total power budget.

The program is mainly implemented in Python, therefore we list all dependent Python libraries below:

- Numpy 1.25
- Scipy 1.10.1

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

We have included two scripts to automatically set up the cluster and install all required libraries. More specifically, the *setup_cluster.sh* script sets up the network alias first, then downloads and installs the four required software libraries on all machines, and lastly builds the DPS Python package on all machines. The *initialize_hibench_hadoop_spark.sh* script will start the HDFS, Yarn, and Spark processes and generate the input Spark data. The total time to set up the cluster and install the libraries and packages should be under 20 minutes, given adequate downloading speed.

To execute workloads, we provide a Python script, *exp.py*. One can execute one workload with the script by specifying the workloads on two clusters respectively, the power management system, and workload repeating times in the arguments. As stated in Appendix A.2, we include a $run\_experiment.sh$ script to run all workload pairs to reproduce the experiment results in this paper, which will take over 1,000 hours. To run toy examples taking shorter time, one can change the $repeat$ variable in the $run\_experiments.sh$ script to 1 and change all NPB benchmark classes in the NPB_CLASS variable in $src/DPS/config.py$ file to S before building the package., and then execute the $run\_experiment.sh$ script. This would bring the total time down to under 3 hours.