PowerCoord: A Coordinated Power Capping Controller for Multi-CPU/GPU Servers

Reza Azimi‡, Chao Jing‡ and Sherief Reda‡
‡School of Engineering
Brown University, Providence, RI, USA, 02912
†College of Information Science and Engineering
Guilin University of Technology, Guilin, China, 541004
Email: {reza_azimi, sherief_reda}@brown.edu, jingchao@glut.edu.cn

Abstract—Modern supercomputers and cloud providers rely on server nodes that are equipped with multiple CPU sockets and general purpose GPUs (GPGPUs) to handle the high demand for intensive computations. These servers consume much higher power than commodity servers, and integrating them with power capping systems used in modern clusters presents new challenges. In this paper, we propose a new power capping controller, PowerCoord, that is specifically designed for servers with multiple CPU and GPU sockets that are running multiple jobs at a time. PowerCoord coordinates among the various power domains (e.g., CPU sockets and GPUs) inside a server to meet target power caps, while seeking to maximize throughput. Our approach also takes into consideration job deadlines and priorities. Because performance modeling for co-located jobs is error-prone, PowerCoord uses a learning method to adapt to various workloads. PowerCoord has a number of heuristic policies to allocate power among the various CPUs and GPUs, and it uses reinforcement learning to select the best policy during runtime. Based on the observed state of the system, PowerCoord shifts the distribution of selected policies. We implement our power cap controller on a real multi-CPU/GPU server with low overhead, and we demonstrate that it is able to meet target power caps while maximizing the throughput, and balancing other demands such as priorities and deadlines. Compared to prior published techniques, our results show that PowerCoord improves the throughput by an average of 14.4% under power caps.

I. INTRODUCTION

Given that the power consumption of servers vary depending on their load, supercomputers and cloud providers in general use power capping mechanisms to limit power consumption to safe levels that meet the electrical specifications (e.g., circuit breaker ratings) and the cooling infrastructure capacity [1], [2]. A centralized or hierarchical power capping system is continuously engaged and once its senses unsafe power levels, it instructs the individual server nodes to cap their power consumption to certain levels [1], [3]. A power capping controller on each server enforces its local power by scaling down the power consumption of the CPU, which in turn reduces the power consumption of the whole server [4], [5].

We observe that multi-CPU/GPU servers create three unique challenges for power capping controllers. First, these servers have multiple CPU sockets and GPUs, each with its own power domain controller (e.g., RAPL [6]), and as a result, meeting a given server power cap must involve coordination among the various domain controllers on the same server. Second, workloads often shift among the CPUs and the GPU, which requires the controller to shift power budgets between the CPU(s) and the GPU(s), while still maintaining the cap. Third, multi-CPU/GPU servers often host multiple jobs to fully utilize their resources; these jobs have various priorities and deadline requirements that have to be taken into consideration during capping to mitigate the impact of capping on performance. Based on our observations, we propose a new coordinated power capping technique that is specifically devised for server nodes with multiple CPUs and GPUs. The big challenge we address is to dynamically find the share or coordinates among the power budgets of various domain controllers in a multi-CPU/GPU server to meet target power caps, while maximizing the performance within the power cap. Our PowerCoord controller also takes into consideration execution scenarios where a mixture of jobs with various priorities and deadlines run on the server.

- Our power cap controller, PowerCoord, dynamically coordinates among the power budgets of various domain controllers in a multi-CPU/GPU server to meet target power caps, while maximizing the performance within the power cap. Our PowerCoord controller also takes into consideration execution scenarios where a mixture of jobs with various priorities and deadlines run on the server.
- We propose multiple heuristic capping policies that work for different scenarios of workload characteristics. These policies coordinate and shift the power budget among the various power domains (e.g., CPU sockets and GPUs) of the server to maximize the performance.
- Because each proposed policy works for different workload characteristic, we devise a novel BestChoice algorithm that uses reinforcement learning methodology to choose among the policies in an online fashion. Based on the observed state of the system, BestChoice learns to shift the distribution of selected policies and automates the process of matching workload characteristics to policy selection. BestChoice continuously updates itself with performance feedback from the server.
- We implement our power capping controller on a server with two Xeon CPU sockets, a Nvidia P40 GPU card and

This work was partially supported by NSF (Grant No. 1438958), the National Natural Science Foundation of China (Grant No. 61802085).
128 GB of DDR4 DRAM. Our controller shows effective operation with negligible overhead across a wide range of workload and power capping scenarios. Compared to a previous approach (POWsched [7]), our approach delivers an average of 14.4% job throughput improvement.

The organization of this paper is as follows. In Section II, we use real workload and power traces from a multi-CPU/GPU server to motivate our work. In the methodology section III, we describe the main components of our PowerCoord controller. In Section IV, we provide a comprehensive experimental evaluation of our technique. In the related work section, Section V, we contrast our technique against previous work, and finally we summarize the main conclusions of our work in Section VI.

II. MOTIVATION

Servers with multiple CPU sockets and multiple GPU cards typically run a mixture of jobs to increase their resource efficiency [8], [9], [10]. A typical job can rarely use all the available resources of a modern server. Figure 1a shows the power consumption of our server with two CPU sockets and a GPU card when a mixture of jobs are running on the node over time and no power capping is enforced. Jobs are submitted at different times and resources can get idle at some points in time. Figure 1b shows the breakdown of power consumption between each CPU socket and GPU domains. The figure shows that the power consumption of each domain depends on the resource utilization and characteristics of the running jobs.

When the total server power needs to be capped, power consumption of each CPU and GPU needs to be reduced. In practice, each power domain (e.g., CPU socket or GPU) has a power controller to actuate a target budget. However, the challenge is to coordinate the power budget of all domains to maximize the performance of the server and meet the performance requirement of various jobs. To maximize the performance, the power budget must be shifted dynamically from idle domains to the active domains that require more power. When all domains are busy and power needs to be capped, the power budget must be divided based on the scheduled jobs on each domain.

Depending on the resource usage and characteristics of running jobs, power capping affects the performance of various workload differently. Figure 2 shows the normalized throughput of various CPU and GPU benchmarks running on our server alone for different caps. A workload’s performance could be modeled as a function of power cap when a single benchmark is running on the system [11], [2]; however, when multiple jobs are co-located on the system, explicit mathematical modeling of performance is not practical for two reasons:

1) Based on the job scheduling decisions, different mixture of jobs are running on the system. As the number of jobs increases, either complex workload classification is needed or a different model is required for each job mixture, which is not a scalable approach.

2) Models are error-prone and require to be updated for any software and hardware changes.

Table I shows the runtime of a CPU and a GPU benchmark co-located under a power cap. Results are reported in form of the runtime of (CPU, GPU) benchmarks and normalized to the runtime of benchmark alone under the same power cap. Only two benchmarks are co-located together for each experiment: a CPU only and a GPU benchmark.

<table>
<thead>
<tr>
<th>CPU benchmark</th>
<th>GPU benchmark</th>
<th>bh</th>
<th>cloverleaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>ft</td>
<td>(2.1x, 1.3x)</td>
<td>(2.7x, 1.0x)</td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td>(1.4x, 1.1x)</td>
<td>(2.0x, 1.2x)</td>
<td></td>
</tr>
</tbody>
</table>

Table I shows the runtime of benchmarks executing alone. *Jacobi* and *tealeaf* use the GPU and a single CPU core, while *ft* and *ep* use 16 CPU cores. Normalized performance is defined as throughput ratio of benchmarks with and without power capping.

Fig. 1. (a) Total power consumption of a multi-CPU/GPU server when running a mixture of jobs over time, and (b) power consumption of each CPU socket and the GPU. No power capping is enforced.

Fig. 2. Effect of power capping on different benchmarks executing alone. *Jacobi* and *tealeaf* use the GPU and a single CPU core, while *ft* and *ep* use 16 CPU cores. Normalized performance is defined as throughput ratio of benchmarks with and without power capping.
where the priority of job \(a\) set of \(n\) subject to the power constraints. More specifically, we assume the problem, where the goal is to maximize the performance optimization of the power cap.

In multi-CPU/GPU servers, each CPU socket or GPU card is an independent power domain that has its independent power controller to monitor and actuate a its power budget. Our controller, PowerCoord, receives the total power cap for the server from the cluster’s power coordinator as well as the running jobs’ information. PowerCoord also monitors the power consumption of the server, and the power consumption of each domain. PowerCoord then determines the power budget for each power domain so that the server’s total cap is met, while seeking to maximize throughput. The controller of each power domain receives its budget from PowerCoord and enforces it locally.

Both capping and scheduling decisions are hierarchical decisions. PowerCoord focuses on intra-server power capping, where the server’s total power cap is given as input from cluster-level power capping controller (e.g. Dynamo or DPC [1], [3]) that is responsible for coordinating power among different servers. PowerCoord focuses on the server-level optimization of the power cap.

Power capping is formulated as a constrained maximization problem, where the goal is to maximize the performance subject to the power constraints. More specifically, we assume a set of \(n\) running Jobs = \(\{job_1, \ldots, job_n\}\) over a period of time. Let \(f_i, d_i,\) and \(pr_i\) be the finish time, deadline, and priority of \(job_i\) respectively. The deadline is defined based on the runtime of the job and does not include the queuing time in the cluster’s scheduler. At any time, multiple jobs can be running on the server on its set of power domains \(H:\)

\[
H \triangleq \{CPU_1, \ldots, CPU_m, GPU_1, \ldots, GPU_k\},
\]

where \(m\) denotes the number of CPU sockets and \(k\) denotes the number of GPUs.

Let \(p_i\) and \(b_j\) be the power consumption and budget of power domain \(j \in H\) respectively. Each power domain has an independent controller that enforces the budget (\(p_j \leq b_j\)). Let \(p_j^{\text{min}}\) and \(p_j^{\text{max}}\) be the minimum and maximum possible power consumption of domain \(j \in H\) respectively, and let \(C\) and \(P\) denote the total power cap and total power consumption of the server. The total power consumption of the server is the sum of its power domains plus the power consumption of other components such as DRAM, motherboard, fans denoted as \(p_{\text{others}}\). The goal is to allocate the power budget for the domains such that the total power consumption of the entire server never exceeds the total cap \(C\). The proposed power capping problem is formulated as:

\[
\begin{align*}
\max_{b_j \in H} & \sum_{i=1}^{n} pr_i \times \mathbb{1}\left(f_i(\{b_j | j \in H\}) \leq d_i\right) \\
\text{subject to:} & \sum_{j \in H} b_j \leq C - p_{\text{others}}, \quad \forall j \in H, \quad p_j^{\text{min}} \leq b_j \leq p_j^{\text{max}}.
\end{align*}
\]

where \(f_i(\{b_j | j \in H\})\) is the finish time of the \(job_i\) which is the function of the power budgets on different power domains. The function \(\mathbb{1}(f_i(\{b_j | j \in H\}) \leq d_i)\) is equal to 1 if \(f_i \leq d_i\); i.e., \(job_i \in Jobs\) finished before its deadline; otherwise, it is equal to 0. Equation (1b) is the constraint on total power consumption of the server. Equation (1c) defines the upper and lower bound of the budgets to be feasible to actuate.

Solving this optimization problem requires complex performance models and is error-prone. To solve the proposed problem, heuristic algorithms must be used in practice in form of policies. A policy selection algorithm then chooses a suitable policy based on the observed state of the server. The proposed PowerCoord controller has the following main components:

1) A set of heuristic Policies where each policy coordinates the total power cap among different power domains while trying to maximize the performance. We observed that different policies are required as each policy performs well for different workload characteristics.

2) Based on the observed state of the server, our BestChoice algorithm adaptively shifts the distribution of selecting Policies to coordinate the power.

A. Proposed Policies

To coordinate the power budget among different power domains while maximizing the performance, we propose the following policies. These policies use different techniques and parameters to allocate the power budget.

1) Uniform policy (U): Uniform power allocation divides the available budget \((B = C - p_{\text{others}})\) uniformly among all power domains. The main motivation for the uniform policy is to show a baseline of achievable performance.

2) Power proportional policy (P): The goal of the power proportional policy is to shift power budget from the domains that are not consuming their allocated power budget to the ones that are consuming their budget. We define \(\alpha_j\) as the ratio between the power consumption of domain \(j\) to its
budget. Power proportional policy allocates the budget, $B$, proportional to $\alpha_j^p$ values. Thus,
\[
\alpha_j^p = \frac{p_j}{b_j},
\]
\[
b_j = \min \left( p_j^\text{min} + \frac{\alpha_j^p}{\sum_{l \in H} \alpha_l^p} \times (B - \sum_{l \in H} p_l^\text{min}), p_j^\text{max} \right).
\]

The intuition here is that if a power domain does not currently consume its allocated budget, then this policy reduces its power budget and allocates more power to the domains that are currently consuming power near their budget. If there is a budget surplus after all budgets are calculated, i.e., $B - \sum_{j \in H} b_j > 0$, this policy allocates the surplus to the domains that have budgets below their maximum power consumption.

3) Power-Deadline proportional policy (PD): The goal of the power-deadline policy is to allocate more power to the domains that are running jobs closer to their deadline. To do so, we first look at which domains are not idle and define $H_{\text{active}}$ as the set of power domains that are running a portion of a job. We define $\alpha_j^d$ as the ratio that defines how critical is the state of jobs on domain $j \in H_{\text{active}}$. A power domain that is running a job closer to its deadline is considered more critical based on this policy, and thus has greater value of $\alpha_j^d$. If the power domain is idle $j \notin H_{\text{active}}$, then $\alpha_j^d$ is zero. If all domains are idle $|H_{\text{active}}| = 0$, we assign uniform ratios to all $\alpha_j^d = \frac{1}{|H|}$. Let $Jobs_j$ denote the set of jobs that are running on domain $j \in H_{\text{active}}$. If a job has a set of CPU cores on two sockets, then it exists on both sockets job sets. Let $t_i$ denote the runtime of $job_i$; thus, the remaining runtime ratio $(d_i - t_i)/d_i$ indicates how close a job is to meet its deadline, where it is equal to 1 when a job gets scheduled and decreases to zero as the job approaches its deadline. Let $m_j$ denote the remaining runtime ratio of the closest job to its deadline. We calculate the budget for each controller as follows.

\[
m_j = \min_{i \in Jobs_j} \frac{d_i - t_i}{d_i},
\]
\[
\alpha_j^d = \begin{cases} 
\frac{e^{(-\rho m_j)}}{\sum_{l \in H_{\text{active}}} e^{(-\rho m_l)}} & \text{if } j \in H_{\text{active}}, \\
0 & \text{otherwise}
\end{cases}
\]
\[
b_j = \min \left( p_j^\text{min} + \frac{\alpha_j^p \times \alpha_j^d}{\sum_{l \in H} \alpha_l^p \times \alpha_l^d} \times (B - \sum_{l \in H} p_l^\text{min}), p_j^\text{max} \right),
\]

where $\rho$ is a parameter that controls the sensitivity of $\alpha_j^d$ to $m_j$. We use the same definition of $\alpha_j^p$ as the one in power proportional policy to consider the power needs of different domains. After all budgets are calculated, we use the same clean-up procedure as power proportional policy to make sure all of the power budget is allocated to the domains.

4) Power-Deadline-Priority proportional policy (PDP): The goal of power-deadline-priority policy is to consider both the priorities of the jobs and their deadlines. PDP allocates more power to the domains that are running high priority jobs and are closer to their deadlines. Let $\alpha_j^{dp}$ denote the ratio that defines how critical is the state of jobs on power domain $j \in H_{\text{active}}$. The power domains that are running closer to deadline jobs with high priorities, have greater $\alpha_j^{dp}$ value and receive greater portion of budget. Similar to PD policy, if all computing units are idle $|H_{\text{active}}| = 0$, we assign uniform ratios to all $\alpha_j^{dp} = \frac{1}{|H|}$. We use the same definition of $m_j$ as the PD policy and define $ap_j$ as the average priority of all jobs running on domain $j$. For power domains that are running more than one job such as CPU sockets, the average priority is calculated based on the number of cores each job is using from the socket. If $c_{ij}$ denotes the number of CPU cores that $job_i$ is utilizing from domain $j$, then we calculate the average priority of $Jobs_j$ based on their $c_{ij}$:

\[
ap_j = \frac{\sum_{i \in Jobs_j} p_i \times c_{ij}}{\sum_{i \in Jobs_j} c_{ij}},
\]
\[
\alpha_j^{dp} = \begin{cases} 
\frac{\sum_{l \in H_{\text{active}}} m_j p_l \times \frac{\alpha_j^d}{\alpha_l^d}}{|H_{\text{active}}| \times \sum_{l \in H_{\text{active}}} \frac{\alpha_j^d}{\alpha_l^d}} & \text{if } j \in H_{\text{active}}, \\
0 & \text{otherwise}
\end{cases}
\]
\[
b_j = \min \left( p_j^\text{min} + \frac{\alpha_j^p \times \alpha_j^{dp}}{\sum_{l \in H} \alpha_l^p \times \alpha_l^{dp}} \times (B - \sum_{l \in H} p_l^\text{min}), p_j^\text{max} \right),
\]

where $\tau$ is the sensitivity parameter. We use the same definition of $\alpha_j^{dp}$ as the one in power proportional policy to consider the power needs of different domains. We used different mathematical function to account the job deadline in $\alpha_j^{dp}$ compared to $\alpha_j^d$ to have heterogeneous policies. After all budgets are calculated, we allocate the surplus as in the case of the power proportional policy to make sure all the budget is allocated to the domains.

B. BestChoice: A Best Policy Selector

Because a large space of system parameters must be considered to select among our policies, we propose a novel learning method for policy selection based on observed state of the system. Under dynamic system state, PowerCoord uses Reinforcement Learning (RL) in an online fashion. We define $state$ as a vector of runtime metrics listed in Table II which include jobs running on each domain, power consumption of each domain, the total power cap, and power consumption of server. We define $action$ as choosing a policy from the set of available policies to coordinate budget for different domains. $Reward$ is defined based on the objective function defined in Equation (1a) and we also subtract the priority of jobs that miss their deadlines to magnify the penalty of bad decisions taken by the BestChoice. That is, the reward, $R$, is given by

\[
R = \sum_{i=1}^{n} p_{r_i} \times \mathbb{I}(f_i \leq d_i) - \beta \sum_{i=1}^{n} p_{r_i} \times \mathbb{I}(f_i > d_i),
\]  

where $\beta$ determines the magnitude of penalty. In our experiments, we choose $\beta = 3$. Maximizing the weighted throughput defined in Equation (4) maximizes the objective function defined in Equation (1a).

The most famous RL method is Q-learning [12], [13]. As the state-space of our problem is large and continuous, Q-learning does not work. Using a neural network to predict the Q(s, a) proved to be unstable in many environments. PowerCoord uses actor-critic methodology that has been shown to perform well in complex real-world scenarios such as robotic applications [14]. Actor-critic has two main components: 1) a critic that approximates the state value function \( V(s) \), and 2) an actor that predicts the probability of different actions to maximize the expected reward. State value function \( V(s) \) predicts the best expected reward being in state \( s \). Actor-critic methods combine the benefit of policy search methods with the learned value functions methods.

Because of large state-space, we leverage neural networks for both actor and critic functions. We divide time to epochs with fixed length \( e \). Assume \( s' \) and \( a' \) are the state-action pair from the last epoch. At each epoch, the critic network gets the current state, \( s \), and predicts the state value function \( V(s) \) in the forward path of critic neural network. The actor network gets the current state as input at each epoch and returns the probability distribution of all actions to maximize the expected reward. We select the policy based on the probability distribution predicted by the actor network. The reward from the previous state-action is used to update the neural networks. In the backward path of actor network, the weights are updated by minimizing \( D(s', a', s) - V(s') \times (-\log \text{Prob}(a'|s')) \), where \( D(s', a', s) \) is the discounted rewards of the previous state-action \( (s', a') \) pair followed by the current state \( s \); i.e.,

\[
D(s', a', s) = R(s', a') + \gamma \times V(s),
\]

where \( \gamma \) is the discount factor that determines how much impact the future reward has on the expected reward from current state and action. \( R(s', a') \) is the reward collected from taking action \( a' \) after being in state \( s' \). We choose \( \gamma = 0.9 \) as any action taken by the controller does not appear in the reward unless a job finishes or removed by the job scheduler at deadline. By minimizing the \( (D(s', a', s) - V(s')) \times (-\log \text{Prob}(a'|s')) \) at backward path of actor network, the probability of actions that achieves less rewards are reduced leading the convergence of discounted rewards and state value function. In the backward path of critic network, weights are updated to minimize the predicted value and observed discounted reward, i.e., \( (D(s', a', s) - V(s'))^2 \). Algorithm 1 summarizes the proposed BestChoice algorithm. We save the model at the end of one-time initialization phase.

**IV. EXPERIMENTAL RESULTS**

**A. Experimental Setup**

1) **Platform:** We run our experiments on a dual-socket Xeon server, where each of the E5-2680 v4 Xeon processors has 14 cores running at 2.4 GHz for all of 28 cores. The system has 128 GB of DDR4 memory. Our server is equipped with an NVIDIA P40 GPGPU card with 24GB of device memory. The server consumes about 500W at maximum load. Ubuntu Server 16.04 with kernel 4.4 is installed on the server with the gcc 5.4, python 2.7, and CUDA 8. We used MPICH 3.2 for message passing. Tensorflow is used for training and using our reinforcement learning algorithm at runtime [15]. We use SLURM as the job scheduler [16].

2) **Power measurement and control:** The server is equipped with Intelligent Platform Management Interface (IPMI) which we use to measure the total power using the lmsensors library. We leverage the power management utilities offered by RAPL [6]. To measure and control the power consumption of CPU sockets, we directly read and write to the Module Specific Registers (MSR). We sample the total and per socket powers every one second. To power cap the GPU domain, we implemented a feedback controller that reads the GPU power every 50 ms and adjusts the GPU’s frequency using NVML library.

3) **Jobs:** To evaluate the performance of PowerCoord, we use different mixture of CPU and GPU benchmarks. For CPU jobs, we use NPB benchmarks suite and leverage different

**NVIDIA’s driver offers power capping but the assigned power cap must be above 125 Watts.**
number of MPI ranks and class sizes to create jobs with different length and resource utilization. We use different GPU benchmarks with various input sizes to create GPU jobs with different length. Table III summarizes our pool of jobs and their problem sizes. The length of our jobs are between 1 to 5 minutes and CPU workloads use between 4 to 16 cores.

Job deadlines are usually provided in Service Level Agreements (SLA) and used for job scheduling. SLURM automatically terminates jobs that do not meet their deadlines. Since capping will always stretch the runtime, we ran the workloads initially without any capping and used 1.3× their runtime as deadline for capping experiments. This 30% extension is used to prevent SLURM from terminating the jobs when power caps are applied. Other values for deadline extensions can be applied with the natural expectations that lower values will lead to more killed jobs when caps are applied, and higher values will allow more jobs to run when caps are applied. Our techniques work effectively regardless. We create different job submission traces for our experiments. In each trace, jobs are selected from our job pool in Table III to have different mixture of resource utilization over time.

4) Performance metric and comparison: In our experiments, we use job throughput as our performance metric measured as the number of jobs finished before their deadline per unit of time. We compare the performance of PowerCoord against 1) our proposed policies without BestChoice component, and 2) POWsched. Similar to our P policy, POWsched dynamically coordinates power among the power domains based on their previous power consumption. It reduces the power budget of the domains that are not using their budget and allocate it to the domains that can use more power uniformly.

5) PowerCoord Implementation: PowerCoord is implemented in Python. Here are the implementation details specific to each component of PowerCoord:

1) Policies: Policies calculate the budget and send them to the power controllers using the server/client architecture implemented using Linux sockets.

2) BestChoice: Both actor and critic neural networks are implemented using Tensorflow. The actor network is a neural network with a 150 neurons hidden layer. The critic network is another one hidden layer neural network with 100 neurons. We used Adam Optimizer with learning rate of 0.001 and .01 to train our actor and critic network respectively.

6) Overhead & runtime analysis: We implemented the PowerCoord controller to have low overhead. The power monitor and controller is implemented in less than 500 lines of code with few control flow instructions. On average its process gets scheduled 10 ms on a single core. One iteration of PowerCoord’s takes 100 ms on average with a max of 200 ms on a single core. A forward pass of our neural networks takes 3 ms with a maximum of 5 ms and a backward path takes on average 40 ms. Both paths are calculated only every 90 seconds. All of our overhead measurements are recorded during our experiments when power is capped and system is loaded. We limit the memory utilization of Tensorflow. Running our PowerCoord controller consumes less than 0.9MB of main memory.

B. Evaluation

To evaluate the performance of proposed PowerCoord controller, we perform two sets of experiments.

• Static Scenario where we evaluate the performance of proposed policies statically without BestChoice.

• Dynamic Scenario where we evaluate the performance of proposed PowerCoord controller when BestChoice decides the policy in dynamic experiments. We compare its performance with running policies statically and POWsched [7].

1) Static Scenario: In this set of experiments, we evaluate the performance of proposed policies and compare them with POWsched [7]. We use four different user job submission traces each with the length of 48 high and low priority jobs. We assume low-priority jobs have priority of one in all traces. In trace 1 and 4, high-priority jobs have priority of three while in trace 2 and 3 they have priority of ten.

Figure 4 shows the job throughput for each class of job’s priority achieved by each policy. Figure 4 shows different policies have different performance for each trace and power cap. Each trace is running a mixture of different jobs with various characteristics and resource utilization. None of the policies performs the best in all traces. As an example, PDP performs the best for trace 1, while PD performs the best for trace 3 when the power cap is 350 W. As expected, reducing the total power cap results in longer runtime and more jobs miss their deadlines. Thus, the throughput of the server decreases. Compared with POWsched, the proposed P, PD and PDP policies improve the throughput by 16% in the best case (trace 2 for 400 W cap) and achieves the same throughput in the worst case (trace 2 for 350 W cap).

The results in Figure 4 show that a single policy does not perform the best for all traces. Additionally, the intuition behind each policy is necessarily true for all state of the system in term of job mixture and power consumption. As an example, intuitively the priority aware policy (PDP) must deliver the best job throughput for high priority jobs. However, results show for trace 2 PDP does not deliver the most throughput for high priority jobs, or PDP has the highest throughput.
for low priority jobs despite in trace 3 while the power cap is 400 W. Both observations emphasize the fact that heterogeneous policies are required for different system states. They also highlight the role of BestChoice in PowerCoord to dynamically select the best policy to coordinate the power budgets at runtime.

2) Dynamic Scenario: In the second set of experiments, we evaluate the performance of PowerCoord controller with BestChoice for following scenarios:

- Dynamically changing the total power cap while the job submission rate is fixed throughout the experiment.
- Dynamically changing the job submission rate while the total power cap is fixed.

We used two separate job traces in the two experiments. In each experiment, we used the same trace to compare different power capping scenarios. Each trace is about two and half hours long. We assume two priority of high and low for jobs. The low priority jobs have a priority of one and the high priority jobs have priority of three and ten. The high priority jobs simulate the case when jobs are for paid users and premium paid users.

**Dynamic power cap:** In the first set of our dynamic experiments, we dynamically change the power cap from 350 W to 400 W and again back to 350 W. The job submission rate is fixed and selected to keep the system busy all the times. The goal is to show that PowerCoord can adapt to variation in total power cap. Figure 5(a) shows the total power cap and power consumption of the server. Power consumption of each CPU socket and GPU is shown in Figure 5(b). Figure 5(a) shows that power is successfully capped for different total power caps. The small spikes (<1% total cap) seen in Figure 5(a) are due to the delay of power controllers. The first level of circuit breakers are at the rack and row level. Circuit breakers are designed to tolerate these spikes based on their time-current trip curve and also as power is aggregated across servers. The spikes in our results are small enough both in term of magnitude and time length to be considered negligible.
Dynamic job rate: In the second set of our dynamic experiments, we fixed the total power to cap to 350 W and varied the job submission rate to vary the load on the server. The goal is to show that PowerCoord can adapt to variation in load. Although the job mixture varies in dynamic power cap experiment as well, the load was enough to keep the system at power cap all the times. We reduced the job submission rate in the dynamic job rate to have moments where no capping is required. Figure 7(a) shows the number of running jobs on the server over time which varies between 0 to 7. Figure 7(b) shows the total power consumption which is under the cap all the times. The power coordination between CPU sockets and GPU is shown in Figure 7(c).

Figure 8 shows the job throughput for high and low priority jobs achieved by each method. PowerCoord delivers the most throughput compared with other capping scenarios. Similar to dynamic cap scenario, static uniform policy (U) achieves the least throughput as it does not shift power from domains that are not using their budget. PowerCoord improves the throughput by 2.3% and 5.2% for high and low priority jobs respectively compared to the best static policy (PDP). Overall, PowerCoord improves the job throughput by 4% in compared to the PDP policy. Compared with POWsched, PowerCoord improves the throughput of high and low priority jobs by 14.8% and 6.4% resulting in 9.5% overall job throughput improvement. Comparing both dynamic experiments together, the throughput in dynamic power cap experiment are higher as the job submission rate and the average power cap is higher. Also in dynamic power cap experiment, PowerCoord improves the throughput more as power is always capped in this experiment.

V. RELATED WORK

Power capping has been extensively studied for CPU workload both at the cluster and node level [4], [1], [23], [5], [24]. None of prior works considered power capping in CPU-GPU servers where a mixture of jobs are running on the system. The method [25] is only applicable if the workload uses both CPU and GPU for computation. Tsuzuku et al. considered a single workload running on the server and solved power capping using performance modeling [11]. The work in [26] collocate jobs on the same CPU-GPU server while modeling the contention among workloads and the total power cap of the system. In our work, we consider jobs are scheduled by the job scheduler. Chiesi et al. proposed job characterization for job scheduling [27]. The proposed methodology is highly dependent on the target architecture. The characterization of the job have to be repeated based on the new target architecture. Ellsworth et al. proposed POWsched [7] which dynamically cap the power of servers with multiple power domains. Although they have not evaluate their methodology on multi-CPU/GPU servers, their methodology is applicable for servers with multiple power domains. We used their algorithm for comparison.

Machine learning approaches are widely used to manage power and resource utilization. Chen et al. used reinforcement learning to maximize the performance of CPUs under a power cap [12]. Tesauro et al. used reinforcement learning for power management where the goal is to minimize the power consumption while the constraint is defined on the performance [28]. Hipster used a combination of heuristic methods and Q-learning to increase the energy and resource efficiency of clouds [13]. N. Liu et al. used deep reinforcement
learning to reduce power consumption of servers [29]. Ukidave et al. collocated GPU jobs to increase the utilization using collaborative filter [9]. Cao et al. used neural networks to predict the performance of workloads under power caps using CPU performance counters [2]. To the best of our knowledge, no prior works considered using reinforcement learning to coordinate the power in a multi-CPU/GPU server while running a mixture of workloads.

VI. CONCLUSIONS

This paper investigated power capping techniques for multi-CPU/GPU servers. Multi-CPU/GPU servers introduce new challenges for power capping because the power budget of multiple domains need to be coordinated and a mixture of jobs are running on the server at any point in time. We proposed PowerCoord to dynamically control the power of CPU sockets and GPUs to meet the total power cap while seeking to maximize the performance of the server. In our work, we considered multiple running jobs while previous works assumed a single job. We proposed different heuristics policies that shift power between different domains. As each policy maximizes the throughput for certain workload characteristics, we used reinforcement learning to adaptively shift the distribution of selected policies based on the observed state of the system. Our PowerCoord controller takes priorities and deadlines of various jobs into the account. We implemented PowerCoord on a real multi-CPU/GPU server with low overhead. We evaluated the performance of PowerCoord on dynamic scenarios and showed that it can adaptively maximize the server’s throughput and outperform previous methods in the literature.

REFERENCES


