Trapped Capacity: Scheduling under a Power Cap to Maximize Machine-Room Throughput

Ziming Zhang
Department of Computer Science and Engineering
University of North Texas
Denton, TX 76203
Email: zimingzhang@my.unt.edu

Michael Lang
UltraScale Systems Research Center
Los Alamos National Lab
Los Alamos, NM 87544
Email: mlang@lanl.gov

Scott Pakin
Applied Computer Science Group
Los Alamos National Lab
Los Alamos, NM 87544
Email: pakin@lanl.gov

Song Fu
Department of Computer Science and Engineering
University of North Texas
Denton, TX 76203
Email: song.fu@unt.edu

***Abstract*—Power-aware parallel job scheduling has been recognized as a demanding issue in the high-performance computing (HPC) community. The goal is to efficiently allocate and utilize power and energy in machine rooms. In practice the power for machine rooms is well over-provisioned, specified by high energy LINPACK runs or nameplate power estimates. This results in a considerable amount of trapped power capacity. Instead of being wasted, this trapped power capacity should be reclaimed to accommodate more compute nodes in the machine room and thereby increase system throughput. But to do this we need the ability to enforce a system-wide power cap. In this paper, we present *TracSim*, a full-system simulator that enables users to evaluate the performance of different policies for scheduling parallel tasks under a power cap. *TracSim* simulates the executing environment of an HPC cluster at Los Alamos National Laboratory (LANL). We use real measurements from the LANL cluster to set the configuration parameters of *TracSim*. *TracSim* enables users to specify the system topology, hardware configuration, power cap, and task workload, and to develop resource configuration and task scheduling policies aiming to maximize machine-room throughput while keeping power consumption under a power cap by exploiting CPU throttling techniques. We leverage *TracSim* to implement and evaluate three resource scheduling policies. Simulation results show the performance of those policies and quantify the amount of trapped capacity that can effectively be reclaimed.**

I. Introduction

Power consumption and energy efficiency have become imperative issues for researchers and engineers in the high-performance computing (HPC) community [20], as the power demand of large-scale HPC systems can easily exceed the power supply of an entire city [23]. Efficient allocation and effective utilization of power and energy in machine rooms are demanded by industry and government sectors. However, the power for machine rooms, in practice, is well over-provisioned, specified by high energy LINPACK runs or nameplate power estimates [19].

A direct way to control the power consumption of machine rooms is to set an upper bound on power, called a *power cap*, and enforce that a machine room cannot use more power than the assigned power cap [4]. Drawing more power would likely trip circuit breakers and lead to system downtime and additional stress on the hardware. However, the idea of setting power caps on a per-job basis and managing them from a system-wide rather than a per-node perspective has not yet been investigated thoroughly. Meanwhile, HPC systems in production environments do not make full utilization of the provided power. In particular, HPC systems are generally configured for worst-case power usage, which leads to over-provisioning. This implies the existence of *trapped power capacity*, which is the amount of power supplied to a machine room but not used to yield useful cycles. Instead of being wasted, the trapped power capacity should be reclaimed to accommodate more compute nodes in the machine room and thereby increase system throughput. Our goal is not to reduce energy or power consumption but rather to maximize the use of the existing power infrastructure, for example the power distribution units (PDUs).

Maximizing the utilization of the trapped capacity to complete as many parallel jobs as possible per unit time without exceeding the power cap is challenging, given how dynamic modern systems are. Tests on production HPC systems are costly and often not feasible. To make things worse, there is no practical and widely accepted simulator that can reflect the power consumption under different job and resource scheduling policies in real HPC environments.

In this paper, we present *TracSim*, a high-level macro-scale simulator that enables users to evaluate the performance of different policies for scheduling parallel tasks under a power cap. *TracSim* simulates the executing environment of an HPC system at Los Alamos National Laboratory (LANL). We use real measurements from the LANL cluster to set the configuration parameters of *TracSim*. *TracSim* enables users to specify the system topology, hardware configuration, power cap, and task workload and to develop job and resource scheduling policies aimed at maximizing machine-room throughput while keeping power consumption under the power cap by exploiting CPU throttling techniques. We leverage *TracSim* to implement and evaluate four task scheduling policies: three power-aware and one performance-dominant. The simulation results show the performance of the policies in terms of the trapped capacity utilization and system throughput. To the best of our knowledge, *TracSim* is the first simulator that estimates the power usage of HPC systems by simulating a production execution environment to test job and resource scheduling policies under a power cap.

Though this work focuses on HPC use cases, i.e., tightly synchronized multi-node jobs, it is also useful in data centers where multi-node jobs are not tightly coupled but could still be

managed dynamically by priority rather than with static node power caps. This paper makes the following contributions:

1) the introduction of a new, power-aware job-scheduling algorithm (PDRC) that executes workloads with high throughput while honoring a given, system-wide power cap,
2) an analysis of the performance and power consumption of multiple job-scheduling algorithms (including PDRC) and a quantification of the amount of trapped power capacity these leave in the system, and
3) the presentation of TracSim, a simulation framework informed by application power and performance measurements taken on a real, 150-node (2400-core) cluster and capable of predicting both workload performance and node and system power consumption.

The rest of paper is organized as follows. The related research is presented and discussed in Section II. Section III details the architecture and the major components of *TracSim*. The design and implementation of *TracSim* is described in Section IV. We implement three power-aware job scheduling policies and use *TracSim* to analyze their performance in Section V. Section VI draws some conclusions from our findings and notes some avenues for future research.

II. Related Work

Characterizing power usage is an important research topic. Existing approaches can be broadly classified into three categories: power measurement from real hardware [13], power estimation by simulations [3], and power characterization by quasi-analytical performance modeling [22]. Direct power measurements are accurate but costly and applicable only to existing systems. Power estimation by simulations works for both existing and future computer systems and can provide detailed power usage breakdown. Quasi-analytical performance modeling can be accurate but requires a deep understanding of each individual application running on the system.

Simulating HPC clusters also provides a flexible way to evaluate power-aware task scheduling policies. In order to simulate the execution of parallel jobs, a simulator needs to model both the performance and power consumption of a system. Several simulators are available. For example, Grobelny et al. presented the FASE framework to predict the performance and power consumption of HPC applications based on an application characterization scheme [11]. FASE enables users to test different architectural designs. Economou et al. propose a server power usage estimation framework that explores hardware utilization to achieve accurate estimation of the server's power usage [5]. Jackson et al. have developed an HPC cluster simulator named Maui, which allows users to test scheduling policies with regard to jobs' performance and resource utilization [12]. However, Maui cannot provide information about the cluster's power consumption.

Power profiling in production computer systems provides valuable data and knowledge for developing power simulators. Fine-grained power profiling techniques measure the power usage of individual hardware components, such as CPU [17], memory [25], disk [26] and other devices [24]. In contrast, coarse-grained power profiling aims to characterize system-wide power dynamics, such as the macropower framework [28]. Kamil et al. profiled HPC applications on several test platforms and projected the power profiling results from a single node to a full system [15]. Ge et al. studied the influence of software and hardware configurations on the system-wide power consumption [8]. They found that characteristics of HPC applications affect the power usage of a system.

To control the power usage of HPC systems, power capping [4] is an promising and effective approach. System operators can balance the performance and power consumption of clusters by adjusting the maximum amount of power (a.k.a. power budget) that can be consumed by the clusters. Pelly et al. present a dynamic power provisioning and capping method at the power distribution unit (PDU) level [21]. They propose to shift the slack power capacity to servers with growing power demand by using a heuristics policy. For HPC jobs, many factors affect the power usage including hardware configurations and resource utilization. Femal et al. developed a hierarchical management policy to distribute power budget among clusters [6]. Kim et al. investigate the relation between CPU voltages and system performance and power efficiency [16]. By exploiting the dynamic voltage scaling (DVS) technologies, they propose a task scheduling policy which aims to minimize the energy consumption while satisfying the specified performance requirements.

The novelty of this work is that we analyze multiple power-cap-aware job scheduling strategies using a simulator informed by actual sub-rack-level power and performance measurement of HPC applications running on a cluster. Using this framework we are able to investigate scheduling policies potentials in achieving high CPU utilization while keeping the power usage under a given power cap.

III. Architecture of TracSim

We aim at developing a simulator that enables users to evaluate the performance of different policies for scheduling parallel jobs under a power cap. It should simulate the execution environment of a production HPC system and be flexible and scalable to allow users to define different system topology, hardware configuration, power cap and task workload. It also needs to provide a well-defined API for users to implement different resource allocation and reconfiguration policies.

Figure 1 depicts the architecture of the *TracSim* simulator. *TracSim* consists of four major components: *workload*, *job*, *node* and *cluster*. A *workload* represents one type of application with a given problem size and a number of tasks (its *concurrency*). A *job* is an instance of a workload type with user-specified performance requirements and a priority level. As is common in HPC systems, a parallel job contains a number of tasks that are scheduled to run on multiple nodes, where a task is usually a rank in an MPI job. A *node* includes the configuration information of the major hardware devices including CPU, memory, disk, and network, the identity of a running job task, and a power monitor that calculates the power usage of executing the task on this node. There are a number of nodes in a *cluster*. In addition, a *job scheduler* in a cluster decides which job will run on which node. Users can define a power cap as the upper bound of power usage of the cluster and implement policies that adjust the resource configurations of nodes to use the trapped capacity in the cluster.

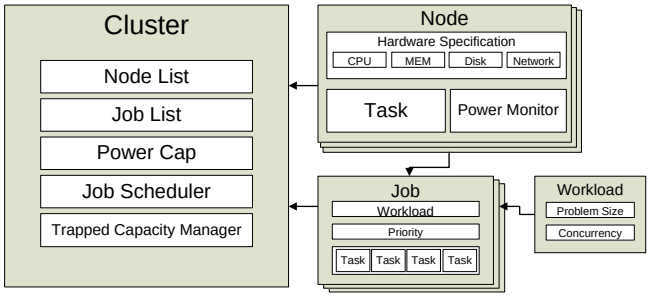


Fig. 1: Architecture of *TracSim*.

We now present the design details of the four major components in *TracSim*: *workloads*, *jobs*, *nodes*, and *clusters*.

A. Workloads and Jobs

TracSim defines a *workload* class for each category of application it simulates. HPC applications display varying load which is manifested as the dynamic utilization of system resources, including CPU, memory, disk and network when run on a real-world system. The corresponding power usage also changes at runtime. To model these dynamics, we could directly use the time series of the continuous measurements of the resource utilization and power usage from a production system to represent an application's workload. However, the time and space complexity of this approach is intimidating. An efficient method is to leverage statistical analysis of the time series, such as data distributions or statistics, to model the workload dynamics [27]. In *TracSim*, we use statistics of the measured performance and power usage data while running different applications on an actual cluster to model the applications' workloads.

Two important attributes of a workload are the *concurrency* and *problem size* as listed in Table I. They determine the number of tasks (in turn the number of nodes) and the amount of computation performed by each task.

A *job* is an instance of a *workload* with user specified performance requirements and system recorded execution information. A user provides the priority of a job during the job submission. The *job scheduler* then selects a set of qualified and available nodes to run the job's tasks and schedules the tasks' execution based on the job's priority. The number of selected nodes is the concurrency of the job's workload. The scheduler also updates the job's execution state, execution time, and completion time. Based on a power usage model (described in Section IV) that uses measurements of power consumption from an HPC cluster at LANL, *TracSim* calculates the power usage of running the job as follows:

$$job.Power = \sum_{node \in job.Nodes} node.Power \qquad (1)$$

In many production HPC systems a node is the minimum scheduling unit. That is, nodes are not shared across jobs. Hence, by aggregating the power consumption of those nodes executing tasks of a job, *TracSim* can simulate the job's total power usage. In addition, the statistics of the job's execution performance are also recorded.

TABLE I: Major attributes of components in *TracSim*.

Attribute	Description
Workload.ProblemSize	Problem size of a *workload*.
Workload.Concurrency	Number of tasks in a job of a *workload*.
Job.Workload	Workload type of a *job*.
Job.Priority	Priority value of a *job*.
Job.Nodes	A set of nodes scheduled to run tasks of a *job*.
Job.Perf	Performance statistics of a *job*.
Job.Power	Power usage of a *job*.
Job.State	Execution state of a *job*.
Job.SubTime	Timestamp of *job*'s submission.
Job.StartTime	Timestamp of starting *job*'s execution.
Job.FinishTime	Timestamp of *job*'s completion.
Node.Config	Hardware configuration of a *node*.
Node.Utilization	Resource utilization of a *node*.
Node.Task	Running task on a *node*.
Node.Power	Power consumption of a *node*.
Cluster.DirtyNodes	A set of nodes assigned to run jobs in a *cluster*.
Cluster.IdleNodes	A set of idle nodes in a *cluster*.
Cluster.JobList	A set of jobs running on a *cluster*.
Cluster.PowerCap	Power cap of a *cluster*.

TABLE II: Major programming functions provided by *TracSim*.

Method	Description
Workload.setConcurrency	Set the concurrency of a *workload*.
Workload.setProblemSize	Set the problem size of a *workload*.
Job.setWorkload	Set the workload type of a *job*.
Job.getWorkload	Get the workload type of a *job*.
Job.allocNode	Assign a set of nodes to run a *job*.
Job.suspend	Suspend the execution of a *job*.
Job.resume	Resume the execution of a *job*.
Job.setPriority	Set the priority of a *job*.
Job.setSubmissionTime	Set the submission time of a *job*.
Node.setFreq	Set the CPU frequency of a *node*.
Node.getPower	Get the power consumption value of a *node*.
Node.assignTask	Assign a task of a job to run on a *node*.
Cluster.setPowerCap	Set the power cap of a *cluster*.
Cluster.setTracManager	Set the trapped capacity management policy of a *cluster*.
Cluster.setJobScheduler	Set the job scheduling policy of a *cluster*.

B. Nodes and Clusters

Node is a representation of a computer or a server in an HPC system. From the hardware's perspective, a node comprises multiple hardware devices. *TracSim* simulates the hardware composition of a node by providing a set of configuration parameters to allow users to define the node's hardware specification. We include those configuration parameters that affect the power consumption of the node, including the number of processor cores, CPU frequency, and various fixed costs.

From the software's perspective, a node provides computing/storage services and runs application tasks. One task can be run on the node at a time. The *Node.Task* attribute keeps tracking the currently running task on the node. The *power monitor* exploits the measurements of resource utilization from the LANL cluster to simulate the dynamics of resource

utilization which determines the node's power consumption according to the power usage model as described in Section IV.

A *cluster* consists of a number of *nodes* which are connected by a network. For each cluster, *TracSim* maintains a list of nodes belonging to that cluster. When the *job scheduler* dispatches a job's tasks to run on the nodes of a cluster, the cluster's job list is updated. The *PowerCap* of the cluster determines the maximum amount of power that the cluster can use at any time. The *job scheduler* and *trapped capacity manager* are two important components that control the overall power consumption and the utilization of the trapped capacity in a cluster. The former decides which job tasks will run on which cluster nodes at what time and the latter changes the hardware configurations, such as the number of active processor cores and the CPU frequency, according to user-defined policies. In *TracSim*, a fixed amount representing the power usage of a cooling system based on the measurements from real systems can be added to the cluster's power consumption.

C. Job Scheduler

The *job scheduler* maps tasks of the submitted jobs to the available nodes in a cluster or across clusters. Consequently, it is also responsible for updating the execution states of jobs and the resource utilization of nodes and for calculating the power usage of jobs and nodes.

In production computer systems, job scheduling tools such as SLURM, PBS, and MOAB decide the execution sequence of jobs and node assignments based on certain properties of jobs including their priorities and/or deadlines. Following this common practice, *TracSim* provides a priority-based job scheduling policy by default. Algorithm 1 outlines this policy. It is implemented by using the attributes and functions provided by *TracSim*, as listed in Tables I and II respectively.

Algorithm 1 Default priority-based job scheduling policy in *TracSim*

```
priority = 1;
for all job ∈ cluster.JobList and job is not started do
  if job.priority = priority and job.Workload.Concurrency
  ≤ ||cluster.IdleNodes|| then
    list ← ∅;
    Job scheduler assigns nodes requested by job to list;
    cluster.IdleNodes ← cluster.IdleNodes − list;
    cluster.DirtyNodes ← cluster.DirtyNodes ∪ list;
    for all node ∈ list do
      node.assignTask(job);
    end for
  end if
  if all the jobs with priority are either started or completed
  then
    priority ← priority + 1;
  end if
end for
```

The algorithm is a basic backfill with no dynamic priority and no specific job placement. At every scheduling time point, the *job scheduler* selects the job with the highest priority to schedule. The scheduler searches for a set of idle nodes whose number matches the *concurrency* value of the job's workload. If successful, the selected nodes are marked as "dirty nodes" which indicates that those nodes are not available for scheduling until the currently running job terminates. In this way, the *job scheduler* keeps allocating the available nodes to those jobs that have been submitted but not yet started until there are not enough nodes available for scheduling a new job.

When a job finishes execution or is terminated, the nodes allocated to the job are released. The *job scheduler* is then invoked in an attempt to allocate the available nodes to the next job with the highest priority.

Even when there are sufficient available nodes for a lower-priority job but not enough for a higher-priority job, the *job scheduler* reserves those available nodes for the higher-priority job until enough nodes are released, because the scheduler cannot foresee when the lower-priority job can finish in real systems and tries to avoid starving the higher-priority job. This is only one of many possible methods of scheduling priority jobs. Other job scheduling policies can be implemented and added to *TracSim*.

TABLE III: Measurements of application performance and power consumption from running xRAGE on a LANL HPC cluster.

CPU Freq.		Execution Time		Power Consumption	
GHz	*norm.*	*sec.*	*norm.*	*Watt*	*norm.*
1.2	0.40	1,525.61	2.38	24,442.19	0.49
1.5	0.50	1,239.06	1.93	27,170.10	0.54
1.8	0.60	1,007.50	1.57	30,123.53	0.60
2.1	0.70	867.55	1.35	34,981.00	0.70
2.4	0.80	772.18	1.20	37,486.77	0.75
2.7	0.90	706.77	1.10	44,434.47	0.89
3.0	1.00	640.97	1.00	49,890.20	1.00

IV. Implementation Issues of *TracSim* and Trapped Capacity Management Policies

In this section, we focus on the implementation issues, including the setting of parameters and the power usage model used in *TracSim*. We also present two trapped-capacity management policies to demonstrate how to use *TracSim* to manage resource and power.

A. Performance Profiling and Parameter Setting

TracSim simulates the executing environment of a cluster at Los Alamos National Laboratory (LANL). We use real measurements from the LANL cluster to set the configuration parameters of *TracSim*. The cluster contains 150 nodes. Each node is equipped with two 8-core Intel E5-2670 Sandy Bridge processor. The total number of CPU cores in the cluster is 2400. The frequency of each core can be set from 1.2 GHz to 3.0 GHz. (3.0 GHz is in fact the effective speed achieved when Turbo Boost is active and all cores are busy.) The nodes are connected by a QLogic InfiniBand QDR network.

We selected two micro-benchmarks and four parallel scientific benchmarks as applications to run on the cluster. The micro-benchmarks that we tested are STREAM [18] which performs four vector operations on long vectors, and DGEMM (double-precision general matrix-matrix multiplication) from Intel's Math Kernel Library [9], optimized for the processors in the cluster and run on a set of matrices small enough to fit in the level 3 cache. The scientific applications that we tested are xRAGE [10], which simulates radiation hydrodynamics,

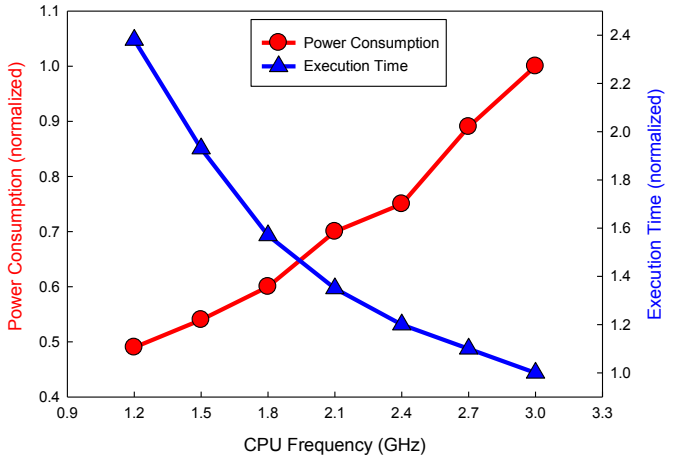


Fig. 2: Execution time and power consumption of running xRAGE on the LANL cluster.

POP [14], which simulates global ocean circulation, VPIC [2], which simulates magnetic reconnection and laser-plasma interactions, and Partisn [1], which solves the neutral-particle Boltzmann transport equation. Each benchmark application was run on all 2400 cores in the cluster. We profiled the execution time and system power consumption under different hardware configurations.

These programs were selected to represent a gamut of scientific applications from completely compute-bound (DGEMM) to memory-bound (STREAM). Table III lists the execution time and cluster-wide power consumption profiled from running xRAGE with different CPU frequencies on the cluster. From the table, it is clear that higher CPU frequency leads to better application performance but also higher power consumption. The profiling results of the other five benchmarks display the similar trend. The details are covered in a related work [19] and are not repeated here.

Based on these measurements, we set the values of some configuration parameters in *TracSim*. For instance, six types of *workload* correspond to the six types of applications tested on the LANL cluster. The CPU frequency in $Node.Hardware$ has seven possible values 1-7 corresponding to the seven frequency levels from 1.2 GHz to 3.0 GHz. The users can specify the cluster scale (i.e., the number of nodes), the problem size and number of tasks of *workload*, *job*'s priority ("low", "medium", and "high") and submission time, the number of processor cores of *node*, and the power cap of *cluster*. *TracSim* calculates a job's execution time and power consumption by using the models that are presented in the next section.

B. Power Usage and Performance Models

Figure 2 plots the normalized power consumption and execution time when the benchmark application xRAGE was run on the LANL cluster. With the increase of CPU frequency, the power consumption increases while the execution time decreases.

Assume a job of a workload type w is scheduled to run on n nodes in a cluster. There are N nodes in total in the cluster and the highest level of CPU frequency is F while the CPU frequency level for running the job is f. Then the power usage of the job can be calculated as

$$Power(w,n,f) = Power(w,N,F) \times \frac{n}{N} \times norm_{power}(w,f), \quad (2)$$

where $norm_{power}$ is a scaling term of the power consumption which either comes from empirical data such as the last column of Table III or is approximated by a relation such as $norm_{power}(w,f) = f/F$. (The latter has an average error of 5.59% relative to Figure 2.) Our scaling method is feasible because in production HPC systems the CPU utilization of active nodes is always very high, and hence the power level of a single node with the highest CPU frequency can be scaled to any CPU frequency level [7].

For HPC applications, we assume that higher concurrency leads to shorter execution time. Following the preceding notation, the execution time of the job is calculated as

$$Exectime(w,n,f) = Exectime(w,N,F) \times \frac{N}{n} \times norm_{exectime}(w,f). \quad (3)$$

Similarly, the normalization term, $norm_{exectime}$, can be determined from the profiled execution time, such as the 4th column in Table III. According to Figure 2, it can also be calculated as $norm_{exectime}(w,f) = F/f$ whose average error rate is 5.37%.

We design the power usage and performance models to be generic and simple to ensure the simulator is applicable to different systems and applications. We have tested six benchmark applications on an existing LANL cluster and the proposed models are accurate in characterizing the execution time and power consumption. Since controlling DVFS settings on nodes is not expensive, broadcast over InfiniBand or Ethernet on a per-job basis would not be an issue. Although detailed application-specific behavior would be valuable for building accurate workload models, this may compromise the efficiency and applicability of the simulator. Since most applications running on LANL supercomputers are CPU- or memory-intensive, CPU throttling is the most effective way that we control the power consumption. The models are a balance between complexity and speed to estimate the power for scheduling the job.

C. Trapped Capacity Manager

TracSim allows users to specify the system topology, hardware configuration, power cap and task workload. It also provides a set of functions, as listed in Table II, that facilitate users to develop and evaluate resource management policies. The *trapped capacity manager* (TCM) enforces the policies aiming to maximize the machine room throughput while keeping power consumption under a power cap by exploiting CPU throttling techniques which are widely used in production HPC systems.

As an illustration, we have designed and implemented a *power-aware dynamic resource configuration (PDRC)* policy. The policy consists of two mechanisms. One is power-level increment (PI) which aims to utilize the trapped capacity of a cluster by increasing the nodes' CPU frequencies (and therefore performance), which drives up their power usage. The other is power-level decrement (PD) which intends to avoid power cap

Algorithm 2 Power-level increment (PI) in the PDRC policy.

```
list ← ∅;
for all job ∈ cluster.JobList such that job is started do
    list ← list ∪ {job};
end for
for all priority ∈ 1 to k do
    while all job not running at the highest CPU freq do
        for all job that job ∈ list do
            if job.Priority = priority then
                for all node ∈ job.Nodes do
                    node.setFreq(node.Freq + 1);
                end for
            end if
        end for
        if Power(cluster) > cluster.PowerCap then
            for all job ∈ list do
                if job.Priority = priority then
                    for all node ∈ job.Nodes do
                        node.setFreq(node.Freq − 1);
                    end for
                end if
            end for
        end if
    end while
end for
```

violations by reducing the nodes' power usage. The pseudocode of adjusting the resource configurations for PI is presented in Algorithm 2. The PI mechanism keeps the power usage level close to but less than the power cap in order to maximize the system performance. It allocates the trapped capacity to reduce the execution time of higher-priority jobs by increasing the CPU frequency of the nodes running those jobs until the power cap is reached. Scheduling is determined strictly by priority; we currently do not take into account the application's CPU utilization. The function $Power(cluster)$ in Algorithm 2 applies the power usage model described in Section IV-B to calculate the power consumption of a cluster.

Algorithm 3 Power-level decrement (PD) in the PDRC policy.

```
list ← ∅;
for all job ∈ cluster.JobList such that job is started do
    list ← list ∪ {job};
end for
for priority = k to 1 do
    while all job not running at the lowest CPU freq and
    Power(cluster) > cluster.PowerCap do
        for all job ∈ list do
            if job.Priority = priority then
                for all node ∈ job.Nodes do
                    node.setFreq(node.Freq − 1);
                end for
            end if
        end for
    end while
end for
```

When the power cap is to be violated, the PD mechanism sacrifices jobs' performance for reduced power consumption. Algorithm 3 shows the pseudocode of PD. To reduce the power usage, PD first selects nodes that run lower-priority jobs to decrease their CPU frequency. If changing the CPU frequency of those nodes to the lowest level still cannot solve the power cap violation, PD tries to degrade the performance of jobs with higher priorities until the cluster's power usage is no greater than the power cap.

The PDRC policy is invoked and enforced by TCM in the following three cases. 1) When the system is initiated, the *job scheduler* starts to schedule jobs to run. Since the nodes in the cluster are in an idle state, the power usage of the cluster is at the lowest level. As the scheduled jobs begin execution using the initial resource configurations which usually are the lowest hardware configuration, an abrupt jump in the cluster's power usage occurs. If there still exists certain trapped capacity after the job scheduling, TCM performs PI to exploit this extra trapped capacity for better job performance. Otherwise, PD is invoked to handle a possible power cap violation. 2) When a job is completed or terminated, some nodes are released. When the *job scheduler* allocates the newly available nodes, TCM executes PI and/or PD to utilize the trapped capacity and/or to enforce the power cap. 3) When the power cap of a cluster changes (by applying a dynamic power capping strategy), PI or PD is invoked by TCM to remedy an increase or decrease of the power cap respectively.

Algorithm 4 Knapsack scheduling policy.

```
window ← ∅;
while not all jobs are started do
    Select jobs with certain priority into window
    while window is not empty do
        Perform knapsack on all jobs ∈ window, and attain a
        set of jobs
        Adjust CPU frequencies if needed
        Run selected jobs set
        Remove selected jobs from window
    end while
end while
```

We implement Zhou et al.'s power-aware job scheduling policy [29] in TracSim. This policy draws an analogy between job scheduling and the 0–1 knapsack problem. The requested number of nodes and the total power consumption are two key attributes that represent the underlying characteristics of an HPC job. Inside a queue full of pending jobs there are numbers of different combinations of jobs that can run simultaneously, which yield different scenarios of system-wide power consumption and resource utilization. The method associates one job with a gain value and a weight in terms of the basic definition of a 0–1 knapsack problem. Essentially, the gain value denotes the number of nodes a job requests, and the weight represents the estimated power consumption of the job. For a given system-wide power cap and a set of pending jobs, the scheduler attempts to select a set of jobs maximizing the system utilization. The 0–1 knapsack problem can be solved in pseudo-polynomial time, which ensures the relatively low overhead of enforcing the knapsack policy on a production HPC infrastructure. All the pending Jobs out of the scheduling window will not be considered and scheduled by the policy. After all the jobs inside the scheduling window are either started or completed, the scheduler will select a new set of jobs and locate them inside the scheduling window. Since the original policy does not consider the CPU throttling technique,

we enhance the policy by practicing CPU frequency adjustment and evaluate the potential of this policy in terms of trapped capacity.

The policy is implemented as in Algorithm 4. There are two specific cases that need additional discussion. First, it is possible that an optimal set of selected jobs requires more than the total number of available nodes. In this case, the scheduler raises the CPU frequency for power estimation and re-runs the knapsack process until the optimal set of selected jobs can be allocated to the available resources. Second, when the power cap is altered, the scheduler concurrently adjusts the CPU frequencies of all engaged nodes to adapt to the new power cap. In general, the knapsack policy is a utilization-dominant method.

V. Performance Evaluation

We have implemented *TracSim* in Java. The prototype has about ten thousand SLOCs. We have employed it to simulate an HPC cluster and conducted experiments to evaluate its performance. In this section, we present the experimental results.

A. Experimental Setup

The cluster that we simulate contains 150 nodes each of which has 16 processor cores. Each core provides seven frequency levels from 1.2 GHz to 3.0 GHz. To simulate jobs of different sizes in real systems, three types of jobs are tested: 1) a large-size job list in which every submitted job in the list needs 50–80 nodes; 2) a small-size job list where each job needs fewer than 50 nodes; and 3) a mixed-size job list where one half of the jobs are small-size and the other half are large-size. Each job is assigned a priority from one of the three values: 1 (high), 2 (medium) and 3 (low). There are six types of workloads in the system. At the start of a simulation, 30 jobs with randomly selected workload types are submitted to the cluster. The base time (with maximum frequency and concurrency) for each workload is determined by empirical data.

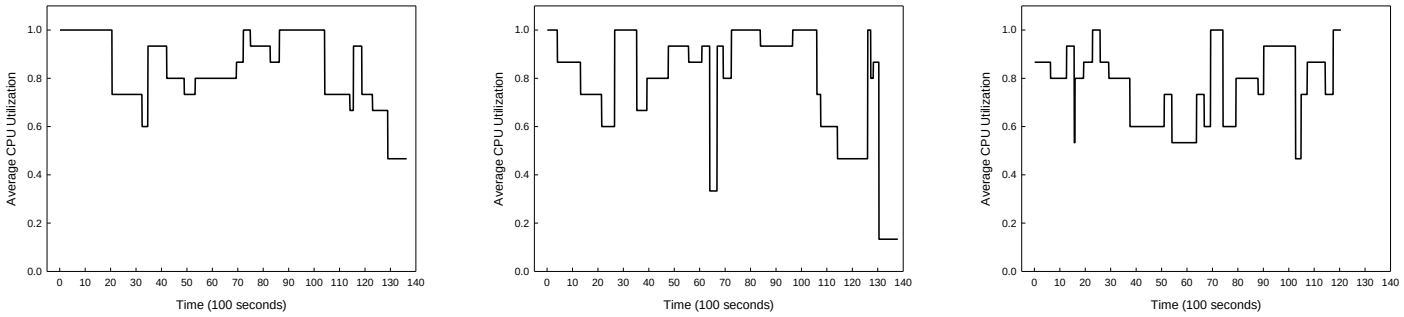
B. Experimental Results

1) CPU utilization: The resource utilization, especially the CPU utilization, is an important performance metric of the job scheduler. Figure 3 depicts the average CPU utilization with the three types of jobs by applying the priority-based job scheduling policy (as described in Section III-C). The three graphs show the job scheduler can effectively utilize the processing capacity as in 72.3% (large jobs), 72.4% (mixed-size jobs) and 87.4% (small jobs) of time, the average CPU utilization is 75.0% or above. The overall average CPU utilization reaches 77.5% (large-size jobs), 78.3% (mixed-size jobs) and 83.8% (small-size jobs).

2) Trapped power capacity utilization: In the next set of experiments, we focus on the utilization of the trapped capacity by testing four policies. In addition to the *power-aware dynamic resource configuration (PDRC)* policy (described in Section IV-C) and *knapsack* policy, we have implemented two static resource configuration policies in which the CPU frequency of a node can only be at one of the two levels. When the node is scheduled to run a job task, TCM sets the CPU frequency of the node's processor cores to the highest level (3.0 GHz in the experiments), while the CPU frequency is reduced to the lowest level (1.2 GHz in the experiments) when the node becomes idle. The static resource configuration policy reflects a common practice in production HPC systems. When there is a possible power cap violation, TCM adjusts the resource configuration by giving bias towards either power saving or system performance. In the former, which is called a *static power-precedence* policy, the job scheduler selects some running jobs to suspend so that TCM can change the CPU frequency of the corresponding nodes to the lowest level until the power cap is complied with. In the latter, called a *static performance-precedence* policy, no running job is suspended which aims to maintain the system performance.

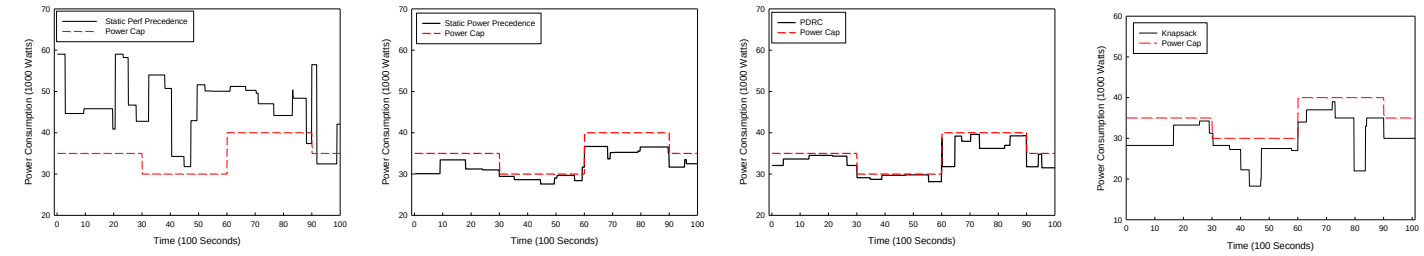
To evaluate the adaptability of these policies to changes of the power cap, the power cap is increased or decreased every 3000 seconds to mimic the cases where the power budget is cut/regained or the electricity rate changes. Figure 4 presents the cluster-wide power consumption by applying the four resource configuration policies. From the graphs, we can see the PDRC policy achieves the best utilization of the trapped capacity because it dynamically adjusts the CPU frequency according to changes of jobs and the power cap. Only 5.1% trapped capacity is unused by this policy and there is no power cap violation. The knapsack policy shows that 19.2% trapped capacity is unused. Among the two tested static resource configuration policies, the power-precedence policy outperforms the performance-precedence policy in terms of the trapped capacity utilization with 16.4% and -28.1% (negative means power cap violation) unused trapped capacity respectively. Moreover, the former experiences no power cap violation, while the latter causes the cluster's power usage exceeds the power cap in 74.3% of time. In total, the amount of unused trapped capacity of the cluster is 7.21 kWh (PDRC), 26.12 kWh (knapsack), 21.94 kWh (static power-precedence) and -29.85 kWh (static performance-precedence) for running only 30 jobs in around 3 hours as shown in Figure 5. If we scale the execution to the period of one year, PDRC can reduce energy consumption by 43,011.2 kWh compared with the static power-precedence policy on one cluster. Given that xRAGE draws a maximum of 49,890W on 150 nodes (Table III), this improvement from PDRC enables an increase in cluster size of 16 nodes (11%) without requiring any additional power infrastructure. When the power cap is enforced, i.e., xRAGE draws 34,981W on 150 nodes (Table III), PDRC enables 23 more nodes (16%) to be added to the cluster.

The impact of resource configuration policy on the resource utilization is also studied. Figure 6 plots the average CPU utilization by employing the four policies. The figure shows PDRC is slightly better than the static performance-precedence policy as their achieved CPU utilization can reach 80.3% and 78.9% respectively. The static power-precedence policy yields the lowest CPU utilization as 65.7% because of its aggressive job suspensions in face of power cap violations. In addition, the resource utilization differs when the cluster runs different types of jobs. Small-size jobs lead to the highest CPU utilization followed by the mix-size jobs and then the large-size jobs as their CPU utilization can reach 87.8%, 80.3% and 77.6% respectively. This is reasonable because as the job size decreases it is more likely to find a small job task to fill an idle slot of a node. The knapsack policy maximizes node utilization and achieves the best average CPU utilization among the four



(a) Small-size jobs. (b) Mixed-size jobs. (c) Large-size jobs.

Fig. 3: Average CPU utilization with three types of jobs.



(a) Static performance-precedence policy. (b) Static power-precedence policy. (c) PDRC policy. (d) Knapsack policy.

Fig. 4: Cluster-wide power consumption when running mixed-size jobs

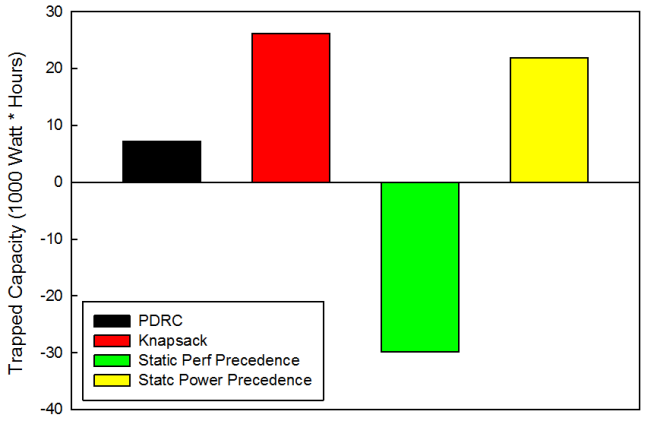


Fig. 5: Utilization of the trapped capacity by employing four resource configuration policies to run 30 jobs in about 3 hours on a 150-node cluster.

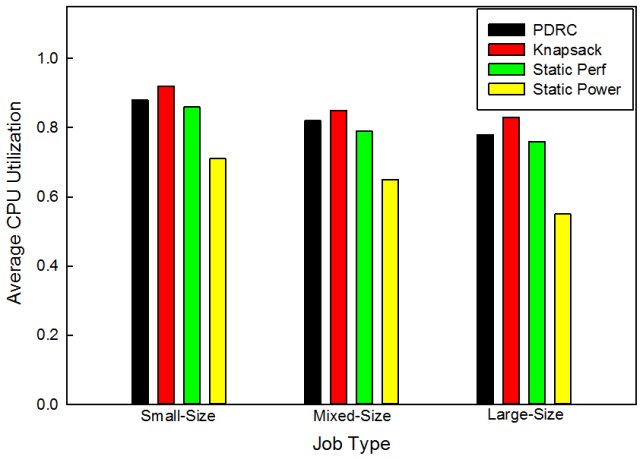
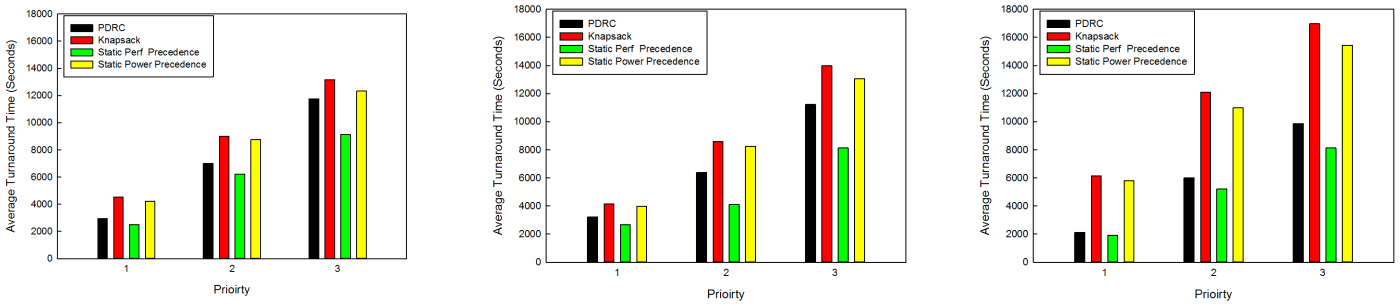


Fig. 6: CPU utilization by running three types of jobs with four resource configuration policies.

policies considered.

3) Job turnaround time: The turnaround time of a job, which refers to the total time taken between the submission and the completion of the job, is an important metric to evaluate the performance of jobs. Figure 7 depicts the average turnaround time of three types of jobs with three levels of priority. From this figure we can see the priority of a job imposes the most significant impact on its turnaround time. For example, by applying the PDRC policy to running small-size jobs, the turnaround time of jobs with the highest priority is only 41.9% and 25.6% of that of jobs with the medium and lowest priorities respectively. Among the four resource configuration policies, the static performance-precedence policy achieves the best job turnaround time as it requires greater power usage for gaining better performance. Between the other three policies which comply with the power cap requirement, PDRC outperforms the static power-precedence and knapsack policies in terms of job performance. The job turnaround time with PDRC is 69.1% at best and 91.8% at worst of that achieved by static power-precedence policy. Because the knapsack policy focuses



(a) Small-size jobs. (b) Mixed-size jobs. (c) Large-size jobs

Fig. 7: Average turnaround time of jobs with three types of workload and three priority levels.

more on maximization of node utilization rather than reclaiming trapped capacity, it yields the worst average turnaround time among the four policies considered. The type of jobs also affects the turnaround time. Large jobs are completed earlier than mixed-size and small jobs. In the cases where PDRC is applied, the turnaround time of large jobs is 78.7% (priority level 1), 88.3% (priority level 2) and 86.1% (priority level 3) of that of small jobs. This is counter-intuitive. Apparently the performance gained by strong-scaling the workloads outweighs the increase in waiting time needed to schedule the larger jobs.

To gain a deeper understanding of the execution of jobs on the cluster, we plot the distribution of jobs with three priority levels on the nodes as shown in Figure 8. The TCM employs the PDRC policy while running mixed-size jobs. A vertical view of the figure presents the spatial distribution of jobs on the nodes and a horizontal view depicts the jobs' execution in time. Three gray-scale color represent the three priority levels of jobs, while a blank area means the corresponding nodes are idle. The figure shows the job scheduler schedules jobs according to their priorities and reserves nodes that cannot immediately run a high-priority job in order to avoid lower-priority job blocking higher-priority ones. Figure 9 displays the CPU frequency of nodes set by the TCM. It is clear that the PDRC policy dynamically adjusts the CPU frequency according to the workload and to changes in the power cap.

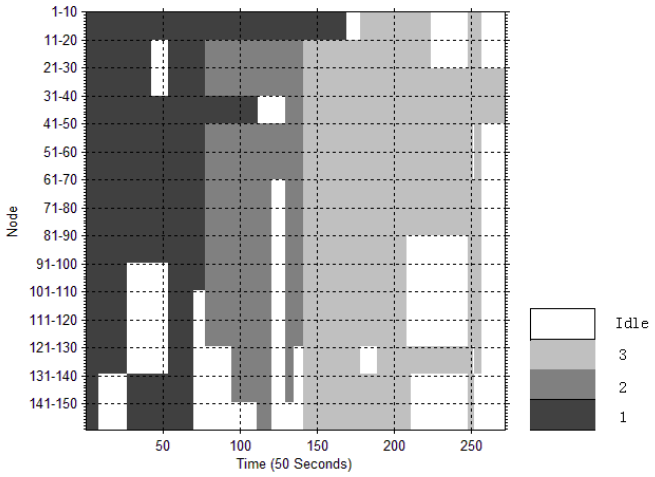


Fig. 8: Spatial and temporal distribution of mixed-size jobs with different priorities on nodes.

VI. Conclusions

Power consumption and energy-aware resource management are increasingly important in the design and operation of large-scale HPC systems. A deep understanding of the correlation between the various design factors and the system's power usage is valuable for developing power-saving and energy-efficient resource policies while meeting the expected performance requirements. In this work, we introduce the *TracSim* simulator, which enables users to simulate a high-performance computing environment and to evaluate the performance of different policies for adjusting resource configurations and scheduling parallel jobs under a power cap. The novelty of our approach is that we enable users to develop and test resource scheduling policies that are aware of both power caps and job priorities and that we use real measurements from an HPC system to set configuration parameters. The simulation results presented provide insights on how to efficiently utilize the trapped capacity to maximize the machine room throughput.

In this work, we proposed a *power-aware dynamic resource configuration* (PDRC) scheduling policy, which schedules jobs based not only on their priority but also with regard to a system-wide power cap, which can be changed dynamically. Among the four resource scheduling policies evaluated on *TracSim*, PDRC achieves the highest utilization (94.9%) of trapped capacity, outperforming the power-precedence policy by 11.3%, under a dynamic power cap. This improvement can be used to power an additional 16% more nodes in the cluster. In contrast, the performance-precedence policy violates the power cap by 28.1%. Moreover, PDRC achieves the highest CPU utilization—reaching 80.3%—and the lowest job turnaround time in the experiments. The validation of the proposed policy in a real HPC system is ongoing. Currently, we attempt to utilize scripting language and internal power meters to implement these proposed policies in HPC cluster at LANL.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and suggestions. This work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract DE-AC52-06NA25396. The

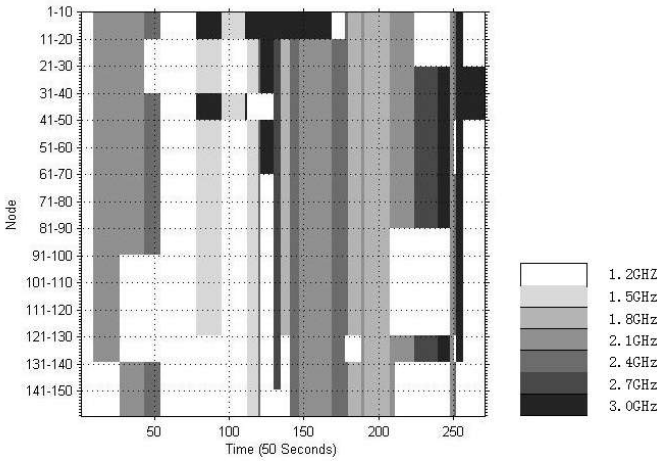


Fig. 9: CPU frequency of nodes running mixed-size jobs with the PDRC policy.

publication has been assigned the LANL identifier LA-UR-14-20294.

REFERENCES

[1] R. S. Baker. A block adaptive mesh refinement algorithm for the neutral particle transport equation. *Nuclear science and engineering*, 141(1):1–12, 2002.

[2] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan. Ultra-high performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5), 2008.

[3] M. Brown and J. Renau. ReRack: power simulation for data centers with renewable energy generation. *SIGMETRICS Performance Evaluation Review*, 39(3):77–81, 2011.

[4] J. Choi, S. Govindan, B. Urgaonkar, and A. Sivasubramaniam. Profiling, prediction, and capping of power consumption in consolidated environments. In *Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS)*, 2008.

[5] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full-system power analysis and modeling for server environments. In *Proceedings of Workshop on Modeling, Benchmarking, and Simulation in conjunction with ACM Symposium on Computer Architecture (ISCA)*, 2006.

[6] M. Femal and V. W. Freeh. Boosting data center performance through non-uniform power allocation. In *Proceedings of IEEE International Conference on Autonomic Computing (ICAC)*, 2005.

[7] R. Ge, X. Feng, and K. W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proceedings of ACM/IEEE Supercomputing Conference (SC)*, 2005.

[8] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658–671, 2010.

[9] P. Gepner, V. Gamayunov, and D. L. Fraser. Effective implementation of DGEMM on modern multicore CPU. *Procedia Computer Science*, 9:126–135, 2012.

[10] M. Gittings, R. Weaver, M. Clover, et al. The RAGE radiation-hydrodynamic code. *Computational Science & Discovery*, 1(1), 2008.

[11] E. Grobelny, D. Bueno, I. Troxel, A. D. George, and J. S. Vetter. Fase: A framework for scalable performance prediction of HPC systems and applications. *Simulation*, 83(10):721–745, 2007.

[12] D. Jackson, H. Jackson, and Q. Snell. Simulation based HPC workload analysis. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2001.

[13] V. Jiménez, F. J. Cazorla, R. Gioiosa, M. Valero, C. Boneti, E. Kursun, C.-Y. Cher, C. Isci, A. Buyuktosunoglu, and P. Bose. Power and thermal characterization of POWER6 system. In *Proc. of ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[14] P. W. Jones, P. H. Worley, Y. Yoshida, J. White, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP). *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

[15] S. Kamil, J. Shalf, and E. Strohmaier. Power efficiency in high performance computing. In *Proceedings of IEEE Parallel & Distributed Processing Symposium (IPDPS)*, 2008.

[16] K. H. Kim, R. Buyya, and J. Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *Proceedings of IEEE International Symposium on Cluster Computing and Grid (CCGrid)*, 2007.

[17] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of ACM ISCA*, 2003.

[18] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[19] S. Pakin and M. Lang. Energy modeling of supercomputers and large-scale scientific applications. In *Proceedings of IEEE International Green Computing Conference (IGCC)*, 2013.

[20] S. Pakin, C. Storlie, M. Lang, et al. Power usage of production supercomputers and production workloads. *Concurrency and Computation: Practice and Experience*, 2014. To appear.

[21] S. Pelley, D. Meisner, P. Zandevakili, et al. Power routing: dynamic power provisioning in the data center. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[22] S. L. Song, K. Barker, and D. Kerbyson. Unified performance and power modeling of scientific workloads. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing (E2SC 2013)*, 2013.

[23] U.S. Environmental Protection Agency. EPA report to Congress on server and data center energy efficiency, Aug. 2007.

[24] T. Ye, L. Benini, and G. De Micheli. Analysis of power consumption on switch fabrics in network routers. In *Proceedings of ACM Design Automation Conference (DAC)*, 2002.

[25] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *Proceedings of ACM Design Automation Conference (DAC)*, 2000.

[26] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[27] H. Zhang and H. You. Comprehensive workload analysis and modeling of a petascale supercomputer. In *Proceedings of Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) in conjunction with IEEE Parallel & Distributed Processing Symposium (IPDPS)*, 2012.

[28] Z. Zhang and S. Fu. macropower: A coarse-grain power profiling framework for energy-efficient cloud computing. In *Proceedings of IEEE International Performance Computing and Communications Conference (IPCCC)*, 2011.

[29] Z. Zhou, Z. Lan, W. Tang, and N. Desai. Reducing energy costs for IBM Blue Gene/P via power-aware job scheduling. In *Proceedings of the 17th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), in conjunction with IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.