Practical Resource Management in Power-Constrained, High Performance Computing

Tapasya Patki
University of Arizona
tpatki@email.arizona.edu

Anjana Sasidharan
Amazon, Inc.
anjans@amazon.com

Matthias Maiterth
Ludwig Maximilian University
maiterth@cip.ifi.lmu.de

David Lowenthal
University of Arizona
dkl@cs.arizona.edu

Barry Rountree
Lawrence Livermore National Laboratory
rountree4@llnl.gov

Martin Schulz
Lawrence Livermore National Laboratory
schulz6@llnl.gov

Bronis de Supinski
Lawrence Livermore National Laboratory
bronis@llnl.gov

ABSTRACT

Power management is one of the key research challenges on the path to exascale. Supercomputers today are designed to be worst-case power provisioned, leading to two main problems — limited application performance and under-utilization of procured power.

In this paper, we propose RMAP, a practical, low-overhead resource manager targeted at future power-constrained clusters. The goals for RMAP are to improve application performance as well as system power utilization, and thus minimize the average turnaround time for all jobs. Within RMAP, we design and analyze an adaptive policy, which derives job-level power bounds in a fair-share manner and supports *overprovisioning* and *power-aware backfilling*. Our results show that our new policy increases system power utilization while adhering to strict job-level power bounds and leads to 31.2% (18.5% on average) and 53.8% (36.07% on average) faster average turnaround time when compared to worst-case provisioning and naive overprovisioning respectively.

1. INTRODUCTION

The Department of Energy (DoE) has set an ambitious target of achieving an exaflop under 20 MW. While procuring this amount of power poses a problem, utilizing it efficiently is an even bigger challenge. Supercomputers today are typically targeted toward High Performance Linpack-like applications [33] and designed to be *worst-case provisioned*—all nodes in the system can run at peak power simultaneously, and thus applications are allocated all the available power on a node. However, most real HPC applications do not utilize this allocated power per node, leading to inefficient use of both nodes and power.

An example of this can be found by data we collected on Vulcan (see Figure 1), which is a high-end BlueGene/Q system located at Lawrence Livermore National Laboratory (LLNL). Vulcan is the ninth-fastest supercomputer in the world, and has procured power of 2.4 MW. However, our study shows that over a 16-month period, applications used only 1.47 MW on average. The only exception was the burn-in phase, which ran High Performance Linpack, as marked in the figure. This under-utilization of power has many negative ramifications, such as the use of lower water temperature than needed for cooling—which leads to additional power wasted on water chillers.

Ideally, supercomputing centers should utilize the procured power fully to accomplish more useful science. *Hardware overprovisioning* (or overprovisioning, for short) has been recently proposed as an alternative approach for designing power-limited supercomputers and improving performance [32,36]. The basic idea is to buy more compute capacity (nodes) than can be fully powered under the power constraint, and then reconfigure the system dynamically based on application characteristics such as scalability and memory intensity. Prior work has shown that on a dedicated cluster system, overprovisioning can improve individual application performance by up to 62% (32% on average) [32].

Initial research in the area has looked at managing resources on overprovisioned systems by deploying Integer Linear Programming (ILP) techniques to maximize throughput of data centers under a strict power budget [38]. While this is an interesting study, the proposed algorithm is not fair-share, and is not practical enough to be deployed on a real HPC cluster. This is because each per-job scheduling decision involves solving an NP-hard ILP formulation, incurring a high scheduling overhead and limiting scalability. Additionally, ILP-based algorithms may lead to low resource utilization as well as resource fragmentation, both of which are major concerns for high-end supercomputing centers [13, 17, 18, 21]. While allowing jobs to be *malleable* (change node counts to grow/shrink at runtime) might help

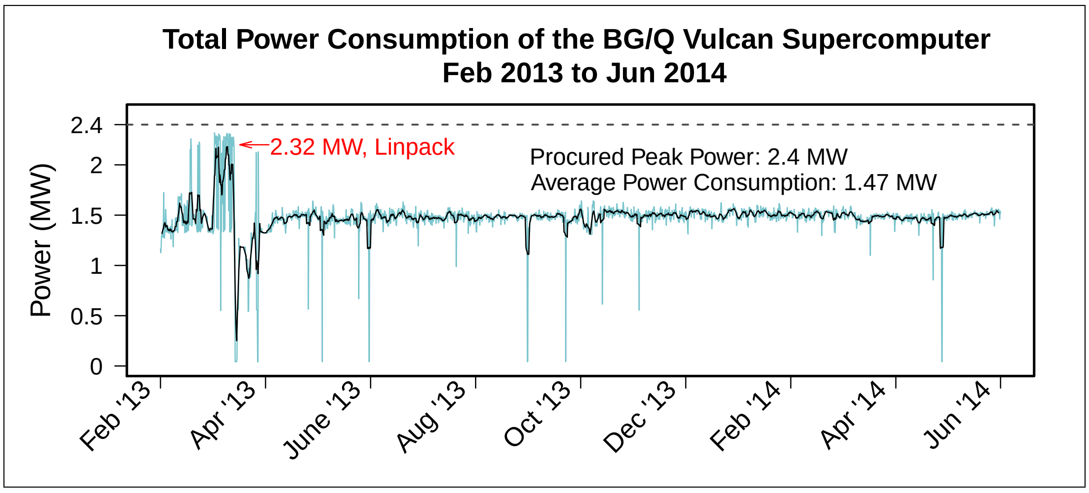


Figure 1: Power Consumption on Vulcan

address some of these problems, less than 1% of scientific HPC codes are expected to support malleability due to the data migration, domain decomposition and scalability issues involved.

In this paper, we present the design and implementation of *RMAP* (Resource MAnager for Power), a practical resource manager with minimal scheduling decision overhead (O(1)) that targets future power-constrained, overprovisioned systems. Within *RMAP*, we support the usage of job configurations and implement novel power-aware scheduling policies. The main focus of this paper is the design, implementation, and comparison of three policies that are targeted at power-constrained, hardware overprovisioned systems: a baseline policy for safe execution under power bound, a naive policy that uses overprovisioning, and an adaptive policy designed to improve application performance by using overprovisioning in a power-aware manner. The goal of the latter strategy is to provide faster job turnaround times as well as to increase overall system resource utilization. We accomplish this by introducing *power-aware backfilling*, a simple, greedy algorithm that allows us to trade some performance benefits of overprovisioning to utilize power better and to reduce job queuing times.

We make the following contributions in this paper:

- We design two novel policies with overprovisioning for which *RMAP* derives the job-level power bound based on a "fair sharing" strategy. The first is called *Naive*, in which the idea is to find the best configuration under the derived job-level power bound. The second is an adaptive policy, which uses power-aware backfilling to optimize for average turnaround time as well as to improve power utilization. We refer to this policy as the *Adaptive* policy for the rest of this paper.

- We develop and validate a model to predict execution time and total power consumption for a given application *configuration*, in order to support overprovisioning within *RMAP*. Our model uses less than 10% of the data for training, and the average errors for both performance and power prediction are under 10%.

- We demonstrate that the *Adaptive* policy leads to better overall turnaround times, adjusts to different job trace types and varying global power bounds, and improves system power utilization. We also demonstrate that by behaving altruistically and by allowing some degradation in their execution time, users can get better individual job turnaround times, and the system can further improve both power utilization and average turnaround time.

In addition, we also implement a simple, baseline policy, *Traditional*, that guarantees safe, correct execution under a power-constraint for non-overprovisioned systems. The *Adaptive* policy performs 18.5% better than the *Traditional* policy and 36.1% better than the *Naive* policy in terms of average per-job turnaround time. Notably, the *Naive* policy actually performs worse than the *Traditional* policy, which shows policies (such as our *Adaptive* policy) must be carefully designed in a power-constrained environment.

The rest of the paper is organized as follows. Section 2 motivates our work. Sections 3 to 5 present the design and implementation of *RMAP* and our model. We discuss our results in Sections 6 and 7. We describe related work in Section 8 and summarize in Section 9.

2. MOTIVATION

In this section, we motivate the need for overprovisioning-based scheduling.We first discuss power profiles of HPC applications and show that applications do not utilize the allocated power efficiently. Then, we discuss hardware overprovisioning.

2.1 HPC Application Power Profiles

In order to study HPC application power profiles, we selected eight strongly-scaled, load-balanced, hybrid MPI + OpenMP applications (described below) and gathered power and performance data for these at 64 nodes on the *Cab* cluster at LLNL. *Cab* is a 1,200-node, Intel Sandy Bridge server cluster, with 2 sockets per node and 8 cores per socket. We measured per-socket power with Intel's *Running Average Power Limit* (RAPL) technology [23, 35]. The maximum power available on each socket was 115 W. Note that we

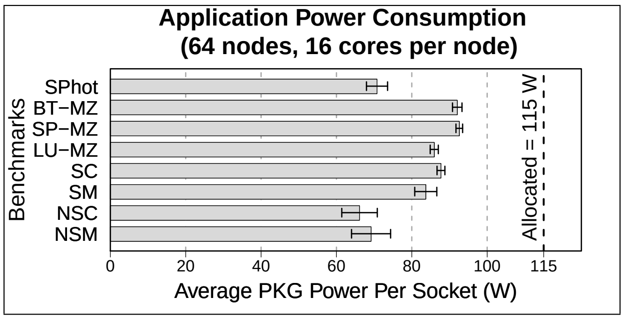


Figure 2: Application Power Consumption

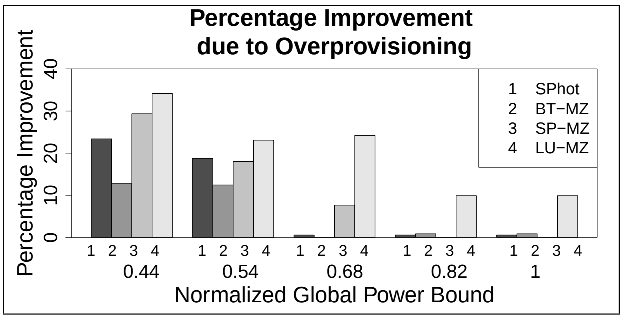


Figure 3: Hardware Overprovisioning Results

only measured socket power, as support to measure memory power was not available.

We used four real HPC applications for our study. These include SPhot [27] from the ASC Purple suite [26], and BT-MZ, SP-MZ and LU-MZ from the NAS suite [1]. SPhot is a 2D photon transport code that solves the Boltzmann transport equation. The NAS Multi-zone benchmarks are derived from Computational Fluid Dynamics (CFD) applications. BT-MZ is a the Block Tri-diagonal solver, SP-MZ is the Scalar Penta-diagonal solver, and LU-MZ is the Lower-Upper Gauss Seidel Solver. We used Class D inputs for NAS, and for SPhot, the NRuns parameter was set to 16,384.

In addition to these real applications, we added four synthetic benchmarks to our dataset to cover the extreme cases in the application space. These are (1) Scalable and CPU-bound (SC), (2) Not Scalable and CPU-bound (NSC), (3) Scalable and Memory-bound (SM), and (4) Not Scalable and Memory-bound (NSM). The CPU-bound benchmarks run a simple spin loop, and the memory-bound benchmarks do a vector copy in reverse order. Scalability is controlled by adding communication by using the `MPI_Alltoall()` function. We used MVAPICH2 version 1.7 and compiled all codes with the Intel compiler version 12.1.5. We used the scatter policy for OpenMP threads.

Figure 2 shows data for application power consumption for the eight applications running at 64 nodes, 16 cores per node, and maximum power per node. Each bar represents the average power consumption per socket (averaged over 128 sockets on 64 nodes) for an application. The minimum and maximum power consumed per socket by the application are denoted with the help of error bars. While all applications were allocated 115 W per socket, they only used between 66.1 W (NSC) to 92.6 W (SPMZ) of power (81.0 W on average). On average, this led to a utilization of only 71.4% of the allocated socket power.

2.2 Hardware Overprovisioning

As discussed in Section 1, a cluster is said to be *hardware overprovisioned* with respect to power if it has more nodes than it can simultaneously fully power. Such a cluster can be essentially "reconfigured" on a per-application basis, depending on the memory-boundedness and scalability characteristics of the application. .

The benefits of overprovisioning rely on determining a *configuration*, $(n \times c, p)$ that leads to the best performance under a power bound, where n is the number of nodes, c is the number of cores per node, and p is the power per socket. Note that this assumes that applications are somewhat flexible in terms of the number of nodes and/or the number of cores per node on which they can run on (*moldable*).

We emulated overprovisioning by enforcing socket-level power caps with Intel's RAPL technology. The minimum RAPL socket power cap that could be enforced on our architecture was 51 W, and maximum power cap was 115 W. We ran our applications with five package power values — 51 W, 65 W, 80 W, 95 W, and 115 W. We gathered data for each configuration ranging from 8 to 64 nodes as well as 8 to 16 cores per node in increments of four and two for each application respectively. Turbo Boost was disabled when the power caps were enforced, except for the 115 W power bound, in which Turbo Boost was enabled (as this is the maximum socket power possible). The highest non-Turbo frequency was 2.6 GHz, and the highest Turbo frequency was 3.3 GHz.

The maximum global power bound for our cluster was $64 \times 2 \times 115\ W$, which is 14,720 W. In order to analyze various degrees of overprovisioning, we chose five global power bounds for our study — 6,500 W, 8,000 W, 10,000 W, 12,000 W and 14,720 W. These were determined by the product of (1) the number of nodes and (2) the minimum and maximum package power caps possible per socket (51 W and 115 W).

Note that the worst-case provisioned configuration is $(n_{max} \times 16, 115)$, where n_{max} is the maximum number of nodes one can run at peak power without exceeding the power bound. This utilizes all the cores on a node (16 nodes) and runs as many nodes as possible at the maximum power level (115 W per socket).

We measured execution time and total power consumed for each of the benchmarks in the configuration space discussed earlier. Figure 3 shows results of overprovisioning when compared to worst-case provisioning for four HPC benchmarks. For our dataset, we saw a maximum improvement of 34.2%, and an average improvement of 11.4% in performance compared to worst-case provisioning. Previous results on a 32-node cluster have indicated that application performance can improve by up to 62% [32, 36].

3. POWER-AWARE SCHEDULING

In this section, we first discuss HPC scheduling basics and backfilling. We then discuss the design challenges for *RMAP* and present the details of the three scheduling policies we evaluate within *RMAP*.

3.1 Basics

At HPC installations, users typically submit jobs by specifying a node count and an estimated runtime. The job executes when the resource manager acquires the specified number of nodes, and the estimated runtime is used to set a deadline ($t_{deadline}$) for the job. If the job exceeds this deadline, it is killed. Depending on the job-size, most HPC users are required to use specific partitions. For example, on most high-end clusters, there is a *small* partition specifically targeted for small-sized jobs or for debugging, and a general-purpose *batch* partition for medium and large-sized jobs. Figure 4 shows the running and pending jobs on Vulcan, and the number of requested nodes by users on Vulcan averaged over a 24 hours. We observe that there are several medium and large-sized jobs.

Resource requests are maintained in a *job queue*, and users are allocated a set of dedicated nodes based on a scheduling policy set by the system administrator. One such policy is First-Come First-Serve (FCFS), which services jobs strictly in the order that they arrive. FCFS tends to cause a convoy effect when a job requesting more resources (large node count) ends up blocking several other smaller jobs. Policies that do not dedicate nodes to jobs, such as *gang scheduling* [4, 15, 39], are not considered to be a feasible options in supercomputing. This is because memory demands for HPC applications are typically quite high—which leads to large paging costs[1]. With gang scheduling, jobs can time-share nodes in a coordinated manner. While this may improve utilization and overall job turnaround time in some cases, it is not considered to be a feasible option in supercomputing due to the high context-switching and paging costs involved.

3.2 Backfilling

Backfilling [24,29,30,41] addresses the convoy effect present in algorithms such as FCFS by executing smaller jobs out of order on idle nodes and by improving utilization—in turn reducing the overall average turnaround time. There are two variants of backfilling: *easy* and *conservative*. Easy backfilling allows short jobs to move ahead and execute out of order as long as they do not delay the *first* queued job. Conservative backfilling, on the other hand, only lets short jobs move ahead if they do not delay *any* queued job. Easy backfilling performs better for most HPC workloads [30].

Backfilling is usually implemented as a greedy algorithm and picks the *first fit* from the job queue. The *first-fit* might not always be the *best-fit*, and a job further down the job queue may better fit the available hole being backfilled. Finding the *best-fit* involves scanning the entire job queue, which increases the job scheduling overhead significantly [40].

3.3 Design Challenges

In addition to managing and allocating nodes, power-aware schedulers strive to (1) enforce job-level power bounds in a fair manner, (2) optimize individual job performance under the job-level power bound, (3) minimize the amount of unused (leftover) power in the system, and (4) optimize overall system throughput.

For simplicity, we assume that all jobs submitted to the system have equal priority, use MPI+OpenMP, and are moldable (not restrictive in terms of the number of nodes on which they can be executed). We also assume that the global power bound on the cluster is $P_{cluster}$, and that the cluster has $N_{cluster}$ nodes. We derive a power bound for each job fairly by allocating it a fraction of $P_{cluster}$ based on the fraction of $N_{cluster}$ that it requested. Thus, $P_{job} = \frac{n_{req}}{N_{cluster}} \times P_{cluster}$.

[1]In fact, many HPC installations utilize operating systems that do not page.

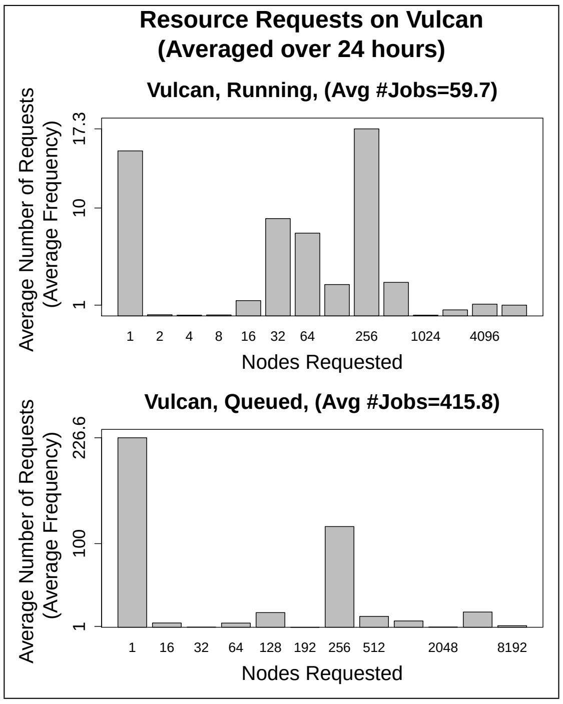


Figure 4: Resource Requests on Vulcan

It is important to note that this allocation for P_{job} can be easily extended to a priority-based system by using weights (w_{prio}) for the power allocation. Thus, $P_{job} = w_{prio} \times \frac{n_{req}}{N_{cluster}} \times P_{cluster}$. For example, higher priority jobs could be allocated more power by using a w_{prio} greater than 1, and lower priority jobs could be allocated a w_{prio} of less than 1. We do not address priorities in this paper.

Once we have a job-level power bound, we can optimize for individual job performance under that power bound. To do this, we use overprovisioning, as discussed in Section 2.2.

To address the third and the fourth challenges, a scheduler could (1) dynamically redistribute the unused power to jobs that are currently executing, or (2) suboptimally schedule the next job with the unused (available) power and nodes.

Dynamically redistributing power to executing jobs to improve performance can be challenging, mostly because allocating more power per node may result in limited benefits (see Figure 2). In order to improve performance and to utilize power better, the system may have to change the number of nodes or the number of cores per node at runtime. However, varying the node count at runtime (malleability) is not possible with the current MPI standard. In addition, there is a data decomposition and migration overhead associated with dynamically changing the node and core counts of a job [25].

To address these challenges, we consider the idea of extending traditional backfilling to a power-aware scenario. Backfilling attempts to utilize as many nodes as possible in the cluster by breaking FCFS order. Similarly, our new approach will attempt to greedily utilize as much global power as possible by scheduling a job with currently avail-

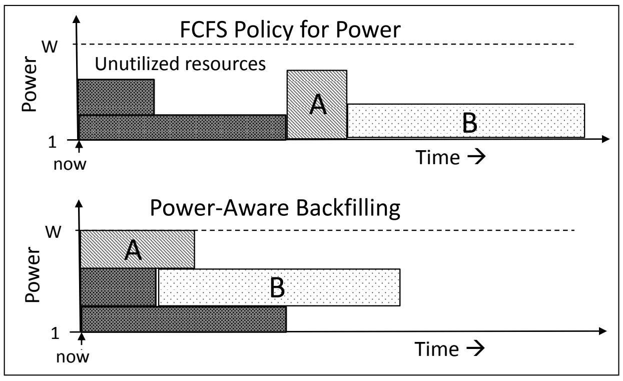


Figure 5: Advantage of power-aware backfilling

Policy	Input to Policy	Description
Traditional	(n_{req}, t_{req})	Pick the *packed* configuration $(c = 16, p = max = 115W)$
Naive	(P_{job}, t_{req})	Pick the optimal configuration under the derived job power limit
Adaptive	$(P_{job}, t_{req}, thresh)$	Use power-aware backfilling to select a configuration

Table 1: Job scheduling policies. All policies require that the user provide a number of nodes (n_{req}) and maximum job time (t_{req}); for the latter two policies, (n_{req}) is turned into the job power bound (P_{job}) as described in the text. Finally, *Adaptive* accepts an optional *thresh*, which is the maximum acceptable slowdown compared to t_{req}.

able power. In most cases, this will involve trading some performance benefits that were obtained from overprovisioning. The key idea is to schedule a job with less power than what was requested (derived using fair share) and schedule it with a suboptimal configuration, and to do so with execution time guarantees. Power-aware backfilling can adapt to both extremely power-constrained scenarios as well as scenarios in which there is too much leftover power. It is important to note that we do this in addition to normal node-level backfilling.

Figure 5 shows an example of this. In this example, we assume that `Jobs A and B` are currently waiting in the queue. `Job A` has requested more power than what is currently available in the system. Traditionally, `Job A` will have to wait in the queue until enough power is available. This leads to wasted power. Instead, our approach is to schedule `Job A` immediately with the *available* amount of power (which is less than what `Job A` requested). This may potentially slow `Job A` down, but improves the overall turnaround time for `Job A` as well as the other jobs in the queue, and utilizes available power better.

The key idea is to use power-aware backfilling while adhering to the user-specified time deadline for the job. Using overprovisioning under a job-level power bound will exceed the user's performance expectations in most cases (as discussed previously). However, as there are no hard guarantees, allowing users to trade their job's execution time for faster turnaround time (due to shorter wait queue time) is an added incentive. This allows the user to specify the maximum slowdown that their job can tolerate. Our focus is to primarily trade the benefits obtained from overprovisioning and utilize all the power to run jobs faster/schedule more jobs and hence accomplish more science.

Keeping cluster resources utilized (both nodes and power) via backfilling leads to better average turnaround times for the jobs, which in turn increases throughput. We thus focus on minimizing the average turnaround time in this paper. The policy that we develop is called the *Adaptive* policy, and we discuss it in the next section.

3.4 Scheduling Policies

We now discuss the power-aware scheduling policies that we implemented in *RMAP*. Each of these policies needs to obtain job configuration information given a power bound. The details of how these configurations are determined are presented in Sections 4 and 5, which discuss the low level implementation details and the model.

Users specify nodes and time as input, along with an optional threshold value for *Adaptive*. We derive, P_{job}, which is the job-level power bound based on the user input, as discussed in the previous subsection. This derived job-level power bound is used as an input to the scheduling policies described below (see Table 1). All three policies use basic node-level backfilling.

3.4.1 The *Traditional* Policy

In this policy, the user is allocated a configuration with their requested node count that uses all the available cores on a node at maximum possible power per node. A job that requests large node counts may exceed the global power bound of the system. In such a scenario, the *Traditional* policy allocates as many nodes (with all cores on the node and maximum power per node) as it can to the job without exceeding the system-wide budget (thus, an unfair job-level power allocation). An alternative option would be to let users know that this job is not runnable due to power constraints.

More formally, let c_{max} be the maximum number of cores per socket, p_{max} the maximum package power per socket, $P_{(n\times c,p)}$ the actual power consumed by the job in the $(n \times c, p)$ configuration, and $P_{cluster}$ the global power bound on the cluster. Then, for a job requesting n_{req} nodes for time t_{req}, the *Traditional* policy does the following:

- If $P_{(n_{req}\times c_{max},p_{max})} \leq P_{cluster}$, allocate the configuration $(n_{req} \times c_{max}, p_{max})$
- Else, allocate the configuration $(n_{max} \times c_{max}, p_{max})$, where $n_{max} = max(N)$, and $N = \{n : P_{(n\times c_{max},p_{max})} \leq P_{cluster}\}$

3.4.2 The *Naive* policy

In this policy, we overprovision with respect to the job-level power bound. Given the derived job-level power bound, $P_{job} \leq P_{cluster}$, and an estimated runtime, t_{req}, the *Naive* policy will allocate a configuration $(n \times c, p)$, such that it leads to the best time t_{act} under that power bound. Hence, $t_{act} = min(T)$, where $T = \{t_{(n\times c,p)} : P_{(n\times c,p)} \leq P_{job}\}$.

It is important to note that if $t_{act} > t_{req}$, the system will set the deadline $t_{deadline}$ for the job to be t_{act} instead of t_{req} during job launch, so that the job does not get killed prematurely. This may happen in the scenario where the

ID	Configuration $(n \times c, p)$	Total Power (W)	Time (s)
C1	$(6 \times 16, max = 115)$	796.4	447.9
C2	$(8 \times 12, 65)$	783.8	415.3
C3	$(8 \times 10, 80)$	738.2	439.2

Table 2: List of configurations for SP-MZ

user's performance estimates are inaccurate and cannot be met with the derived power bound, and the best performance level that the *Naive* policy can give to the job under the specified power bound P_{job} is worse than t_{req}. In the scenario that $t_{act} < t_{req}$, $t_{deadline}$ is not updated until job termination (if the job terminates sooner). *RMAP* will kill the job after $t_{deadline}$. The main purpose for t_{req} is to have a valid deadline in case the job fails or crashes. User studies suggest that t_{req} is often over-estimated (by up to 20%) [43].

3.4.3 The *Adaptive* policy

The goal of our *Adaptive* policy is to allow both (1) users to receive better turnaround time for their jobs, and (2) the overall system to greedily minimize the amount of unused power and achieve better average turnaround time for all jobs. Similar to the *Naive* policy, the inputs are a (derived) job-level power bound and duration. However, the *Adaptive* policy considers these values as suggested and uses power-aware backfilling. It also trades the raw execution time of the application as specified by the user for potentially shorter turnaround times. The user can also specify an optional *threshold (th)*, which denotes the percentage slowdown that the job can tolerate. When *th* is not specified, it is assumed to be zero (no slowdown).

The *Adaptive* policy uses the suggested job-level power bound to check if the requested amount of power is available at the current time. If it is, it obtains the best configuration under this power bound (similar to the *Naive* policy). If not, it determines a suboptimal configuration based on currently available power and the threshold value. The advantage for the user is that the job wait time may be significantly reduced. From an administrative point of view, this leads to better resource utilization (in terms of nodes and overall power) and better throughput.

More specifically, if $P_{job} \leq P_{avail}$, the *Adaptive* policy uses the same mechanism as the *Naive* policy. However, when $P_{job} > P_{avail}$, it determines $t_{act} = min(T)$, where $T = \left\{t_{(n \times c,p)} : P_{(n \times c,p)} \leq P_{avail}\right\}$, and schedules the job immediately with the configuration $(n \times c, p)$ with time t_{act} as long as $t_{act} <= (1 + th) \times t_{req}$. This helps reduce the wait time for the job while meeting a performance requirement. Algorithm 1 gives an overview of this policy.

3.5 Example

As an example, consider SP-MZ from the NAS Multizone benchmark suite [1]. Some configurations for SP-MZ (Class C) are listed in Table 2.

We now discuss four scenarios, under the assumption that $750W$ of power and 10 nodes are currently available in the cluster and that `Job A` is currently executing and terminates in $1000s$ from the current time. Let us also assume that the user has requested 6 nodes for $450s$, and that based on the global cluster power bound, this translates to a job-level power bound of $800W$.

3.5.1 *Scenario 1, Traditional Policy*

In this scenario, *RMAP* allocates configuration `C1` to the job, and the user has to wait until enough power is available (until `Job A` terminates).

3.5.2 *Scenario 2, Naive Policy*

Here, *RMAP* allocates configuration `C2` to the job. Again, the user has to wait until `Job A` terminates and enough power is available to meet the request.

3.5.3 *Scenario 3, Adaptive Policy (unbounded)*

In this scenario, *RMAP* first checks if enough power ($800W$) is available in the system. Because there isn't, it chooses `C3`, which is the best configuration given that only $750W$ of power is available. Picking `C3` reduces the wait time of the job significantly (by $1000s$).

3.5.4 *Scenario 4, Adaptive Policy (threshold=0%)*

A threshold value of 0% means that the user cannot compromise on performance. *RMAP* determines that `C3` does not violate the performance constraint ($450s$) and can be launched immediately with currently available power ($750W$). It is important to distinguish this case from Scenario 2, which will *always* pick `C2`.

In scenarios 1 and 2, $750W$ of power is unused for $1000s$. Scenarios 3 and 4, on the other hand, utilize power more efficiently.

4. RMAP IMPLEMENTATION

We implemented *RMAP* within the widely-used, open source resource manager for HPC clusters, SLURM [44]. SLURM has been deployed on several of the Top500 [2] supercomputers. It provides a standard framework for launching, managing and monitoring jobs executing on parallel architectures. The `slurmctld` daemon runs on the head node of a cluster and manages resource allocation. Each compute node runs the `slurmd` daemon for launching tasks. `Slurmdbd`, which also runs on the head node, collects accounting information with the help of a MySQL interface to the `slurm_acct_db` database.

As described earlier, *RMAP* supports overprovisioning and implements three power-aware scheduling policies that adhere to a global, system-wide power budget. We refer to our extension of SLURM as P-SLURM. RMAP can be implemented with other resource managers in a similar manner.

The key to the three scheduling policies is the ability to produce execution times for a given configuration under a job-level power bound. Table 3 shows the information that is required within P-SLURM. We refer to this as the `job_details_table`, and we added this table to the existing `slurm_acct_db`.

We developed a model to predict the performance and total power consumed for application configurations in order to populate this table. The details of this model are presented in Section 5. Furthermore, to understand and analyze the benefits of having exact application knowledge, we also included another table within the SLURM database (with the same schema) that contains an exhaustive set of empirically measured values (as per the details discussed

Algorithm 1
***Adaptive* Policy**

Inputs:
(1) Currently available power, P_{avail},
(2) Requested power and time, P_{job}, t_{req},
(3) Threshold value (optional), th, which is the percentage slowdown the job can tolerate,
(4) Information about job power profiles and configurations. More specifically, a way to determine, $P_{(n\times c,p)}$, the actual measured power for the configuration $(n \times c, p)$, and the time $t_{(n\times c,p)}$ it takes.
Note that th is assumed to be zero percent if it has not been provided as an input.

```
if $P_{job} \leq P_{avail}$ then
    Use the Naive policy.
else
    Determine $t_{act} = min(T)$, where $T = \{t_{(n\times c,p)} : P_{(n\times c,p)} \leq P_{avail}\}$

    if $t_{act} <= (1 + th) \times t_{req}$ then
        Schedule the job (immediately) with the configuration $(n \times c, p)$ with time $t_{act}$
    else
        Job waits in queue, as it does not meet the slowdown requirements.
    end if
end if
```

Field	Description
`id`	Unique Index (Primary)
`job_id`	Application ID
`nodes`	Number of nodes
`cores`	Number of cores per node
`pkg_cap_0`	PKG Power Cap (Socket 0)
`exec_time`	Execution Time
`tot_pkg`	Total PKG Power

Table 3: Schema for Job Details Table. Values for `exec_time` and `tot_pkg` can be measured or predicted via our model.

in Section 2). For simplicity, both these tables were populated in advance and the scheduler queried the database for information when making decisions, making the decision complexity O(1). The model can also be used to generate values dynamically without needing a database. However, this may incur a scheduling overhead and call for advanced space-search algorithm implementations within the scheduler (such as hill climbing). We do not address this issue in this paper.

5. PREDICTING PERFORMANCE AND POWER

In this section, we discuss the models that *RMAP* deploys in its policies. The models predict execution time and total power consumed for a given configuration (number of nodes, number of cores per node, and power cap per socket). As discussed in Section 2, we first collected exhaustive power and performance information. We ranged the node counts from 8 to 64, core counts from 8 to 16, and power from 51 W to 115 W. The dataset we built contained 2840 data points, with 5 different power caps, 15 different node counts, 5 different core counts per node and 8 applications.

We then used 10% of this data for training and obtained application-specific linear regression parameters that allowed us to predict application execution time and total package power consumption at a given configuration. A logarithmic polynomial regression of degree two was used for this purpose. We limited our power predictions to package power

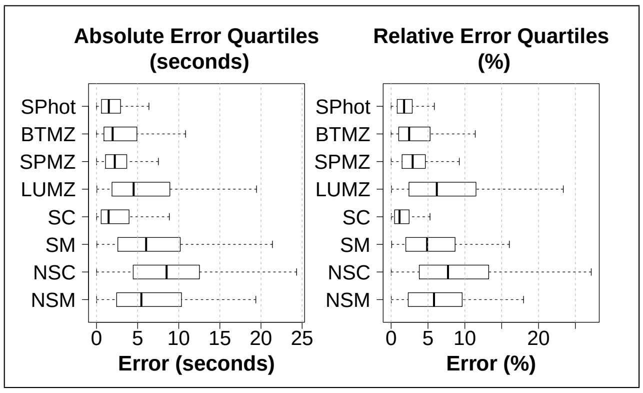
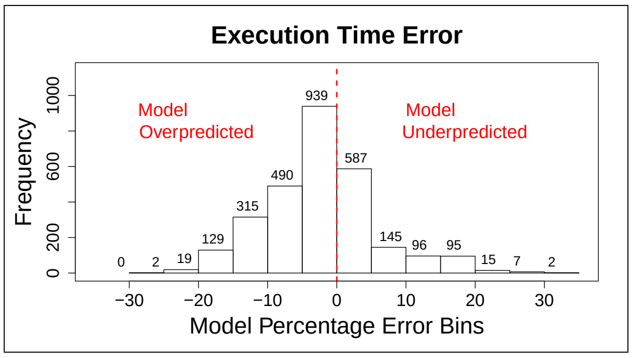


Figure 6: Error Quartiles of Regression Model

only as memory power measurements were unavailable on our cluster.

Next, we validated our models with our previously measured data. When using only 10% of data for model training, the average error for execution time is below 10%, and the maximum error for the same is below 33%. Figure 6 shows the absolute (seconds) and relative (percentage) error quartiles for all benchmarks when predicting execution time at arbitrary configurations. For all benchmarks, the third quartile is under 13.2% with the median below 7.7%.

Figure 7 shows the detailed error histograms for both performance and power across the entire dataset (2840 points). The x-axes on both graphs represent the error bins, and the y-axes represent the frequencies. Regions with over-predicted and under-predicted values have been identified. It is important to note that if we over-predict the power consumed by a job, we may block the next job in the queue due to lack of enough power. On the other hand, under-predicting the power may lead to a situation where we exceed the cluster-level power bound (worst-case scenario). In our model, for 96% of our data, the under-prediction for power was no more than 10%, and the worst case was under 15%. This issue can be addressed by giving *RMAP* a conservative cluster-level power bound that is 15% less than the actual power bound, or by relying on the fact that most



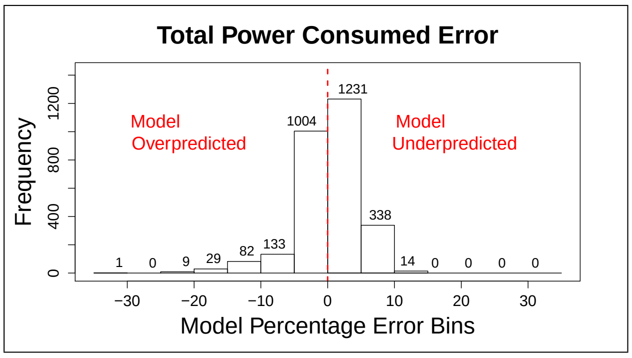


Figure 7: Range of Prediction Errors

supercomputing facilities are designed to tolerate these kind of surges [3].

6. EXPERIMENTAL DETAILS

To set up our simulation experiments for *RMAP*, we populate the `job_details_table` with application configuration information, as discussed in Section 4. In all our experiments, we assume the same architecture as *Cab*. We consider a homogeneous cluster with 64 nodes and global power bounds ranging from 6,500 W to 14,000 W, based on the product of the number of nodes and the minimum and maximum package power caps that can be applied to each socket (51 W and 115 W). Each node has two 8-core sockets.

We generate job traces from a random selection of our recorded configuration data as inputs for the simulator. Each trace has 30 jobs to ensure a reasonable simulation time. The total simulation time with all traces, power bounds, node counts and policies was about 3 days (approximately 30 minutes for each trace).

We use a Poisson process to simulate job arrival [14, 15]. Job arrival rate is sparse on purpose (to make queue times short in general), so we can be conservative in the improvements that we report with *Adaptive*. We select the following types of job traces to evaluate our scheduling policies.

- Traces with *small-sized* and *large-sized* jobs: To identify scenarios where one power-aware scheduling policy may be preferred over another, we create trace files with small-sized and large-sized jobs. Users are allowed to request up to 24 nodes in the former, and have to request at least 40 nodes for the latter.

- *Random* traces: For completeness, we also generate two random job traces. Users were allowed to request up to 64 nodes for these traces. We refer to these traces as *Random Trace 1* and *Random Trace 2*. The two traces differ in the job arrival pattern as well as job resource (node count) requests, thus exhibiting different characteristics. While both traces were created using the same number of jobs and arrival rate parameter, Random Trace 1 has many of the jobs arrive early in the trace, whereas arrival times are more uniform in Random Trace 2.

7. RMAP RESULTS

In this section, we discuss our results and evaluate our scheduling policies on a Sandy Bridge cluster (which was described in Section 2) that we simulated with P-SLURM. In our experiments, we assume that all jobs have equal priority. For fairness and ease of comparison, in each experiment we assume that all users can tolerate the same amount of slowdown (threshold value for the *Adaptive* policy). Note that for readability, the graphs are *not* centered at the origin.

All figures in this section compare the *Traditional* and the *Naive* policies to the *Adaptive* policy when the global, cluster-level power bound is varied across the cluster. The x-axis is the global power limit enforced on the cluster (6,500 W–14,000 W), normalized to the worst-case provisioned power (in this case, that equals 64 $\times$ 115 W $\times$ 2, which is 14,720 W). The y-axis represents the average turnaround time for the queue, normalized to the average turnaround time of the *Traditional* policy (lower is better). The *Traditional* policy mimics worst-case provisioning. Recall that this allocates per-job power in an unfair manner and always uses Turbo Boost, unlike the other two policies, which are fair-share and use power capping. Also, all three policies have O(1) scheduling decision complexity, so we do not compare them against each other for scheduling overhead.

We start by evaluating the model discussed in Section 5 when applied to *RMAP* and its policies. Then, we compare and analyze the three policies by applying them to different traces at several global power bounds. We then analyze two traces in detail; more specifically to discuss how altruistic behavior on part of the user can improve turnaround time, and how the *Adaptive* policy can improve system power utilization.

7.1 Model Evaluation Results within RMAP

In this section, we compare the impact of using our model for predicting application configuration performance and power within *RMAP*. Figure 8 compares the average turnaround time for the first random job trace at 5 different power caps. Both configuration performance as well as total power consumed is being predicted for each job in the trace. The former is used for determining execution time, and the latter is used to determine available power. We observe that for *Traditional* and *Adaptive*, our model is accurate (always under 10.1%, and 3.9% on average across the two policies). We observe similar results on the other traces.

While performance prediction introduces error and affects overall turnaround times, we observe that the errors introduced by over-prediction of the total power consumed by a configuration propagates and impacts the turnaround time more. This is because scheduling as well as backfilling decisions can be significantly affected when they depend on avail-

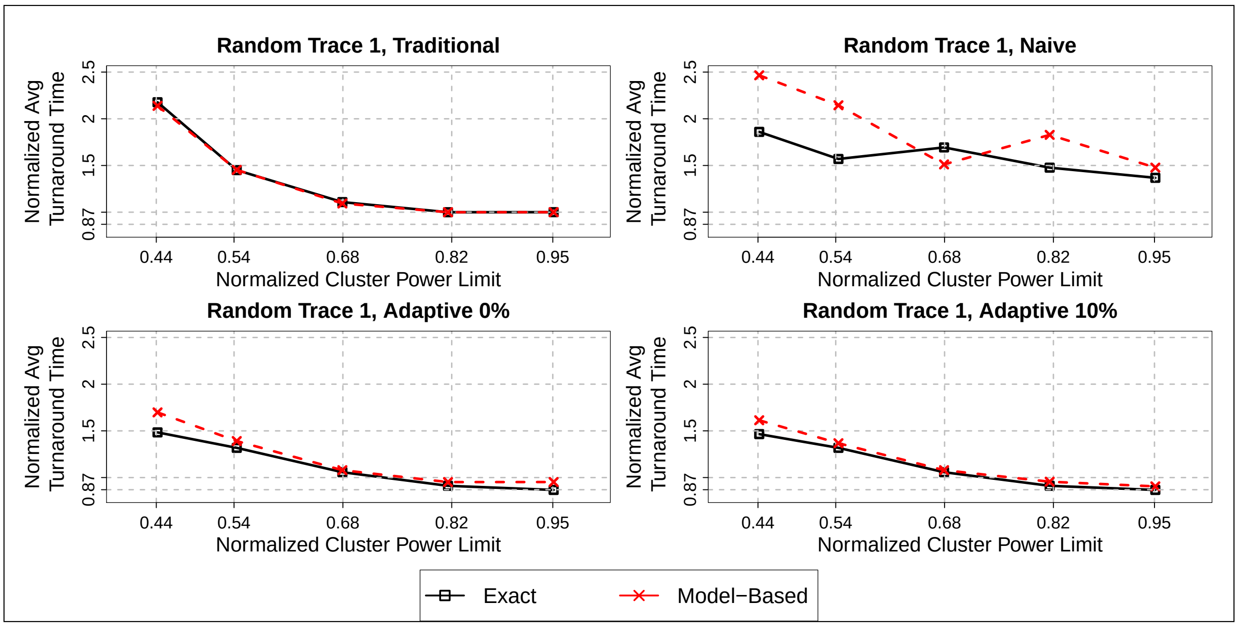


Figure 8: Model Results on the Random Trace

able power. For example, at a lower cluster power bound, if we over-predict the power consumed by a small amount (even 3%), we might not be able to schedule the next job or backfill a job further down in the queue, resulting in added wait times for all queued jobs. This is especially true for *Naive* at lower global power bounds.

In the subsections that follow, we take a conservative view on *Adaptive* and establish the *minimum improvements* that our algorithm can provide. We accomplish this by using oracle information for the *Traditional* and *Naive* policies, which are our baselines, and by using the model for the *Adaptive* policy.

7.2 Analyzing Scheduling Policies

In this subsection, we compare and analyze the power-aware scheduling policies within RMAP on the four different traces at various global power bounds.

7.2.1 Trace with Large-sized Jobs

Each job in this trace file requests at least 40 nodes. For all enforced global power bounds, the *Adaptive* policy leads to faster turnaround times than the *Traditional* and *Naive* policies, primarily because it fairly shares power and uses power-aware backfilling to decrease job wait times. Figure 9 shows that the *Adaptive* policy with a threshold of 0% improves the turnaround time by 21.7% when compared to the *Naive* policy and by 14.3% when compared to the *Traditional* policy on average (up to 47.3% and 24.6%, respectively). *Adaptive* policy with a threshold of 10% further improves the overall turnaround time by 16.1% on average when compared to the *Traditional* policy.

At lower global power bounds, the *Traditional* policy serializes the jobs, leading to longer wait times and larger turnaround times. The *Naive* policy always allocates the optimal configuration under the user-specified power bound, and this may lead to longer wait times if the best configuration uses a large number of nodes.

7.2.2 Trace with Small-sized Jobs

Figure 10 depicts the results for small-sized jobs and compares all three policies. Each job in this trace file requests a maximum of 24 nodes. For small-sized jobs, the average turnaround time of the *Traditional* policy is significantly better than that of the *Adaptive* policy. This is because original wait times for small jobs are fairly short, leaving limited opportunity for the *Adaptive* policy to backfill for power. Additional improvement is a result of running the jobs at a higher frequency in Turbo mode (3.0 GHz with Turbo on 16 cores, 2.6 GHz otherwise) due to smaller power requirements. When using power-aware policies, the small-sized jobs can be run on a small, but separate partition (such as *pSmall*) with the *Traditional* policy if needed, as discussed in Section 3.

For the small-sized job trace, the *Adaptive* policy with a threshold of 0% leads to 16.6% worse average turnaround time (21.1% maximum degradation in average turnaround time) when compared to the *Traditional* policy. The *Adaptive* policy with a threshold of 10% depicts similar results. At the highest power bound though, the *Traditional* policy fails to adapt and the *Adaptive* policy improves the overall turnaround time by 17.1%.

7.2.3 Random Traces

Figure 8 (from the previous subsection) compares the three policies. For both the random traces, the *Adaptive* policy with a threshold of 0% does 18.52% better than the *Traditional* policy and 36.07% better than the *Naive* policy on average (up to 31.2% and 53.8%, respectively).

In some scenarios, the policies lead to larger turnaround times at higher global power bounds. An example of this is the *Naive* policy at 10,000 W (corresponding to value of 0.68, when normalized, in Figure 8). This happens because of two reasons. One, this policy strives to optimize individual job performance, so it sometimes chooses configurations with large node counts under the power bound for minor gains in performance (less than 1% improvement in execution time). This leads to other jobs in the queue incurring longer wait times and increased serialization of jobs. The other reason for this trend is aggressive and inefficient backfilling (picking the *first-fit* instead of the *best-fit*).

Figure 11 (page 9) depicts the impact of varying threshold values on the *Adaptive* policy for the large-sized and the random traces. The *Adaptive* policy is compared to the baseline *Naive* policy (which does not support thresholding). Threshold values that tolerate a slowdown of 0% to 30% in application performance are shown. For large jobs, thresholding helps the user improve the turnaround time for their job by greatly decreasing queue time. However, when there is not enough queue time to trade for, as in the case of extremely small-sized jobs, it is expected that adding a threshold will lead to larger turnaround times. The random traces shown here have a mix of small-sized and large-sized jobs. For all our traces, the *Adaptive* policies with thresholding up to 30% either improve the overall turnaround time (by up to 4%) or maintain the same turnaround time when compared to the *Adaptive* policy with a threshold of 0%. The unbounded *Adaptive* policy, which assumes that the job can be slowed down indefinitely, is also shown for comparison, and this leads to worse turnaround times.

For the large trace, *Adaptive* with a 0% threshold does 33.8% better on average than the unbounded *Adaptive*. Slowing down by 10% to 30% improves the average turnaround time by 2.1% on average (up to 4%) when compared to *Adaptive* with 0% thresholding. For the other three traces, the improvement obtained by slowing down the jobs is under 2.6% on average when compared to *Adaptive* with a threshold of 0%, and this depends on the power bound as well as the job mix. It is important to note that each data point in these graphs represents the average across all jobs in the trace at a particular global power bound. We analyze per-job performance for a single trace at a fixed global power bound in the next subsection.

7.3 Analyzing Altruistic User Behaviour

We now present detailed results on the large-sized job trace in a power-constrained scenario, where only 6,500 W of cluster-level power is available (50% of worst-case provisioning). We pick this scenario because most important jobs in a high-end cluster typically have medium-to-large node requirements. Recall that each job is requesting at least 40 nodes, so all these jobs are allocated the entire 6,500 W ($P_{cluster}$) with the *Traditional* policy, leading to unfair power allocation, sequential schedules and no opportunity for backfilling; the scheduler runs out of power even when enough nodes are available. The trace contains 30 jobs and is a dynamic queue.

Figure 12 shows individual job turnaround time for the *Traditional* policy, and for the *Adaptive* policy with 0% and 20% thresholding. The absolute values of average turnaround times for the job trace for all the policies are shown in Table 4. We limit the graph to the main policies we focus on.

Allocating power fairly with the *Adaptive* policy with a threshold of 0% leads to better turnaround times for most users (17 out of 30), even when they choose to not be altruistic. The average turnaround time improved by 7.1% for

Policy	**Average Turnaround Time (s)**
Traditional	684
Naive	990
Adaptive, 0%	636
Adaptive, 10%	613
Adaptive, 20%	536
Adaptive, 30%	536

Table 4: Average Turnaround Times

the job queue when compared to the *Traditional* policy in this case. For the *Adaptive* policy with 10% and 20% thresholding, 18 and 22 jobs resulted in better turnaround times respectively, improving the average turnaround time of the job queue by 10.5% and 21.1% when compared to the *Traditional* policy. This demonstrates the benefits of altruistic behavior.

Altruistic users also get better turnaround times when compared to the *Adaptive* policy with a threshold of 0%. For example, when the threshold was set to 20%, 13 users got better turnaround times (up to 57.7% better, for job 30; and 12.8% on average) than what they did with a threshold of 0%. 14 users had the same turnaround time, and for 3 users, the turnaround times increased slightly (by less than 1.9%). The average turnaround time for the queue improved by 21.1%, as discussed previously. These benefits come from power-aware backfilling as well as hardware overprovisioning.

Figure 13 shows the breakdown of queue time, execution time, and allocated power for the jobs. The fair-share derived job-level power bound for the *Traditional* policy has also been shown.

In some cases, such as for the first 5 jobs in the queue, the turnaround times with the *Adaptive* policy increased when compared to the *Traditional* policy. There are several reasons for this increase. One, all the jobs were allocated significantly more power with the *Traditional* policy (because there was no fair-share derived power bound, resulting in allocating the entire power budget to most jobs) and executed with Turbo Boost enabled (as no power capping was enforced), resulting in better execution times compared to the other policies. Also, depending on when a job arrived, it may have had zero wait time with the *Traditional* policy. In such a case, when the execution time increases, the turnaround time increases as well, because there is no wait time to trade. Despite these issues, the *Adaptive* policy with 0% thresholding improved the turnaround times for 17 out of 30 jobs, which shows the benefits of fair sharing. For this example, the utilization of system power by both the *Traditional* and *Adaptive* policies was fairly high and there was not much leftover power, mostly because this was a tight global power bound (50% of peak) and the jobs were large-sized.

7.4 Power Utilization

We now analyze the *Random Trace 2* in detail in a scenario at 14,000 W, when the global power bound is 95% of peak power. We show that the *Adaptive* policy, even with a 0% threshold, improves system power utilization. Again, we take the conservative view by looking at a sparse job arrival rate in our dynamic job queue (short queue times in general), so we can test the limits of our *Adaptive* policy.

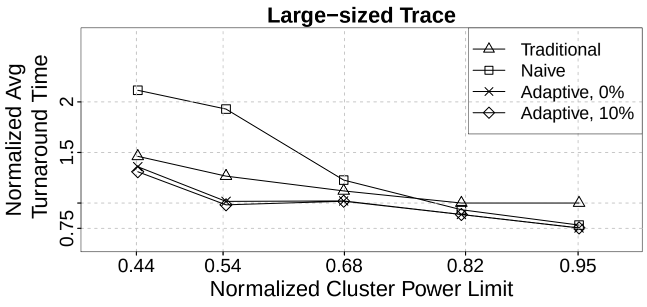


Figure 9: Large-sized Jobs

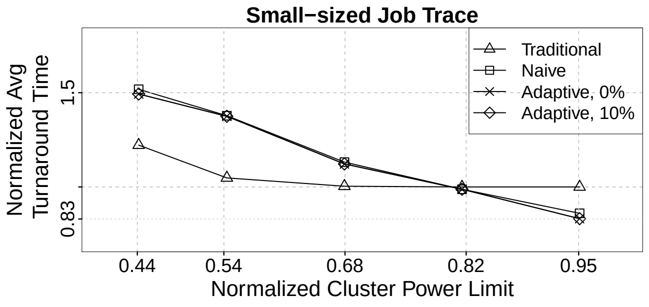


Figure 10: Small-sized Jobs

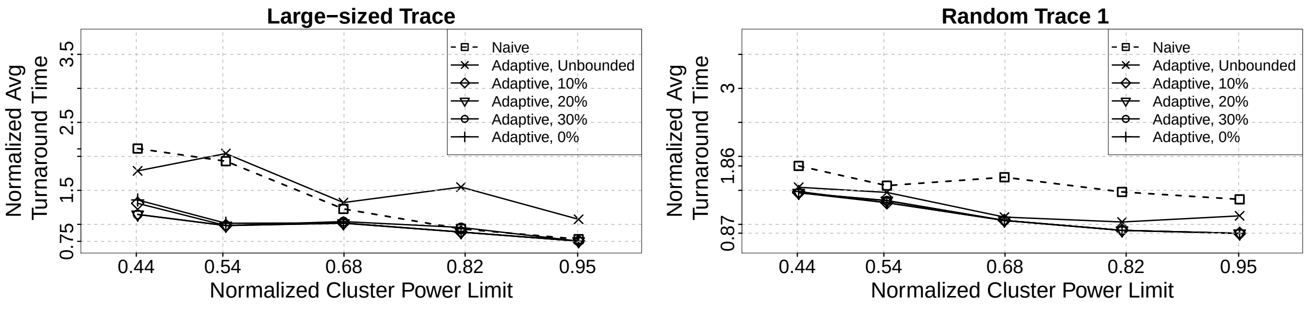


Figure 11: The *Adaptive* policy with varying thresholds, Large and Random Traces

With a sparse arrival rate, we expect significant amount of wasted power in this scenario.

Figure 14 shows a relative timeline for the random trace with current system power data being reported when each job is scheduled (30 data points) for both the *Traditional* policy and the *Adaptive* policy with 0% thresholding. The job arrivals are marked at the bottom with points (blue). The dotted (red) line represents the *Adaptive* policy with 0% thresholding. As can be observed from the graph, the *Adaptive* policy allocates more power (resulting in better utilization), even when there are limited jobs to backfill. The exceptions occur when the *Adaptive* policy runs out of jobs to schedule, while the *Traditional* policy is still scheduling pending jobs (higher wait times).

This can be verified from Figure 15, which shows the corresponding per-job queue times, allocated power, and turnaround times for the *Traditional* policy, and for the *Adaptive* policy with 0% and 20% thresholding. The derived fair-share, job-level power bounds have been shown as well, which apply only to the *Adaptive* policy. The *Adaptive* policy cannot exceed the per-job power bound. The *Traditional* policy has job-level power allocation of 5% more power than that of the *Adaptive* policy in this scenario, as we are looking at 95% of peak power as the cluster power bound (14,000 W).

For this trace, 14 of the 30 jobs did not have to wait in the queue at all (even with the *Traditional* policy). Even when there was no wait time to trade, the *Adaptive* policy improved the turnaround time for 28 of these 30 jobs (except Jobs 6 and 10). It tried to utilize all the power without exceeding the job-level power bound to improve application performance. The average improvement in turnaround time was 13%, and for 8 jobs in the trace this was by more than 2x. The *Traditional* policy fails to utilize the power well, and leads to larger turnaround times.

Another observation here is that at higher global power bounds, as in this example, benefits of the *Adaptive* policy with thresholding (see *Adaptive*, 20% for example) are limited. This is expected as the system is not significantly constrained on power anymore.

7.5 Summary

Our results have yielded three interesting lessons for power-aware scheduling. First, our encouraging results with the *Adaptive* policy show that jobs can significantly shorten their turnaround time with power-aware backfilling and hardware overprovisioning. In addition, by being altruistic, most users will benefit further in terms of turnaround time.

Second, naive overprovisioning, as implemented by the *Naive* policy, can lead to significantly *worse* turnaround times than the non fair-share policy (*Traditional*) for some job traces. An example of this was shown in the random job trace in Figure 8. Careful thought must be put into power-aware scheduling, or average turnaround time may actually *increase*.

Third, the node count requests made by jobs determine the best scheduling policy. The *Adaptive* policy is aimed at the most important jobs in a high-end cluster, which are those jobs that request the more resources. If most jobs are small, a simpler scheme such as the *Traditional* policy is often superior.

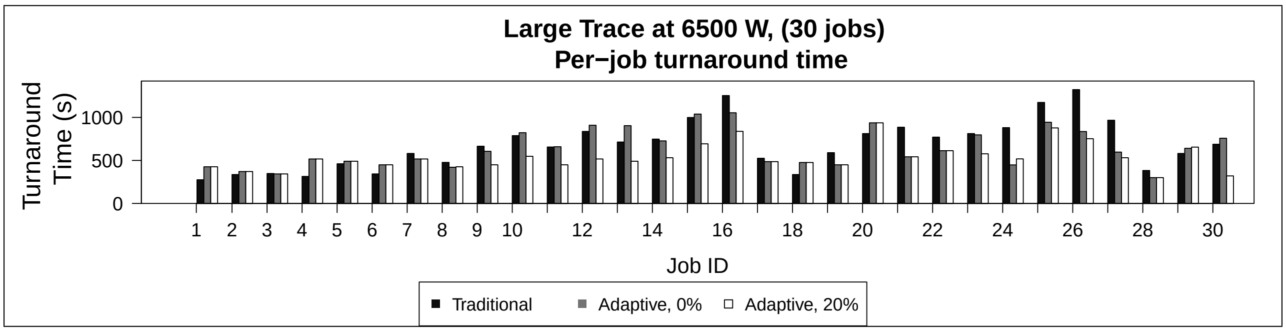


Figure 12: Benefits for Altruistic User Behavior

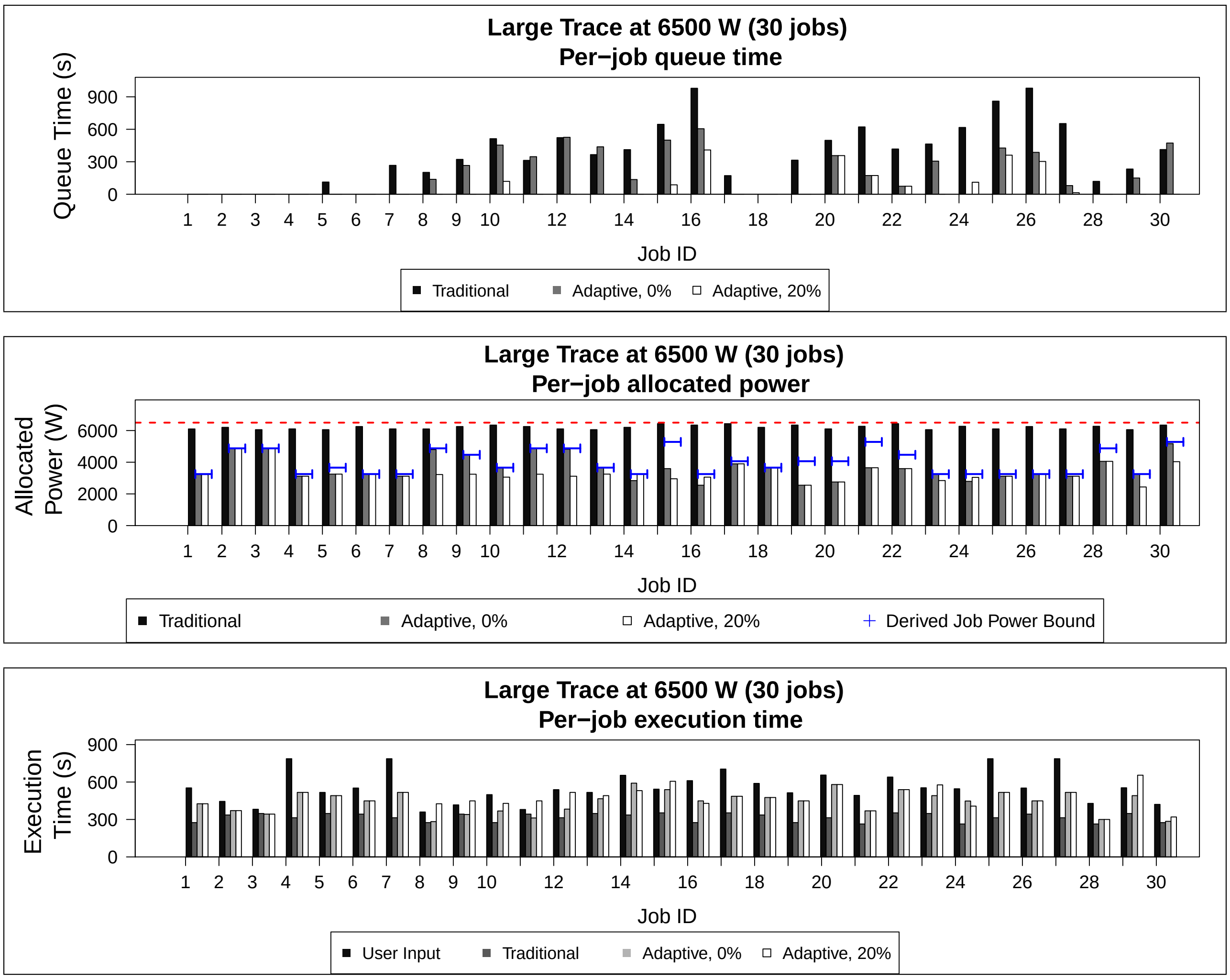


Figure 13: Queue Times, Allocated Power and Execution Times for the Large Trace

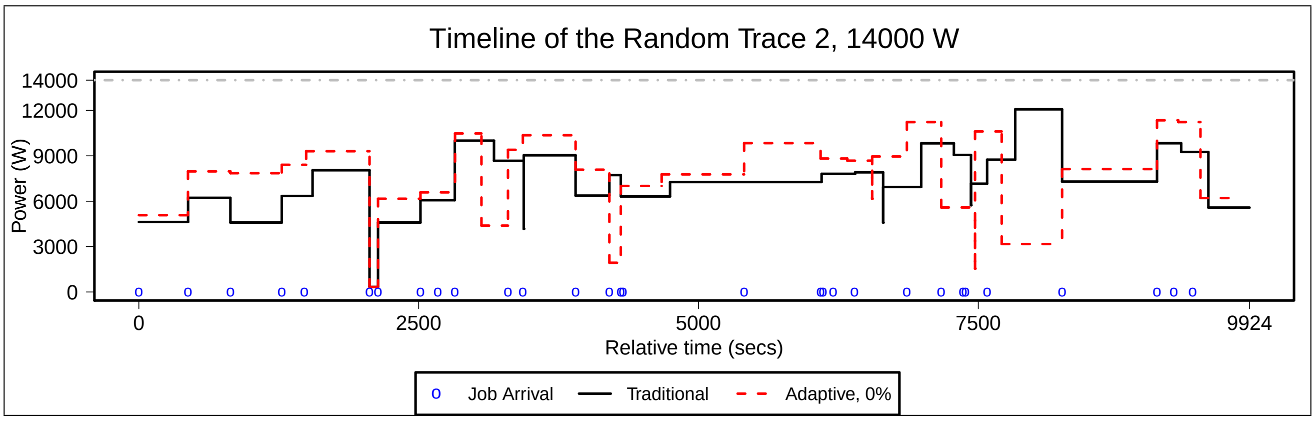


Figure 14: Timeline of Scheduling decisions

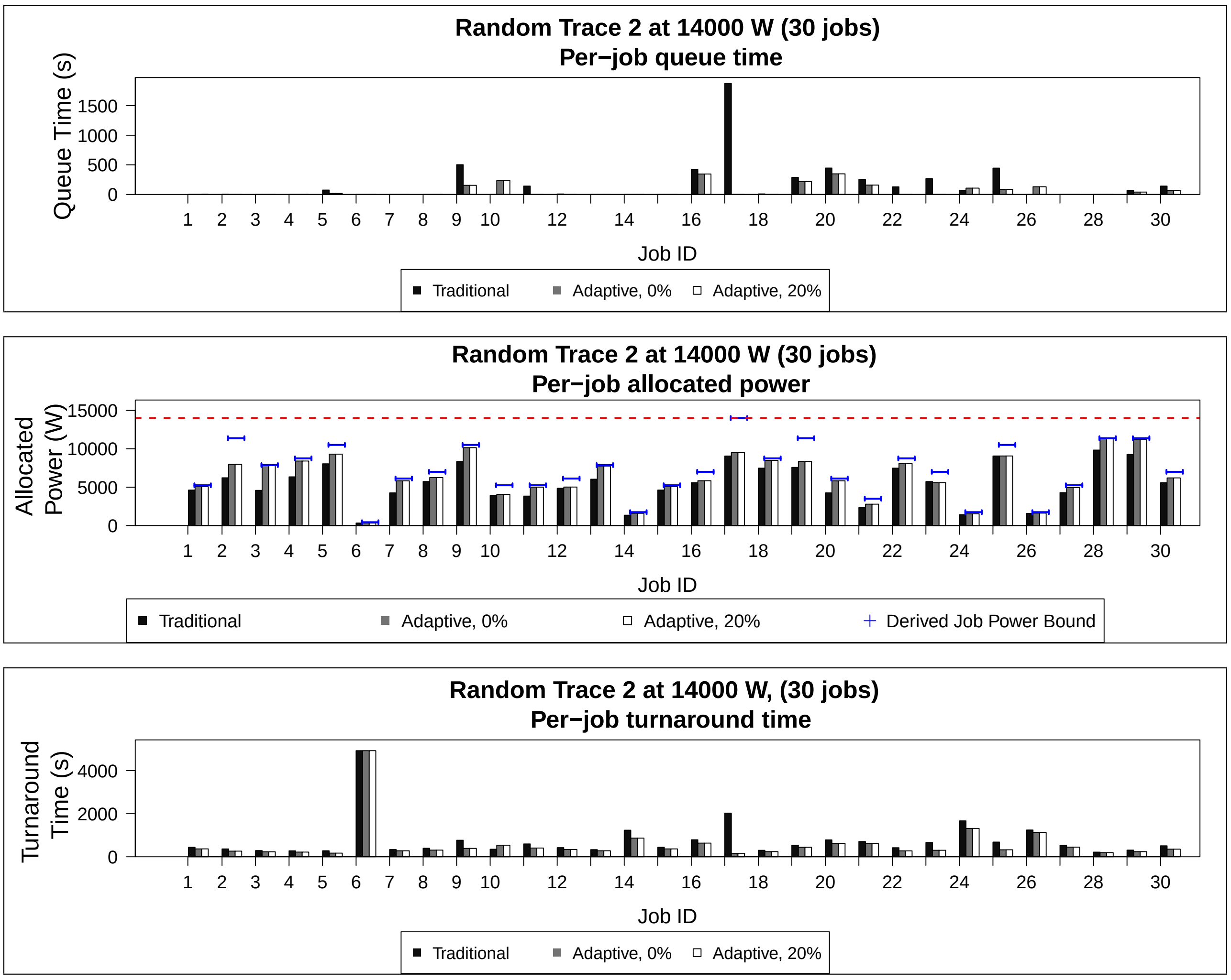


Figure 15: Using power efficiently

8. RELATED WORK

Job scheduling for parallel systems with a focus on backfilling algorithms has been studied widely [6, 16, 19, 20, 24, 29, 30, 40–43]. These studies have examined the advantages and limitations of various backfilling algorithms (conservative versus easy backfilling, lookahead-based backfilling, and selective reservation strategies). Early research in the domain of power-aware and energy-efficient resource managers for clusters involved identifying periods of low activity and powering down nodes when the workload could be served by fewer nodes from the cluster [28, 34]. The disadvantage of such schemes was that bringing nodes back up had a significant overhead. This was overcome by using DVFS-based algorithms [7–12, 31]. Fan et al. [12] looked at power provisioning strategies in data centers. They analyzed power usage characteristics of large collections of servers and proposed a DVFS-based algorithm to reduce energy consumption when CPU utilization is low.

While most of this work identified opportunities for using power efficiently and reducing energy consumption, Etinski et al. [8–11] were the first to look at bounded slowdown of individual jobs and job scheduling under a power budget in the HPC domain. They proposed three DVFS-based policies: Utilization-driven Power-Aware Scheduling (UPAS), Power-Budget Guided (PB-guided), and a Linear Programming based policy called *MaxJobPerf.* The UPAS policy assigns a CPU frequency to the job prior to its launch depending on current system utilization. The PB-guided policy predicts job-level power consumption while meeting a specified bounded slowdown condition. The `MaxJobPerf` policy extends the PB-guided policy to use a reservation condition. Their `MaxJobPerf` policy does better than their other policies due to decreased wait times. They do not consider application configurations and do not analyze overprovisioned systems. Zhou et al. [45] explored knapsack based scheduling algorithms with a focus on saving energy on BG/Q architectures. Zhang et al. further improved this work by using power capping and using leftover power to bring up more nodes when possible. They also explored policies similar to Etinski et al. while using power capping.

Very recently, SLURM developers have looked at adding support for energy and power accounting [22]. However, this work doesn't discuss any new scheduling policies. Bodas et al. [5] explored a policy with dynamic power monitoring to schedule more jobs with stranded power. This work, however has several limitations – the job queue is static and comprises of three jobs, application performance is not clearly quantified, and overall job turnaround times are not discussed. In addition, some of the results depend on the user specifying a frequency of operation for their job and the overhead of dynamic monitoring has not been explored. Sarood et al. [37, 38] developed a policy based on integer linear programming for resource management under a power bound for overprovisioned systems for strongly-scaled applications. This work assumes a specific programming interface with malleability and focuses on maximizing power aware speedup for applications. Our work, on the other hand, applies to general HPC applications, and improves system power utilization and overall job turnaround times. In addition, *RMAP* has significantly less scheduling overhead and derives job-level power bounds in a fair manner.

9. CONCLUSION AND FUTURE WORK

In this paper we discussed *RMAP*, a power-aware resource manager for hardware overprovisioned systems. We designed and implemented three batch scheduling algorithms within *RMAP* using the SLURM scheduler, the best of which is the *Adaptive* policy. The *Adaptive* policy improves the average turnaround time by up to 18.52% when compared to a naive algorithm that uses worst-case power provisioning in addition to increasing system power utilization.

We are currently working on extending *RMAP*. One direction, is to look deeper into existing job queues and analyze them dynamically to determine which scheduling policy will best apply to a set of upcoming jobs. We will also work towards handling different user priorities; this is essential for a full-fledged batch scheduler at an HPC installation. Finally, we will look to integrate our work into realistic next-generation resource managers that are ongoing at multiple sites that support real HPC users.

10. REFERENCES

[1] NASA Advanced Supercomputing Division, NAS Parallel Benchmark Suite v3.3. 2006. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[2] Top500 Supercomputer Sites. June 2013. `http://www.top500.org/lists/2013/6`.

[3] NPFA 70. National electric code 2014. `http://www.nfpa.org/codes-and-standards/document-information-pages?mode=code&code=70`.

[4] A. Batat and Dror Feitelson. Gang scheduling with memory considerations. In *International Symposium on Parallel and Distributed Processing Symposium*, pages 109–114, 2000.

[5] Deva Bodas, Justin Song, Murali Rajappa, and Andy Hoffman. Simple power-aware scheduler to limit power consumption by hpc system within a budget. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, pages 21–30. IEEE Press, 2014.

[6] Robert Davis and Alan Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. In *Technical Report YCS-2009-443, Department of Computer Science, University of York*, 2009.

[7] E.N. Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Power-Aware Computer Systems*, volume 2325 of *Lecture Notes in Computer Science*, pages 179–197. Springer Berlin Heidelberg, 2003.

[8] Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Optimizing Job Performance Under a Given Power Constraint in HPC Centers. In *Green Computing Conference*, pages 257–267, 2010.

[9] Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Utilization driven power-aware parallel job scheduling. *Computer Science - R&D*, 25(3-4):207–216, 2010.

[10] Maja Etinski, Julita Corbalan, Jesus Labarta, and Mateo Valero. Linear Programming Based Parallel Job Scheduling for Power Constrained Systems. In *International Conference on High Performance Computing and Simulation*, pages 72–80, 2011.

[11] Maja Etinski, Julita Corbalan, Jesus Labarta, and

Mateo Valero. Parallel job scheduling for power constrained HPC systems. *Parallel Computing*, 38(12):615–630, December 2012.

[12] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz AndrÃľ Barroso. Power provisioning for a warehouse-sized computer. In *The 34th ACM International Symposium on Computer Architecture*, 2007.

[13] Dror Feitelson. Job scheduling in multiprogrammed parallel systems, 1997.

[14] Dror Feitelson. Workload modeling for performance evaluation. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, pages 114–141, London, UK, UK, 2002. Springer-Verlag.

[15] Dror Feitelson and Morris Jette. Improved utilization and responsiveness with gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 238–261. Springer-Verlag LNCS, 1997.

[16] Dror Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. *Job Scheduling Strategies for Parallel Processing*, pages 1–18, 1995.

[17] Dror Feitelson and Larry Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, pages 1–26, London, UK, UK, 1996. Springer-Verlag.

[18] Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling: A status report. In *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2005.

[19] Dror Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. *Job Scheduling Strategies for Parallel Processing*, pages 1–34, 1997.

[20] Dror Feitelson, Uwe Schwiegelshohn, and Larry Rudolph. Parallel job scheduling - a status report. In *In Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2004.

[21] Eitan Frachtenberg and Dror Feitelson. Pitfalls in parallel job scheduling evaluation. In *Proceedings of the 11th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'05, pages 257–282, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] Yiannis Georgiou, Thomas Cadeau, David Glesser, Danny Auble, Morris Jette, and Matthieu Hautreux. Energy accounting and control with slurm resource and job management system. In *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 96–118. Springer Berlin Heidelberg, 2014.

[23] Intel. Intel-64 and IA-32 Architectures Software Developer's Manual, Volumes 3A and 3B: System Programming Guide. 2011.

[24] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the maui scheduler. In *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 87–102. Springer Berlin Heidelberg, 2001.

[25] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, and Bronis R. de Supinski. Evaluating user-level fault tolerance for mpi applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 57:57–57:62, New York, NY, USA, 2014. ACM.

[26] Lawrence Livermore National Laboratory. The ASCI Purple benchmark codes. `http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html`.

[27] Lawrence Livermore National Laboratory. SPhot–Monte Carlo Transport Code. `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/sphot/`.

[28] Barry Lawson and Evgenia Smirni. Power-aware resource allocation in high-end systems via online simulation. In *International onference on Supercomputing*, pages 229–238, June 2005.

[29] David Lifka. The ANL/IBM SP Scheduling System. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer Berlin Heidelberg, 1995.

[30] A. W. Mu'alem and Dror Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, 2001.

[31] T. Mudge. Power: a first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, 2001.

[32] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring Hardware Overprovisioning in Power-constrained, High Performance Computing. In *International Conference on Supercomputing*, pages 173–182, 2013.

[33] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. High performance linpack. `http://www.netlib.org/benchmark/hpl/`.

[34] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, 2001.

[35] Barry Rountree, Dong H. Ahn, Bronis R. de Supinski, David K. Lowenthal, and Martin Schulz. Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound. In *IPDPS Workshops (HPPAC)*, pages 947–953. IEEE Computer Society, 2012.

[36] O. Sarood, A. Langer, L. Kale, B. Rountree, and B. de Supinski. Optimizing power allocation to CPU and memory subsystems in overprovisioned hpc systems. In *IEEE International Conference on Cluster Computing*, pages 1–8, Sept 2013.

[37] Osman Sarood. *Optimizing Performance Under Thermal and Power Constraints for HPC Data Centers*. PhD thesis, University of Illinois, Urbana-Champaign, December 2013.

[38] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *Supercomputing*, November 2014.

[39] Sanjeev Setia, Mark S. Squillante, and Vijay K. Naik. The Impact of Job Memory Requirements on Gang-scheduling Performance. *SIGMETRICS Perform. Eval. Rev.*, 26(4):30–39, March 1999.

[40] Edi Shmueli and Dror Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 228–251. Springer Berlin Heidelberg, 2003.

[41] Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY LoadLeveler API Project. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer Berlin Heidelberg, 1996.

[42] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *Scheduling Strategies for Parallel Processing, LNCS 2357*, pages 55–71. Springer-Verlag, 2002.

[43] Dan Tsafrir, Yoav Etsion, and Dror Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):789–803, 2007.

[44] Andy Yoo, Morris Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. 2003.

[45] Zhou Zhou, Zhiling Lan, Wei Tang, and Narayan Desai. Reducing energy costs for ibm blue gene/p via power-aware job scheduling. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, pages 96–115. Springer Berlin Heidelberg, 2014.