Practical Resource Management in Power-Constrained, High Performance Computing

Tapasya Patki, David K. Lowenthal
University of Arizona
{tpatki,dkl}@email.arizona.edu

Anjana Sasidharan[*]
Amazon, Inc.
anjans@amazon.com

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

Barry L. Rountree, Martin Schulz, Bronis R. de Supinski
Lawrence Livermore National Laboratory
{rountree4,schulz6,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% (19% on average) and 54% (36% on average) faster average turnaround time when compared to worst-case provisioning and naive overprovisioning respectively.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

Keywords

Power-constrained HPC, Resource Management

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 [34] and designed to be *worst-case provisioned*—all nodes in the system can run at peak power simultaneously, and thus applications are allocated all available power on a node. However, most real HPC applications do not use this allocated power per node, leading to inefficient use of both nodes and power.

An example of this can be found in data that 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 with the only exception being the burn-in phase. 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 recently been proposed as an alternative approach for designing power-limited supercomputers and improving performance [32,39]. 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 explored 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 an interesting research approach, the proposed algorithm is not fair-share and is not sufficiently practical for deployment 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, 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 address some of these problems, less than 1% of scientific HPC applications are

[*]This work was carried out when Ms. Sasidharan was a student at the University of Arizona and is not endorsed by Amazon, Inc.

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3550-8/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2749246.2749262.

expected to support malleability due to the data migration, domain decomposition and scalability issues involved.

We present the design and implementation of *RMAP* (Resource MAnager for Power), a practical resource manager with minimal scheduling overhead (O(1)) that targets future power-constrained, overprovisioned systems. This paper focuses on the design, implementation, and comparison of three policies within *RMAP*: a baseline policy for safe execution under a power bound; a naive policy that uses overprovisioning; and an adaptive policy that is 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 share” strategy. The first is the *Naive* policy, which tries to find the best performing *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 show that users can improve job turnaround times further by altruistically allowing some degradation in their execution time.

Our simple baseline policy, the *Traditional* policy, guarantees safe, correct execution under a power-constraint for systems that are not overprovisioned. The *Adaptive* policy provides 19% and 36% better average per-job turnaround time than the *Traditional* and *Naive* policies respectively. Since the *Naive* policy performs worse than the *Traditional* policy, power-constrained environments require policies such as the *Adaptive* policy.

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

This section motivates the need for overprovisioning-based scheduling. We discuss power profiles of HPC applications and show that applications do not use the allocated power efficiently. We then discuss hardware overprovisioning.

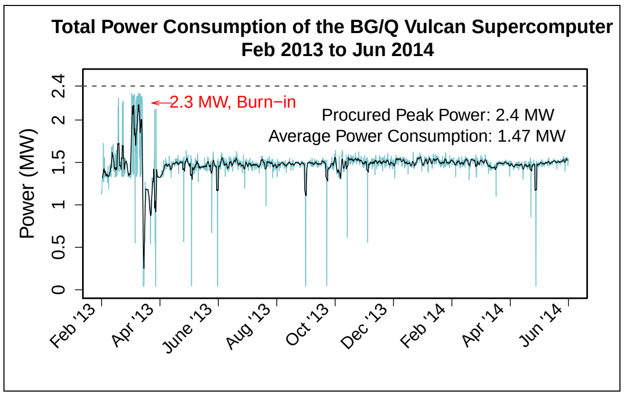


Figure 1: Power Consumption on Vulcan

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 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, 36]. The maximum power available on each socket was 115 W. We only measured socket power, as support to measure memory power was not available due to BIOS restrictions.

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.

We also used four synthetic benchmarks in 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 perform a vector copy in reverse order. Scalability is controlled by adding `MPI_Alltoall` communication. 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 by error bars. While all applications were allocated 115 W per socket, they only used between 66 W (NSC) to 93 W (SPMZ). On average, they only used 81 W or 71% of the allocated socket power.

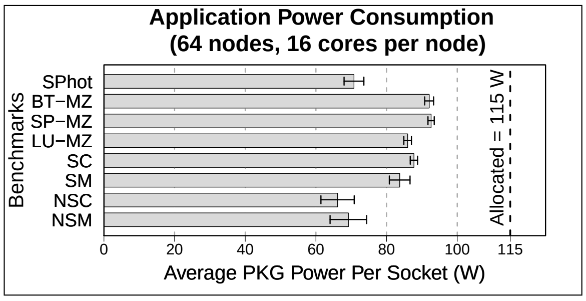


Figure 2: Application Power Consumption

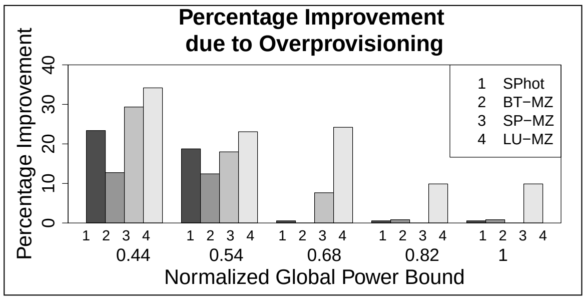


Figure 3: Performance with Overprovisioning

2.2 Hardware Overprovisioning

A cluster is *hardware overprovisioned* with respect to power if it has more nodes than it can fully power simultaneously. Such a cluster can essentially be "reconfigured" based on an application's memory-boundedness and scalability.

The hardware cost for an overprovisioned system depends on the cost of the underlying processor architecture. For example, if more high-end processors are purchased, the hardware cost will increase; however, the system may be more efficient overall because more jobs will complete in its lifetime. The cost might *not* increase, though: power-inefficient processors typically have a lower unit price than power-efficient processors. Thus, a hardware cost budget provides a choice between more power-inefficient processors leading to better job throughput and performance under a power constraint, or fewer, power-efficient nodes on a non-overprovisioned cluster (which may lead to wasted power).

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. This benefit requires 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 (*moldable*).

We emulated overprovisioning by enforcing socket-level power caps with Intel's RAPL technology. The minimum RAPL socket power cap that we could enforce (within the processor's specification) was 51 W, and the 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 from 8 to 64 nodes (increments of 4) and 8 to 16 cores per node (increments of 2). We disabled Turbo Boost when we enforced power caps, except for the 115 W power bound, for which we enabled Turbo Boost. 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).

With n_{max} being the maximum number of nodes that one can run at peak power without exceeding the power bound, the worst-case provisioned configuration is $(n_{max} \times 16, 115)$.

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 at the five global power bounds discussed above. The x-axis is normalized to the worst-case provisioned power (14,720 W). For our dataset, we saw a maximum improvement of 34% (11% on average) in performance compared to worst-case provisioning. Note that choosing the correct application-specific configuration is important even when there is no global power bound, as shown by LU-MZ at 14,720 W, and is increasingly critical under a power constraint. Previous results have indicated that overprovisioning can improve performance by up to 62% [32, 39].

3. POWER-AWARE SCHEDULING

This section discusses HPC scheduling basics and backfilling. We then discuss the design challenges for *RMAP* and present the details of our three scheduling policies.

3.1 Basics

Users at HPC sites 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; the estimated runtime is used to set a deadline ($t_{deadline}$) for the job. The job is killed if it exceeds this deadline. Depending on the job-size, most HPC users are required to use specific partitions. For example, most high-end clusters have a *small* debug partition that specifically targets small-sized jobs, and a general-purpose *batch* partition for medium and large-sized jobs.

Resource requests are maintained in a *job queue*, and users are allocated dedicated nodes based on a scheduling policy. One such policy is First-Come First-Serve (FCFS), which services jobs strictly in the order 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* [4,15,40], are not feasible on supercomputers because memory demands for HPC applications are typically quite high—which leads to large paging costs[1].

3.2 Backfilling

Backfilling [24,29,30,42] addresses the convoy effect caused by FCFS by executing smaller jobs out of order on idle nodes and by improving utilization—in turn reducing the overall average turnaround time. Backfilling has two variants: *easy* and *conservative*. Easy backfilling allows short jobs to execute out of order as long as they do not delay the *first* queued job. Conservative backfilling, on the other hand, only lets

[1]Many HPC sites use operating systems that do not page.

short jobs move ahead if they do not delay *any* queued job. Easy backfilling performs better for most workloads [30].

Backfilling frequently uses a greedy algorithm that picks the *first-fit* from the job queue. The *first-fit* might not always be the *best-fit*, and a job further down the queue may fit the hole being backfilled better. Finding the *best-fit* involves scanning the entire job queue, which increases job scheduling overhead significantly [41].

3.3 Design Challenges

Power-aware schedulers must enforce job-level power bounds as they manage and allocate nodes. They need to optimize for overall system throughput as well as individual job performance under the job-level power bounds, which means they must minimize the amount of unused (leftover) power.

For simplicity, we assume that all jobs 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 (n_{req} being the number of requested nodes). Thus, $P_{job} = \frac{n_{req}}{N_{cluster}} \times P_{cluster}$.

This allocation for P_{job} can be extended easily 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 $w_{prio} > 1$, and lower priority jobs could be allocated using $w_{prio} < 1$. This paper does not explore priorities further.

At any given point in time (say t), the available power in the cluster, P_{avail_t}, can be calculated by subtracting the total power consumption of running jobs from the cluster level power bound. P_{avail_t} is used to make power-aware scheduling decisions. Thus for r running jobs at time t,

$$P_{avail_t} = P_{cluster} - \sum_{j=1}^{r} P_{job\_j}$$

The performance of an individual job can be optimized by using overprovisioning with respect to the job-level power bound, P_{job}. To optimize system throughput and to minimize unused power, 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 cores per node) at runtime. However, varying the node count at runtime (malleability) is not possible with the current MPI standard. In addition, dynamically changing the node and core counts of a job would incur data decomposition and migration overhead [25].

We explore extensions of traditional backfilling for a power-aware scenario. Traditional backfilling attempts to utilize as many nodes as possible in the cluster by breaking the FCFS order. Similarly, our new greedy approach, *power-aware backfilling*, attempts to use as much global power as possible by scheduling a job with currently available power. Most cases involve sacrificing some performance benefits attained from overprovisioning. The key idea is to schedule

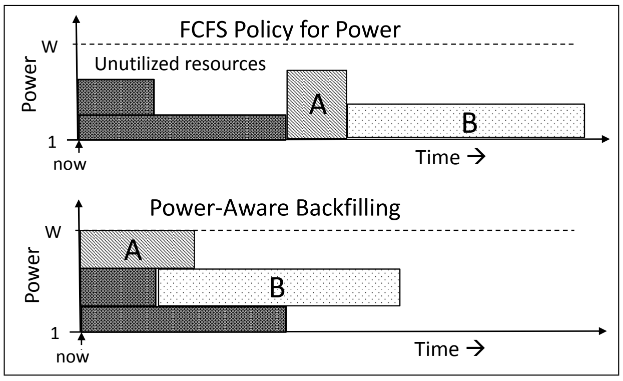


Figure 4: Advantage of Power-Aware Backfilling

a job with less power than 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 extremely power-constrained scenarios and to scenarios with too much leftover power. Our approach builds on standard backfilling.

Figure 4 shows an example in which we assume that jobs `A` and `B` are currently waiting in the queue. Job `A` requested more power than is currently available in the system. Traditionally, Job `A` waits in the queue until enough power is available, which wastes resources. Our approach schedules Job `A` immediately with the *available* power. While we increase the execution time of Job `A`, our approach improves the overall turnaround time of Job `A` as well as the other jobs in the queue, and utilizes power better.

The key idea is to use power-aware backfilling while adhering to the user-specified time deadline for the job. Overprovisioning under a job-level power bound will often exceed the user's performance expectations. However, allowing users to specify a maximum slowdown for their job and thus trade their job's execution time for a faster turnaround time is an added incentive. We focus primarily on trading some of the benefits obtained from overprovisioning to utilize all available power to run jobs faster and to schedule more jobs.

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 per-job 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 the *Adaptive* policy. We derive, P_{job}, which is the job-level power bound based on the user input, as discussed in the previous subsection. This job-level power bound is an input to our scheduling policies (see Table 1). All three policies use basic node-level 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

3.4.1 The *Traditional* Policy

In this policy, the user is allocated a configuration with their requested node count that uses all available cores on a node at maximum possible power. A job that requests large node counts may exceed the system's global power bound. In this case, 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 (an unfair job-level power allocation). Alternatively, we could reject the job 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 total 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 allocates the $(n_{req} \times c_{max}, p_{max})$ configuration if $P_{(n_{req} \times c_{max}, p_{max})} \leq P_{cluster}$.

Otherwise, it allocates the $(n_{max} \times c_{max}, p_{max})$ configuration to the job, where n_{max} is the maximum n such that $P_{(n \times c_{max}, p_{max})} \leq P_{cluster}$.

Note that the job will have to wait in the queue if enough resources (nodes and power) are not available when the scheduling decision is being made (if $P_{avail_t} < P_{job}$).

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 allocates the $(n \times c, p)$ configuration that leads to the best time t_{act} under that power bound. Thus, $t_{act} = min(T)$, where $T = \{t_{(n \times c,p)} : P_{(n \times c,p)} \leq P_{job}\}$.

If $t_{act} > t_{req}$, the system sets the deadline $t_{deadline}$ for the job to t_{act} instead of t_{req} during job launch, so that the job does not get killed prematurely. This scenario occurs if the 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 provide 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. *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%) [44].

Again, note that the job may have to wait in the queue until enough power is available to schedule it.

3.4.3 The *Adaptive* policy

This policy's goal is to allow (1) users to receive better turnaround time for their jobs, and (2) the system to minimize the amount of unused power to 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 specify an optional *threshold (th)*, which denotes the percentage slowdown that the job can tolerate. When *th* is not specified, we assume that it is zero (no slowdown).

The *Adaptive* policy uses the suggested job-level power bound to check if the requested amount of power is currently available. If so, 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. The administrative advantage is better resource utilization (in terms of nodes and overall power) and throughput.

More specifically, if $P_{avail_t} > P_{job}$, the *Adaptive* policy uses the same mechanism as the *Naive* policy. However, when $P_{avail_t} < P_{job}$, it determines $t_{act} = min(T)$, where $T = \{t_{(n \times c,p)} : P_{(n \times c,p)} \leq P_{avail_t}\}$, and schedules the job immediately with the $(n \times c, p)$ configuration with time t_{act} as long as $t_{act} <= (1 + th) \times t_{req}$. Thus, the job's wait time is reduced while meeting the performance requirement.

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

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 three scenarios in which 750 W of power and 10 nodes are available in the cluster, and job `A` that is currently executing terminates in 1000s. Also, a user has requested 6 nodes for 450s, and, the derived job-level power bound is 800 W.

3.5.1 ***Scenario 1, Traditional Policy***

Here, *RMAP* allocates configuration `C1` to the job but it waits until job `A` terminates and enough power is available.

3.5.2 ***Scenario 2, Naive Policy***

RMAP allocates configuration `C2` to the job but also waits until job `A` terminates and enough power is available.

3.5.3 ***Scenario 3, Adaptive Policy (threshold=0%)***

A threshold value of 0% means that the user cannot compromise on performance. Under the *Adaptive* policy, *RMAP* checks if enough power (800 W) is available in the system. It then determines that `C3` does not violate the performance constraint (450s), and job `A` can be launched immediately with the currently available power (750 W). We distinguish this case from Scenario 2, which will *always* pick `C2`.

Picking `C3` reduces the wait time of the job significantly (by 1000s). Also, in scenarios 1 and 2, 750 W of power is wasted for 1000s. In this scenario, power is utilized more efficiently and turnaround time for the job is reduced.

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

Table 3: Schema for Job Details Table

4. RMAP IMPLEMENTATION

We implemented *RMAP* within the widely-used, open source resource manager for HPC clusters, SLURM [45]. SLURM is used on several Top500 [2] supercomputers. It provides a standard framework for launching, managing and monitoring jobs 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 similarly be implemented within other resource managers.

Our scheduling policies require the ability to produce execution times for a given configuration under a job-level power bound. Table 3 shows the information that P-SLURM requires. We refer to this as the `job_details_table`, and we added this table to the existing `slurm_acct_db`. Values for `exec_time` and `tot_pkg` can be measured or predicted.

We developed a model to predict the performance and total power consumed for application configurations in order to populate this table. Section 5 presents the details of this model. Furthermore, to understand and to 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 in Section 2). For simplicity, we populated both tables 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 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 that we built contained 2840 data points, with 5 different power caps, 15 different node counts, 5 different core counts per node and 8 applications.

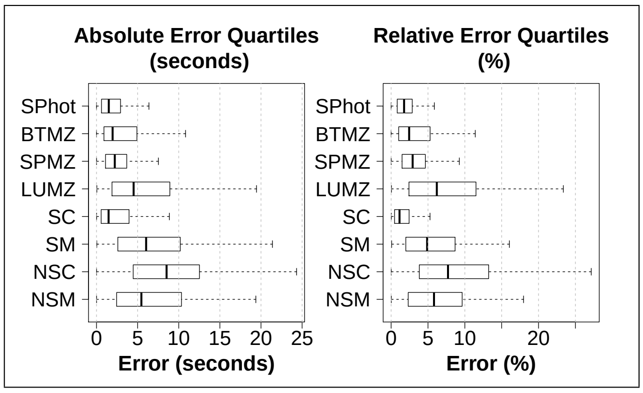


Figure 5: Error Quartiles of Regression Model

We used 10% of this data for training and obtained application-specific linear regression parameters that allow us to predict application execution time and total package power consumption at a given configuration. We used a logarithmic polynomial regression of degree two. We limited our power predictions to package power only as memory power measurements were unavailable on our cluster.

We validated our models with our previously measured data. When using only 10% of the data for model training, the average error for execution time is below 10%, and the maximum error is below 33%. Figure 5 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%, and the median is below 8%.

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 us to exceed the cluster-level power bound (worst-case scenario). In our model, for 96% of our data, the under-prediction 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 (15% less than the actual bound), or by relying on the common practice of designing supercomputing facilities to tolerate such surges [3].

6. EXPERIMENTAL DETAILS

In order 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 consider 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 P-SLURM. 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).

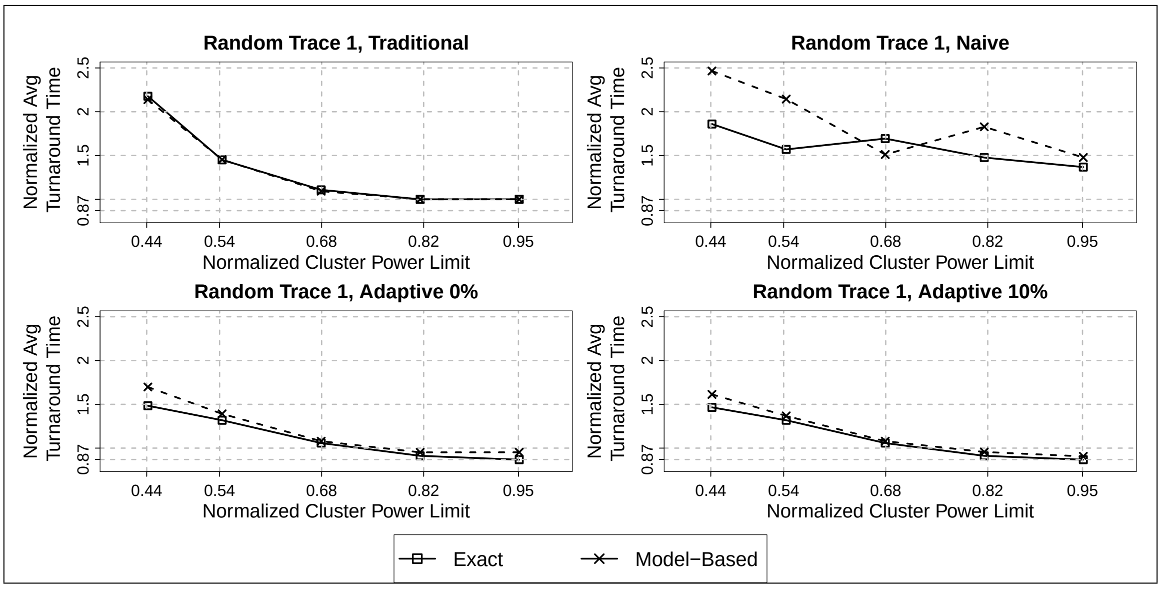


Figure 6: Model Results on the Random Trace

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 the *Adaptive* policy. We select the following types of job traces to evaluate our scheduling policies.

- Traces with *small-sized*[2] and *large-sized* jobs: To identify scenarios which may favor one power-aware scheduling policy over another, we create trace files with small-sized and large-sized jobs. Users could 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 could 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 have the same number of jobs and used the same arrival rate parameter in the Poisson process, *Random Trace 1* has many 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 slowdown (threshold value for the *Adaptive* policy). For readability, we do *not* center our graphs 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 × 115 W × 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 at 14,720 W (lower is better). The *Traditional* policy mimics worst-case provisioning, which unfairly allocates per-job power 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) decision complexity, so we do not compare their scheduling overheads.

We start by evaluating the model discussed in Section 5 when applied to *RMAP* and its policies. We then compare and analyze the three policies by applying them to different traces at several global power bounds. Finally, we analyze two traces in detail to explore how altruistic behavior on the 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

This section explores the impact of using our model for predicting application configuration performance and power. Figure 6 compares average turnaround time for *Random Trace 1* at 5 different global power caps. Configuration performance and total power consumed are predicted for each job in the trace. The former is used for determining execution time, and the latter is used to determine available power. For the *Traditional* and *Adaptive* policies, our model is accurate (error is always under 10%; and is 4% on average across the two policies). We observe similar results with the other traces.

While performance prediction introduces error and affects overall turnaround times, the errors introduced by overpre-

[2]Due to space limitations, we do not present the results on the small-sized trace in this paper. Our results indicate that the *Traditional* policy always does better for traces with several small jobs, as there is limited wait time [33].

diction of the total power consumed by a configuration propagates and impacts the turnaround time more. Scheduling and backfilling decisions can be significantly affected when they depend on available power. For example, at a lower cluster power bound, if we overpredict 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, particularly for the *Naive* policy at lower global power bounds.

In the subsections that follow, we conservatively establish the *minimum improvements* that the *Adaptive* policy can provide. For this purpose, we use oracular information for the *Traditional* and *Naive* policies, which are our baselines, and the model for the *Adaptive* policy.

7.2 Analyzing Scheduling Policies

In this subsection, we compare and analyze the power-aware scheduling policies on different job traces.

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 7 shows that the *Adaptive* policy with a threshold of 0% improves the turnaround time by 22% when compared to the *Naive* policy and by 14% when compared to the *Traditional* policy on average (up to 47% and 25%, respectively). The *Adaptive* policy with a threshold of 10% further improves the overall turnaround time by 16% 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, which can lead to longer wait times when the best configuration uses a large number of nodes.

7.2.2 Random Traces

Figure 6 (from the previous subsection) compares the three policies for *Random Trace 1*. The *Adaptive* policy with a threshold of 0% does 19% better than the *Traditional* policy and 36% better than the *Naive* policy on average (up to 31% and 54%, respectively), for both the random traces.

Policies may lead to larger turnaround times at higher global power bounds in some cases, such as the *Naive* policy at 10,000 W (a normalized value of 0.68 in Figure 6). 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). Thus, other jobs in the queue incur longer wait times.

Figure 8 depicts the impact of varying threshold values on the *Adaptive* policy for the large-sized and the random traces (*Random Trace 1*). We compare it to the baseline *Naive* policy, which does not exploit slowdown thresholds. We show threshold values that tolerate a slowdown of 0% to 30%. For large jobs, thresholding helps the user improve the turnaround time for their job by greatly decreasing queue time. However, when queue wait times are short, as with small-sized jobs, we expect that adding a threshold will lead to larger turnaround times. The random traces have a mix of small-sized and large-sized jobs. For our traces, the *Adaptive* policy with thresholding up to 30% either improves average turnaround time (by up to 4%) or maintains 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, which we show for comparison, leads to worse turnaround times.

For the large trace, the *Adaptive* policy with a threshold of 0% does 34% better on average than the unbounded *Adaptive* policy. Slowing down by 10% to 30% improves the average turnaround time by 2% on average (up to 4%) when compared to the *Adaptive* policy with 0% thresholding. For the other three traces, the improvement obtained by slowing down the jobs is under 3% on average when compared to *Adaptive* policy with a threshold of 0%. This improvement depends on the power bound as well as the job mix. The numbers reported in this section are averaged across all global power bounds for the traces. We analyze per-job performance for a single trace at a fixed global power bound in the next subsection.

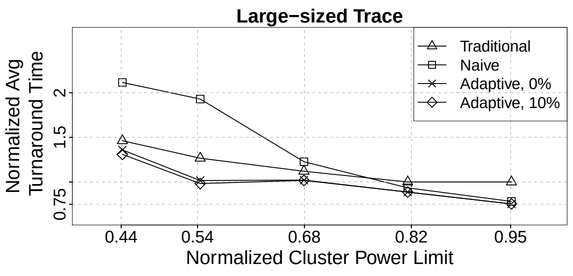


Figure 7: Large-sized Jobs

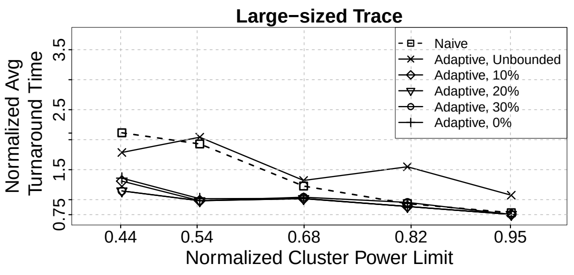
Policy	Average Turnaround Time (s)
Traditional	684
Naive	990
Adaptive, 0%	636
Adaptive, 10%	613
Adaptive, 20%	536
Adaptive, 30%	536

Table 4: Average Turnaround Times

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. Each job requests at least 40 nodes, so all jobs are allocated the entire 6,500 W ($P_{cluster}$) with the *Traditional* policy (as the scheduler runs out of power), leading to unfair power allocation, sequential schedules and no opportunity for backfilling. The trace contains 30 jobs.

Figure 9 shows individual job turnaround time for the *Traditional* policy, and for the *Adaptive* policy with 0% and 20% thresholding. Table 4 shows the absolute values of average turnaround times for the job trace for all policies. We limit the graph to our main policies.



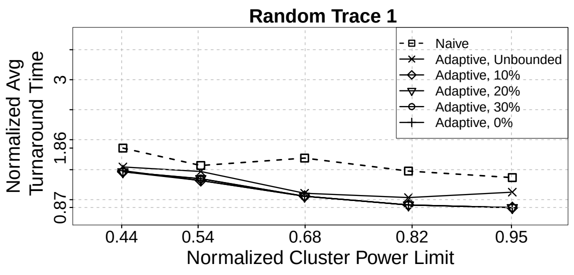


Figure 8: The *Adaptive* Policy with Varying Thresholds, Large and Random Traces

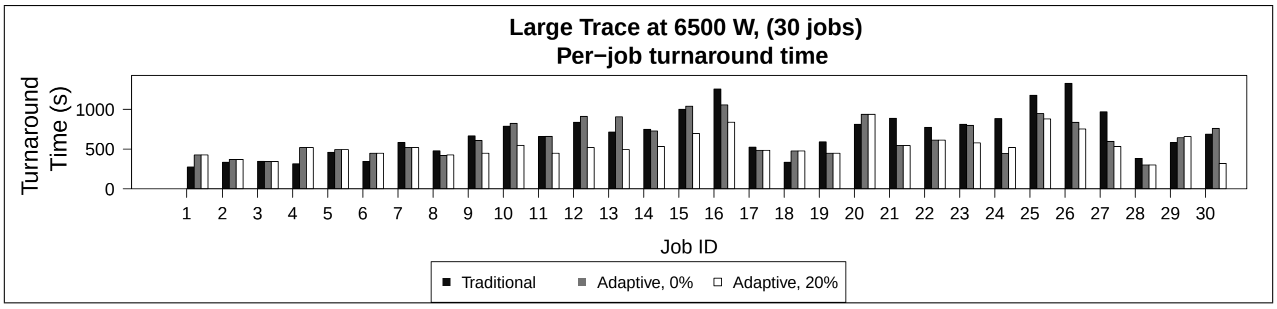


Figure 9: Benefits for Altruistic User Behavior

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 though they are not altruistic. The average turnaround time improved by 7% for 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, improving the average turnaround time of the job queue by 11% and 21% when compared to the *Traditional* policy, which 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 58% better, for job 30; and 13% on average) than 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 2%). The average turnaround time for the queue improved by 21%, as discussed previously. These benefits come from power-aware backfilling as well as hardware overprovisioning.

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. Several reasons explain this increase. First, all jobs were allocated significantly more power with the *Traditional* policy (because no fair-share derived power bound was used, resulting in allocating the entire power budget to most jobs) and executed with Turbo Boost enabled (as no power capping was enforced), resulting in faster execution 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, with the *Adaptive* policy, when the job's execution time increases, its turnaround time increases as well, because there is no queue wait time to trade for. Despite these issues, the *Adaptive* policy with 0% thresholding improved the turnaround times for 17 out of 30 jobs, which shows the benefits of altruism. For this example, the utilization of system power by both the *Traditional* and *Adaptive* policies was high, leaving little unused power, mostly because the global power bound was tight (50% of peak) and the jobs were large-sized.

7.4 Power Utilization

We now analyze a random job trace (*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, our results are conservative due to the sparse job arrival rate in our dynamic job queue (short queue times in general) so we can test the limits of our *Adaptive* policy. With a sparse arrival rate, we expect significant unused power.

Figure 10 shows the per-job allocated power and turnaround time 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 *Traditional* policy has job-level power bounds 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 wait in the queue at all (even with the *Traditional* policy). Even with no wait time, the *Adaptive* policy improved the turnaround time for 28 of these 30 jobs (except Jobs 6 and 10). It tried to utilize all power without exceeding the job-level power bound to improve application performance. The average improvement in turnaround time was 13%, and by more than 2x for 8 jobs in the trace. The *Traditional* policy fails to utilize the power well, and leads to larger turnaround times. Also, at

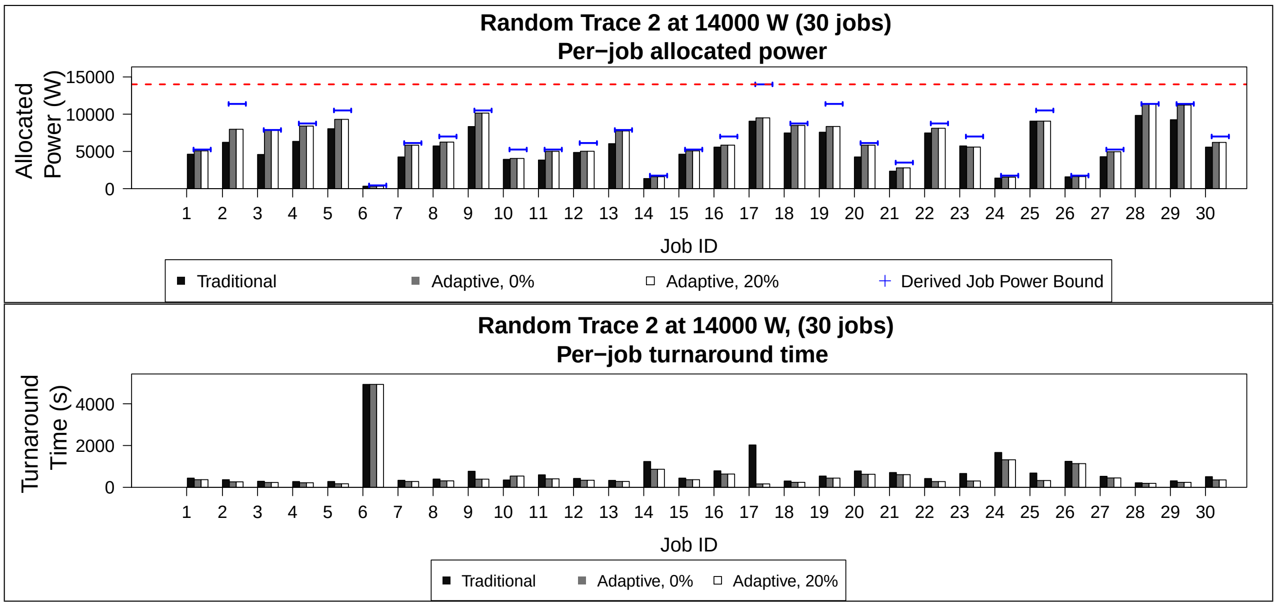


Figure 10: Power Utilization

higher global power bounds, as in this example, benefits of the *Adaptive* policy with thresholding (see *Adaptive*, 20% for example) are limited, which is expected as the system is not significantly power constrained.

7.5 Summary

Our results have yielded three important 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, as with the random job trace and the results that Figure 6 shows. Power-aware scheduling requires advanced policies or average turnaround time may actually *increase.*

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

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, 41–44]. 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, 35]. The disadvantage of such schemes was that bringing nodes back up had a significant overhead.

DVFS-based algorithms can avoid this cost [7–12, 31]. Fan et al. [12] looked at power provisioning strategies in data centers and proposed a DVFS-based algorithm to reduce energy consumption for server systems.

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; however, they did not consider application configurations or power capping and did not analyze overprovisioned systems. Zhou et al. [47] explored knapsack-based scheduling algorithms with a focus on saving energy on BG/Q architectures. Zhang et al. [46] further improved this work by using power capping and using leftover power to bring up more nodes when possible.

Recently, SLURM developers have looked at adding support for energy and power accounting [22]. However, this work does not 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 three jobs, application performance is not clearly quantified, and overall job turnaround times are not discussed. Sarood et al. [37, 38] developed an ILP-based policy 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. As discussed earlier, this scheme has a high scheduling overhead, and less than 1% of real HPC codes are expected to support malleability. Our work, specifically the *Adaptive* policy, 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 leads to 19% faster average turnaround time when compared to the traditional algorithm that uses worst-case power provisioning. It also increases 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 upcoming jobs. We will also work towards handling different user priorities, which is essential for a production batch scheduler. Finally, we will look to integrate our work into realistic next-generation resource managers being developed at multiple sites that support real HPC users.

10. ACKNOWLEDGMENTS

We thank Livermore Computing and their support staff for providing us with the appropriate permissions required to access the MSRs. We also thank Dr. Ghaleb Abdulla for helping us gather data from Vulcan. This material is based upon work supported by the National Science Foundation under Grant No. 1216829. In addition, part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-669277).

11. 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. November 2014. `http://www.top500.org/lists/2014/11`.

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

[4] Anat 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] Elmootazbellah 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 AndrÃl' 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/spho%t/`.

[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] Ahuva 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] Trevor 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] Tapasya Patki, Anjana Sasidharan, Matthias Maiterth, David Lowenthal, Barry Rountree, Martin Schulz, and Bronis de Supinski. Practical Resource Management in Power-Constrained, High Performance Computing. TR 01-15, University of Arizona, January 2015. `http://www.cs.arizona.edu/people/tpatki/tr01-15.pdf`.

[34] Antoine Petitet, Clint Whaley, Jack Dongarra, and Andy Cleary. High Performance Linpack. `http://www.netlib.org/benchmark/hpl/`.

[35] 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.

[36] 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.

[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] Osman Sarood, Akhil Langer, Laxmikant V. Kale, Barry Rountree, and Bronis R. 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] Ziming Zhang, Michael Lang, Scott Pakin, and Song Fu. Trapped Capacity: Scheduling under a Power Cap to Maximize Machine-room Throughput. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, pages 41–50. IEEE Press, 2014.

[47] 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.