I/O Aware Power Shifting

Lee Savoie and David K. Lowenthal
Dept. of Computer Science, The University of Arizona

Bronis R. de Supinski, Tanzima Islam, Kathryn Mohror, Barry Rountree, Martin Schulz
Lawrence Livermore National Laboratory

***Abstract*—Power limits on future high-performance computing (HPC) systems will constrain applications. However, HPC applications do not consume constant power over their lifetimes. Thus, applications assigned a fixed power bound may be forced to slow down during high-power computation phases, but may not consume their full power allocation during low-power I/O phases. This paper explores algorithms that leverage application semantics—phase frequency, duration and power needs—to shift unused power from applications in I/O phases to applications in computation phases, thus improving system-wide performance. We design novel techniques that include explicit staggering of applications to improve power shifting. Compared to executing without power shifting, our algorithms can improve average performance by up to 8% or improve performance of a single, high-priority application by up to 32%.**

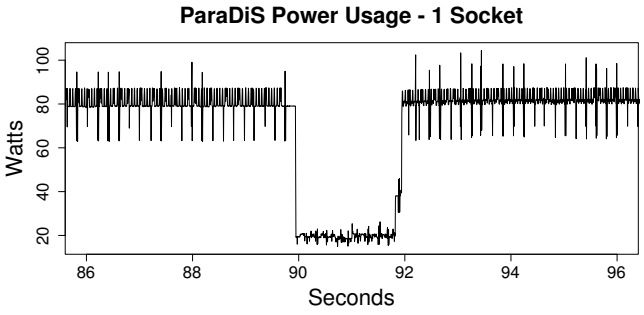

Fig. 1. ParaDiS computation (high) and I/O (low) power use

I. Introduction

Power is a significant concern for upcoming exascale systems [1]. Today's fastest systems only have tens of petaflops of capability while using about 10MW. Without significant software and hardware changes, we will not meet exascale power budgets. Because of these future power constraints, applications will soon execute with strict power limits and will rely on advances in system software to meet them.

Many researchers have explored single application performance under a power budget. One solution provides each application an amount of power based on the number of nodes that it uses and only shifts power *within* those nodes [2]. However, we can significantly improve system performance by shifting power *across* concurrently running applications. Figure 1 shows that ParaDiS uses more than 80W of power during computation, but when it enters an I/O phase at around 90 seconds, power use drops to around 20W. This pattern is typical of applications with periodic I/O phases [3], [4]. If the per-socket power limit is between 25W and 80W, ParaDiS will have to slow down during computation phases to meet the power limit, but it will have excess power during I/O phases. We can dynamically shift this excess power to applications that are executing computation phases to improve system-wide performance without affecting the "donor" application.

In this paper, we develop and analyze algorithms that leverage application semantics to shift power during I/O phases. The semantics capture key characteristics of computation and I/O phases, including their start times, frequencies, durations, and power needs. We implement our algorithms in a simulator, called *PowerShifter*, to study scenarios that vary job size distribution, number of jobs, application power consumption, cluster power limit, and I/O phase frequency and duration.

We investigate several power shifting strategies. Our base strategy, which often performs well, simply shifts power whenever an I/O phase occurs. We also design and implement more sophisticated algorithms that potentially delay some applications to achieve a schedule that limits the number of applications that incur I/O concurrently. Thus, an application in an I/O phase can more easily identify a target "partner" application for power shifting. *PowerShifter* handles several challenges, including job failures, job arrivals, and job completions, by changing the schedule on the fly, in response to these unpredictable events. This paper makes several contributions:

- The first I/O-aware power shifting algorithms;
- A demonstration that a simple, on-demand algorithm improves average performance, while complex algorithms are effective for systems with few, large jobs;
- An implementation of our algorithms in *PowerShifter* and a study of the impact of workload characteristics including job power needs, job count, and job sizes;
- A validation of *PowerShifter* for three real applications: Cactus, ParaDiS, and LAMMPS.

PowerShifter shows that our algorithms improve performance of a single application up to 32% and average performance up to 8% compared to executing without power shifting. Moreover, scheduling I/O phases explicitly can improve performance by 12% over straightforward algorithms. Many of our experiments are driven by traces from real machines.

This paper is organized as follows. Section II details our assumptions, Section III presents our novel algorithms, and Section IV describes *PowerShifter*. Section V discusses our implementation on a real cluster and its use in validating *PowerShifter*, and Section VI presents our experimental results.

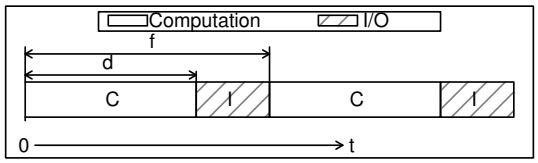

Fig. 2. Basic model

II. Assumptions

We assume a high-performance computing (HPC) system with N nodes and a global power bound *P*. Each application can use a different node count; the sum of the node counts is less than or equal to N. In the default case (i.e., no power shifting), we assume a uniform power bound of P/N per node.

We assume that applications alternate between computation phases and I/O phases. I/O phases generally record checkpoints or visualization data. Checkpoints save application state periodically to enable a restart if a failure occurs. Visualization data similarly represents periodic system state. Thus, both types of I/O generally occur periodically. We group them in the same category as *I/O phases*, which generally use little power because techniques such as DMA allow the CPU to idle. The power needs of a computation phase depend on the mix of CPU operations, cache hits, and main memory accesses, so different applications (phases) react differently to additional power. We assume that computation phases consume more power than I/O phases in most cases.

We define the following parameters: each computation (or I/O) phase lasts d time units, consumes p power, and starts every f time units (see Figure 2). In most cases, the value of d for computation phases is chosen as a multiple of the length of a program-level timestep. In general, an application may have multiple types of computation or I/O phases, such as checkpointing and visualization output. For simplicity, we assume each application has one type of each phase. We initially assume that I/O phase duration is independent of the degree of I/O concurrency (relaxed in Section VI-D).

We assume that *PowerShifter* has access to the parameters described above (d, f, and p) for each application. Thus, it can determine when I/O and computation will occur. Phase timing is often available in explicit user configurations or most checkpoint or visualization data libraries. We can obtain power information by profiling [5] or program analysis. Most of our algorithms (see Section III) need less information—just the power needs of the *current* phase. For simplicity, we only consider CPU power. We could potentially shift power between other components to achieve additional benefit. We also ignore any extra available power due to idle nodes.

We allow the set of executing applications to change over time due to job arrivals and completions. Users typically submit jobs by specifying a node count and an estimated run time. The resource manager maintains requests in a job queue and allocates nodes to them, using space sharing [6]. We assume that the scheduler uses a first-come, first-served (FCFS) approach, but our I/O-aware power shifting algorithms are easily extended to handle backfilling-based algorithms.

Algorithm 1 *Spread* algorithm

```
1:  function ENTER_IO
2:      decrease my.allocatedPower by my.freePower
3:      my.apps = apps in compute phase that can use power
4:      my.donationAmount = my.freePower/sizeof(my.apps)
5:      for i in my.apps do
6:          increase i.allocatedPower by my.donationAmount
7:  function EXIT_IO
8:      for i in my.apps do
9:          decrease i.allocatedPower by my.donationAmount
10:     increase my.allocatedPower by my.freePower
```

III. Algorithms

We present two types of algorithms. *Spread* and *Priority* are *shifting* algorithms, which are concerned with dynamically shifting power between applications. By contrast, *Stagger* and *Control* are *scheduling* algorithms in that they attempt to reduce the overlap of I/O phases across applications, thus increasing the opportunity for power shifting. Each shifting algorithm can either run on its own, or it can be combined with either of the scheduling algorithms. Each algorithm is described in greater detail below. We use the variable *my.freePower* to denote the difference between an application's allocated power and its consumed power in the I/O phase.

A. Spread

Algorithm 1 outlines *Spread*, our baseline power shifting algorithm for improving the performance of as many applications as possible. *Spread* only uses the knowledge that an application is in an I/O phase; it ignores the phase's frequency and duration. At the start of an I/O phase, *Spread* allocates excess power evenly among all applications in computation phases that are not running at full power. The donated power is then reclaimed at the end of the I/O phase.

B. Priority

Priority is similar to *Spread* except that it assumes applications are ordered by their relative importance by some mechanism, and it attempts to finish high priority applications as quickly as possible. At the start of an I/O phase, an application donates its unused power to the highest priority application that is not yet running at full power. A single application may donate power to multiple applications if the highest priority application is running at or near full power.

C. Stagger

Spread and *Priority* often perform well (see Section VI). However, if too many applications enter their I/O phases concurrently, the unused power may exceed the power needs of the remaining applications, so some power may be wasted, as the leftmost pane of Figure 3 shows. However, since *PowerShifter* knows the I/O frequency and duration of each application, we can schedule I/O phases intelligently to reduce the number of applications in I/O phases at any one time.

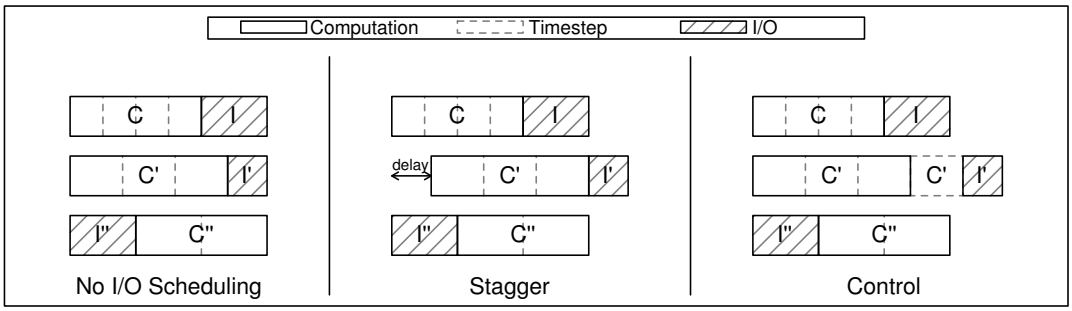

Fig. 3. How *Stagger* and *Control* schedule I/O phases

Algorithm 2 *Stagger* algorithm

```
1: function APPLICATION_ENTER(AppType a)
2:     my.g = findExistingGroup(a)
3:     if my.g != NULL then
4:         delay a
5:     else
6:         my.g = newGroup()
7:     my.g.add(a)
8: function APPLICATION_EXIT(AppType a)
9:     my.g.remove(a)
10:    if my.g.size < Threshold then      ▷ my.g is too small
11:        my.g.delete()
12:        for i in my.g do ▷ Add remaining apps to groups
13:            Application_Enter(i)
```

Algorithm 3 *Control* algorithm. *ShiftingAlg* refers to the shifting algorithm in use.

```
1: function ENTER_IO
2:     for i in allApps except me do
3:         if i.phase == Computation then
4:             powerNeed += i.powerNeed-i.allocPower
5:     cond1 = my.freePower <= powerNeed
6:     cond2 = (curTime - my.lastIOSched) > (k × d)
7:     if cond1 or cond2 then
8:         my.phase = IO
9:         ShiftingAlg.Enter_IO()
10: function EXIT_IO
11:    my.phase = Computation
12:    my.lastIOSched += f
13:    ShiftingAlg.Exit_IO()
```

Stagger, which Algorithm 2 describes, groups applications and then staggers the I/O phases within each group. When application a starts, *Stagger* adds a to an existing group, or if no suitable group is found, it creates a new group. It then delays a such that a's first I/O phase will not overlap with the next I/O phase of any other application in a's group. This mechanism prevents overlap with the next I/O phase, although future I/O phases might overlap. The calculation of I/O phase timing may be imperfect because it does *not* consider the effect of power limits or power shifting. The middle pane of Figure 3 shows how *Stagger* reduces I/O phase overlap.

While delaying applications could increase run time, these delays are amortized over an application's (usually many) timesteps. Applications are only delayed when they first join a group, which happens either when the application arrives or when the application's current group is disbanded because it became too small. Thus, the improvement from better power utilization usually outweighs the cost of the delays. Also, *Stagger* limits the delay that it assigns to an application to a small percentage of its expected run time.

Stagger is our only algorithm that explicitly handles application failures, since it may need to redo group assignments. It treats a failure as a job completion and a restart as a job arrival by calling *Application_Exit* or *Application_Enter*, respectively.

D. Control

Control (see Algorithm 3) assumes that the runtime system can control I/O phase timing, which could, for example, occur in a checkpointing library. Thus, *Control* can minimize overlap of I/O phases across applications without incurring delays. While many current HPC applications do not conform to our assumption, we include *Control* because its approach may become more important in the future. Exascale systems may need to defer I/O phases to avoid excessive overhead, and I/O phase libraries will need awareness of current system conditions such as potential imminent failures to determine when I/O phases are necessary. Thus, system control of I/O phases, especially checkpoints, may become common.

The right pane of Figure 3 shows the basic operation of *Control*, which allows an application to enter an I/O phase only if it will not waste power. Otherwise, the application continues computing and attempts to enter an I/O phase later, such as at the end of its next timestep.

Control allows an application's I/O phase to proceed, regardless of power needs of other applications, if it has been delayed by more than $k \times d$, where d is the interval between I/O phases and k is a constant (0.5 in our experiments). In addition, when *Control* delays an I/O phase, it also reduces the length of the following computation phase so that the next I/O phase is not affected. These mechanisms prevent the situation in which an I/O phase is delayed sufficiently long to impact reliability or post-mortem visualization negatively.

IV. Simulator

We developed *PowerShifter*, a simulator that explores the effects of shifting power across applications. Simulation allows us to execute experiments quickly and to investigate situations that current hardware or applications do not support. *PowerShifter* simulates applications executing on a power-constrained HPC system. Its input includes a set of applications, their computational and I/O phase parameters, and which shifting and scheduling algorithms to use. It outputs the run time of each application, including the cost of any delays incurred by the power shifting algorithm.

PowerShifter divides time into discrete steps. It starts applications at the beginning of each timestep if nodes are available. It then enforces delays that arise from *Stagger*, calculates power needs of each application based on its current phase, and re-allocates any unused power based on the power shifting algorithm. It then advances one timestep, which involves calculating the amount of work each application performs, which depends on the application's current power allocation and its maximum possible power usage in its current phase.

At the end of a timestep, *PowerShifter* updates application phases (e.g., ending an I/O phase) and injects exponentially distributed failures. When a failure occurs, the application's progress is reset to its last checkpoint, a restart penalty is assessed, and then the application continues. This cycle continues until all applications complete.

Since *PowerShifter* discretizes time, all significant events (such as phase changes) occur at the boundaries of timesteps. In some cases, such as the start and end of I/O phases, we reduce this error, but some error inevitably arises from the size of the timesteps. However, as we show in Section V-B, our simulations closely match a real implementation.

V. Implementation and Validation

We implemented our techniques in a prototype that dynamically reallocates power between executing applications. We use it to validate *PowerShifter* based on real system results.

A. Implementation

Our prototype consists of three parts: a wrapper, a runtime, and a controller. The wrapper uses the MPI profiling interface to report start and end times of both computation and I/O. This component is linked into all applications.

The runtime runs in the background on each node, communicates with the wrapper and controller, and implements node-level power caps using RAPL (Running Average Power Limit) [7]. It can pause and resume the application using the signals *SIGSTOP* and *SIGCONT*, and it accesses MSRs (Model Specific Registers) via *libmsr* [8] and *librapl* to set power limits and to record power usage and CPU activity. While we use RAPL, our prototype could be adapted to any cluster that supports power capping.

Our prototype's final component is the controller. One copy of the controller runs during each experiment, and it communicates with the runtime on each node. It makes power allocation and application pause/continue decisions based on node-level information on application state and power needs, and then implements these decisions via the runtimes on each node. The controller is configurable to simplify development and use of new power shifting algorithms. It is also application aware—it groups nodes by applications—so power shifting algorithms can make decisions at the node or application level.

Our prototype runs on Cab, which is a cluster of 1,296 Intel Xeons that uses SLURM [9] for resource management. It uses a job submission script that requests a node allocation large enough for all application jobs with one extra node for the controller. The script uses SLURM to start the controller on one node and the runtime on all other nodes. It then uses SLURM to start jobs on disjoint subsets of nodes. The script exits when all jobs, including the controller, have completed.

The controller and runtimes start first, and each runtime registers with the controller. Eventually, an application starts and calls `MPI_Init` on each of its nodes. The wrapper intercepts this call, calls `PMPI_Init`, and sends a signal to the runtime to indicate that the application has started. Control is then returned to the application. The runtime sends a signal to the controller to indicate that the application has started on that node. It also includes information about the application, including the application's power needs. The controller responds with an initial power limit, which the runtime implements using MSRs.

The wrapper monitors the application for transitions between computation and I/O phases. Our prototype requires a user-inserted `MPI_Pcontrol` call to mark each transition between phases; a production implementation could observe the transition less intrusively. The wrapper forwards this information to the runtime, which sends it to the controller. Depending on the power shifting algorithm, the controller can lower the power limit on that node and raise the power limits on other nodes. The controller can also pause or resume the application at any time, as required by *Stagger*.

Communication between the controller and runtimes has negligible overhead because messaging occurs in the background and thus does not directly affect application execution time. RAPL latency is another source of overhead. Our experiments exhibit approximately 50 ms lag between when the runtime sets a power cap and when the processor begins to execute at it, which is a 0.2% overhead in the worst case.

We use our prototype to validate *PowerShifter*. However, we note that it improved application run times up to 26%.

B. Simulator Validation

To validate *PowerShifter*, we compare it to our prototype using LAMMPS, ParaDiS, and Cactus. LAMMPS [10] is a molecular dynamics simulation from the ASC Sequoia benchmark suite. ParaDiS [11] is a production dislocation dynamics simulation that operates on dynamically changing, unbalanced data set sizes across MPI processes. We used the "Copper" input. Cactus [12] is a framework that numerically solves Einstein's equations [13] via adaptive mesh refinement. These applications are used for production science and use periodic I/O phases.

Experiment	Set of Applications	Percentage Error
1	LAMMPS	0.2%
	ParaDiS	1.7%
	Cactus	0.4%
2	LAMMPS	0.1%
	Cactus	0.4%
3	ParaDiS	0.5%
	ParaDiS	0.9%
	ParaDiS	0.7%
	ParaDiS	1.2%
4	Cactus	0.1%
	Cactus	0.3%
	Cactus	0.8%
	Cactus	0.2%

TABLE I
VALIDATION RUNS

We first executed each application at various power levels to obtain simple models of their response to power limit changes. *PowerShifter* uses these models to predict their run times under our power shifting algorithms. We then ran various tests using our prototype, and then re-ran the same tests in *PowerShifter*. Table I presents the accuracy of *PowerShifter*'s run time predictions relative to the median run time of at least five actual runs.

All predictions have less than 2% error and most cases have less than 1% error. Analysis indicates that *PowerShifter*'s prediction of the increase in performance due to an increase in allocated power is the most significant source of error. The worst case (not shown in Table I) occurred in a test run with LAMMPS, in which the error when increasing power from 60W to 80W was 3.3%.

VI. RESULTS

We present the results from our *PowerShifter* simulator for both large and small numbers of applications.

A. Setup

We execute two broad groups of experiments. The first group (which covers the experiments in Section VI-B) is based on real data. Specifically, the node count and job arrival time are taken from the RICC job trace from the Parallel Workloads Archive [14]. The percentage of time a job spends in I/O is either based on Darshan log files [15] obtained from actual runs on Intrepid (an IBM Blue Gene/P ALCF machine), or it is provided as an input parameter. The power limit for all tests was 60W per node, unless otherwise stated. Other parameters, which are not available in traces, are determined randomly; how we choose these is detailed later in this subsection.

In the second group (which covers the experiments in Section VI-C), all parameters are fixed and identical across all applications. This group is intended to study specific scenarios found on capability machines.

Unless otherwise stated, the random parameters for each simulation are chosen as follows. We estimate the number of application timesteps using the run time from the RICC trace. We choose the timestep time from a uniform distribution between 0.1 and 2 seconds (applications generally perform several timesteps between I/O phases). We choose I/O phase time from a uniform distribution between 30 seconds and 5 minutes. We choose the power consumption during computation from a uniform distribution between 80W and 100W per node, and we choose I/O phase power from a uniform distribution between 35W and 45W per node.

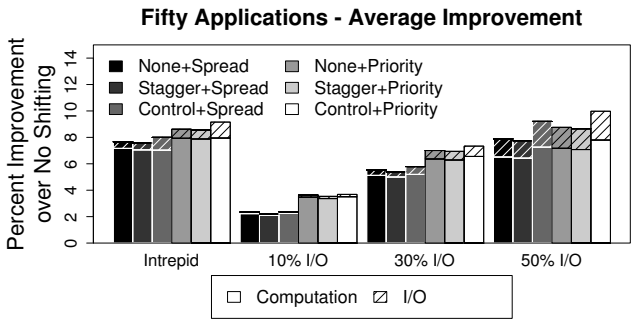

Fig. 4. Average percentage improvement with 50 applications

We randomly select roughly 80% of applications to enter I/O phases after a fixed number of program-level timesteps; the remaining applications enter I/O phases after a fixed amount of wall-clock time. These mechanisms mirror real applications, which can specify the interval between I/O phases in either timesteps or time. We calculate the interval between I/O phases based on the timestep time, I/O phase time, and percentage of time spent in I/O phases, which was a simulation input.

To emulate the response of applications to changes in the power limit, we ran ParaDiS at several different power levels. We use this data to produce a third degree polynomial that estimates performance under a given power limit. We then assign a random factor to each application to scale this polynomial, and use the scaled polynomial to calculate that application's response to changes in the power limit. We choose the factor such that the maximum improvement due to increasing the power limit for a given application ranges from 10% (representing a memory-bound program) to 250% (representing a CPU-bound program). We use 250% based on the range of frequencies on *Cab* when setting power bounds on ParaDiS from 40W to 90W, which was 1.33GHz to 3.0GHz.

In addition to using the Intrepid log files mentioned previously, we run tests in which we vary the percentage of time that applications spend in I/O from 10% to 50%. While 50% is unusually large, we use it as our upper bound to study the effect of increasing I/O phase time on our algorithms.

For *Priority*, we assign priorities in reverse order of CPU-intensiveness; applications that are highly CPU-intensive get the highest priority, which preferentially allocates power to applications that can benefit the most from it. Doing this on a real system would require knowledge of the CPU-intensiveness of all running applications, which is an orthogonal problem for which much related work exists [16], [17].

B. Large Numbers of Applications

Our first set of experiments models a capacity system with many concurrent applications. These experiments simulate 50 applications running on a 512 node cluster.

1) Average Improvement: This section includes two sets of experiments. In the first set (labeled "Intrepid" in Figure 4), we

take the percent of time that applications spend in I/O from a Darshan log [15] recorded on Intrepid, which represents about 1/3 of its workload. The log does not contain any information about text-based I/O, so I/O time is underrepresented for some applications. Thus, we remove any application for which I/O is less than 2% of its run time. We also remove any application for which I/O is more than 90% of its run time, as we assume it was an I/O benchmark or similar program. In the second set of experiments, which Figure 4 also shows, we fix the percentage of time spent in I/O for all applications, which allows us to investigate the effect of the amount of time that applications spend in I/O on the effectiveness of power shifting. In both sets of experiments, we randomly inject failures with an MTTF of 41 minutes across the entire cluster.

We ran each experiment several times with different random parameters. Figure 4 shows the average percentage improvement obtained across all runs. The legend indicates the algorithms used in the format *SchedulingAlgorithm+ShiftingAlgorithm*. For simplicity, throughout this section, if we refer to just a single algorithm, any comments apply to *all* combinations of scheduling and shifting algorithms that include that algorithm. For example, if we make comments about "*Spread*", they apply to *None+Spread*, *Stagger+Spread*, and *Control+Spread*.

Performance can improve because an application executes computation faster due to power shifting or because it executes fewer I/O phases. The latter can occur if an application measures the interval between I/O phases in time rather than timesteps, so it may execute fewer I/O phases when it runs faster. Also, the *Control* algorithm can push the final I/O phase beyond when the application completes. To show these two sources of improvement, we divide the improvement into the portion due to faster computation (solid fill) and that due to dropping I/O phases (hatched).

The results based on real I/O data show an improvement of about 8%. The other experiments demonstrate that as applications spend more time in I/O phases, the benefit of power shifting increases. Experiments that use *Priority* as the shifting algorithm perform better than equivalent experiments that use *Spread*, because more power is allocated to applications that will benefit the most from it. However, *Priority* shifts power unfairly, so assigning priorities based on CPU-intensiveness may not be suitable for a production system. Surprisingly, runs with no scheduling algorithm perform better than equivalent runs with *Stagger+Spread* or *Stagger+Priority*. This effect occurs because *Stagger* incurs delays to schedule I/O phases even when the delays provide little benefit.

In all experiments, relatively little improvement comes from dropped I/O phases, which is consistent with the majority of applications taking their I/O phases after a fixed number of timesteps. *Control* occasionally pushes an I/O phase past the end of the run, which results in a larger benefit from dropped I/O phases for *Control+Spread* and *Control+Priority* than for the other algorithms. Besides this additional performance benefit, *Control* provides little benefit over runs with no scheduling algorithm unless applications spend a large percentage of time in I/O phases. With many running applications, the probability that power is wasted due to a large percentage of the applications executing I/O phases at once is small. As the time spent in I/O phases increases, this probability becomes larger, and so does the benefit of *Control+Spread* and *Control+Priority*.

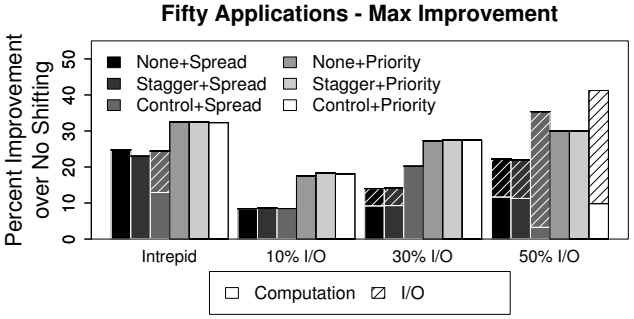

Fig. 5. Maximum percentage improvement with 50 applications

2) Maximum Improvement: Figure 5 shows the maximum improvement over 50 applications with each algorithm. In these experiments, we do not inject failures since they can occur at different times relative to checkpoints in different runs, which can result in a large performance improvement that is unrelated to power shifting. We again run each test several times with different random parameters. We calculate the maximum improvement achieved by any application in each run; the data that Figure 5 shows is the median of these maximums. We again indicate the improvements due to faster computation (solid fill) and reduced I/O phases (hatched).

As expected, runs with *Priority* achieve larger improvements than equivalent runs with *Spread*, and some of its improvements are in excess of 30%. In many cases, it achieves these improvements without reducing the time spent in I/O phases, as denoted by the lack of hatching on many of the bars. The two *Control* runs at 50% I/O show the largest improvements, but they were both for a short running application that took two I/O phases in the baseline run, and *Control* shifted one of them past the end of the run. Thus, the percent improvement is large, but the absolute improvement is relatively small.

3) Improvement Under Different Power Limits: The relationship between an application's computation power, I/O phase power, and system-imposed power limit can impact the effectiveness of power shifting. To explore this dimension, we execute several tests that vary the global power limit such that the power allocated to each node ranges from the minimum to the maximum amounts. In these experiments, we use 40W and 80W for the I/O and computation power per node for all applications. As in Section VI-B2, we do not inject failures to ensure that all improvement is due to power shifting.

Figure 6 shows results for I/O phases that take 10%, 30%, and 50% of total execution time. The x-axis is the global power limit divided by the number of nodes, i.e., the power limit per node *in the absence of power shifting*. For clarity, we only show *None+Spread* and *Control+Spread*. The title states the percentage of time in I/O phases at 60W.

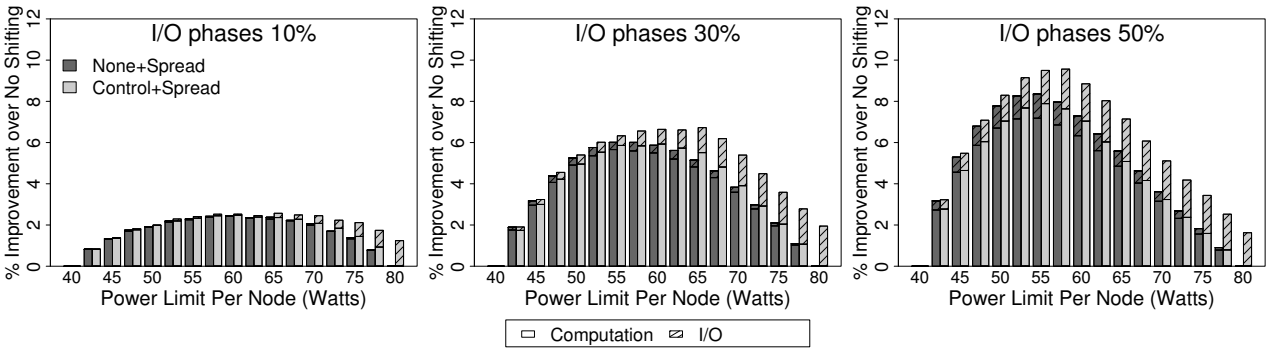

Fig. 6. Average percentage improvement under different power limits with I/O phases consuming 10% (left), 30% (middle), and 50% (right) of the time

At the lowest power limit (40W per node), power shifting cannot provide any benefit because applications never have unused power. Similarly, when the power per node is 80W, all applications always have enough power, so power shifting is unnecessary. *Control+Spread* can provide some benefit at 40W or 80W if it shifts I/O phases past the end of the run, but power shifting provides no benefit.

The leftmost graph of Figure 6 has an inflection point at 60W per node. Below this power limit, half of the nodes can be in I/O phases at the same time without leaving power unused. Thus, *None+Spread* and *Control+Spread* have similar performance. When the power limit is higher, less than half of the nodes can be in an I/O phase at a time without some unused power, so the benefit of *Control+Spread* increases. However, *Control+Spread* has to delay more I/O phases to ensure that power is not lost, so a large percentage of its improvement is due to dropped I/O phases. The middle and rightmost graphs show similar patterns, except that performance improvement peaks at lower power limits.

C. Small Numbers of Applications

Our second set of experiments covers scenarios with small numbers of concurrent applications, similar to those found on capability machines. Reducing the number of applications increases the probability that a scheduling algorithm will be required to exploit I/O power shifting opportunities fully. Thus, this section explores scenarios in which scheduling algorithms make a significant difference.

In the next two subsections, all parameters are fixed and identical across all applications. We set computation power to 80W per node, I/O power to 40W per node, and the power limit to 60W per node. These experiments are explicitly designed to explore the limits of *Stagger* and *Control* when I/O phases are aligned in specific patterns; thus, we do not inject failures because they would disturb the alignment.

1) Perfect Alignment of I/O Phases: In these tests, several applications with identical attributes (I/O phase time, computation phase time, node count, and response to power limit changes) start at the same time. Thus, all of their I/O phases occur at the same time, which means that no power can be shifted unless I/O phases are adjusted. This represents a near-optimal situation for *Stagger* or *Control*.

Figure 7 shows the improvement over the baseline (no power shifting) for different numbers of applications. We omit runs using *None+Spread* and *None+Priority* because they achieve no performance improvement; in this scenario, the shifting algorithms provide no benefit unless they are paired with a scheduling algorithm. *Control* generally performs slightly better than *Stagger*; both shift I/O phases to ensure that they do not overlap, but *Control* does so without incurring the cost of delays. For both algorithms, the magnitude of the improvement increases as the percentage of time spent in I/O phases increases: more time in I/O phases implies more opportunity to shift power.

The benefit of *Control* over *Stagger* is more pronounced with odd numbers of applications than with even numbers of applications because of the non-linear relationship between power and performance. With an even number of applications, *Control* allows applications to enter I/O phases in 2 groups—first, half of the applications execute their I/O phases, and when they finish, the other half execute theirs. Thus, all applications are at their maximum power level for a short period. With an odd number of applications, one of the applications becomes an "odd application out" and must execute its I/O phase separately from the other applications to avoid unused power. Thus, with an odd number of applications, applications are given less extra power for a longer time, which results in better improvement than giving applications a large amount of power for a short time.

When applications spend 50% of their time in I/O, scheduling I/O phases becomes difficult, as the widely varying performance at different numbers of applications shows. Because applications spend roughly the same amount of time in I/O phases as in computation phases, *Stagger* can align even numbers well. For example, with four applications, the first application is in an I/O phase at the same time as the third, and the second application is in an I/O phase at the same time as the fourth. Thus, half of the applications are in I/O phases at any given time, so little power is unused. With an odd number of applications, more than half of the applications are in I/O phases at the same time, so some power cannot be used.

Control performs relatively poorly at 50% because of the limit on how long it will delay an I/O phase. For even numbers of applications, it allows half of the applications to enter I/O

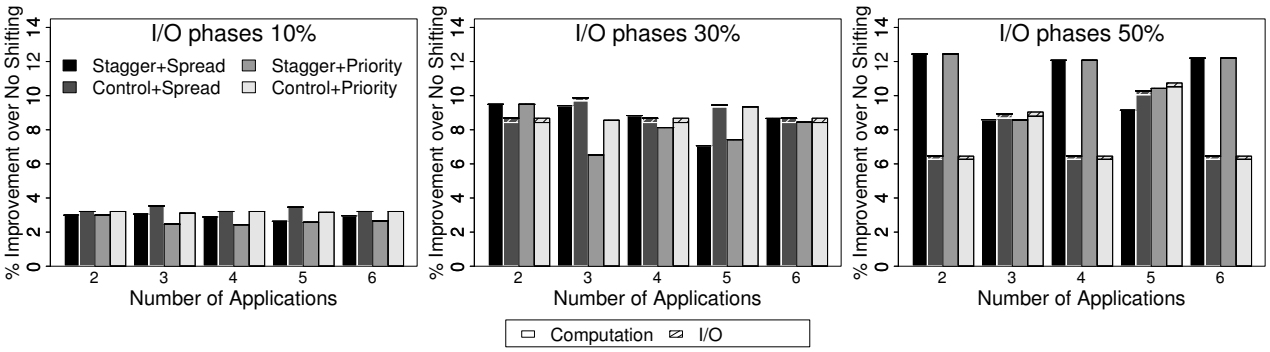

Fig. 7. Average percentage improvement with I/O phases aligned and consuming 10% (left), 30% (middle), and 50% (right) of the time.

phases and delays the I/O phases of the other half. However, halfway through the I/O phase, the second set of applications reaches the I/O phase delay limit, so they start their I/O phases. Thus, all applications are in I/O phases for a significant period, which leads to unused power. For odd numbers of applications, a similar effect happens, but it is mitigated by the different number of applications in the two groups, which causes applications to speed up by different amounts and naturally staggers their I/O phases.

2) Partial Alignment of I/O Phases: Figure 8 shows identical tests to Figure 7 except that application start times are offset by half the length of an I/O phase. Thus (in the absence of power shifting), the first half of the I/O phase of application i overlaps with the second half of the I/O phase of application $i-1$. In this more realistic situation, *Spread* and *Priority* have some opportunity to shift power, even without a scheduling algorithm. *Stagger* and *Control* exhibit roughly similar behavior to that in Figure 7 except that *Control* can achieve larger improvements when applications spend a large percentage of time in I/O phases. With partially aligned I/O phases, the amount of delay required to stagger I/O phases is significantly less, so *Control* is still effective even when applications are in I/O phases 50% of the time.

None+Spread and *None+Priority* can achieve little improvement with two applications because when both are in I/O phases, no other applications are available to use the extra power. As the number of applications increases, this situation is mitigated; when there are six applications, *None+Spread* and *None+Priority* achieve large speedups. As the number of applications and the time that they spend in I/O increases, we also see that *None+Spread* begins to perform better than *Stagger+Spread*. As noted previously, *Stagger* will always impose delays, even when they provide no benefit.

In the rightmost graphs of Figures 7 and 8, several algorithms improve performance more than 12%. In these cases, applications spend half of their time in I/O phases, and half of the applications are in I/O phases at any given time, which is the best case for power shifting. Applications spend half of their time in computation and benefit from power shifted from other applications during the entire computation phase. They then spend the other half of their time in I/O phases, providing power that can be shifted to other applications.

D. I/O Contention

Our results are encouraging. One key assumption that we make is that I/O time is constant: that is, writing a given amount of data to the file system always takes the same amount of time. This assumption may not hold in practice. In particular, I/O operations may incur slowdown when the number of nodes concurrently performing I/O increases. We now consider potential implications of this fact on our results.

For *Spread*, *Priority*, and *Stagger*, the number of nodes in I/O phases at any given time is essentially random: jobs start at random times, enter I/O phases at random intervals, and remain in I/O phases for random lengths of time. Thus, effects of increased I/O overlap are generally counteracted by periods of decreased I/O overlap. However, power shifting compresses computation time but not I/O time, so the overall percentage of time that jobs spend in I/O will increase, which increases the probability of I/O phases overlapping. Consider the situation in which jobs spend 10% of their time in I/O phases, which, according to Figure 4, results in a roughly 2% performance improvement. If we assume the increase in I/O overlap is proportional to the performance improvement from power shifting, I/O phases should overlap about 2% more during the run. Since I/O phases take up 10% of execution time, this increased overlap takes up about 0.2% of total run time, which is negligible. Also, I/O system contention may occur without power shifting, so *PowerShifter* is unlikely to *initiate* contention for large systems.

If applications spend significant time in I/O (e.g., 50%), the speedup is larger (8% on average in Figure 4). In this extreme case, I/O phases will overlap roughly 4% more due to power shifting, which may noticeably decrease the speedups achieved by power shifting. However, this extreme of high I/O time for all applications is unlikely. We expect an inflection point after which the benefit of power shifting decreases; any slowdown due to overlapped I/O phases simply moves this inflection point to the left.

Control actively manages when I/O phases occur, so the impact of I/O overlap is different. *Control* is essentially a "peak shaving" algorithm: when a "peak" of many nodes in I/O phases exists, it "shaves off" this peak into the following valley. This results in fewer overlapped I/O phases for jobs that were part of the peak, at the cost of possibly increasing

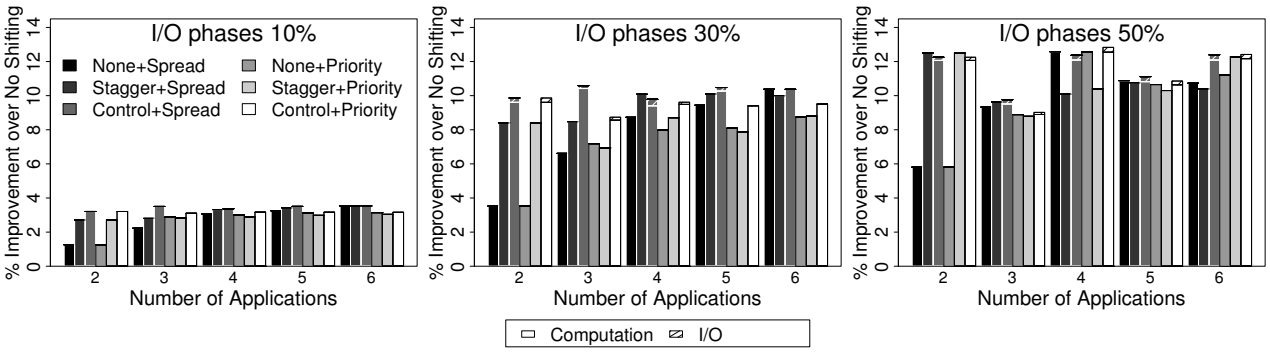

Fig. 8. Average percentage improvement with I/O phases partially aligned and consuming 10% (left), 30% (middle), and 50% (right) of the time.

the I/O overlap for jobs in the valley. Thus, the normal case for *Control* is to reduce I/O overlap for a large number of jobs while potentially increasing I/O overlap for a smaller number of jobs. Therefore, we expect that I/O overlap will have little negative effect—and possibly a positive one—on *Control*.

VII. Related Work

Many systems and techniques move power within a single application. They generally exploit load imbalance to shift power from nodes off of the critical path to those on it. Marathe et al. [2] move power by resetting power caps based on Intel's Running Average Power Limit (RAPL) [18]. Other work saves energy on off-critical path nodes rather than shifting power [19], [20], [21]. *PowerShifter* differs from these systems in that it shifts power *across* applications.

Some recent work explores shifting power between different applications. Ellsworth et al. [22] developed a system called POWsched that collects excess power from nodes executing below their power caps and reallocates it to nodes executing near their power caps. POWsched does not require any semantic information, and it considers nodes individually rather than grouping them by application. Another system by Liu et al. [23] uses hardware counters to classify phases as CPU, I/O, or "undetermined" and then shifts power from inferred I/O phases to inferred CPU phases. These systems are simpler than *PowerShifter* because they do not require semantic information about applications. However, they are vulnerable to issues such as quickly changing power consumption leading to hysteresis, and they cannot schedule applications to increase the opportunity for power shifting. In addition, POWsched may briefly allocate uneven amounts of power to different nodes within an application, thus creating computational imbalance. *PowerShifter* avoids these problems by requiring semantic information about applications.

Several researchers have studied HPC application execution under power bounds. This includes overprovisioning [24], [25] as well as work in scheduling [26], [27], [28], [29]. This work is complimentary to our own; our algorithms could be integrated into these systems to provide additional benefit.

Our work is mostly orthogonal to the checkpointing approach, as long as we can determine the checkpoint interval and duration. We could integrate our techniques into checkpoint libraries such as BLCR [30] or SCR (The Scalable Checkpoint/Restart Library) [31]. The growing scale of systems creates challenges for checkpointing, which have given rise to approaches like burst buffers [32], [33] and asynchronous transfers of checkpoints [34]. These techniques will impact the power draw and cause more frequent [31] and more complicated power shifts from different components (e.g., the CPU, the memory system and the I/O system). These developments will likely require extensions to our I/O-aware power shifting techniques, but the key idea behind our approach—integrating knowledge of the checkpointing process into power shifting decisions—will remain critical.

VIII. Summary

Power use is a major challenge for near-future exascale systems. System software must be redesigned in order to achieve the high performance required for scientific discovery while respecting system power limits. In this paper, we explored an opportunity for significant performance improvement by shifting power from applications that are executing I/O phases to those executing computation phases. We designed and implemented two algorithms for shifting power and two algorithms for scheduling I/O phases and evaluated them in a simulator, which we validated on a real system. We showed that staggering applications to avoid I/O phase overlap and externally controlling I/O phases can provide significant benefit. Results from *PowerShifter* show that our algorithms achieve significant performance improvement over static, per-node power caps.

Our algorithms and results could impact future system software that must optimize performance under a power constraint. For example, job schedulers can implement our algorithms, or variations of them, based on system usage characteristics. Contrary to naive expectations, we found that straightforward algorithms performed well with a large number of concurrent jobs as found on capacity systems. This finding will benefit schedulers for capacity systems that may already be overloaded with the overhead of managing a large number of jobs, since the simpler algorithms are less computationally expensive. On the other hand, capability systems may justify the use of more complex and expensive algorithms.

ACKNOWLEDGMENT

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-669729. In addition, this material is based upon work supported by the National Science Foundation under Grant No. 1216829 and No. 1526015.

REFERENCES

[1] K. Bergman *et al.*, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf, Sep. 2008.

[2] A. Marathe, P. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A Run-Time System for Power-Constrained HPC Applications," in *Supercomputing*, Nov. 2015.

[3] M. Diouri, O. Gluck, L. Lefèvre, and F. Cappello, "Energy Considerations in Checkpointing and Fault Tolerance Protocols," in *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2012, pp. 1–6.

[4] E. Meneses, O. Sarood, and L. V. Kale, "Assessing Energy Efficiency of Fault Tolerance Protocols for HPC Systems," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 2012, pp. 35–42.

[5] K. W. Cameron, R. Ge, and X. Feng, " High-Performance, Power-Aware Distributed Computing for Scientific Applications," *IEEE Computer*, vol. 38, no. 11, 2005.

[6] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, "Theory and Practice in Parallel Job Scheduling," *Job Scheduling Strategies for Parallel Processing*, pp. 1–34, 1997.

[7] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B. System Programming Guide, Part 2." http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf.

[8] K. Shoga and B. Rountree, "libmsr," https://github.com/scalability-llnl/libmsr.

[9] A. Yoo, M. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2003, vol. 2862, pp. 44–60.

[10] "ASC sequoia benchmark codes," http://asc.llnl.gov/sequoia/benchmarks/, 2009.

[11] V. Bulatov, W. Cai, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis, "Scalable Line Dynamics in ParaDiS," in *Supercomputing*, Nov. 2004.

[12] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H. C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf, "Cactus Tools for Grid Applications," *Cluster Computing*, vol. 4, no. 3, pp. 179–188, 2001.

[13] E. Seidel and W. Suen, "Numerical Relativity as a Tool for Computational Astrophysics," *Journal of Computational and Applied Mathematics*, vol. 109, no. 1-2, pp. 493–525, 1999.

[14] M. Kurokawa, "Parallel Workload Archives," http://www.cs.huji.ac.il/labs/parallel/workload/l_ricc.

[15] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749269

[16] C.-H. Hsu and W.-C. Feng, "Effective Dynamic Voltage Scaling Through CPU-Boundedness Detection," in *Proceedings of the 4th International Conference on Power-Aware Computer Systems*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 135–149. [Online]. Available: http://dx.doi.org/10.1007/11574859_10

[17] V. W. Freeh, F. Pan, N. Kappiah, D. Lowenthal, and R. Springer, "Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005.

[18] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory Power Estimation and Capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10, 2010, pp. 189–194.

[19] R. Ge, X. Feng, W. Feng, and K. W. Cameron, " CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters ," in *International Conference on Parallel Processing*, 2007.

[20] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1175–1185, 2008.

[21] B. Rountree, D. Lowenthal, B. de Supinski, M. Schulz, V. Freeh, and T. Bletch, "Adagio: Making DVS Practical for Complex HPC Applications," in *International Conf. on Supercomputing*, Jun. 2009.

[22] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz, "Dynamic power sharing for higher job throughput," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15*. ACM, 2015, pp. 80:1–80:11. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807643

[23] Z. Liu, J. Lofstead, T. Wang, and W. Yu, "A Case of System-Wide Power Management for Scientific Applications," in *International Conference on Cluster Computing*, Sept 2013.

[24] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. de Supinski, "Exploring Hardware Overprovisioning in Power-Constrained, High Performance Computing," in *International Conference on Supercomputing*, Jun. 2013.

[25] O. Sarood, A. Langer, L. V. Kale, B. Rountree, and B. R. de Supinski, "Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems," in *IEEE International Conference on Cluster Computing*, Sept 2013, pp. 1–8.

[26] T. Patki, A. Sasidharan, M. Melarth, D. K. Lowenthal, B. Rountree, M. Schulz, and B. de Supinski, "Practical Resource Management in Power-Constrained, High Performance Computing," in *High-Performance Distributed Computing*, Jun. 2015.

[27] O. Sarood, A. Langer, A. Gupta, and L. V. Kale, "Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget," in *Supercomputing*, Nov. 2014.

[28] M. Etinski, J. Corbalan, J. Labarta, and M. Valero, "Optimizing Job Performance Under a Given Power Constraint in HPC Centers," in *Green Computing Conference*, 2010, pp. 257–267.

[29] ——, "Linear Programming Based Parallel Job Scheduling for Power Constrained Systems," in *International Conference on High Performance Computing and Simulation*, 2011, pp. 72–80.

[30] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/i=1/a=067

[31] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinksi, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10, LLNL-CONF-427742*, November 2010.

[32] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the Role of Burst Buffers in Leadership-Class Storage Systems," in *Symposium on Mass Storage Systems and Technologies, MSST 2012*, April 2012.

[33] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, "A User-level Infiniband-based File System and Checkpoint Strategy for Burst Buffers," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*, May 2014.

[34] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinksi, N. Maruyama, and S. Matsuoka, "Design and Modeling of a Non-blocking Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'12, LLNL-CONF-554431*, November 2012.