I/O Aware Power Shifting

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

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

ABSTRACT

Power on future clusters will be strictly limited. This will in turn force some flavor of power limits on individual applications. However, high-performance computing applications often do not consume a fixed amount of power over their lifetime. When applications execute low-power I/O phases, we can improve system-wide performance (that is, of all applications) if we shift power to applications executing high-power, computation-bound phases.

This paper studies algorithms that leverage application semantics—specifically the knowledge of I/O phase frequency and duration as well as power needs—to shift power from applications in I/O phases to those in compute phases. We investigate several algorithms, including ones that explicitly stagger applications to improve power shifting. We design, implement, and validate our novel techniques in a simulator we call *PowerShifter*. Compared to executing without power shifting, our algorithms achieve an average improvement in application performance of up to 8% with a maximum improvement of 27%.

1. INTRODUCTION

Power is a significant concern for upcoming exascale systems due to the 20 megawatts power budget set by the U.S. Department of Energy (DOE). In fact, a recent DOE report stated that power was the most difficult challenge for exascale [4]. Today, the fastest supercomputers in the world have theoretical peaks only in the tens of petaflops while using in excess of 10 megawatts. Based on this trend, experts predict that without significant changes in software, we will not meet this power budget and instead exceed it by at least an order of magnitude [3, 4, 26]. Because of these power constraints on future clusters, applications will soon execute with a strict power limit, and will rely on advances in system software to help them meet the limits while still maintaining performance.

While there has been systems software research in budgeting power while maximizing performance of single applications in isolation [27], there is little research when it comes to *set of applications*. As we show in this paper, there is a significant opportunity to improve overall system performance by shifting power across concurrently running applications. When applications execute in I/O phases, they generally use less power than they use when they execute in computation phases [10, 28]. Thus, the power demand of applications varies over time, and overall performance can be improved if power is shifted dynamically between applications in an intelligent manner.

This paper focuses on developing and analyzing algorithms that leverage application semantics to shift power from applications in I/O phases to those in compute phases. We assume knowledge of key characteristics of both computation and I/O phases, including their start times, execution frequencies, durations, and power needs. We implemented our algorithms in a simulator called *PowerShifter* and used it to study a range of scenarios, including job size distribution, number of jobs, application power consumption, cluster power limit, and I/O phase frequency and duration.

Within *PowerShifter* we investigate several power shifting strategies. Our base strategy, which often performs quite well, shifts power on demand, i.e., whenever an I/O phase occurs. We also design and implement more sophisticated algorithms that potentially delay some applications so as to achieve a schedule in which the minimum number of applications incur I/O concurrently. This helps to avoid situations in which an application in an I/O phase cannot identify a target "partner" application for power shifting. In addition, *PowerShifter* handles several challenges, including job failures, job arrivals, and job departures. *PowerShifter* changes the schedule when necessary, on the fly, in response to these unpredictable events.

In this paper, we make several contributions.

- We develop the first algorithms that we know of that leverage application semantics to shift power between different applications during I/O phases.
- We show that a simple, on-demand algorithm that shifts power is quite effective in improving average performance.
- We show that complex scheduling algorithms for power shifting are effective on capability systems with few, large jobs.
- We implement our algorithms in *PowerShifter*, which allows us to explore parameters including differing application power needs and job count and size distributions.
- We validate *PowerShifter* with an implementation using three real applications: LAMMPS, ParaDiS, and Cactus.

Results with *PowerShifter* show that using our algorithms, the average improvement (over all applications) is up to 8% with a maximum improvement of 27%, as compared to executing without shifting power between applications. Moreover, scheduling I/O phases explicitly can account for an 12% improvement over straightforward algorithms.

The rest of this paper is organized as follows. Section 2 provides background, and Section 3 provides our assumptions used

in this paper. Section 4 details the algorithms we used, Section 5 describes our *PowerShifter* simulator, and Section 6 discusses our implementation on a real cluster and its use in validating *PowerShifter*. Section 7 presents our experimental results. Finally, Section 8 discusses related work and Section 9 summarizes the paper.

2. BACKGROUND

Power-constrained supercomputing has received both significant and recent attention. Primarily, this is because exascale systems will likely have their power use bounded [4]. The reaction to power-constrained systems has largely been to focus on individual applications and develop techniques to execute those applications faster given a fixed amount of power. The straightforward idea is to give each application an amount of power directly based on the number of nodes it uses and for the run-time system (operating on behalf of that application) to shift that power only *within* that application [27].

In this paper, we specifically focus on optimizing the performance of many (or all) applications by allowing power to be shifted from applications in I/O phases to those in computation phases. We specifically study applications whose power needs vary because they alternate between computation phases (which are CPU intensive) and I/O phases, (which are not CPU intensive). We note that the CPU intensity of a computation phase depends on the mix of CPU operations, cache hits, and main memory accesses; different applications (and application phases) react differently to additional power. This paper considers shifting CPU power only, and we also ignore any extra power that is available due to idle nodes.

I/O phases generally occur because of either checkpointing or visualization output. With checkpointing, applications save their state to the I/O system periodically, and if a failure occurs, the application is re-started from the most recently saved checkpoint. Visualization output similarly represents periodic system state (without, of course, any analogy to a restart operation). The key is that both types of I/O generally occur periodically in HPC applications. In this paper we group both kinds of I/O into the same category and refer to them simply as *I/O phases*. From a power perspective, I/O phases generally use little power, because techniques such as DMA or asynchronous I/O and network operations cause the CPU to idle and wait.

Overall, we focus on shifting power from applications executing I/O phases to applications in computation phases. In this way, we aim to speed up applications when they are computing, without negatively affecting applications in I/O phases. In the next section, we discuss the application and system model that we assume in the paper.

3. ENVIRONMENT AND ASSUMPTIONS

We assume a high-performance computing (HPC) system that consists of a total of N nodes, with some number of applications executing, each with a potentially different node count. The sum of the nodes in use by all applications is less than or equal to N. The HPC system is prescribed a global power bound *P*. In the absence of an intelligent multi-node or multi-application strategy, we assume that this bound imposes a uniform power bound per node for each application, so each node would receive P/N power. As stated previously, we allow a node to have more or less than P/N, depending on application characteristics.

3.1 Phase Characterization

In this paper, we assume that all HPC application I/O is exposed. This allows our *PowerShifter* system to know precisely when applications are in I/O phases and computation phases. This can be done by having the application expose each computation and I/O phase in advance of execution. This information is often already present through explicit configuration by the user or the use of checkpoint or "visualization-dump" libraries, which maintain this information for their own purposes. We note that with our simpler algorithms (see Section 4), less information is actually needed (just when applications enter and exit the *current* I/O phase).

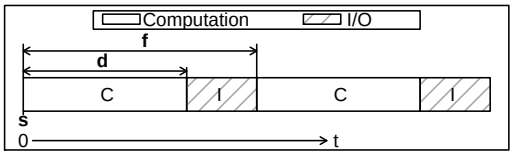

Figure 1: Basic model.

We assume that each application consists of two types of phases: computation and I/O. We represent a computation phase with a descriptor $C_{s,f,d,p}$ and an I/O phase with a descriptor $I_{s',f',d',p'}$. The meaning of $C_{s,f,d,p}$ is that the first instance of the computation phase starts at time s, and a computation phase occurs every f time units, and has duration d time units (see Figure 1). The power consumed by the computation phase is denoted by p. The variables in $I_{s',f',d',p'}$ have an identical meaning, but for I/O phases. There can in general be multiple computation or I/O phase descriptors; for example, there could be I/O due to both periodic checkpointing and visualization output. In such a case, we would need to add an identifier per phase. For simplicity, in this paper we assume that there is only one computation phase descriptor and one I/O phase descriptor per application. We also use the variables s, f, and d interchangeably in this paper for both computation and I/O phases.

As can be seen from the denotation of computation and I/O phases, we assume that all phases are periodic. Moreover, for most applications, the value of f for I/O phases will be chosen so that it is a multiple of the length of a program-level timestep (iteration[1]). This is the case for most programs that execute at supercomputing facilities; the I/O phase is most often checkpointing, which by nature executes in a fairly regular, periodic manner. Our experience with applications that save state for visualization (e.g., ParaDiS, a dislocation simulator [5]) is that they also execute these phases in a periodic manner.

We also assume that the power usage information for each application is available—specifically the relative performance improvement by an application when increasing its power allocation. Generally, this information can be obtained by profiling, but it could also be provided by the programmer or inferred by program analysis. Several researchers have developed power profilers such as PowerPack [6].

3.2 Scheduling

In this work we allow the set of applications executing on the machine to change over time. That is, our model allows for random application arrivals and departures. At HPC installations, users typically submit jobs by specifying a node count and an estimated run time. Resource requests are maintained in a job queue, and when the resource manager acquires the set of dedicated nodes that matches a request, users are allocated these nodes. The scheduling policy is space sharing [14], in which compute node resources are exclusively allocated to a single job until its completion. From the point of view of this paper, jobs arrive randomly and complete at

[1]We use timestep and iteration interchangeably in this paper.

Algorithm 1 *Spread* algorithm. Variable *my.freePower* represents the difference for an application between its allocated power and its consumed power in an I/O phase.

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

some point. We assume that each job has the descriptors we described above. In this paper, we assume that the scheduler uses a straightforward first-come, first-served (FCFS) approach. In reality, supercomputers use backfilling-based algorithms that allow jobs to be executed out of order, leading to better resource utilization; however, our power-shifting techniques are effective independent of the particular scheduling strategy.

4. ALGORITHMS

We have designed four algorithms with increasing levels of sophistication for dynamically sharing power between applications. We call these algorithms *Spread*, *Priority*, *Stagger*, and *Control*. Each is described below. In general, the algorithms all shift power; additionally, *Stagger* and *Control* schedule I/O phases explicitly.

4.1 Spread

Spread is our baseline power sharing algorithm for improving the performance of as many applications as possible. *Spread* uses only the knowledge that an application is in an I/O phase and ignores the frequency and duration of the phase. During the I/O phase, *Spread* allocates this excess power evenly among all applications that can profitably use more power to make more progress. When the I/O phase completes, the application receives the power back. Algorithm 1 outlines *Spread*.

4.2 Priority

Priority is our baseline power sharing algorithm for improving the performance of one high-priority application. The motivation here is that on some systems, there may be a strict priority order of applications, and *PowerShifter* handles this case by finishing the high-priority application as quickly as possible. Like *Spread*, *Priority* uses only the knowledge that an application is in an I/O phase, ignoring phase frequency and duration. However, during the I/O phase, *Priority* simply allocates *all* of the excess power of an application to the highest-priority application that can make use of the power. As we know the power needs of each application, we can simply traverse the priority list of applications and stop when we reach the first application that will improve given more power. *Priority* may shift power to multiple applications in the event that the first application that can use power does not need all of the available excess power. We assume that priorities are provided by the system administrator and assigning them is outside the scope of this work.

4.3 Stagger

Spread and *Priority* perform quite well in many situations (see Section 7). However, situations exist in which power cannot be profitably used by either algorithm because we exert no control over when the applications reach their respective I/O phases. It becomes possible, especially as the I/O phase frequency and duration increase, to have enough applications enter their I/O phase concurrently that some of the available power cannot be profitably reallocated. A picture of this for three applications is shown in the leftmost pane of Figure 2.

Algorithm 2 *Stagger* algorithm. Functions *Enter_IO* and *Exit_IO* are omitted as they are identical to those in *Spread*.

```
 1: function APPLICATION_ENTER(AppType a)
 2:     g = findFreeGroup(a)
 3:     if g != NULL then
 4:         delay a
 5:         g.add(a)
 6:     else
 7:         g' = createNewGroup()
 8:         g'.add(a)
 9: function APPLICATION_EXIT(AppType a)
10:     g = a.getGroup()
11:     g.remove(a)
12:     if g.size < Threshold then
13:         for i in g do      ▷ Run entry protocol for each app in g
14:             Application_Enter(i)
15:         g.remove()
```

If instead the applications hit their I/O phases in a sufficiently disjoint manner, all power will be profitably used. Fortunately, as mentioned above, *PowerShifter* knows the I/O frequency and duration for each application. This allows us to schedule the applications intelligently to reduce (or ideally minimize) the amount of applications that are in I/O phases at any one time.

Our *Stagger* algorithm, detailed in Algorithm 2, accomplishes this by placing applications into groups. Within each group, *Stagger* calculates when the next I/O phase will occur for each application in the group and then staggers these phases so that only one application in the group is in an I/O phase at a time. *Stagger* avoids overlap of the next I/O phase for each application but, as a greedy approach, cannot guarantee avoiding overlap for all future I/O phases of the applications. When calculating the time of I/O phases, we do *not* take into account the effect of power limits or power sharing. When *Stagger* creates an I/O phase schedule, it implements that schedule by adding small delays to some applications to force their I/O phases to occur at the intended times (see the middle pane of Figure 2).

While it is possible that delaying applications can increase run time, the likely large number of time steps present in HPC applications allow these delays to be amortized. Therefore, it is likely that the improvement from better power utilization is well worth these delays. Also note that 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. In addition, *Stagger* limits the amount of delay that it will assign to any single application to a small percentage of the expected run time of that application. We expect *Stagger* to be most effective with jobs of similar size in terms of number of nodes. This is because staggering I/O phases of jobs that have vastly different node counts provides minimal benefit due to the wide disparity in power needs.

4.4 Control

The *Control* algorithm (see Algorithm 3) assumes that the run-

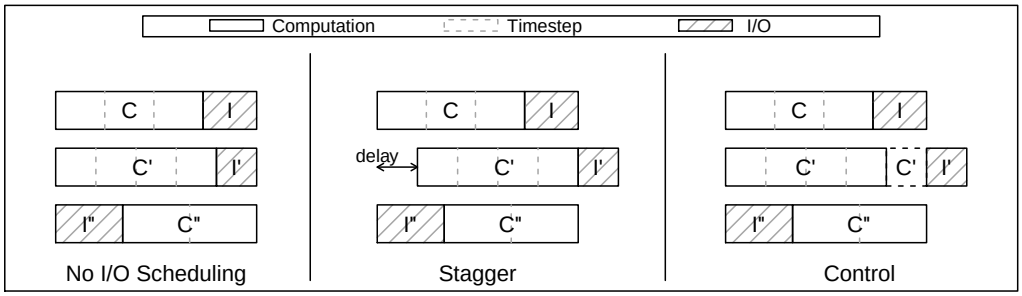

Figure 2: Picture of how *Stagger* and *Control* schedule I/O phases.

Algorithm 3 *Control* algorithm. Variable *my.lastScheduledIOTime* stores the expected time of the most recent I/O event. Also, the code to prevent an application from starving (never executing an I/O phase) is omitted for brevity.

```
1:  function ENTER_IO
2:      for i in allApps except me do
3:          if i.phase == Computation then
4:              neededPower += i.neededPower-i.allocatedPower
5:      cond1 = my.freePower <= neededPower
6:      cond2 = (currentTime - my.lastScheduledIOTime) > (k × d)
7:      if cond1 or cond2 then
8:          my.phase = IO
9:          Spread.Enter_IO()                    ▷ See Algorithm 1
10: function EXIT_IO
11:     my.phase = Computation
12:     my.lastScheduledIOTime += f
13:     Spread.Exit_IO()                         ▷ See Algorithm 1
```

time system has the ability to control I/O phase timing externally to the application. Conceptually, this could represent either an application that uses system level checkpointing or an application that executes an API call at the end of each timestep to determine if it is time to execute an I/O phase For example, if an application uses the checkpointing library SCR [29], it will call an SCR routine to determine if a checkpoint is needed at that time and only perform the I/O if the SCR library determines it is necessary. Thus, *Control* has the ability to minimize overlap of I/O phases across applications without incurring delays in any application.

While *Control* makes assumptions that are not common for the majority of current HPC applications (e.g., external control of I/O phase timing), we include it because it provides an approach that may become more important and common going forward. At exascale it may be necessary to defer I/O phases to avoid excessive overhead, and I/O phase libraries will need to be aware 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 of checkpoints, may become more prevalent.

The right pane of Figure 2 shows the basic operation of *Control*, which works as follows: applications request to enter I/O phases at appropriate times based on their I/O interval. When an application requests to enter an I/O phase, it will be allowed to do so only if that will not cause power to be wasted; that is, if some set of other applications can profitably use the full amount of power that will become available due to this application's I/O phase. If this is not true, the application will not enter an I/O phase and will attempt to do so again at the next opportunity, presumably at the end of the next timestep.

There are two special cases in the *Control* algorithm. First, if no applications are in I/O phases, the first application that requests to enter an I/O phase will be able to. This prevents situations in which an application is never allowed to execute an I/O phase because there are not enough other applications to use its excess power. Second, if an application's I/O phase has been delayed by more than $k \times d$, the next I/O request will be granted. Here, d is the length of the interval between I/O phases and k is a constant (currently 0.5, or half the time between I/O phases). This prevents the situation in which an I/O phase is delayed sufficiently long to have a negative impact on reliability or on the post-mortem visualization. In addition, when an application's I/O phase is delayed, the amount of time until the next I/O phase will be reduced by the delay time to ensure that the I/O phase interval will be unchanged over the entire program[2]. As with k above, the idea is to avoid a negative impact on reliability or visualization.

5. SIMULATOR

In order to test the effects of power shifting on application execution, we developed *PowerShifter*, a simulator that can explore the effects of power shifting between applications. Using a simulation allows us to execute experiments quickly and investigate situations not supported by current hardware or applications.

5.1 Description

PowerShifter simulates a set of applications executing together on a hypothetical power-constrained HPC system. The input to *PowerShifter* includes a set of applications, together with a descriptor for the computational and I/O phase of each as described in Section 3, as well as a power shifting algorithm to use. The descriptor can be explicitly specified or selected randomly from distributions that are based on our experiences with real applications. The output from *PowerShifter* is the run time of each application, including the cost of any delays incurred by the power sharing algorithm.

PowerShifter runs by dividing time into discrete time steps. At the beginning of each time step, any applications that arrive are started. In addition, any paused applications (if the *Stagger* power shifting algorithm is used) that have reached the end of their pause period are allowed to continue. At this point, if we are using *Stagger*, *PowerShifter* lines up I/O phases, one after the other, in time, which may include pausing additional applications. It then calculates power needs for each application based on its current phase (computing, I/O, or paused) and re-allocates any unused power based on the chosen power shifting algorithm. Then, *PowerShifter* advances one time step, which involves calculating the amount of work each application is able to perform. This is a function of

[2]It may be impossible to avoid a slight decrease in the I/O phase interval if an I/O phase is delayed near the end of the application.

the application's current power allocation relative to the amount of power it could profitably use, the current phase, and the response to changes in its power limit.

At the end of the time step, each application's current phase is updated, which may involve entering or exiting an I/O phase. *PowerShifter* can then inject failures based on an exponential distribution and a user-provided MTTF. When a failure occurs, the application is placed back to the last checkpoint, a restart penalty is assessed, and then the application continues from that point. This overall cycle continues until all applications have completed.

Injecting failures is orthogonal to how the algorithms handle the failure. *Spread*, *Priority*, and *Control* do not take any specific action on a failure. On the other hand, *Stagger* has to potentially redo group assignment. *Stagger* accomplishes this by simply treating a failure as a job departure and a restart as a job arrival—so calls `ApplicationEnter` and `ApplicationExit`, respectively (see Algorithm 2).

One limitation of *PowerShifter* is that time is discretized, which forces all significant events (such as phase changes) to occur at the boundaries of time steps. In some cases, such as the start and end of I/O phases, we account for and reduce this error. However, in general, there will be some error that is related to the size of the time steps. In the next section we show that the simulator does in fact match our implementation quite closely.

6. IMPLEMENTATION AND VALIDATION

We have implemented our techniques in a prototype that dynamically reallocates power between executing applications. We use our prototype to validate *PowerShifter* as well as provide some results on real hardware to get an idea of the potential of our approach. We discuss the implementation and validation in turn.

6.1 Implementation

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

Our prototype runs on Cab, which is a cluster of 1,296 Intel Xeons housed at Lawrence Livermore National Laboratory. Cab uses SLURM [39] for resource management, and our prototype is is orchestrated via a shell script that is submitted to MOAB. The script requests a node allocation that is large enough for all the jobs that will be run, with one additional node used for the controller. When the requested node allocation is granted, the script uses SLURM to start the controller on one of the nodes and the runtime on all other nodes. It then uses SLURM to start jobs on disjoint subsets of nodes, excluding the node the controller is running on. The script exits when all jobs, including the controller and all the runtimes, have completed.

The runtime runs in the background on each node, communicates with the wrapper and the controller, and implements node-level power limits using the RAPL (Running Average Power Limit) MSR (Model Specific Register) interface [18]. It pauses and resumes the application as necessary using the signals `SIGSTOP` and `SIGCONT`, and accesses MSRs via `libmsr` [38] and `libraplto` set power limits and record power usage and CPU activity for debugging and analysis. In our prototype, the runtime is a user-space process. While our prototype requires access to RAPL MSRs from user space, it could be adapted to run on any cluster that provides power-limiting capability.

The final component in our prototype is the controller. Only one copy of the controller is running during each experiment over several jobs. The controller communicates with the runtimes on each of the nodes, and then based on node-level information (specifically regarding application state and power needs), the controller makes power allocation and application pause/continue decisions. The individual runtimes carry out these decisions on their respective nodes. The controller is configurable and allows new power sharing algorithms to easily be developed and used. The controller is application aware—it groups nodes by applications—so power sharing algorithms are free to make decisions at the node level or application level.

Operationally, the controller and runtimes start up first, and each of the runtimes 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 indicating that the application has started on that node. Control is then returned to the application. The runtime sends a signal to the controller indicating that the application has started on that node. It also includes information about the application, including the application's power needs during computation and I/O.

As mentioned in Section 2, our prototype assumes that the power usage information about the application is known. The controller responds to the runtime by giving it an initial power limit, which the runtime implements using MSRs (no communication with the application is required to set the power level). At some point after this , the application enters an I/O phase. Our prototype currently has the application communicate to the wrapper via a user-inserted `MPI_Pcontrol` call; this is just one way to handle this, and a full-scale implementation could handle this in less intrusive ways. The wrapper forwards this information to the runtime, which sends it to the controller. At this point, depending on the power sharing algorithm that the controller is running, it has the option of lowering the power limit on the node that is in an I/O phase and possibly also raising the power limit on some subset of other nodes. As noted above, the controller also has the option to pause or resume the application at any time—this is also dependent on the power sharing algorithm that the controller is running.

6.2 Simulator Validation

In order to validate our *PowerShifter* simulator, we compare it to our prototype using LAMMPS, ParaDiS, and Cactus. LAMMPS is used as a molecular dynamics simulation and is from the ASC Sequoia benchmark suite [1]. ParaDiS [5] is a production dislocation dynamics simulations application that operates on dynamically changing, unbalanced data set sizes across MPI processes. We used the "Copper" input set. Cactus [16, 2] is a framework that numerically solves Einstein's equations [37] via adaptive mesh refinement.

We chose these applications because each of them represents actual applications run on modern HPC systems and uses periodic I/O phases. We first executed each application at various power levels to create a simple model of how the application responds to power limit changes. Specifically, we fit a third degree polynomial to this data and then scaled the polynomial based on a number selected from a uniform interval (see Section 7.1 for more details). This model was used by *PowerShifter* to predict the run time of these applications running under various power sharing algorithms. We compared the run time predictions from *PowerShifter* with the median run time of at least five actual runs. The results are provided in Table 1.

In all cases, *PowerShifter* was able to predict application run time with less than 2% error and, in most cases, less than 1% error. The most significant source of error that we noticed was accounting for increase in performance due to an increase in allocated power. The worst case (not shown in Table 1) occurred with one experiment with LAMMPS, in which its error when increasing power from

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 1: Validation runs.

60W to 80W was 3.3%. We note that overall, our implementation showed improvements in application runtimes as high as 26%.

7. RESULTS

This section describes the results of our *PowerShifter* simulator. In turn, we describe sets of simulated experiments on systems with large numbers of applications and then small numbers of applications.

7.1 Setup

In most of our simulation results, the number of nodes and the job arrival times were taken directly from the a job trace (see below). Unless other wise stated, the power limit for all tests was 60W per node. Unfortunately, many application characteristics that we need for our simulation are not available in traces—including number of iterations, power consumption, I/O phase duration and frequency, and application response to changes in allocated power. We derived these application characteristics as follows, based on the real HPC applications described in the previous section (LAMMPS, ParaDiS, and Cactus).

- We estimated the number of application iterations using the run time from the trace, and we chose the iteration time from a uniform distribution between 0.1 and 2 seconds (applications will generally perform several iterations between I/O phases).
- We chose the power consumption during computation from a uniform distribution between 80W and 100W per node, and we chose I/O phase power from a uniform distribution between 35W and 45W per node.
- We chose I/O phase time from a uniform distribution between 30 seconds and 5 minutes.
- Some applications enter I/O phases after a fixed number of iterations, and others enter I/O phases after a fixed amount of time. In our simulation, roughly 80% of applications enter I/O phases after a fixed number of iterations, and the rest enter I/O phases after a fixed amount of time.
- The number of iterations or amount of time between I/O phases was calculated based on the iteration time, I/O phase time, and percent of time spent in I/O phases, which was an input to the simulation.

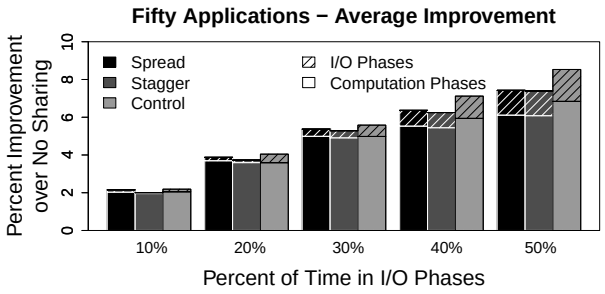

Figure 3: Average percentage improvement with 50 applications.

- To emulate the response of applications to changes in the power limit, we fit the third degree polynomial discussed in Section 6.2 to data from runs of ParaDiS at several different power levels. The minimum improvement due to an increase in power (10%) represents a memory-bound program, and the maximum improvement (250%) represents a CPU-bound program. The 250% number is modeled after the range of frequencies on the *Cab* cluster that we observed when setting power bounds on ParaDiS of 40 W to 90 W, which was 1.2GHz to 3.0GHz.
- We generally investigate a range of percentage of time an application spends in I/O from 10% to 50%. While 50% is larger than typically encountered, we use it as our upper bound as a way to study effects on our algorithms of increasing I/O phase time.

7.2 Large Numbers of Applications

Our first set of experiments cover scenarios with large numbers of concurrent applications, similar to those found on capacity machines. In this subsection, all experiments used 50 applications running on a simulated 512 node cluster, except as noted for those in Section 7.2.3. We used a portion of the RICC trace from the Parallel Workloads Archive [21], which was produced from the RIKEN Integrated Cluster of Clusters in Japan. In these tests, we randomly injected failures with an MTTF of 41 minutes across the entire cluster, except as noted in Section 7.2.2.

7.2.1 Average Improvement

The average percentage improvement of our tests is shown in Figure 3. This experiment covers 50 applications across several runs with different sets of random parameters. There are two possible sources of performance improvement. First, an application executes computation faster due to power shifting, and second, an application may execute fewer I/O phases. The latter situation occurs in two cases. First, if an application measures the interval between I/O phases in time rather than iterations, it can execute fewer I/O phases when it runs faster. Second, the *Control* algorithm may push the final I/O phase of an application beyond the end of the application. To show these two sources of improvement, we have divided the improvement into the portion due to faster computation (solid fill) and the portion due to dropping I/O phases (crosshatched). We do not display *Priority* in this experiment, because *Priority* is intended to provide maximum (not average) improvement to one (or a small set) of applications.

As expected, as the amount of time applications spend in I/O phases increases, the benefit of power shifting also increases. At this large number of applications, *Stagger* performs slightly worse

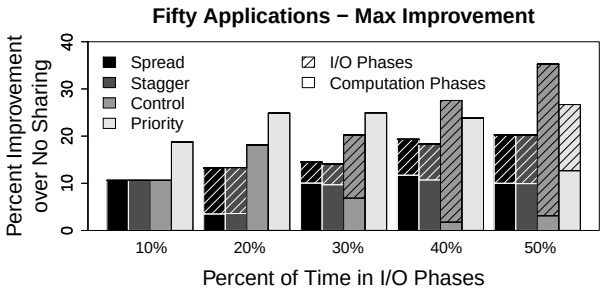

Figure 4: Maximum percentage improvement with 50 applications.

than *Spread*, simply because *Stagger* continues to incur delays to stagger I/O phases even when the benefit is minimal. When applications spend 10% of time in I/O phases, the benefit of *Control* over *Spread* is small; when there are a large number of 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* over *Spread*. Also, a relatively small amount of the improvement comes from dropped I/O phases, which is consistent with the majority of applications taking their I/O phases after a set number of iterations rather than a set amount of time. *Control* occasionally pushes an I/O phase past the end of the run, which results in a larger benefit from dropped I/O phases than the other two algorithms. However, this benefit is only a fraction of the total benefit of *Control* over *Stagger* and *Spread*.

7.2.2 Maximum Improvement

Figure 4 shows the maximum improvement over the fifty applications with each algorithm. We did not inject failures in this test, because a failure that occurs soon after a checkpoint in one run and just before a checkpoint in a different run can create a large performance improvement that is not related to power shifting. For *Priority*, we assigned priorities in reverse order of CPU-intensiveness; applications that are highly CPU-intensive get the highest priority. Each run is executed several times with different random parameters. The graph shows the median of the maximum improvement achieved by any application for a given run. As in Figure 3, we have subdivided the improvements due to faster computation (solid fill) and dropping I/O phases (crosshatched).

For *Spread*, *Stagger*, and *Control*, the majority of the improvement comes from applications that are able to avoid some I/O phases. However, *Priority*, which is focused on executing one application as fast as possible, achieves the majority of its improvement from simply executing computation phases faster. *Priority* is also able to achieve larger maximum improvements than the other algorithms. The only exceptions to this are *Control* at 40% and 50%. In both of those cases, *Control*'s large improvement was for a short running application that took one I/O phase in the baseline run, and *Control* shifted that I/O phase past the end of the run. Thus, the improvement from *Priority* is more beneficial overall.

7.2.3 Effect of Power

The relationship between an application's computation power, I/O phase power, and system-imposed power limit can also affect the effectiveness of power shifting. To explore this dimension, we executed several tests in which we varied the global power limit such that the power allocated to each node ranged from the minimum to maximum amounts. Unlike the previous two subsections, these experiments used only 20 applications because of otherwise excessive simulation time. In addition, here the I/O and computation power per node were 40W and 80W, respectively. In this test, the cluster was large enough to run all jobs, so jobs were scheduled when they arrived.

We executed tests for cases in which I/O phases took up 10%, 30%, and 50% of execution time. The results are shown in Figure 5. The power limit shown on the x-axis is the global power limit divided by the number of nodes; thus it is the power limit per node *in the absence of power shifting*. In all three graphs, only *Spread* and *Control* are shown; the other algorithms had performance similar to or worse than *Spread*. In these graphs, the percentage shown in the title is the percent of time spent in I/O phases at 65W.

At the lowest power limit (40W per node), no benefit from power shifting is possible because no application will ever have unused power. Similarly, when the power limit is 80W per node, all applications will always have enough power; this is analogous to executing without a power limit. In the cases where *Control* does produce a benefit at 40W or 80W, this is either because it shifted one I/O phase past the end of the run or because failures occurred at favorable times relative to checkpoints—no benefit was achieved due to power shifting.

In the leftmost graph of Figure 5, there are two important inflection points. When the power limit is less than 60W per node, up to half of the nodes can be in I/O phases at the same time without power being lost, so *Spread* and *Control* have similar performance. (Note here that different applications have different numbers of nodes.) When the power limit is above 60W per node, less than half of the nodes can be in an I/O phase at a time without power being lost, so there the difference between *Spread* and *Control* increases. Finally, above 70W per node, there is sufficient power in the system that power will be lost when even just a few nodes are in I/O phases at the same time. Hence, the benefit of power shifting diminishes. The middle and rightmost graphs show similar patterns, though performance falls off after the inflection point more quickly for the rightmost graph.

7.3 Small Numbers of Applications

Our second set of experiments cover scenarios with small numbers of concurrent applications, similar to those found on capability machines. Reducing the number of applications increases the probability that straightforward algorithms such as *Spread* that do not attempt to schedule I/O phases will be insufficient to get the full benefit of power shifting. Accordingly, this section explores scenarios in which there is a significant difference between *Spread* and more sophisticated algorithms such as *Stagger*.

While aligning all applications so that their I/O phases occur concurrently represents an unlikely situation, we include them to investigate the potential benefits of *Stagger* and *Control*. Accordingly, for the following two subsections, we fix application characteristics instead of using the general ones identified in Section 7.1. In these tests, the global power limit was set halfway between I/O phase power and computation power for each application.

7.3.1 Perfect Alignment of I/O Phases

This subsection presents situations in which several applications with identical attributes (I/O phase time, computation phase time, number of nodes, and response to power limit changes) are started at the same time. This means that all of their I/O phases also happen at the same time, which gives *Spread* no opportunity to share power; it represents a near-optimal situation for *Stagger* or *Control*. We do not display *Priority* in this experiment for the same reason

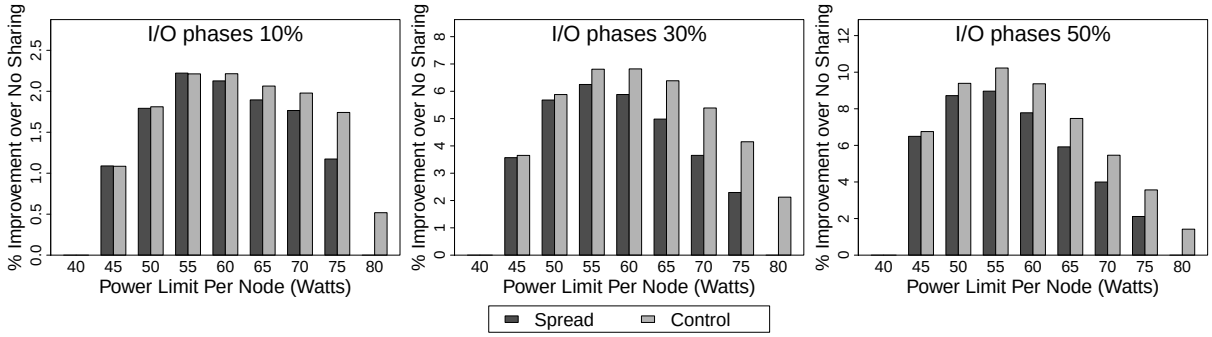

Figure 5: Average percentage improvement under different power limits with I/O phases consuming 10% (left), 30% (middle), and 50% (right) of the time.

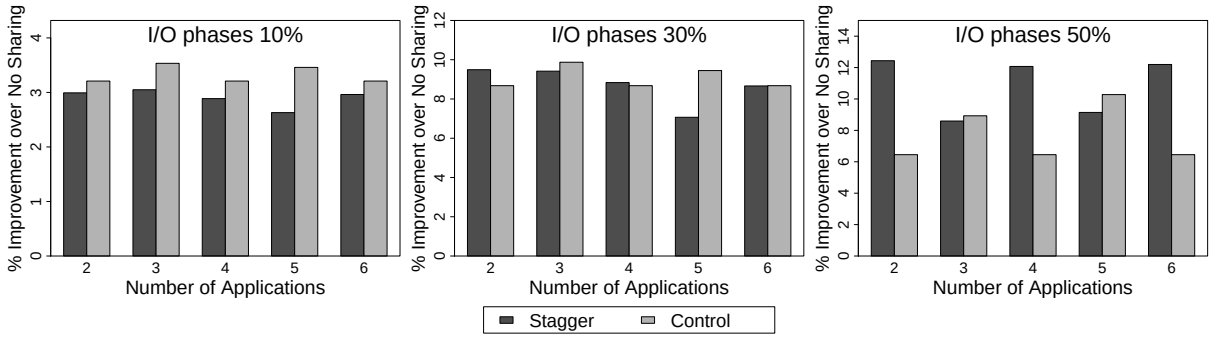

Figure 6: Average percentage improvement with I/O phases aligned and consuming 10% (left), 30% (middle), and 50% (right) of the time.

as in Section 7.2.1.

Figure 6 shows, for three different I/O phase percentages, improvement over the baseline (no power shifting) for different numbers of applications using *Stagger* and *Control*. As expected, *Stagger* achieves similar improvement to *Control*; both shift I/O phases to ensure they do not overlap, but *Control* does so without incurring the cost of delays. Also note that the magnitude of the improvement increases as the percent of time spent in I/O phases increases. This is expected because more time in I/O phases means there is more opportunity to shift power. As discussed above, *Spread* achieves no improvement in any of these experiments and so is omitted.

Several aspects of this figure deserve more attention. First, in all 3 graphs there is a dip in the performance of *Stagger* when five applications are executing. This occurs because *Stagger* creates a second group for the fifth application, so the first and fifth applications execute their I/O phases at roughly the same time, which reduces the efficiency of the power shifting.

Second, the *Control* algorithm consistently achieves better performance for odd numbers of applications than for even numbers of applications. This is due to the non-linear relationship between power and performance—shifting extra power to an application at a low power limit results in more improvement than shifting the same amount of power to an application that is already at a high power level. When there is 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 their I/O phases. This puts all applications at their max power level for a short period of time. When there is 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 prevent power being lost. The ultimate result is that applications are given a smaller amount of extra power for a longer amount of time, which results in better improvement than giving applications a large amount of power for a short amount of time.

Both algorithms run into difficulty in the rightmost graph (I/O phases 50% of the time), as evidenced by their widely varying performance at different numbers of applications. Note that in this case, applications spend roughly the same amount of time in I/O phases as they do in computation phases. Thus, for even numbers of applications, *Stagger* ends up aligning applications well; taking four applications as an example, the first application is in an I/O phase at the same time as the third application, and the second application is in an I/O phase at the same time as the fourth applications. Thus, half of the applications are in I/O phases at any given time, so little power is lost. However, if there is an odd number of applications, more than half of the applications will end up in I/O phases at the same time, so power is guaranteed to be lost.

Control shows relatively poor performance here because it is up against the limit on the amount of time it will delay an application. For even numbers of applications, it allows half of the applications to enter I/O phases and delays the I/O phases of the other half of the applications. However, halfway through the I/O phase, the second set of applications reaches the limit on the amount of time an I/O phase can be delayed and start their I/O phases. Thus, all of the applications are in I/O phases for a significant period of time,

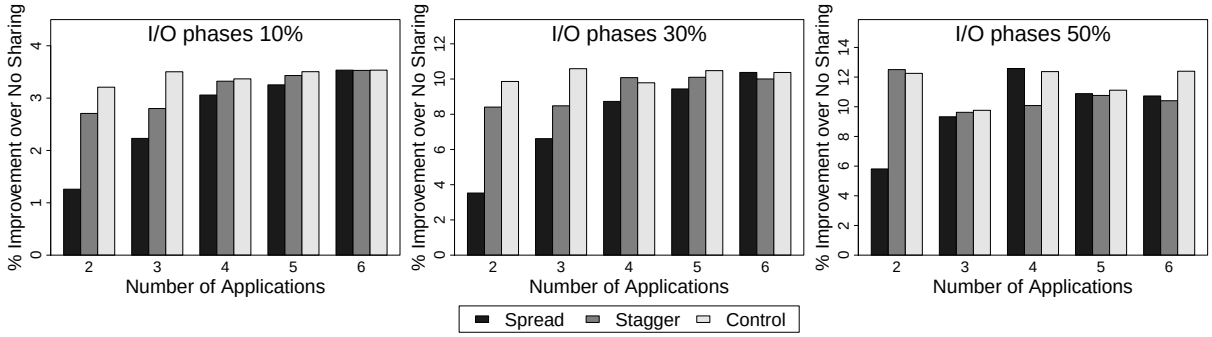

Figure 7: Average percentage improvement with I/O phases partially aligned and consuming 10% (left), 30% (middle), and 50% (right) of the time.

which reduces the efficiency of the algorithm. For odd numbers of applications, a similar effect happens, but it is mitigated by the fact that the two groups of applications are different sizes, which causes applications to be sped up by different amounts and begins to naturally stagger the I/O phases.

7.3.2 Partial Alignment of I/O Phases

Figure 7 shows identical information to Figure 6 except that application start times are staggered 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 $i - 1$. This is a more realistic situation in which *Spread* has at least some opportunity to shift power. *Stagger* and *Control* exhibit roughly similar behavior as that in Figure 6 except that *Control* is able to achieve larger improvements when applications spend a large percentage of the time in I/O phases. As noted in the previous section, when applications spend 50% of the time in I/O phases, *Control* is up against its limit in the amount of time it will delay an I/O phase, so several I/O phases end up overlapping. With partially aligned I/O phases, the amount of delay required to stagger I/O phases is significantly less, so *Control* is well within its limit.

At two applications, *Spread* is only able to achieve a small amount of improvement because when both applications are in I/O phases, there are no other applications to accept shifted power. As the number of applications increases, this situation is mitigated, until at six applications, *Spread*'s performance is indistinguishable from that of *Control*. We also see that with enough applications spending a high enough amount of time in I/O phases, *Spread* starts to outperform *Stagger*. This is because *Stagger* will continue to stagger I/O phases and incur the cost of delays, even when it is unnecessary.

In the rightmost graph of Figure 7, *Spread* achieves a dramatic improvement with four applications, and *Stagger* and *Control* achieve similar improvements in some of the tests with two, four, and six applications. In all of these cases, applications spend half their time in I/O phases, and half of the applications are in I/O phases at any given time. This is the best possible scenario for power shifting: applications spend half of their time in computing and benefit from power shifted from other applications during the entire computation phase, and then spend the other half of their time providing power that can be shifted to other applications. As these figures show, that can result in improvements in excess of 12%.

8. RELATED WORK

This section discusses work related to this paper. We discuss power-constrained supercomputing, power shifting within a single application, and advanced checkpointing techniques and their impact.

8.1 Power-Constrained Supercomputing

The power limit specified by the Department of Energy for exascale systems is 20 megawatts. Several researchers have studied different aspects of executing high-performance computing (HPC) applications under a power bound.

One approach is to use *overprovisioning*, in which there are more nodes than can be fully powered without violating the facility power bound. System software then determines how to "reconfigure" the nodes for a particular application [30, 34] or for multiple applications [31, 33, 12, 13]. Mechanisms such as Intel's Running Average Power Limit (RAPL) [8] allow power to be explicitly allocated to different machine components, including the CPU and memory.

One disadvantage of RAPL is that it only adjusts the frequency and voltage settings, which may not be best for performance. There are other techniques that have been used to limit power consumption while maintaining good performance. One common technique is dynamic concurrency throttling, where the number of active cores is reduced [9, 7, 22, 23].

8.2 Power Shifting Within an Application

There have been many systems and techniques developed to move power within a single application. Generally speaking, the idea is to exploit load imbalance to shift power from nodes off of the critical path to those on the critical path. Marathe et al. [27] move power by resetting RAPL-based power caps. Their work was motivated by work on saving energy on off-critical path nodes (rather than shifting power), which included systems such as Adagio [32], CPU-Miser [15], and Jitter [20]. *PowerShifter* differs from all of these systems in that it shifts power *between* applications, rather than within an application.

8.3 Power Shifting Between Applications

There has been some recent work on shifting power between different applications. Ellsworth et al. [11] collects excess power by determining the per-node difference between the power cap and the consumed power. Nodes with sufficiently large differences donate power to nodes with small (or zero) difference. Liu et al. [25] classifies phases—by inspecting hardware counters—as CPU, I/O, or "undetermined" and then shift power from inferred I/O phases to inferred CPU phases.

PowerShifter differs from the above two approaches in two ways.

First, *PowerShifter* uses semantic information about applications, which allows avoidance of situations in which, for example, quickly changing power consumption leads to hysteresis. Second, *PowerShifter* uses this semantic information to create two sophisticated algorithms (*Stagger* and *Control*).

8.4 Advanced Checkpointing

Checkpointing is still the most dominant technique to achieve resilience and can either be managed by the system transparently to the application, as in BLCR [17], or explicitly performed by the application itself. The latter requires modifications to the application, but provides additional optimization potential in terms of timing checkpoints and reducing their size and is the most common type of checkpointing on HPC systems.

Our work is mostly orthogonal to the type of checkpointing used, as long as we can determine the checkpoint interval and duration either by application instrumentation or by integrating our techniques into checkpoint libraries such as BLCR [17] or SCR (The Scalable Checkpoint/Restart Library) [29]. However, with the growing scale of systems, checkpointing itself faces challenges, which require new approaches like the use of NVRAM in the form of in-system burst buffers [24, 36], asynchronous transfers of checkpoints [35], or checkpoint compression [19]. The use of such techniques will impact the power draw and cause more complicated shifts of varying magnitudes from different components (from the CPU, to the memory system all the way to the I/O system) and at potentially higher frequencies [29]. These developments will likely require extensions to our power shifting scheme to maintain global power efficiency, yet will make the key idea behind our approach—integrating knowledge of the checkpointing process into power shifting decisions—even more critical.

9. SUMMARY

A major challenge for near-future exascale systems is the DOE's 20 MW power bound. System software will need a major overhaul in order to achieve the high performance required for scientific discovery while still honoring this power bound. In this paper, we explore an opportunity for significant performance improvement by shifting power from applications executing in I/O phases to those executing in computation phases. We present the design and implementation of four different algorithms for shifting power—two straightforward and two more complex—and evaluate them in both a simulator, and in a validation in a real implementation. We showed that staggering applications to avoid I/O phase overlap and externally controlling I/O phases can provide significant benefit. Results in *PowerShifter* show that our algorithms achieve significant performance improvement compared to no power sharing.

The algorithms we developed and the results from our simulator and implementation have the potential to have impact on 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 their system usage characteristics. Contrary to naive expectations, we found that straightforward algorithms performed quite well with a large number of concurrent jobs as found on capacity systems. This finding will be of benefit to 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, on capability systems, more complex algorithms may be necessary and may be worth a more expensive scheduling algorithm.

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.

10. REFERENCES

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

[2] 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*, 4(3):179–188, 2001.

[3] S. Amarasinghe and et al. ExaScale Software Study: Software Challenges in Extreme Scale Systems. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`, Sept. 2009.

[4] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, P. Kogge, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. `http://www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf`, Sept. 2008.

[5] 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.

[6] K. W. Cameron, R. Ge, and X. Feng. High-Performance, Power-Aware Distributed Computing for Scientific Applications. *IEEE Computer*, 38(11), 2005.

[7] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction. In *International Conference on Supercomputing*, 2006.

[8] 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*, ISLPED '10, pages 189–194, 2010.

[9] Y. Ding, M. Kandemir, P. Raghavan, and M. Irwin. A Helper Thread Based EDP Reduction Scheme for Adapting Application Execution in CMPs. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[10] 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)*, pages 1–6, 2012.

[11] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz. POW: System-wide Dynamic Reallocation of Limited Power in HPC. In *Symposium on High-Performance Distributed Computing*, Jun 2015.

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

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

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

[15] 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.

[16] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit: Design and Applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2003. Springer.

[17] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[18] 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`.

[19] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. De Supinski, and R. Eigenmann. McrEngine: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[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*, 68:1175–1185, 2008.

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

[22] D. Li, B. R. de Supinski, M. Schulz, K. W. Cameron, and D. S. Nikolopoulos. Hybrid MPI/OpenMP Power-Aware Computing. In *24th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2010.

[23] D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. de Supinski, and M. Schulz. Power-Aware MPI Task Aggregation Prediction for High-End Computing Systems. In *24th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2010.

[24] 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.

[25] 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.

[26] R. Lucas et al. Top Ten Exascale Research Challenges: DOE ASCAC Subcommittee Report. `http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf`, Feb 2014.

[27] 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 *International Supercomputing Conference*, July 2015.

[28] 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*, pages 35–42. IEEE, 2012.

[29] 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.

[30] 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*, June 2013.

[31] 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*, June 2015.

[32] B. Rountree, D. Lowenthal, B. de Supinski, M. Schulz, V. Freeh, and T. Bletch. Adagio: Making DVS Practical for Complex HPC Applications. In *International Conference on Supercomputing*, June 2009.

[33] 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.

[34] 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*, pages 1–8, Sept 2013.

[35] 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.

[36] 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.

[37] E. Seidel and W. Suen. Numerical Relativity as a Tool for Computational Astrophysics. *Journal of Computational and Applied Mathematics*, 109(1-2):493–525, 1999.

[38] K. Shoga and B. Rountree. libmsr. `https://github.com/scalability-llnl/libmsr`.

[39] A. Yoo, M. Jette, and M. 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. Springer-Verlag, 2003.