Dynamic Power Sharing for Higher Job Throughput

Daniel A. Ellsworth, Allen D. Malony
University of Oregon
Eugene, Oregon, USA
{dellswor,malony}@cs.uoregon.edu

Barry Rountree, Martin Schulz
Lawrence Livermore National Laboratory
Livermore, California, USA
{rountree4,schulzm}@llnl.gov

ABSTRACT

Current trends for high-performance systems are leading towards hardware overprovisioning where it is no longer possible to run all components at peak power without exceeding a system- or facility-wide power bound. The standard practice of static power scheduling is likely to lead to inefficiencies with over- and under-provisioning of power to components at runtime. In this paper we investigate the performance and scalability of an application agnostic runtime power scheduler (POWsched) that is capable of enforcing a system-wide power limit. Our experimental results show POWsched is robust, has negligible overhead, and can take advantage of opportunities to shift wasted power to more power-intensive applications, improving overall workload runtime by as much as 14% without job scheduler integration or application specific profiling. In addition, we conduct scalability studies to determine POWsched's overhead for large node counts. Lastly, we contribute a model and simulator (POWsim) for investigating dynamic power scheduling behavior and enforcement at scale.

CCS Concepts

•**Software and its engineering** → **Power management;**

Keywords

RAPL; hardware over-provisioning; HPC; power bound

1. INTRODUCTION

Scalable parallel applications have been the driving force behind the evolution of large systems with their ever-increasing demands for processor, memory, and network performance. This evolution over the past decade has followed a "horizontal" scaling strategy to increase floating-point operations per second (*flops*) and input-output operations per second (*iops*) by adding more of the latest hardware. Unfortunately, powering a massive cluster at the maximum power draw of all hardware components simultaneously is a major technical and cost challenge that will become infeasible for future machines. Because few applications are able to fully exploit all components at peak capacity [10], providing maximum power is often unnecessary. These observations will force system designers to rethink scaling strategies for high-end systems from the ground up.

A possible alternative is *hardware over-provisioning*, where more hardware is available than can be powered at maximal draw at any time [9]. Power systems and system scales are designed for the common case, requiring mechanisms to prevent the system from exceeding the predetermined maximal power (i.e., a system wide power bound must be enforced). New technologies, such as Intel's *Running Average Power Limit (RAPL)*, which provide a software configurable and hardware enforced power cap per socket, are key to this approach. Their use requires a power distribution algorithm to allocate the available power across the system. The naïve solution is to assign equal power to all sockets across the cluster. One consequence of the naïve approach is that it wastes power[1] on the applications that do not execute at the fixed power limit. Alternatively, a dynamic system-wide power scheduler can detect and reallocate the wasted power resources to efficiently utilize a hardware over-provisioned system.

In this work, we evaluate the dynamic power reallocation strategy implemented in *POWsched* [3]. POWsched is a dynamic power scheduler that enforces a global power bound and uses a simple heuristic, based on current per socket power consumption and allocation, to guide scheduling decisions. POWsched is agnostic to the applications running on the system and does not coordinate with the job scheduler. While even better performance is expected with job scheduler integration, POWsched has been observed to reduce the overall runtime versus the naïve solution in power constrained settings. When power is plentiful, overall runtimes are within a standard deviation of the unbound time.

Specifically, our research contributes the following:

- An experimental evaluation of a dynamic power scheduler on an HPC-class system with RAPL power control capable of enforcing a global power bound.
- A scalability study of the POWsched algorithm on BG/Q to determine its overhead on large numbers of sockets.

[1]The power is "wasted" in the sense that it limits the hardware resources that could otherwise be assigned to other concurrent jobs.

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807643

- A simulator for experimenting with power enforcement at scale. This addresses the current lack of large-scale clusters exposing RAPL-like power adjustment at runtime.

Section 2 provides a brief background on power research in HPC systems. The environment and design of POWsched is described in Section 3. We discuss the implementation of POWsched, POWmon, and POWsim in Section 4. Section 5 discusses our experimental results. First, we report a range of workload execution experiments on a RAPL-enable HPC machine demonstrating POWsched's implementation. Second, we show the scaling results characterizing POWsched overhead. Lastly, we give results from the POWsched simulation. Section 6 discusses future work and conclusions.

2. BACKGROUND

Existing work on power consumption and management is primarily focused on per job optimization of total energy consumed. While total energy consumption is reduced when the rate of consumption is decreased without increasing runtime, reducing total energy consumption is a different goal than enforcement of a system-wide power limit[2].

2.1 Power Optimization

Most existing power scheduling work seeks to simultaneously minimize power consumption and computation runtime. Hoffmann [5] observes that there is a class of applications that operate with real-world time constraints and the minimum computation runtime is equivalent to one that completes just ahead of the deadline. Bambagini et al. [1] use such timing constraints for power optimization in real-time embedded devices with periodic inputs. Mistral [7] uses a target request latency for VM migration and activation to reduce data center energy costs while maintaining service quality. The majority of existing HPC workloads are not interactive and are unlikely to benefit from real-world (clock) time-based deadlines.

Adagio [12] uses DVFS to conserve energy for instrumented processes. Adagio uses hardware performance counters and instrumented MPI calls to measure program progress and define task boundaries. Deadlines are based on the estimated time of the last communication participant rather than real-world time. The progress measures are used independently, per processor, to estimate the frequency for the next scheduling interval, attempting to minimize power without impacting runtime. Adagio, unlike our current work, is uncoordinated across nodes and does not guarantee that a global power budget will be maintained.

Green Queue [13] uses DVFS control and a precomputed database of observations to reduce power consumption when compared with the default power consumption of an application. Prior to running a workload for power savings, static analysis is done to help instrument the workload and runs in differing configurations are done to develop profile information. Machine learning is used to develop models, which are applied at runtime to guide clock frequency selection. For the experiments reported in the paper, Green Queue produced an average power savings of 12.5% with an average performance loss of 5.2%. The power and time cost in instrumentation and model production were not discussed.

[2]Energy is a quantity measured in joules. Power is a rate measured in watts, $\frac{\text{joules}}{\text{seconds}}$.

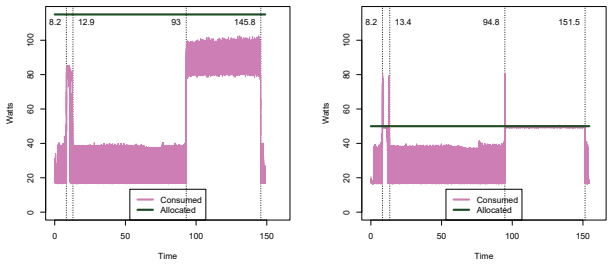

Figure 1: Spikes above the allocation are an artifact of the 100ms POWmon sampling interval being smaller than the 1 second RAPL window. Limits at 115 watts left and 50 watts right.

The general problem of fine-grained resource scheduling in modern systems is quite complicated due to the number of software configurable elements. PTRADE [6] explores power and performance optimization of an application on a single host in the presence of different configuration elements on different hardware platforms. A heartbeat from the application is used to measure application runtime performance (progress). To avoid exploring the configuration space and developing application profiles before use, PTRADE adapts the model used for application configuration at runtime based on observed performance side-effects. POWsched is extremely simple in contrast, only observing power consumption and only adjusting the power allocation. On the other hand, POWsched operates across multiple nodes of an HPC cluster.

Patki et al. [9] explore over-provisioning of hardware. They explore the runtime of differing node and processor counts under differing global power bounds and show that the optimal runtime for a given bound is not necessarily the configuration using all available processors. Patki et al. [10] show that power estimates given at job submission time, together with the flexibility to reduce actual power allocation, can be used to reduce the time from job submission to job completion. Our current work does not interface with the jobs or job scheduler in anyway. Also, POWsched makes allocation adjustments during execution. We expect large performance gains are possible with job scheduler integration and plan to explore such integration in future work.

2.2 Effect of Bound

Rountree et al. [11] and Fukazawa et al. [4] investigate runtime performance of applications under fixed power bounds. Results from this work show a non-linear correlation between power allocation and overall runtime as power bounds are lowered. Additionally, the work by Rountree et al. [11] shows that under the same power constraint, processors of the same model, have different performance characteristics. The relationship between power bound, consumption, and runtime are fundamental to the POWsched heuristic and the POWsim simulation model.

Applications may not use power at a consistent rate throughout their execution. When application power consumption, c, is beneath the power allocation, a, negligible impact to application runtime is expected. For instance, Figure 1 shows time measurements in seconds of an 8-node (2 sockets

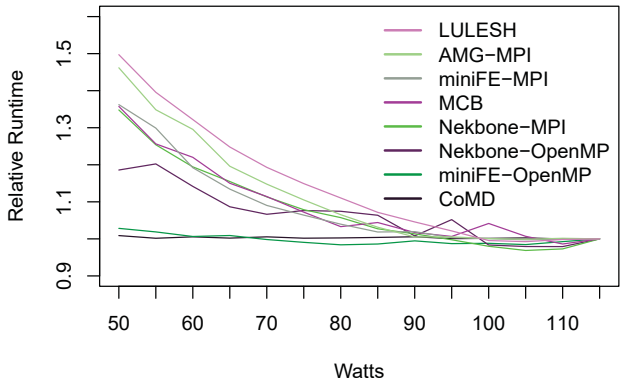

Figure 2: Runtime effect of decreasing bounds for different CORAL benchmarks and parameters.

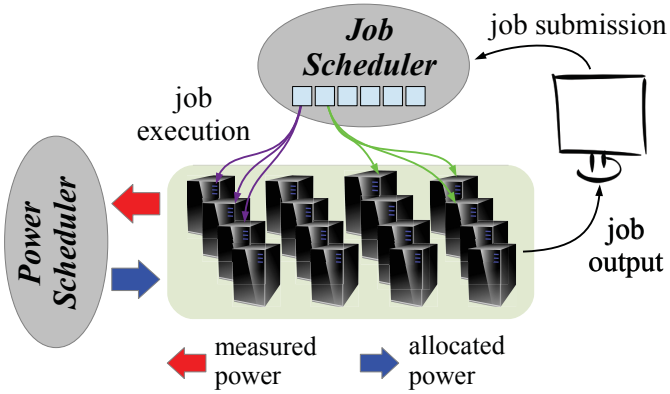

Figure 3: High-level model of system interactions

per node) run of the *miniFE* benchmark from the CORAL benchmarks with power bounds of 115 watts and 50 watts. A "smear" plot is shown where the power consumption of each socket is displayed. miniFE has a variable power consumption ("power signature") over its execution and between sockets. The 115 watts case is effectively unbounded. However, when power constraints are imposed, certain phases of the miniFE computation become power-limited (with less smearing), and other phases have enough power to operate at their full rate. Runtime duration, as a percentage of phase time, increases only for phases where unbound consumption would exceed the runtime cap. In the miniFE experiment, we observe the total time increasing by almost 5 seconds, mostly due to the second phase of its computation being power-constrained.

A polynomial impact to application runtime is observed as the power allocation is pushed further beneath the unbound consumption. Intuitively, power consumption is directly related to transistor switching power and the number of active transistors are directly related to the instruction stream. Generally the formula for the switching power loss is given as $W = \eta C V^2 f$ [2]; the watts W lost are directly related to the square of the voltage V and the frequency f of switching. The voltage and frequency must be increased together resulting in a nonlinear relationship between watts and instruction execution speed, dominated by the V^2 term. Figure 2 shows the runtime effects of decreasing allocation across several CORAL benchmarks.

3. SCHEDULING APPROACH

Our work targets large-scale high-performance computing (HPC) systems, primarily with an eye to future exascale platforms. HPC systems represent a substantial capital investment and are typically shared, batch-scheduled resources. An HPC system is composed of many compute *nodes*, each with a number of processing elements, including CPUs and accelerators. Users of the system typically submit *jobs* with a desired number of nodes to a job scheduler where each job is queued. The scheduler will schedule a job to run when an adequate number of nodes become available. We will call a subset of the nodes assigned to a job a *partition* or *enclave*, and will assume that any particular node is a member of only one enclave at a time.

The HPC environment is highly parallel and concurrent. User jobs are typically multi-node, highly-parallel applications and several jobs will run simultaneously on an HPC system. A job's start time is determined by node availability and a job's end time is based on the actual time to complete execution (or maximum time allocation) of the job. Although the HPC machine is *space-partitioned*, in that each job has its own processing resources, certain shared resources (e.g., network, file system, power) are used by concurrently executing jobs, potentially impacting the runtime behavior across jobs.

One of the major challenges in the move from current petascale to future exascale computation is increasing computational power within realistic electrical power consumption. The current approach of designing power systems to sustain peak power at all times, even though few jobs consume energy at that rate, is unrealistic. Hardware overprovisioning is likely the only way to achieve the increase in computing power while maintaining the power budget, however new approaches are required to distribute the available power and enforce that components stay within their assigned power limits. Exceeding the total system bound could physically damage the HPC cluster or the supporting power infrastructure.

We assume future hardware platforms will support an interface with properties similar to Intel's *Running Average Power Limit (RAPL)*. In current systems, components supporting RAPL can enforce a configurable maximum rate of energy consumption over a sliding temporal window. The particular techniques used to enforce the limit are selected and implemented completely by the hardware. The RAPL interface in our testbed uses *model-specific registers (MSRs)* (accessible via *libmsr* [8]) to allow software to interact with the hardware power management facilities.

A mechanism like RAPL alone is insufficient for running in an over-provisioned environment. RAPL only enables setting a hardware enforced power bound for individual components. A global power scheduler is needed to control the individual power bounds across components and ensure that the total sum of all bounds is below the total system bound.

Figure 3 shows a high-level view of the interaction between a potential power scheduler and an HPC cluster. The job scheduler is responsible for assigning jobs to hardware resources as well as starting and stopping the jobs. The power scheduler is solely responsible for analyzing power measure-

ments from the cluster and providing updated power allocations to all cluster components. The HPC cluster itself is primarily concerned with executing jobs from the scheduler, but also provides the integrated infrastructure for power measurement and control used by the power scheduler.

3.1 Power Model

The system-wide power scheduler has the primary objective of enforcing a global power limit, L. We can think of the HPC system as having an infinite amount of energy, but having a global maximum limit to the instantaneous rate at which energy can be used. Power-optimization and energy-aware techniques reduce the energy consumed [1, 13, 12], often by reducing the power while maintaining the runtime, allowing more of the hardware over-provisioned system to be used concurrently. These techniques do not provide a guarantee that the global rate of energy consumption remains within a fixed bound. Reduced energy consumption and optimal runtimes are secondary objectives for a power scheduler charged with enforcing the global power limit in a hardware over-provisioned system.

A global power limit L is set by facility limitations or administrative policy to protect the power infrastructure from damage due to exceeding capacity. A system is modeled as a set of n sockets. Socket power allocations above the maximum possible power consumption, A_{max}, are pointless and power allocations below a manufacture specified minimum, A_{min}, cannot be reliably enforced by the hardware. Every socket i has a power consumption, c_i, and a power allocation, a_i. The delta between c_i and a_i is the *wasted* allocation and will be noted as w_i. It is assumed that the hardware enforces $c_i \leq a_i$ or equivalently $a_i = c_i + w_i$ with $0 \leq w_i$. Thus, the total power allocated to the system is $\sum a_i$ and the total power consumption is $\sum c_i$. Further, due to the hardware enforcement, $\sum c_i \leq \sum a_i$.

An application's runtime is roughly the same for any a_i such that $a_i > c_i$. Runtime should only be impacted when a_i is less than the amount an application would consume if there was no power bound. This conclusion is consistent with Fukazawa et al. [4] and our own experiments.

3.2 Static Scheduling

A *static* power scheduler makes a decision about how to schedule power prior to the job launch. A naïve scheduling strategy would be to allocate an equal amount of power to each socket, $a_i = \frac{L}{n}$, over the lifetime of the machine. Since $\sum c_i \leq \sum a_i$, trivially this strategy maintains $\sum c_i \leq L$. Two existing systems at the Lawrence Livermore National Lab (LLNL) use this strategy presently. While meeting the technical requirement of enforcing a global power bound, the naïve static strategy is expected to under perform.

A more refined static power scheduler could attempt some optimization of power distribution if it is aware at scheduling time of an application's expected power consumption. Rather than allocating an equal amount of power to each socket, the static scheduler could allocate an equal amount of wasted allocation, w_i, to each socket. The allocation per socket for such a scheduler can be computed using $a_i = c_i + w_{avg}$ where $w_{avg} = \frac{1}{n}(L - \sum c_i)$.

For the more refined static approach, the scheduler must know *a priori* the corresponding c_i and w_{avg} values across the system. The behavior of a job can change based on the parameters used for execution and there is also an expectation of greater uncertainty in behavior as systems are scaled due to increasing runtime complexities and interactions with other jobs. For long lived clusters, where numerous jobs of various sizes asynchronously enter and exit the system, $\sum c_i$ across the system is expected to vary greatly over time as jobs enter and leave the system. Even within a single job, different phases may consume energy at different rates. Knowledge of per socket power consumption in advance of execution is therefore not feasible in the general case.

3.3 Dynamic Scheduling

Static power scheduling at job launch time cannot maintain w_{avg} across the full machine in the presence of dynamic job power consumption and missing knowledge of future jobs. A dynamic approach to power scheduling is likely required to respond to the dynamic power consumption observed at runtime. Rather than attempting to set a_i once at job start time, a dynamic scheduler can periodically adjust any a_i in the system, even when there is an active job running on the socket.

Extending the model to include time, the scheduler must guarantee for all times t that $\sum c_i^t \leq L$. A basic dynamic scheduler strategy may assume that the power consumption of a running job remains fairly consistent over time, represented by the heuristic $c_i^t \approx c_i^{t-1}$. At time t, the scheduler can know the values c_i^{t-1} and a_i^{t-1}, as reported by the socket, as well as L. The updated per socket allocation can be computed as $a_i^t = c_i^{t-1} + w_{avg}^{t-1}$.

Using the formulation above, a dynamic power scheduler can maintain w_{avg} without any control of the job scheduling. If the scheduler is able to maintain $w_{avg} > 0$ then all applications are expected to complete with their unbounded runtime since runtime is not degraded when $a_i^t > c_i^t$. The power scheduler only requires c_i^t and a_i^t for all sockets as input to set all a_i^{t+1} during runtime.

Up to now, there has been an assumption that there is sufficient power to run all scheduled jobs at the optimal power consumption, $c_i^t < a_i^t$ for all i and t. This assumption requires a job scheduler that is guaranteed to never oversubscribe power. Due to the challenges discussed for static power scheduling, requiring the job scheduler to produce a schedule that never oversubscribes power and can consume the full system wide power allocation is not practical.

A power reading where $c_i^t = a_i^t$ could indicate that the power is set to exactly what the application using the socket can consume. Alternatively, $c_i^t = a_i^t$ could indicate that a_i^t was too low and that the hardware reduced consumption on the socket, degrading application performance. Such sockets could potentially benefit from additional power allocations to them.

The responsiveness of a dynamic power scheduler to increased consumption, using the formulation in this section, is expected to be impacted by both the scheduling interval and the per socket wasted power allocation due to the assumption $c_i^t \approx c_i^{t+1}$ and hardware enforcement of $c_i^t \leq a_i^t$.

4. DESIGN AND IMPLEMENTATION

To evaluate the power scheduler approach discussed above, we developed a trio of tools: POWsched, POWmon, and POWsim. The first two implement the dynamic power scheduling apparatus on a real HPC platform that is outfitted with power control utilities. The third enables us to assess sched-

uler dynamics across a greater scope of application power behaviors and system scales.

4.1 POWsched

POWsched is a *dynamic* power scheduler based on the model and approach discussed above. Scheduling decisions in POWsched are made per socket and are completely agnostic with respect to job, enclave, and node. POWsched maintains a system-wide power bound without job scheduler coordination using only per socket observed power consumption to guide power scheduling across a cluster.

Pseudocode for the scheduler is provided in Algorithm 1. The scheduling task is performed in three phases during each scheduling interval. Our experiments use a 1 second interval. In Phase 1, POWsched collects recent consumption readings from all sockets. In Phase 2, power is greedily recovered from the existing allocations for later distribution. In Phase 3, additional power is given to sockets that may be able to use the power. At the end of Phase 3, POWsched sleeps the remainder of the scheduling interval.

Separation of power allocation into two phases is needed to guarantee that the system wide power limit is never exceeded due to communication delays. Recall that $a_i^t \leq L$ must be maintained for RAPL to successfully enforce $c_i^t \leq L$. Assume $a_0^t + a_1^t = L$. If the scheduler computes $a_0^t > a_0^{t+1}$ and $a_1^t < a_1^{t+1}$ and sends a_0^{t+1} and a_1^{t+1} at the same time, communication delays might cause socket 1 to update the allocation before socket 0. For a short interval the allocated power will be $a_0^t + a_1^{t+1} > L$, which is a violation of the system power bound. POWsched must be certain that all sockets receiving a lower allocation have been updated before any sockets receiving a higher allocation are updated.

POWsched does not compute w_{avg}. A target w_i is used to account for the measurement jitter and to greedily reclaim power from under consuming sockets. POWsched assumes the system is oversubscribed and steals a percentage of the allocation for each socket allocated more than the system wide average per socket allocation ($a_i > \frac{L}{n}$) when no power is yielded and very little surplus power is available. When adjusting allocations up, POWsched divides the surplus power evenly across the sockets consuming near their current allocation. When power is abundant, the allocation up behavior is expected to result in a lot of wasted power that can then be greedily collected in the next scheduling interval. When power is scarce, the allocation up and power stealing behavior will eventually converge at a fair allocation across all sockets.

For a homogeneous system, like cab, an equal percentage for all components is a reasonable strategy. Without access to heterogenous systems that support hardware enforced power capping on the accelerators in addition to the CPUs, we were unable to experimentally explore the applicability of our approach for heterogenous systems. We believe the basic approach should be applicable. In future work we would like to explore wether weighting by component may be advantageous.

We implemented POWsched in C using libmsr to access the RAPL MSRs and MPI for collective communication. POWsched is deployed as a separate MPI job, co-resident with the actual workload[3]. Fault tolorence is not explored in the current work as most MPI implementations have extremely limited support for fault tolorence. In future systems, power scheduling will need to be fault tolorent and will likely be provided as part of the system stack by a global operating system.

[3]A workload consists of several concurrent jobs in our experiments.

Algorithm 1 POWsched logic in pseudocode

```
q ← target w_i
C stores {c_0, ··· , c_{n−1}}
A stores {a_0, ··· , a_{n−1}}
M stores {m_0, ··· , m_{n−1}}
numdown ← count of nodes yielding power
interval ← scheduling interval
reclaimfactor ← power to reserve when stealing

procedure MAIN
    while True do
        getReadings                                   ▷ Phase 1
        allocDown                                     ▷ Phase 2
        allocUp                                       ▷ Phase 3
        sleep rest of interval
    end while
end procedure

procedure getReadings
    for all sockets do
        Update c_i with the current reading
    end for
end procedure

procedure allocDown
    numdown ← 0
    for all sockets do
        if c_i < a_i − q then
            Update a_i to max{c_i + q, A_min}
            numdown ← numdown + 1
            Update m_i to False
        else
            Update m_i to True
        end if
    end for
    if numdown= 0 and Σ a_i + n ≥ L then
        for all sockets do
            if a_i > L/n then
                a_i ← a_i − (a_i − L/n) × (1−reclaimfactor)
                m_i ← True
            end if
        end for
    end if
    for all sockets do
        Set the socket to limit a_i
    end for
end procedure

procedure allocUp
    u ← (L−Σ a_i) / (n−numdown)
    for all sockets do
        if m_i then
            a_i ← min{a_i + u, A_max}
        end if
    end for
    for all sockets do
        Set the socket to limit a_i
    end for
end procedure
```

4.2 POWmon

Monitoring power allocation and consumption is done with *POWmon*. POWmon is run as a transparent wrapper around another process on the monitored node and terminates just

App	Nodes	App Only	+POWmon	+POWsched @115W	+POWsched @dyn	≈ Overhead
LU	16	119.77	119.84	120.99	121.25	0.01
LU	4	112.39	112.92	112.05	113.30	0.00
CoMD	16	107.1491	105.3836	107.3378	107.0001	0.00
CoMD	8	109.3181	109.2498	109.9474	110.1558	0.01
CoMD	4	92.4329	91.9755	92.2450	92.7113	0.00
AMG	16	102.573688	103.323772	103.71112	103.71112	0.00
AMG	8	88.667316	88.173036	89.631203	90.110953	0.01
AMG	4	76.821048	76.763169	77.002957	76.873345	0.00

Table 1: Runtimes reported by the workloads. POWsched @115W run forces POWsched to assign 115W per socket over the lifetime of the job. POWsched @dyn allows POWsched to dynamically adjust the per socket allocation with a global bound permitting 115W per socket.

after the wrapped process terminates. Since POWmon writes summary data, it is important that the wrapped process terminate at the end of the monitoring interval and before the job scheduler begins killing processes. The wrapped process can be anything the OS will treat as an executable, including a shell script. To support wrapping MPI applications in which multiple process will be started per node, POWmon uses a shared memory segment to select only one monitor instance per node to record measurements.

POWmon has a 100 millisecond measurement interval and 1 millisecond time resolution. Operating with millisecond resolution can cause some inaccuracy when converting between quantities to rates (such as the conversion between joules and watts), but such inaccuracy is sufficiently small. Monitor sleeps are scheduled based on offsets from the monitor start time. Therefore, errors due to short and long intervals should average out over the lifetime of the run.

4.3 POWsim

Large-scale experimental evaluation of POWsched is complicated by the limited number of existing HPC systems supporting dynamic adjustment of per socket power allocation at runtime. To aid in evaluation of POWsched at scale, we develop *POWsim* to simulate the effects of power bounding on applications running on power-adjustable system configurations that are not presently available. POWsim uses the following model:

POWsim estimates program progress by an instruction measure, I, and tracks energy consumed, in joules E, by applications running on sockets with RAPL-like power capping. The intuition behind the simulation model is that power consumption is directly related to transistor switching power and the number of active transistors are directly related to the instruction stream. The instruction measure is not tight, but captures the dominant behavior observed in our experiments involving runtime under bound.

The amount of instruction work that can be done by the socket, I_s, and energy allocated to the socket, E_s, are estimated by the following formula:

$$I_s = \int_s^e \sqrt{\frac{b - S_-}{S_+ - S_-}}\mathrm{d}t \quad , \quad E_s = \int_s^e b\mathrm{d}t$$

where s is the interval start time, e is the interval end time, b is the socket power bound, S_- is the "idle" socket power consumption, and S_+ is the maximum socket power consumption. E_s is in joules and the integration is natural based on the definition: watts $= \frac{\text{joules}}{\text{seconds}}$. I_s is a measure of the instruction work the socket is capable of over the interval. Execution of a specific hardware instruction with specific input requires a specific number of transistor state transitions. The expression is based on transistor switching power being related to the frequency and square of the voltage. The S_- term captures the power consumed by socket work not related to application progress. The S_+ term captures the power consumed by the socket when the maximum number of transistors are active per unit time.

Applications are modeled as a function from time to instantaneous watts consumed, $w_{(t)}$, when the socket power is unconstrained. We think of a particular run of an application as being a finite ordered sequence of hardware instructions, the execution of each instruction resulting in a specific number of transistor state transitions. The runtime of an application is the time taken to execute the complete sequence:

$$I_p = \int_s^e \sqrt{\frac{w_{(t)} - S_-}{S_+ - S_-}}\mathrm{d}t \quad , \quad E_p = \int_s^e w_{(t)}\mathrm{d}t$$

Over an interval s to e, a computation can be *power bound*, $I_s < I_p$, or *program bound*, $I_s > I_p$. A program bound interval is one in which the instruction stream of the program does not require more switching than the process can support over the interval. A power bound interval indicates that the program can induce more transistor state changes than the process can complete over the interval.

In the simulator, an instance of an application is represented by the application's function, $w_{(t)}$, and the current application time, t_p. Advancing the simulation involves updating the current application time, t_p, for all active applications on the simulated cluster. For simulation intervals in which the application is program bound, t_p, is updated based on the simulation time interval. For simulation steps in which the application is power bound, updating t_p is slightly more complex since the simulation time interval must be converted into a the application time interval, in the program's frame of reference, based on the progress effect of the bound.

For power bound steps, the advancement of t_p is computed by solving for e in the following equation:

$$I_s = \int_{t_p}^e \sqrt{\frac{w_{(t)} - S_-}{S_+ - S_-}}\mathrm{d}t$$

The limiting factor on application progress, due to the power bound, is the amount of instruction work to be done in the socket. I_s provides a measure for the amount of the instruction work the socket is capable of over the simulation

interval. The solution gives the end time, e, in the program's frame of reference, to complete the instruction work done by the socket over the simulation interval.

Presently, POWsim does not model the effects of application communication behavior or scheduling latency due to communication and computation delays. Runtime synchronization of simulated applications currently is a side-effect of simulation determinism. As future work, we plan to enhance the application model with a communication function that would provide a capability to model network effects. Scheduling latencies are also future work for POWsim.

4.4 POW Overhead

In our experiments, POWmon and POWsched appear to interfere negligibly with other applications. Table 1 provides the runtimes, as reported by three CORAL benchmarks, for invocations on 2, 4, 8, and 16 nodes with different modes of monitoring and scheduling enabled. Runtimes were perturbed by less than 0.1% compared to application execution without POWmon and POWsched.

5. RESULTS

We conducted a series of experiments with POWsched on a live HPC system at Lawrence Livermore National Laboratory (LLNL). Results from these experiments are presented in this section first. Next, results from a scaling study POWsched's computation and communication costs on an IBM BG/Q system are presented. The section concludes with results from our simulator.

5.1 Live Power Scheduling Experiments

Our live power scheduling experiments took place on the Cab cluster at LLNL[4]. Most of the experiments used 128 Cab nodes. Logically, we think of the 128 nodes being partitioned into 8 enclaves, each containing 16 nodes (32 sockets with 8 cores each). During each experiment, all enclaves will run simultaneously and each enclave will run a workload of two benchmark apps in sequence, with a 10 second sleep between benchmark apps. Workloads of this form are chosen to ensure a window of unevenness in the maximum power consumption, per node, during the experiment run. The sleep also simulates the window of time expected between completion of one job and the system job scheduler starting another job on the nodes. Workloads with fixed node counts per workload are used rather than individual jobs due to complexities of running and tracking concurrent subjobs in existing job schedulers.

Figures 4, 5 and Table 2 use workloads with 3 application benchmarks (AMG, LULESH, and CoMD). For experiment control, we ran each workload with each socket receiving the maximum power allocation, 115 watts. 115 watts is expected to result in the shortest possible runtime. We also ran experiments where each socket received a specific power allocation (90 watts, 70 watts, and 50 watts), simulating the naïve static power scheduler. The static runs provide a baseline for comparison between POWsched and a system where each active socket is given an equal static allocation based on the global power available (e.g., a system has a

[4]Cab is one of only a few HPC clusters with support for per socket power capping via user space code. Additional information on Cab can be found at http://computation.llnl.gov/computers/cab and on libmsr at https://github.com/scalability-llnl/libmsr.

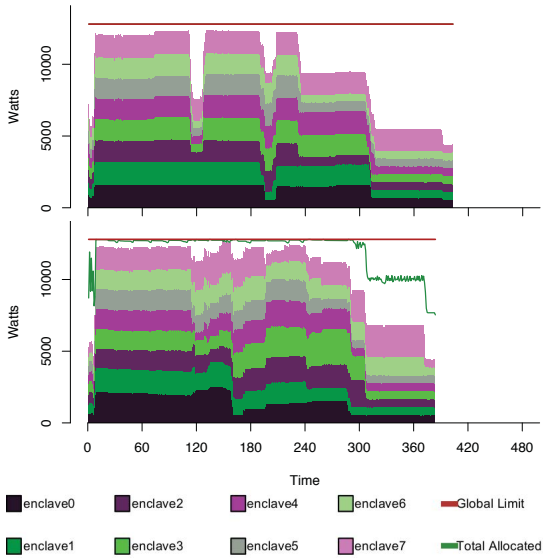

Figure 4: The enclave consumption and global bound for 50 watt forced and central runs. Above 50 Watts Forced. Below 50 Watts Central.

global power bound of 17,920 watts and 256 active sockets, resulting in an average allocation of 70 watts per socket). The runs with a fixed per socket allocation will be referred to as *static*[5]. The experimental runs using POWsched will be referred to as *dynamic*[6] and rely on the same global power bound as the corresponding static run.

Figure 4 shows total power allocation across the 8 enclaves for a 50 watt average bound using static and dynamic scheduling. Table 2 shows the runtime impact of POWsched over 10 runs at each bound with outliers removed. These results use a scheduler interval and RAPL window of 1 second. The time to complete all workloads with POWsched, when power is constrained, is better than the static schedule by more than one deviation. We also note from Table 2 that POWsched clearly is not attempting energy optimization. In all cases roughly 4 megajoules are used to complete the workloads, the primary effect of POWsched is on runtime relative to static required to complete all workloads.

Figure 5 shows per enclave allocation and consumption for the corresponding 50 watt runs, comparing static and dynamic. What is interesting to see is the dynamic spreading of power to workload applications that can use it, some of which are consuming significantly above the 1,600 watts per enclave (32 sockets per enclave) constraint used by the static allocation.

Unallocated, or idle, power is present as side-effect of the greedy reclamation strategy and can be seen in Figure 4 as the space between the total allocated power and the global limit. There is no idle power in the static strategy since the full power limit is allocated across all sockets at all times. We can imagine several co-located clusters sharing a power infrastructure and power schedulers coordinating via some set of policies to reallocate idle power in one system to other

[5]The scheduler statically allocates a particular power setting to all sockets.

[6]The scheduler dynamically adjusts the power settings during execution.

Experiment	Runtime	Stddev	Improvement	kj Alloc	Stddev	kj Used	Stddev
115W static	278.26	9.57		8191.85	281.75	4007.80	97.99
115W dynamic	276.24	4.84	0.7%	5474.75	52.56	3977.02	36.74
90W static	284.63	3.20		6571.76	72.75	3984.68	30.32
90W dynamic	277.13	5.04	2.6%	5339.11	66.21	3979.78	47.356
70W static	323.83	4.90		5829.02	86.82	3904.29	34.08
70W dynamic	278.02	4.97	14.1%	4638.32	68.91	3984.80	37.77
50W static	401.76	5.47		5178.29	72.59	3937.65	37.52
50W dynamic	371.92	13.23	8.7%	4562.48	124.44	4015.64	79.36

Table 2: 128 nodes, 16 nodes workloads per workload, 10 runs, same workload for all runs.

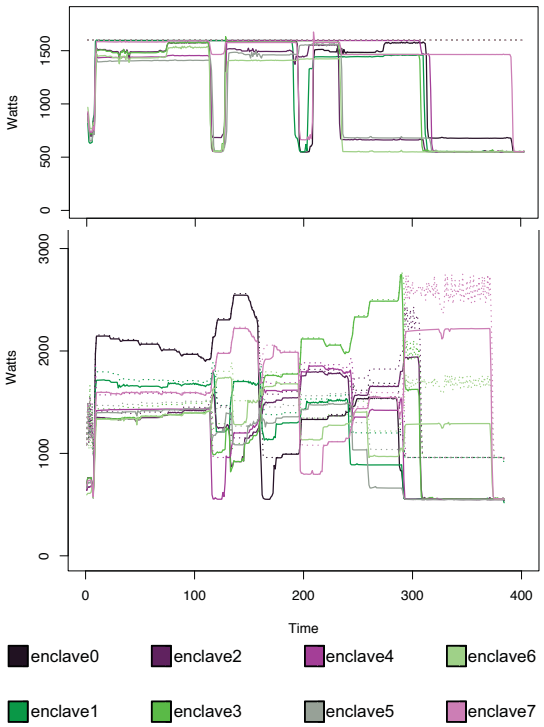

Figure 5: Consumption (solid) and allocation (dotted) over time for two workload placements at 50 watts. Above 50 watts static. Below 50 watts dynamic.

power bound systems. In such a scenario, the global power limit is also a dynamic policy-driven value. Similarly, if needed for scaling, we can imagine a hierarchy of power schedulers where the leaf schedulers control sets of nodes and the interior schedulers control sets of enclaves.

Additional experiments have been conducted at 128 and 256 nodes with more diverse workloads. In these experiments a workload uses 8 nodes and completes 4 random benchmarks with a short random length sleep between benchmarks. Table 3 lists the benchmark apps used[7]. The same workloads are used for each power limit[8]. Table 4 shows the results.

[7]Due to limitations on node over-subscription in the job scheduler, we were unable to launch one MPI process per core for benchmarks that have best performance with MPI only, versus MPI+OpenMP parallelism

[8]Due to limited machine time these experiments were not able to be repeated to generate distributions and timing; only a single run of each workload is reported

Benchmark	Domain	Processes
LULESH	Shock Hydro	27
miniFE	Finite Element	8
miniFE	Finite Element	64
AMG	Linear Solver	32
AMG	Linear Solver	64
MCB	Monte Carlo	32
CoMD	Molecular Dynamics	32
Nekbone	Science App	8

Table 3: Benchmarks used for 8 node workloads in the 128 and 256 node experiments.

	128 Nodes			256 Nodes		
Bound	Forced	Central	%	Forced	Central	%
115W	640.98	648.80	-1.2	717.53	729.82	-1.7
90W	650.98	645.70	0.8	648.69	656.54	-1.2
70W	687.47	693.11	-0.8	717.99	700.35	2.5
50W	821.69	828.24	-0.8	826.45		

Table 4: Times for experimental runs of random workloads with and without POWsched. Due to an MPI Abort in one of the jobs, the 256 node 50W experiment was incomparable.

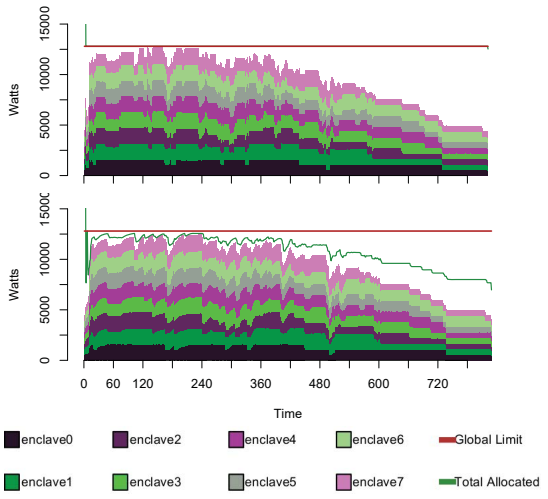

Figure 6: Each enclave contains two concurrently executing 8 node workloads. Above 50 watts static. Below 50 watts dynamic.

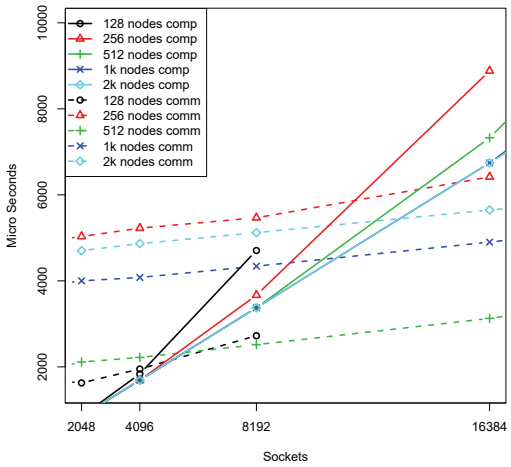

Figure 7: Observed time for communication and computation by simulated socket count.

The outcome from the 128-node experiment indicates that the time to complete all workloads using POWsched is roughly the same as the static power scheduler. We attributed this to a lack of magnitude and diversity in *power consumption intensity* in the randomly generated workloads. Figure 6 highlights this by showing the 50 watt case in which POWsched has idle power for much of the run, in contrast to Figure 4 where POWsched is able to productively assign all of the power available for the first two minutes of execution.

Performance improvement from POWsched over static requires uneven power demand across the system, work for which an evenly set power limit is insufficient for subset of the sockets, and a sufficient global power limit for all jobs in aggregate. If power is plentiful, POWsched is not needed, though the overheads are only slight if POWsched is enabled. If all applications need more than the static allocation, POWsched will attempt to converge to the static allocation but during the convergence period will iteratively alter power allocations, perturbing overall runtime non-uniformly. In the initial experiments, workloads were constructed as permutations of two benchmark applications run in serial. The benchmark settings were such that LULESH was the highest power consumer (around 90 watts per socket) and the other benchmarks were significantly less (around 50 watts per socket). The mix provided ample power to reallocate at 90 and 70 watts and some power for reallocation at 50 watts.

5.2 Scaling Experiment

Few large HPC platforms exist for experimenting with dynamic hardware enforced power bounding. However, we still desired to get a sense for scaling of the POWsched algorithm. For this purpose, we deployed POWsched on the Vulcan IBM BG/Q platform at LLNL[9] and measured the time spent in POWsched communication and computation. Since the BG/Q platform does not support RAPL, we used random numbers for consumption read. Use of random numbers should not disrupt the results since the per node time

[9]Additional information on Vulcan can be found at http://computation.llnl.gov/computers/vulcan.

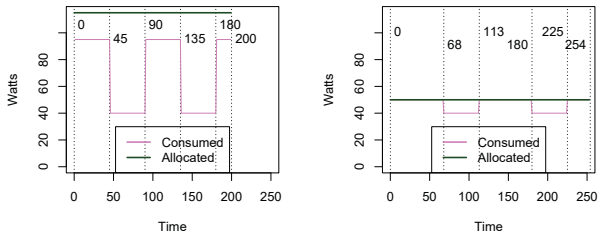

Figure 8: Simulation output showing dilation effects similar to Figure 1. Left 115 watts, right 50 watts.

to read from or write to the RAPL registers should remain constant[10]. The performance of POWsched at scale will be dominated by the time taken to communicate the per socket readings or the time taken to perform computation over the socket readings.

Each scheduler process launched represents a simulated node with 2 sockets and for each run we use 1, 2, 4, 8, 16, 32, or 64 processes per physical BG/Q node. Overall, we use BG/Q node counts from 1 to 8k, allowing us to sweep the space from a single simulated node to 500k simulated nodes. We observed linear scaling for computation and slowly growing communication cost. Linear scaling for computation is expected due to the linear scans conducted by the dynamic scheduler each interval. BG/Q's optimized network for low-latency and high-bandwidth MPI collectives results in slow all-gather communication time growth. Figure 7 shows the cross over region between computation and communication being the dominant time cost. Even at the largest number of simulated nodes, 512k, scheduling communication and computation completes in under 400ms. Depending on system scale and network performance, dynamic centralized power scheduling may be viable.

5.3 Simulation Experiment

POWsim allows exploration of the interplay between application runtime and power bound that is difficult on real systems due to scale and nondeterminism. Our simulated job power functions, $w_{(t)}$, in these experiments are a constant, square wave, or saw tooth. Figure 8 shows the simulation behavior for an application similar to the miniFE execution shown in Figure 1; we note that the runtime penalty for power bound phases using the simulator is greater than those we have experimentally observed.

Figure 9 shows power consumption and allocation for 4 concurrent jobs using dynamic and static schedulers. All jobs use the same amount of power, but the alignment of the job consumption results in very different behavior. When jobs have time-synchronized and mirrored rates of change in power consumption, POWsched has the best opportunity for power reallocation and throughput improvement. If the jobs are time-synchronized, but with identical rates of change in power consumption, POWsched can not produce any throughput improvement. If the rate of change in power

[10]Measurements we made on Cab show an average of 44 and 15 microseconds are required to read and set the RAPL registers, respectively, via libmsr.

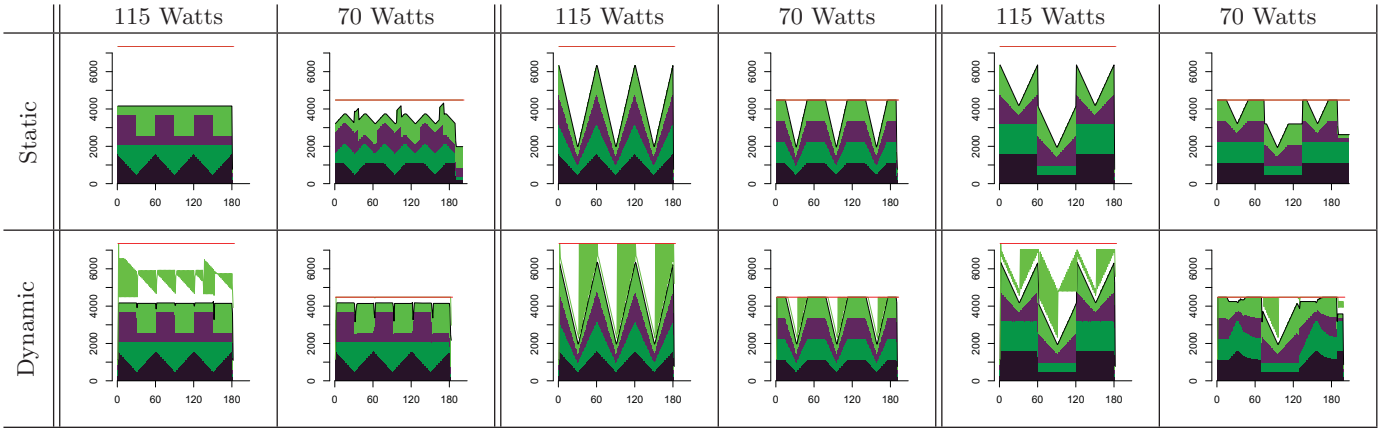

Figure 9: Simulated power allocation and consumption for 4 concurrent jobs. From left to right: best, worst, and improvable.

Parameter	Min	Max
Job Nodes	8	$\frac{\text{clustersize}}{4}$
Job Runtime (seconds)	10	6000
Phase 1 Watts	85	115
Phase 2 Watts	30	115
Phase Period	30	600

Table 5: Simulation parameters

Nodes	Bound	Static	Dynamic	%
1k	115W	40567	40581	-0.0
	70W	44541	43287	2.8
	50W	53249	54955	-3.2
8k	115W	43801	43825	-0.0
	70W	51652	51081	1.1
	50W	63085	65545	-3.9
16k	115W	44414	44429	-0.0
	70W	52656	51873	1.5
	50W	64097	66054	-3.1

Table 6: Simulated runtime with random workloads.

consumption is not identical, POWsched can potentially improve runtime performance.

Using POWsim we simulate the use of POWsched on clusters of 1k, 8k, and 16k nodes. For each node count a random mix of 100 jobs is generated, the same mix is used for each run at that particular node count. Each job is one of the three simulated functions with a random runtime, period, and consumption. Table 5 shows the parameter ranges used. All jobs in the run are queued using FIFO job scheduler before the first time step executes. Table 6 shows the time taken to complete all queued jobs. Simulation results are consistent with the experimental results on random workloads – little effect at 115 watts, improvement around 70 watts, and reduced performance at 50 watts when power is overly constrained.

6. CONCLUSION

The work contributed by this paper is significant for future power limited systems. We have demonstrated the first system-wide dynamic power scheduler that enforces a global power limit on an HPC system with opportunistic power reallocation to improve performance. Decisions to reallocate power across individual sockets are based only on the relation between power allocation and consumption, per socket, across the HPC system. We have shown that power can be allocated efficiently and that workload performance can be improved, compared to static fixed power allocation. When sufficient power is available, POWsched can increase throughput (up to 14% in our experiments) by redistributing waste power. We have also provided a node-level performance-monitor with high temporal resolution and negligible performance impact. In support of future work on large scale power scheduling, we have provided a model for simulating the effects of power bounds on application runtime and implemented a simulator using the model.

While the current RAPL and libmsr technology provide adequate capabilities for the development of socket level power monitoring and control, RAPL is not widely available to allow large-scale power scheduling experiments. The technology and its deployment will likely improve in the next several years. This will allow our techniques to be integrated more broadly and at greater scale. There are also opportunities for improved power scheduling through the integration with job schedulers and incorporation of application-specific knowledge of power consumption. With improved power measurement and control for accelerator and manycore devices, our work can be extended to heterogeneous systems for both power monitoring and scheduling.

7. ACKNOWLEDGMENTS

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-675184). Work by the University of Oregon is supported by the DOE Office of Science, through a Sub-Contract No. 3F-32643 from the University of Chicago, Argonne, LLC (as operator of Argonne National Laboratory), under Prime Contract No. DE-AC02-06CH11357.

8. REFERENCES

[1] M. Bambagini, M. Bertogna, M. Marinoni, and G. Buttazzo. An energy-aware algorithm exploiting limited preemptive scheduling under fixed priorities.

In *8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 3–12. IEEE, 2013.

[2] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho. The energy/frequency convexity rule: Modeling and experimental validation on mobile devices. In *Parallel Processing and Applied Mathematics*, pages 793–803. Springer, 2014.

[3] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz. Pow: System-wide dynamic reallocation of limited power in hpc. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 145–148, New York, NY, USA, 2015. ACM.

[4] K. Fukazawa, M. Ueda, M. Aoyagi, T. Tsuhata, K. Yoshida, A. Uehara, M. Kuze, Y. Inadomi, and K. Inoue. Power consumption evaluation of an mhd simulation with cpu power capping. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 612–617. IEEE, 2014.

[5] H. Hoffmann. Racing and pacing to idle: an evaluation of heuristics for energy-aware resource allocation. In *Workshop on Power-Aware Computing and Systems*, page 13. ACM, 2013.

[6] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. A generalized software framework for accurate and efficient management of performance goals. In *International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2013.

[7] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 62–73. IEEE, 2010.

[8] L. L. N. S. LLC. libmsr. https://github.com/scalability-llnl/libmsr.

[9] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *27th ACM International Conference on Supercomputing*, pages 173–182. ACM, 2013.

[10] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. de Supinski. Practical resource management in power-constrained, high performance computing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 121–132, New York, NY, USA, 2015. ACM.

[11] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond dvfs: A first look at performance under a hardware-enforced power bound. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW),*, pages 947–953. IEEE, 2012.

[12] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: making dvs practical for complex hpc applications. In *23rd ACM International Conference on Supercomputing*, pages 460–469. ACM, 2009.

[13] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A. Snavely. Green queue: Customized large-scale clock frequency scaling. In *Second International Conference on Cloud and Green Computing (CGC)*, pages 260–267. IEEE, 2012.