Performance & Energy Tradeoffs for Dependent Distributed Applications Under System-wide Power Caps

Huazhe Zhang
University of Chicago
Chicago, Illinois
huazhe@cs.uchicago.edu

Henry Hoffmann
University of Chicago
Chicago, Illinois
hankhoffmann@cs.uchicago.edu

ABSTRACT

Large scale parallel machines are subject to system-wide power budgets (or caps). As these machines grow in capacity, they can concurrently execute dependent applications that were previously processed serially. Such application *coupling* saves IO and time as the applications now communicate at runtime instead of through disk. Such coupled applications are predicted to be a major workload for future *exascale* supercomputers; *e.g.,* scientific simulations will execute concurrently with *in situ* analysis. While support for system-wide power caps has been widely studied, prior work does not consider the impact on coupled applications.

We study techniques for maximizing coupled application performance under a system-wide power cap and implement them on a 26-node cluster. We compare to SLURM, a state-of-the-art job scheduler that considers power, but not coupling. The proposed techniques increase mean performance over SLURM by 7–14%. Unlike existing approaches, the proposed techniques also recognize when it is not possible to increase performance and, instead, reduce energy, achieving 18% energy reduction for a 5% performance loss. Finally, the dynamic techniques are resilient to tail behavior and system noise, improving performance in noisy environments by 30–36%.

ACM Reference Format:
Huazhe Zhang and Henry Hoffmann. 2018. Performance & Energy Tradeoffs for Dependent Distributed Applications Under System-wide Power Caps. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3225058.3225098

1 INTRODUCTION

All computer systems are limited by power consumption. Next generation *exascale* supercomputers are predicted to have a strict operating budget of approximately 20 MW, but the total power dissipation at full utilization would far exceed this budget [5]. These systems require sophisticated, distributed *power-capping* mechanisms to assure their power budget will not be exceeded; the United States Department of Energy (DoE) has therefore declared power management a key challenge for exascale [52].

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225098

While power concerns create new problems, exascale's increased capacity creates new opportunities. Specifically, instead of sequentially running dependent jobs that communicate through disk, the size of exascale supercomputers allows these jobs to be *coupled* [2, 5, 9, 29, 52]. That is, two formerly independent jobs can now be run simultaneously and communicate at runtime. For example, scientific simulations can now be run with *in situ* data analysis or visualization providing scientists the insight needed to alter the simulation as it runs [2, 9]. Additionally, separately developed physics simulations can now be run together, sharing their results to provide much greater fidelity [29]. In fact, the DoE has declared resource management for coupled application workloads an additional challenge for exascale [5, 52]. Unfortunately, existing work on enforcing power caps across large scale systems does not account for coupled applications.

For example, several approaches increase overall system performance with unbalanced power allocations [26, 27, 49]. Other approaches shift power as independent applications transition through compute and IO phases [48]. While unbalanced allocation and phase-adaptation are important for coupled applications they are not sufficient. In brief, coupled applications obey the basic principles of pipeline parallelism: the couple's speed is determined by the slowest application. Thus, *optimizing coupled application performance under a power cap often requires slowing down the faster application to shift more power to the slower.* This requirement to slowdown a fast application is unique to coupled applications and is not supported by prior power shifting approaches, which consider only one application at a time [26, 47], or optimize for multiple, independent applications [27, 48, 49]. Furthermore, due to the couple's interaction, sometimes it is not possible to increase performance through any amount of power shifting. Here, the couple scheduler should reduce total power usage to save energy.

We address the unique challenges of coupled applications with *PowerShift*, a family of three techniques for shifting power between dependent applications in a distributed system. All three respect the system-wide power cap and maximize the couple's performance by shifting power from the faster application to the slower one, until both run at the same rate. All techniques rely on existing node-local power cap enforcement (*e.g.,* [30, 31, 44, 56]).

PowerShift-S is a static technique that sets power caps based on individual application profiles. *PowerShift-C* is a *centralized dynamic approach* in which a single decision maker dynamically shifts power. *PowerShift-D* is a *distributed dynamic approach* where nodes put surplus power into a shared pool, and nodes that need more power take from the pool. These techniques provide different tradeoffs in overhead, flexibility, and adaptability.

Figure 1: Performance/power for `cluster` and `pigz`.

We implement *PowerShift* on a real 26 node system. We compare SLURM, a state-of-the-art, power-aware job scheduler [49] and *PowerShift* to a *fair* approach that evenly divides power among all nodes. Compared to *fair* we find that:

- *SLURM* improves mean performance by 3%.
- *PowerShift-S* improves mean performance by 10%.
- *PowerShift-C* improves mean performance by 17%.
- *PowerShift-D* improves mean performance by 15%.

Additional results demonstrate that *PowerShift* detects when no performance improvement is possible and instead reduces energy. Furthermore, *PowerShift D & C* adapt to system noise and tail behavior, automatically shifting power to nodes that have unexpected extra load or long tails, improving performance by 36% and 30%, respectively. Finally, we show that *PowerShift* is *topology-oblivious* and works equally well if the coupled applications are physically separate or mapped to different power domains on the same node.

Critically, *PowerShift* does not measure performance or require any application instrumentation. Instead, it requires some offline profiling and then observes only applications power usage. Thus, *PowerShift*, like SLURM, works with unmodified applications. *PowerShift is the first work to examine the effects of system-wide power capping on the performance of coupled applications, two key challenges identified by the DoE for exascale* [2, 5, 9, 29]. [1]

2 MOTIVATIONAL EXAMPLE

To highlight the challenges of maximizing coupled jobs' performance under a power cap we run the `cluster` scientific simulation (from the Gadget 2.0 suite) as a *frontend* job coupled with `pigz`, a parallel version of gzip, as a *backend* job. We compare 3 approaches: *Fair*, *PowerShift-S*, *PowerShift-C*. The *Fair* approach allocates equal power to all nodes so that the cluster power cap is respected.

We test on a 26-node cluster. Each node is a dual-socket with Xeon E5-2670 v3 processors, 256GB of RAM and 10 GigE Ethernet NIC, supporting Intel RAPL technology [11]. Each processor is 12 cores, with hyperthreading, for a total of 48 virtual cores across both sockets. These nodes are connected with a 32-port software-defined 40 GigE switch.

We set a 1820W cluster power budget. The offline profiles shown in Fig. 1 illustrate the different performance and power tradeoffs for `cluster` and `pigz`. There is one power cap (when the lines cross) where the front and backend performance is equal, and this case would require no work. To the left of that cap, we can increase performance by shifting power from `cluster` to `pigz`. To the right of that cap, we can decrease energy by shifting power away from `pigz`, as `cluster` has reached the point of diminishing returns.

[1] We release *PowerShift* as open source: https://github.com/Powershift2017/Powershift.

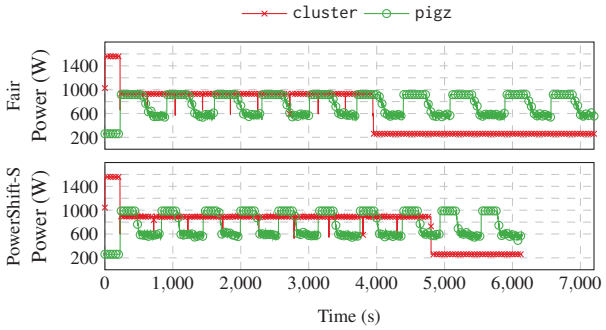

Figure 2: *Fair* (top) vs. *PowerShift-S* (bottom).

Figure 3: Runtimes for different approaches.

`cluster` iteratively produces output. To reduce storage space, `pigz` compresses each output. To produce the charts, we report performance as *iteration/s*. *PowerShift* itself does not measure performance. As shown, when the global power cap is 1820W `cluster` reaches 0.258 iteration/s, whereas `pigz` achieves only 0.143 iteration/s; *i.e.,* `pigz` is about half `cluster`'s speed.

2.1 Static Power Shifting

Fig. 2 compares *Fair* and *PowerShift-S*, highlighting the importance of shifting power from the fast application to the slow one so that they run at the same speed. The top figure shows *Fair* and the bottom shows *PowerShift-S*. We run the coupled workload for 10 iterations, the backend beginning after the first frontend iteration finishes. *Fair* divides the power budget equally among all nodes. At the beginning of the couple's execution, *PowerShift-S* uses the individual profiles to determine an unbalanced power distribution such that the jobs finish close together.

With *Fair*, `cluster` is much faster than `pigz`, requiring 4000 seconds. The couple, however, only finishes when `pigz` is done—at around 7000s. *PowerShift-S* allocates `pigz` more power to balance performance, resulting in around 6000s elapsed time, as shown in Fig. 3. Here, `pigz` still runs fairly slower than `cluster`, because there is a physical limit—imposed by the hardware—on the power we can shift from `cluster` to `pigz`.

2.2 Dynamically Shifting Unused Power

Fig. 3 shows that *PowerShift-S* is much faster than *Fair*, but *PowerShift-C* is faster still. To illuminate this performance gain, Fig. 4 compares *PowerShift-S* and *PowerShift-C*. For visibility, this figure shows just the first 1800 seconds of execution, highlighting the importance of dynamic power shifting to take advantage of application phases. The upper portion of Fig. 4 highlights the two distinct phases within one `pigz` iteration: (1) a power-hungry phase that compresses data

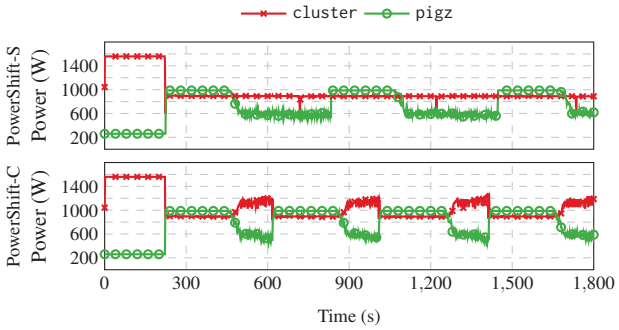

Figure 4: *PowerShift-S* (top) vs. *PowerShift-C* (bottom).

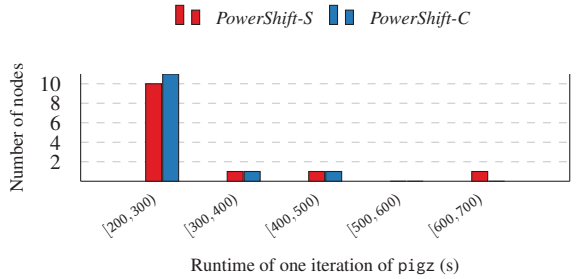

Figure 5: Tail distribution under static and dynamic power shifting. The static approach has a long tail, but the dynamic approach has a much shorter tail.

and (2) a low-power phase that waits for more data. *PowerShift-S* does not make runtime adjustment and cannot take advantage of `pigz`'s low-power phase. *PowerShift-C*, however, detects unutilized power and shifts it to `cluster` for a dramatic speedup. Under *PowerShift-C*, `pigz`'s second phase also finishes much faster and the reason will be introduced in next section. Therefore, *PowerShift-C* completes the couple much faster (in about 4000s, see Fig. 3).

This figure exemplifies power shifting that is not supported by current solutions. When `pigz` enters its low-power phase, power can be shifted away from those nodes without performance loss. Current schedulers can recognize this. The issue is that when `pigz` transitions back to a power-hungry phase, *PowerShift* has to reallocate power from `cluster` to `pigz` despite the fact that both are operating at or near their assigned budgets. *Power shifting between nodes operating at their caps is a unique challenge of supporting coupled application performance, not supported by prior work.*

2.3 Dynamically Shifting Power to Tail Nodes

Dynamic power monitoring also allows shifting power to tail nodes. In coupled applications, tail nodes ruin the performance of not just their own application, but the entire couple. Fig. 5 shows a histogram of completion times for one `pigz` iteration under both *PowerShift-S* and *PowerShift-C*. Static power distribution has a longer tail, with one node completing in the 600-700s range while the maximum completion time for dynamic scheduling is 500s.

Thus, dynamic power shifting in coupled applications both shifts power from one application to another and to tail nodes in either application. Fig. 6 shows the time series of one tail node in the `pigz` application. As noted, `pigz` has two phases. In the second, most of the backend nodes finish and wait for a few tail nodes. The

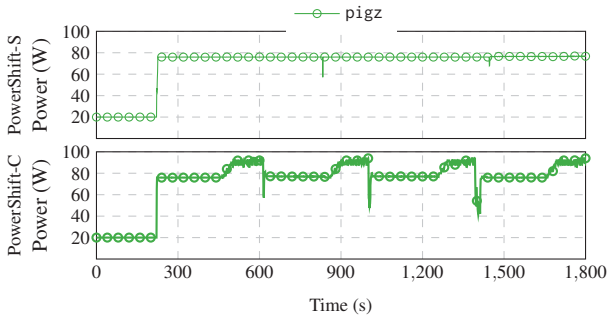

Figure 6: Tail node power time series comparing PowerShift-S (top), to PowerShift-C (bottom).

tail nodes under *PowerShift-S* continue running at the same speed and slow down the entire couple. *PowerShift-C* detects that a few nodes do not go to idle in this phase, so it shifts power from the idle nodes to both the frontend application and the tail nodes. As the bottom figure shows, the tail node gets more power during the second phase and therefore finishes much faster.

3 POWER MANAGEMENT APPROACHES

We discuss the 5 distributed power capping approaches we evaluate. We briefly review *Fair* and SLURM [49]. We then describe our couple-specific approaches: a static power manager using an offline power profile and two dynamic managers, one using a centralized decision mechanism, the other making distributed decisions.

All approaches assume there is a node-level power capping mechanism. Many exist in the literature [31, 44, 56] and Intel RAPL (Runtime Average Power Limiting) is a commercial example available on all current Intel processors [11]. The *Fair* and the *Static* approach do not involve any runtime operation, while other approaches (SLURM, centralized, and distributed) make runtime power measurements. The dynamic approaches follow a standard three step feedback process: (1) observe the current power, (2) decide on a response, and (3) act to implement the decisions.

3.1 Fair Power Allocation and SLURM

In the *Fair* approach, the system has no prior knowledge of the applications running on top of it. Therefore, it allocates power evenly to all nodes. Furthermore, at runtime, *Fair* has no mechanism to make changes adaptively. SLURM is an open-source job scheduler used by many supercomputers and clusters. SLURM provides an integrated system for power capping. SLURM starts with the same initial power distribution as *Fair*, then monitors actual power consumption, and redistributes power from nodes operating below their fair budget to those that are at their budget. This heuristic improves the throughput of independent applications, but is often sub-optimal for coupled applications as the couple's performance is defined by the slowest application.

3.2 *PowerShift-S*: Static Power Allocation

PowerShift-S, the static approach, takes advantage of offline application profiles based on two key concepts:

(1) The couple's performance is determined by the slower of the frontend and backend applications.

Algorithm 1 *PowerShift-S* Decision Algorithm

Require: Power budget P, Max shiftable power P_m
Require: Accuracy threshold A
Require: Performance improvement threshold T
Require: $S_f = \{(power_i, perf_i) | i = 1, 2, \ldots, n\}$ for frontend app
Require: $S_b = \{(power_i, perf_i) | i = 1, 2, \ldots, n\}$ for backend app
Fit S_f,S_b into functions $F_f(power) = perf$, $F_b(power) = perf$
$pow_f = pow_b = P/2$
$perf_f = F_f(pow_f)$, $perf_b = F_b(pow_b)$
$\delta = perf_f - perf_b$ ▷ assume $perf_f >= perf_b$
if $F_f(pow_f - pow_s) >= F_b(pow_b + pow_s)$ **then**
$pow_f = pow_f - pow_s, pow_b = pow_b + pow_s$
else
$pow_f = pow_f - pow_s, pow_b = pow_b + pow_s$
while $|\delta| > A$ **do** ▷ binary search for optimal power
$pow_s = pow_s/2$
if $F_f(pow_f) > F_b(pow_b)$ **then**
$pow_f = pow_f - pow_s, pow_b = pow_b + pow_s$
else
$pow_f = pow_f + pow_s, pow_b = pow_b - pow_s$
$\delta = F_f(pow_f) - F_b(pow_b)$
Power distribution: $\langle pow_f, pow_b \rangle$
Final performance: $perf_{final} = min(F_f(pow_f), F_b(pow_b))$
Fair performance $perf_{fair} = min(F_f(P/2), F_b(P/2))$
if $perf_{final} > (1 + T) * perf_{fair}$ **then**
▷ decide to enter performance or energy mode
Return (pow_f, pow_b) ▷ Enter performance mode
else
search for S_f,S_b, find the highest energy efficiency power state, pow_f and pow_b, that satisfy $F_f(pow_f) >= (1 - T) * perf_{fair}$ and $F_b(pow_b) >= (1 - T) * perf_{fair}$
▷ Enter energy mode

(2) All applications have monotonically non-decreasing power/-performance tradeoffs; *i.e.,* performance may level off with increasing power, but will never get worse.

We assume that we have profiled each application in isolation in the cluster to obtain power performance profiles, but we have never seen the coupled applications together. The profile only requires performance and power information. Scalability is not an issue, as the profile just captures performance as a function of node-level power cap, which does not change with the number of nodes. For this paper, we sample performance at 15 evenly spaced power caps, 3 evenly spaced input sizes, and apply quadratic regression to build the model.

Given the individual application profiles and a total system power budget, *PowerShift-S* maximizes performance by choosing the optimal static power distribution between the frontend and backend applications. In some cases, *PowerShift-S* recognizes that there is no additional performance to be gained, so it enters an *energy savings mode* that maintains a user-defined performance, while reducing energy. Algorithm 1 shows the basic logic of *PowerShift-S*. It requires the power budget P from the user and the maximum power that can be shifted from one application to the other based on the system P_m. It also requires two parameters: (1) the power accuracy A—the tolerance for error as application performance can usually not be perfectly matched—and (2) the performance improvement threshold T—the minimum performance improvement over the *Fair* approach required to avoid energy savings mode.

The algorithm first does a binary search for the highest performance power distribution between the front and back end applications. The power distribution returned is $\langle pow_f, pow_b \rangle$, which are distributed evenly among the front and backend nodes. The performance achieved here is called $perf_{final}$, the performance under *Fair* is called $perf_{fair}$. When $perf_{final} > (1+T) * perf_{fair}$—*i.e.,* there is acceptable performance gain—it enters performance mode and returns power distribution $\langle pow_f, pow_b \rangle$. Otherwise, it enters energy mode, where it searches all profile data points for each application to find the highest energy efficiency such that $F_f(pow_f) >= (1-T) * perf_{fair}$ and $F_b(pow_b) >= (1-T) * perf_{fair}$. It returns the most energy efficient power state within at least $(1-T)$ of max performance for both applications. In our implementation, we choose $A = 0.5$, because such a small difference has little effect on performance. We choose performance improvement threshold $T = 3\%$. The larger T is, the more *PowerShift-S* prefers to optimize for energy, and the smaller T is, the more it prefers to optimize for performance.

Algorithm 2 *PowerShift-C* decision algorithm.

Require: Node number N, power margin M
Require: Optimal static power budget $B = \{b_i | i = 1, 2, \ldots N\}$
while application couples not finished **do**
$\{pow_i | i = 1, 2, \ldots N\} \leftarrow GetAveragePower()$
Power budget of last iteration $\{bl_i | i = 1, 2, \ldots N\}$ ▷ the average power of each node since last iteration
Group all nodes into 3 priority groups
Group 1 $\leftarrow \{Node_i | pow_i < b_i - M\}$
▷ Nodes with extra power
Group 2 $\leftarrow \{Node_i | pow_i >= b_i - M, bl_i >= b_i\}$
▷ Nodes at power limit
Group 3 $\leftarrow \{Node_i | pow_i >= b_i - M, bl_i < b_i\}$
▷ Nodes at power limit needing more power
if Group 1's extra power > Power needed by Group 3 **then**
Shift all needed power from Group 1 to Group 3
Any extra power from Group 1 goes to Group 2
else
Shift power exceeding B from Groups 1 & 2 to Group 3
Any extra power is divided between Group 2 and 3
Sleep(t) ▷ make power shifting every t time

A simple proof of the optimality is as follows: By concept 2, above, both applications have monotonically non-decreasing performance as a function of power. Then, at the optimal state, if we reallocate power from one end to the other, one application slows down, and the other speeds up. But by concept 1, performance is determined by the time when both applications finish; *i.e.,* the slower application is the performance limiter. Therefore, any power shifting from the optimal state will increase the overall runtime.

3.3 Dynamic Power Shifting

We propose 2 dynamic power management approaches: *PowerShift-C* and *PowerShift-D*. Critically for coupled applications, both approaches can take power from high-performance nodes to shift to slower nodes with the result of speeding up the entire couple. *PowerShift-C* has a single centralized decision maker. In *PowerShift-D* decisions are distributed and each node has its own local decision maker. Both approaches take advantage of the offline profiles and their initial power distribution state is the same as *PowerShift-S*. In energy mode, runtime power shift is disabled, because any power state change will cause the coupled applications to move away from the most energy-efficient power state.

3.3.1 **PowerShift-C***: Centralized Power Shifting. PowerShift-C* starts with the optimal static power from *PowerShift-S* and operates

on a typical control loop: observing its environment, deciding on a response, and acting to implement its decisions. During observation, it collects each nodes power usage. To decide a new power distribution, it first groups all nodes into 3 power priority groups (described below) and then it shifts power from those nodes that either (1) have lower priority or (2) simply have unused power to the nodes that are hungry for power with a higher priority. Finally, it acts by sending new power capping information to each node.

Algorithm 2 details *PowerShift-C*. The algorithm requires a *power margin M*; any node that consumes power within M of its local power cap is defined to be *intensively* using power. Otherwise, the node can reduce power. The central decider first groups each node into 1 of 3 priority classes by its distance to its own power cap and whether its current power cap is higher than the optimal static power cap. Higher group numbers mean the node needs power more urgently. Group 1 has power to spare. Group 2 is operating at or near their power cap, but will not increase the couple's performance with extra power. Group 3 is operating at or near the power cap and increased power will increase the couple's performance. *A key difference between PowerShift and prior work, is that PowerShift will move power from Group 2—which has high power utilization—to Group 3—which also has high utilization. Existing schedulers, like SLURM, will only shift from low utilization nodes to high utilization nodes, making them unsuitable for coupled applications.*

After assigning nodes to priority groups, the algorithm checks the shiftable power of each group to make sure that on the next iteration, all nodes in Group 3 will at least get power equivalent to their optimal static power budget. If necessary, the algorithm will force Group 2 to reduce power in favor of Group 3; otherwise, extra power will go to Group 2. Finally, the decision engine idles for a time interval t. This idling both reduces overhead and allows the nodes to settle into their new power consumption.

Front and backend nodes get different amounts based on their performance derivative. $F()$ is the performance power function generated by offline data, p_f is the power acquired by frontend node and p_b is the power acquired by backend node.

$$\frac{\delta F_f(pow_f)}{\delta pow_f} : \frac{\delta F_b(pow_b)}{\delta pow_b} \quad = \quad p_b : p_f \tag{1}$$

This shifting power ratio ensures that all nodes in one end (front or back) get extra power proportional to the performance growth of the other end. Thus, *PowerShift* allocates less power to the end where performance scales faster and more power to the end that scales worse so that both ends work at the same rate.

For power margin M and time interval t in Algorithm 2, we use 2 Watts and 1 second. Choosing M decides how aggressively we want to shift power. Lower M results in higher aggressiveness, at the risk of possibly removing power from nodes that need it. Higher M means lower aggressiveness, at the risk of not being able to fully utilize the unused power. For the time interval, larger intervals make the system reaction time longer and it may miss short duration power shifting opportunities. A shorter time interval exaggerates the system noise but provides more opportunities to react.

3.3.2 **PowerShift-D***: Distributed Power Shifting.* *PowerShift-D* distributes power shifting. The key data structure is a *power pool* that contains unused power. In contrast to *PowerShift-C*, *PowerShift-D* does not have any centralized component. Instead, each node has its own local decision maker to shift power. All local power deciders talk to the power pool asynchronously and make decisions based on their own power state and the power pool.

Algorithm 3 *PowerShift-D* algorithm.

```
Require: Power margin M
Require: Optimal static power budget of itself B
  while not finished do
    Current power p_c ← GetAveragePower()
    Power budget of last iteration b
    Define power state of itself
    Group 1: p_c < b − M                          ▷ have extra power
    Group 2: p_c >= b − M, b > B                  ▷ at power limit
    Group 3: p_c >= b − M, b < B                  ▷ at power limit, need power
    Read p_pool and urgency flag F
    if Group 1 then
      Post extra power to power pool
    else
      if Group 3 then
        Grab unused power from power pool
        When p_pool < B − b, increment F
        After needed power is satisfied, decrement F
      else
        When F = 0, take from power pool
        When F ≠ 0, post b − B to power pool
    Sleep(t)
```

The power pool structure serves as a bulletin board to all the local deciders. It has a floating point variable F to post how much power is in the pool and one integer variable I to post how many nodes are in Group 3 (from above) if any. When $I = 0$, nodes Groups 2 & 3 take power from the power pool. Otherwise, only nodes in Group 3 can take power from the pool, and nodes in Group 1 or 2 have to give up power exceeding their initial power budget if any. Group 3 nodes will always get the power they need. If there is not enough power in the pool, Group 2 nodes will have to release power (even if they are at their budget).

We maintain the invariant that any node adding power to the pool must first reduce its local power usage and any node taking power must decrement unused power in the pool before raising its local power usage. This simple invariant ensures the global power budget is respected with local decision making. Our implementation uses sockets to communicate between local nodes and the power pool.

Algorithm 3 shows the local power decision systems. The rule to choose the power margin M is same as in *PowerShift-C*. The time interval t is different here. Since every local decider can have different values, we use 2 seconds for nodes in Groups 1 and 2, and 1 second for nodes in Group 3, to allow the node to get out of an urgent state faster. *PowerShift-D* can easily integrate with local power managers by coordinating its interaction with the pool with the decisions of the local power manager. Other approaches are not compatible with independent, local power managers.

3.4 Extension Beyond Two Applications

A simple modification version of *PowerShift-S* can work on any arbitrary number of dependent applications. In each loop iteration, we shift power from the fastest application to the slowest, with the goal that all applications run at the same speed. In this way,

the binary search for the optimal performance extends across all applications and get the power distribution of each.

PowerShift-C and *-D* are also trivially extended to any number of dependent applications: the global or local decider follows the same algorithm regardless of the number of applications. This scalability to multiple applications for the dynamic cases is easy because the algorithms do not reason about front and backend applications. Instead they reason about priority levels which are set based entirely on a node's behavior relative to its assigned power cap. This flexibility does rely on the stated assumption that performance is monotonically non-decreasing with increasing power consumption.

3.5 Complexity, Guarantees, and Discussion

Algorithmic Complexity

(1) *Fair* is $O(1)$.
(2) *PowerShift-S* is $O(n * \log P_m)$, given n profile data points and P_m maximum shiftable power. It uses binary search to find the optimal power distribution.
(3) *PowerShift-C*: Executes the *PowerShift-S* algorithm once at launch. At each iteration, it executes Algorithm 2, which is $O(N)$, where N is the number of nodes.
(4) *PowerShift-D*: Executes the *PowerShift-S* algorithm once at launch. The communication with the power pool is $O(1)$.

The complexity of *PowerShift-S* grows with larger n (more profile data points). We do not consider this a scaling problem, because only a small n is needed to have a good picture of power and performance tradeoffs for each application. In our implementation, we use $n = 15$. Additionally, this algorithm is only executed once before launch. *PowerShift-C*'s complexity, however, grows with N, the number of nodes. In a larger cluster, N might be thousands or tens of thousands. The centralized approach may suffer from overload. In that case, we have to provide more computing power and hardware to the central decider or have a multi-tier central decider to solve the scaling problem. This linear scaling does not occur with *PowerShift-D*, which essentially parallelizes the power distribution process, leading to an approach that naturally scales with the number of nodes.

Guarantees

(1) *Fair*: equal power distribution between frontend and backend; equal power distribution between each node; and assurance that the global power budget is met.
(2) *PowerShift-S*: the maximum performance static power distribution with at least T speedup over *Fair*. When T speedup is not achievable, the most energy efficient power distribution within $(1 - T)$ of *Fair*'s performance.
(3) *PowerShift-C* and *PowerShift-D*, both provide speedup when the applications have unbalanced power needs over time or across different nodes. In energy-saving mode, these approaches have the same guarantees as *PowerShift-S*. Due to the simple invariants that nodes first lower their power before contributing to the global power budget and first take from the global power budget before raising their power, these approaches also ensure that the global power cap is respected.

Topology-obliviousness These guarantees are *topology-oblivious*. While we have assumed—for simplicity of discussion—that the front and backend applications are on physically separate nodes, they can actually be scheduled in any manner such that their power control is independent. For example, front and backend processes could be scheduled on separate sockets in a node, or separate cores in a chip, if the processor supports independent core-level power caps. Our empirical analysis shows that performance is the same under different mappings (see Section 4.4).

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setup

Benchmarks We use 9 individual applications including two cosmological simulation benchmarks, `cluster` and `galaxy` from Gadget 2.0 [51], a hydrodynamic simulation benchmark `lulesh` [28], a visualization application `VisIt` [10], a data mining application `kmeans` [6], a data compression application `pigz` [1], and three MapReduce [12] applications: `teragen`, `terasort`, and `teravalidate` [40]. These benchmarks test a range of important cluster applications, both compute-intensive and memory-intensive. All applications run with up to 48 threads per node (the maximum supported on our test machine). All workloads are long running, taking at least a few minutes to complete.

We pair applications from this pool. The frontend applications are: `cluster`, `galaxy`, `lulesh`. The backend applications are: `VisIt`, `kmeans`, `pigz`. These nine pairs are representative of the emerging HPC trend of coupling scientific simulation benchmarks with *in situ* analysis or visualization [2, 9]. The 3 MapReduce applications pair among themselves. In each pair, the output of the frontend application is sent to the backend application as input. In total, we have 11 pairs of coupled applications.

Platform and Metrics We use the 26-node cluster described in Section 2. All nodes run Linux 3.13.0. We use RAPL [11] for both power monitoring and capping. The bulk of our results use an execution model where frontend and backend applications are deployed to physically separate nodes. We reiterate, however, that *PowerShift* is topology-oblivious. We explore a different mapping—where front and backend applications are mapped to different sockets in the same physical nodes in Section 4.4. We use $1/runtime$ as our performance metric, and $1/energy$ as our energy efficiency metric. All comparisons, normalize to the *Fair* approach.

4.2 Performance

We evaluate our eleven coupled applications with five different cluster power budgets ranging from 1820W to 2860W. In all approaches, the sum of the node-level power caps is less than or equal to the cluster power budget at all times.

Table 1 shows harmonic mean performance under each power cap for SLURM and *PowerShift*. SLURM performs better than *Fair* at lower power caps and worse at higher power caps, but *PowerShift* uniformly outperforms both SLURM and *Fair*. SLURM's performance suffers for two reasons. First, moving power from nodes with low-power phases to power-hungry nodes, violates the principal that shifts power from the faster application to the slower one, because the power-hungry one is not necessarily the slower one. Second, SLURM does not have a mechanism to return the power back to the node that gave it up when that node

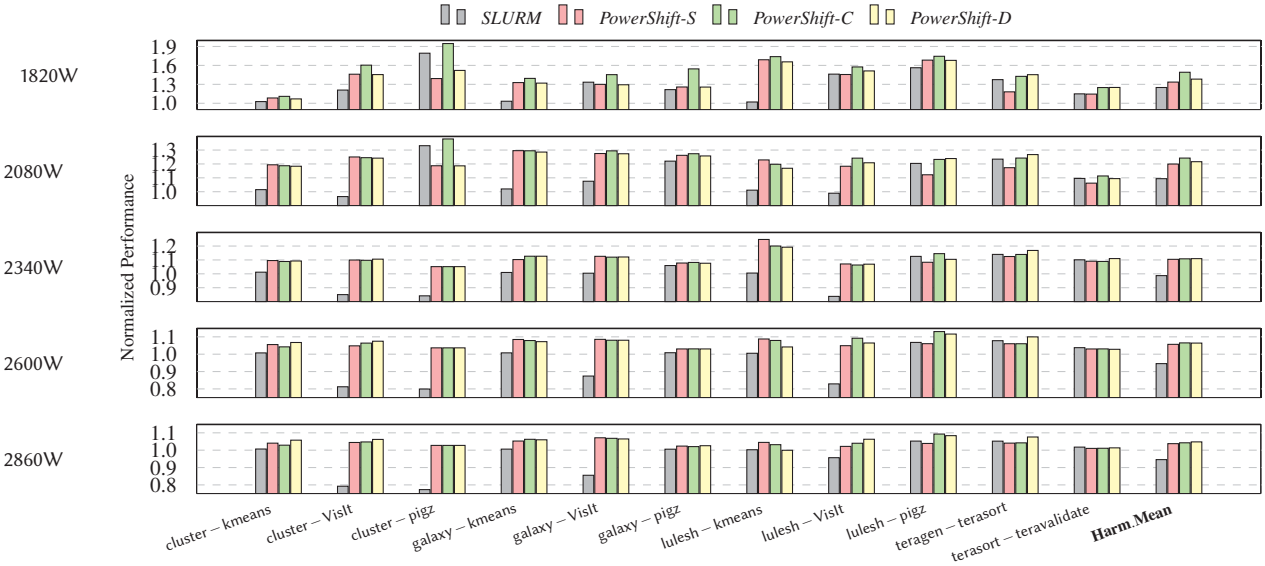

Figure 7: Performance for different PowerShift strategies under different power caps.

Table 1: Comparison of performance under power caps.

Power Cap	SLURM	PowerShift-S	PowerShift-C	PowerShift-D
1820W	1.25	1.28	1.49	1.38
2080W	1.09	1.15	1.24	1.22
2340W	0.99	1.06	1.11	1.11
2600W	0.95	1.01	1.07	1.06
2860W	0.95	1.04	1.04	1.05
Har. Mean	**1.03**	**1.10**	**1.17**	**1.15**

shifts to a phase that requires more power. For example, when running the `cluster-pigz` couple, `cluster` is always power hungry and `pigz` alternates between power-hungry phase and low-power phase. Once `pigz` enters the low-power phase the first time, all the unutilized power is shifted to `cluster` and never returns. As a result, when `pigz` re-enters its power-hungry phase, it suffers great performance loss. This effect happens at both low and high power budgets, however, when the power budget is low, it turns out that focusing all power budget on one application has fairly good performance. But at higher power budgets, the speedup of focusing all power on one application at a time is much smaller and often worse than *fair*.

The key to *PowerShift's* performance is that it understands coupling. In the static case, this understanding allows some performance gain over SLURM. In the dynamic case, *PowerShift* is designed so that it will shift power from one high-power application to another. This design manifest itself in *PowerShift's* three priority groups (Section 3.3.1). This unique feature of *PowerShift* allows it to shift power back from `cluster` to `pigz` in the above example and achieve much greater performance than SLURM.

PowerShift-S consistently outperforms *Fair* by 1–28%, with higher speedup under lower power budgets. *PowerShift-C* outperforms *Fair* by 4–49%, while *PowerShift-D* outperforms *Fair* by 5–38%. When the power budget is relatively low, the performance scales better with power, so *PowerShift* has more potential to boost the slower application's performance. For higher power budgets, it is more likely that application performance stops scaling even with more power.

Fig. 7 shows the performance delivered under each power budget for each coupled application pair. This figure contains one chart for each power cap, the x-axis shows the couple, the y-axis shows performance normalized to *Fair*. The charts show one bar for each of *PowerShift-S*,*PowerShift-C*, and *PowerShift-D*. While results vary per application and power cap, the general trends show that:

(1) All *PowerShift* approaches outperform *Fair*.
(2) *PowerShift-C* and *PowerShift-D* provide higher performance than *PowerShift-S* across all power budgets.
(3) For lower power budgets, *PowerShift-C* out-performs *PowerShift-D* on average. At higher power budgets, *PowerShift-D* can out-perform *PowerShift-C*.

As shown in Fig. 7, *PowerShift-C* and *PowerShift-D* consistently outperform *PowerShift-S*—by as much as 20%. Applications with long tails receive the biggest benefits from runtime power shifting. In our case, applications having significant I/O phases—*e.g.,* MapReduce, VisIt, and Lulesh—or applications having both phases and long tails—*e.g.,* pigz—all get big speedups with dynamic, runtime power shifting compared to *PowerShift-S*'s static allocation.

The tradeoffs between the dynamic approaches *PowerShift-C* and *PowerShift-D* are not as obvious but they do show:

(1) At lower power budgets, *PowerShift-C* outperforms *PowerShift-D*. In *PowerShift-D*, the power pool may not ever be fully drained, but *PowerShift-C* uses global knowledge to allocate all power. At low power budgets, a small amount of unallocated power left in *PowerShift-D*'s pool can have a large effect on performance.
(2) If an application has distinct phases—where each node frequently goes from a power-saturated state to a power-hungry state or vice versa—*PowerShift-D* reacts faster than *PowerShift-C*. First, each local decider in *PowerShift-D* acts asynchronously, providing faster reaction to local power state changes. Second, local decisions have their own control frequency. In our

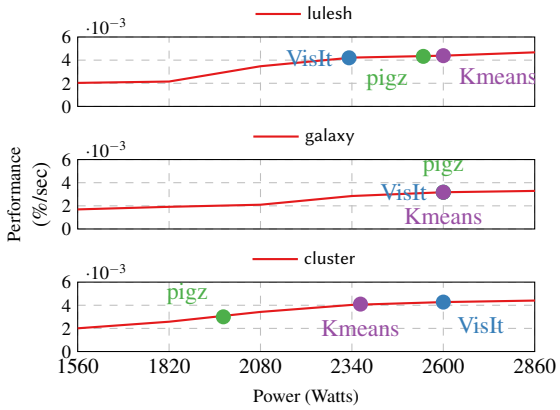

Figure 8: Power distribution of different backends paired with the three scientific frontends.

tests, the local decider runs at double the frequency in the power-hungry state compared to the power-saturated state.

(3) As the cluster grows, the central decider in *PowerShift-C* may have too much work and generate workload imbalance if co-located with frontend or backend nodes. Eventually, it will need a separate node.

4.3 Coupling's Effects on Optimal Power

The prior sections show that*PowerShift* boosts performance compared to fair power distribution. We now demonstrate that power shifting is a hard problem by showing that power allocations that work well for an application in one couple are poor when that application is coupled with a different application; *i.e.,* optimal power allocation is both application- and couple-dependent.

Fig. 8 shows the optimal power distribution when the three scientific frontend applications are paired with the three different backend applications. There are three charts, one for each of the front-end applications. The x-axis shows power and the y-axis shows performance. There is a labeled mark for each backend application showing its optimal static power assignment when coupled with that frontend. As the figure shows, the different frontends place different requirements on the backends. For example, when coupled with `lulesh`, `VisIt` needs the least power of the three backends. When coupled with `cluster`, however, `VisIt` needs the most power of all backends. *These results show that the front and backends cannot be optimized in isolation, but require consideration of the application couple as a whole.*

4.4 Results with Co-located Front and Backends

This section evaluates *PowerShift* in a different topology: when frontend and backend application processes are mapped to separate sockets on the same physical nodes. Other than this process-to-node allocation, the experiments are the same as in Section 4.2. Table 2 summarizes the results, showing that with a different mapping topology, *PowerShift* has very similar results and properties.

Table 2: Performance with co-located couples.

Power Cap	PowerShift-S	PowerShift-C	PowerShift-D
1820W	1.26	1.49	1.46
2080W	1.18	1.36	1.32
2340W	1.04	1.11	1.10
2600W	1.01	1.10	1.10
2860W	1.01	1.06	1.07
Har. Mean	**1.09**	**1.20**	**1.19**

Table 3: Performance of three dependent applications.

	Performance		Energy Efficiency	
Power Cap	**PowerShift-S**	**PowerShift-D**	**PowerShift-S**	**PowerShift-D**
1820W	1.55	1.65	-	-
2080W	1.15	1.21	-	-
2340W	1.07	1.09	-	-
2600W	1.03	1.01	-	-
2860W	.99	.99	1.02	1.02
Har. Mean	**1.13**	**1.15**	-	-

Table 4: Energy efficiency with larger threshold T.

Benchmark	**performance**	**energy efficiency**
cluster-kmeans	1.02	1.31
cluster-VisIt	0.94	1.21
cluster-pigz	0.93	1.23
galaxy-kmeans	0.92	1.33
galaxy-VisIt	0.94	1.41
galaxy-pigz	0.92	1.42
lulesh-kmeans	0.99	1.07
lulesh-VisIt	0.96	1.06
lulesh-pigz	0.96	1.07
teragen-terasort	1.01	1.04
terasort-teravalidate	1.01	1.03
Har. Mean	**0.95**	**1.18**

4.5 Considering Three Dependent Applications

We illustrate how *PowerShift* can scale beyond two dependent applications. In this example, `cluster` generates scientific simulation data, `VisIt` turns this data into a movie, and `pigz` compresses the movie for storage. Table 3 shows the performance and energy efficiency of *PowerShift-S* and *PowerShift-D* normalized to *Fair*. *PowerShift-C* is omitted in this experiment, because its overhead grows significantly beyond two applications. For power caps 1820W, 2080W, 2340W, 2600W, *PowerShift* enters performance mode. At the 2860W cap it enters energy-saving mode. The results are consistent with those for coupled applications. Under lower power caps, *PowerShift* tends to enter performance mode boosting performance significantly. Under higher power caps, the potential performance gain is less and it tends to favor energy saving. By harmonic mean, *PowerShift-S* improves performance of *Fair* by 13%, while *PowerShift-D* achieves a 15% improvement. *These results indicate that* PowerShift *still achieves significant performance gains when working with more than two applications.*

4.6 Energy Savings with High Power Budgets

This section evaluates the energy efficiency when *PowerShift* enters energy-saving mode. The performance improvement threshold T, discussed in Section 3.2 is a key impact factor for energy savings. The results in Fig. 7 show $T = 3\%$; *i.e.,* only very small performance loss is allowed. We now evaluate the energy savings when $T = 10\%$.

Table 4 shows the performance loss and energy efficiency for all pairs of coupled applications under the maximum cluster power budget. *PowerShift* achieves 18% higher energy efficiency with only 5% performance loss based on harmonic mean. The reason for this savings is that with higher threshold T, a wider range of power states can be searched for the most energy-efficient one.

We note that our earlier results show diminishing performance returns for high power budgets (see Table 1). The results in this section, however, indicate that *PowerShift* still provides a benefit at high power budgets if users are willing to sacrifice some performance to save energy. In this case, *PowerShift* improves energy efficiency by 18% on average and by as much as 42% for individual couples. *These results indicate that* PowerShift *provides performance gains at low power budgets or energy savings with high power budgets.*

4.7 *PowerShift*'s Resilience to System Noise

System noise is inevitable in any computing system and prevalent at the large scale of clusters and supercomputers [4]. System noise comes from many kinds of physical and human sources, but its defining characteristic is unpredictability. It often results in great performance fluctuation or slow down on random nodes. It is one of the biggest reasons that large scale systems have severe tail problems. In coupled scientific computations, this system noise slows down not just a single node, but the entire couple.

Fig. 9 shows the performance results of running all eleven couples in a noisy environment. We create the noise by randomly picking a node and running an additional 8-thread program, stressing the computing resources on one node of the cluster but leaving all other nodes in their usual state. This method of generating noise is general, *PowerShift* does not detect the new application, it simply detects increased power usage on one node. Thus this method could be a stand-in for many other noise sources, including temperature fluctuations or manufacturing variation [26].

All experiments are under the 1820W system-wide power budget and everything is normalized to *Fair*. *PowerShift-C* and *PowerShift-D* consistently outperform *Fair* and average 1.36 and 1.30 speedup, respectively. *PowerShift-S*, however, only has 1.07 speedup and in some cases, is beaten by *Fair*. *Fair* can outperform *PowerShift-S* because the static power distribution derived by the offline decider is not optimal once the second application starts. In *Fair*, however, the stressed node may belong to the slower application, and therefore, may not necessarily slow down the process. Both *PowerShift-C* and *PowerShift-D* deal with tail nodes by shifting power from other finished or light-workload nodes. *These results indicate that dynamic power shifting makes a distributed system significantly more resilient to system noise without requiring any application-level changes.*

Although omitted for space, we have tested many different noise levels. The included data shows that *PowerShift* produces a big benefit for even a small noise level. Our omitted experiments show that the higher the noise, the better the dynamic approaches do compared to static allocation.

4.8 Overhead and Scalability analysis

We show the runtime overhead of the central decider from *PowerShift-C*, and the local decider *PowerShift-D*. We have discussed the likelihood of scaling problems in Section 3.5. At our experimental scale, the overhead of both approaches are negligible. Table 5 shows the power and core utilization overhead of both approaches running on power traces. The central decider has $1.23W$ and 4.725% core utilization overhead. The local decider has $0.06W$ and 0.323% core utilization overhead. As the node number grows, local deciders' workload is constant, whereas the central decider's workload grows linearly with node number, shown in Section 3.5. All our results shown in the Section 4 include *PowerShift*'s overhead; *i.e.,* the *PowerShift* software is running with the jobs it manages on the same machines. *These results show that the centralized decision mechanism is practical at small scale, while the distributed mechanism is lower overhead and more likely to scale to large systems.* As the system scale grows, interconnection latency is critical to ensure strong scaling. In our design, it is very easy to account for the latency overhead to the frontend. Increased network overhead will result in longer IO phases for the frontend application. This trend will likely make the static scheme worse (since it uses single-application profiles), but *PowerShift-C and -S* should dynamically adjust to find the optimal. Therefore, we believe in a higher latency interconnection environment, there are more opportunities for dynamic shifting.

Table 5: Comparison of overhead.

workload	**Power**	**Core Utilization**
idle	24.52W	0
central decider	25.75W	4.725%
local decider	24.58W	0.323%

5 RELATED WORK

Power and energy management is extremely important for data centers [13, 16, 21, 38, 53] and it has long been important for mobile and embedded systems [17, 18, 22, 25, 32, 36, 46, 50, 55].

For HPC and data center power management, hooks exist at multiple levels; *e.g.,* the processor [11], load balancer [8], enclosure [43, 53], and the installation itself [16]. Raghavendra et al. present a method for ensuring that power management systems at multiple levels coordinate to stabilize at the desired power [42].

Apart from simply respecting a power cap, though, it is important to deliver performance. Several approaches examine the problem of maximizing performance subject to a power cap. Again, this work can be done at the processor-level [23, 24, 44, 56], DRAM [11], storage [30], and across a data-center [33, 39]. At the data-center or cluster-level, power can be saved by consolidating workloads to use fewer physical machines [14, 27, 34, 35, 37, 54], coordinating co-existing applications [45], and scheduling with green power [20]

Scheduling jobs under a power cap has recently become a major concern for HPC operating systems [7, 15] and job schedulers [3, 19]. Recent work suggests that HPC workloads can actually achieve higher performance by over-provisioning large-scale installations—such that using all nodes at full capacity would drastically violate the power budget—and severely power capping the individual nodes [47]. Additionally, future supercomputer workloads are expected to contain exactly the type of *coupled* applications explored in this paper [5] and the US Department of Energy explicitly lists both power management and coupled workloads as primary challenges to be addressed by future operating systems [52]. Yet, little work

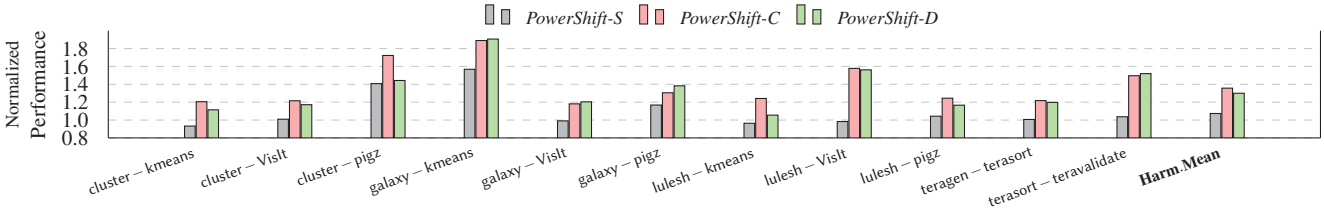

Figure 9: Performance under system noise.

has been done to explore the effect of power capping on dependent jobs.

Closest to *PowerShift* is a collection of prior work that shifts power unevenly among jobs in HPC systems. *Power Routing* shifts power to maintain throughput and minimize power infrastructure; it does not support dependent applications [41]. SLURM will shift power from nodes operating below their budget to those at the budget [49]. Another approach is aware of application-level semantics and will shift power from processes executing IO (and thus not using their full power allocation) to those doing computation (i.e., at their power limit) [48]. Inadomi et al. propose variation-aware power shifting that shifts power within a single parallel job to compensate for manufacturing variations [26].

While these approaches and *PowerShift* all move power from one process to another, *PowerShift* is unique in its ability to speedup dependent, coupled applications. *The key difference is that PowerShift will move power from one process operating near its cap to another that is also near its cap.* This ability is essential for speeding up coupled applications whose performance is dependent on the slower application. To the best of our knowledge no other power shifting system will shift from one high-power process to another to speed up dependent applications.

6 CONCLUSION

This paper presents *PowerShift*, a family of techniques for shifting power among nodes running coupled, distributed applications under a system-wide power budget. Specifically, we propose a static scheme, a centralized dynamic scheme, and a distributed dynamic scheme. The static scheme is low overhead, but is not able to adjust to tail behavior or application phases. Both dynamic schemes do adjust, with the centralized scheme generally achieving higher performance—with higher overhead—while the distributed dynamic scheme trades some performance for reduced overhead and increased flexibility. Compared to prior approaches, *PowerShift* provides two advances: (1) instead of just shifting from low-power to high-power nodes, *PowerShift* will shift power away from high utilization nodes to other high utilization nodes that need power more; and (2) *PowerShift* can recognize when it is not helpful to shift power and instead it will reduce energy. Our results confirm that *PowerShift* has practical benefits, demonstrating improved performance, reduced energy, and dynamic adjustment to tail behavior and system noise. We believe the coupled workloads addressed in this paper will become increasingly important in both data centers and supercomputers because they reduce IO burden and take advantage of system scale. *PowerShift* represents one way of improving the performance, energy efficiency, and tail tolerance of this class of application.

Acknowledgments. This research was supported by the DARPA BRASS program, a DOE Early Career award, and the NSF (CCF-1439156). Experiments were conducted on the NFS-funded RIVER infrastructure (CNS-1405959). We thank Andrew A. Chien, who leads RIVER, and Fan Yang who helped configure RIVER machines for this research.

REFERENCES

[1] Mark Adler. [n. d.]. A parallel implementation of gzip for modern multi-processor, multi-core machines. http://zlib.net/pigz/
[2] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, Jim Ahrens, E. Wes Bethel, Hank Childs, Jian Huang, Ken Joy, Quincey Koziol, Gerald Lofstead, Jeremy Meredith, Kenneth Moreland, George Ostrouchov, Michael Papka, Venkatram Vishwanath, Matthew Wolf, Nicholas Wright, and Kesheng Wu. 2011. Scientific discovery at exascale: Report from the doe ascr 2011 workshop on exascale data management, analysis,and visualization.
[3] Peter E Bailey, Aniruddha Marathe, David K Lowenthal, Barry Rountree, and Martin Schulz. 2015. Finding the limits of power-constrained application performance. In *SC*.
[4] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, and Susan Coghlan. 2006. Operating System Issues for Petascale Systems. *SIGOPS Oper. Syst. Rev.* 40, 2 (April 2006), 29–33. https://doi.org/10.1145/1131322.1131332
[5] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. 2008. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead.
[6] Halil Bisgin and HN Dalfes. 2008. Parallel clustering algorithms with application to climatology. In *Geophysical Research Abstracts*, Vol. 10.
[7] Ron Brightwell, Ron Oldfield, Arthur B. Maccabe, and David E. Bernholdt. 2013. Hobbes: Composition and Virtualization As the Foundations of an Extreme-scale OS/R. In *ROSS*.
[8] Jeff Chase and Ronald P. Doyle. 2001. Balance of Power: Energy Management for Server Clusters. In *HotOS*.
[9] J. Chen, Alok Choudhary, S. Feldman, B. Hendrickson, C. R. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams. 2013. Synergistic Challenges in data-intensive science and exascale computing.
[10] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. 2012. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. 357–372.
[11] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *ISLPED*.
[12] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *OSDI*.
[13] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*.
[14] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *ASPLOS*.
[15] Daniel A. Ellsworth, Tapasya Patki, Swann Perarnau, Sangmin Seo, Abdelhalim Amer, Judicael A. Zounmevo, Rinku Gupta, Kazutomo Yoshii, Henry Hoffmann, Allen D. Malony, Martin Schulz, and Peter H. Beckman. 2016. Systemwide Power Management with Argo. In *IPDPS Workshops*.

[16] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Power Provisioning for a Warehouse-sized Computer. In *ISCA*.

[17] J. Flinn and M. Satyanarayanan. 1999. Energy-aware adaptation for mobile applications. In *SOSP*.

[18] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. 2008. Quanto: Tracking Energy in Networked Embedded Systems. In *OSDI*.

[19] Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power tuning HPC jobs on power-constrained systems. In *PACT*.

[20] Md E. Haque, Inigo Goiri, Ricardo Bianchini, and Thu D. Nguyen. 2015. GreenPar: Scheduling Parallel High Performance Applications in Green Datacenters. In *ICS*.

[21] Urs Hoelzle and Luiz Andre Barroso. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.

[22] Henry Hoffmann. 2015. JouleGuard: Energy Guarantees for Approximate Applications. In *SOSP*.

[23] Henry Hoffmann and Martina Maggio. 2014. PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management. In *ICAC*.

[24] C. Imes and H. Hoffmann. 2016. Bard: A unified framework for managing soft timing and power constraints. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 31–38. https://doi.org/10.1109/SAMOS.2016.7818328

[25] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints. In *RTAS*.

[26] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, Masaaki Kondo, and Ikuo Miyoshi. 2015. Analyzing and Mitigating the Impact of Manufacturing Variability in Power-constrained Supercomputing. In *SC*.

[27] M. A. Islam, X. Ren, S. Ren, A. Wierman, and X. Wang. 2016. A market approach for handling power emergencies in multi-tenant data center. In *HPCA*. https://doi.org/10.1109/HPCA.2016.7446084

[28] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.

[29] D. E. Keyes, Lois Curfman McInnes, C. Woodward, W. D. Gropp, E. Myra, and M. Pernice. 2012. Multiphysics Simulations: Challenges and Opportunities. (10/2012 2012). http://hpc.sagepub.com/content/27/1/4.full.pdf+html

[30] Mohammed G. Khatib and Zvonimir Bandić. 2016. PCAP: Performance-aware Power Capping for the Disk Drive in the Cloud. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 227–240. https://www.usenix.org/conference/fast16/technical-sessions/presentation/khatib

[31] C. Lefurgy, X. Wang, and M. Ware. 2008. Power capping: a prelude to power shifting. *Cluster Computing* 11, 2 (2008).

[32] Matthew Lentz, James Litton, and Bobby Bhattacharjee. 2015. Drowsy Power Management. In *SOSP*.

[33] Ivana Marincic, Venkatram Vishwanath, and Henry Hoffmann. 2017. PoLiMEr: An Energy Monitoring and Power Limiting Interface for HPC Applications. In *E2SC*.

[34] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *MICRO*.

[35] Andreas Merkel, Jan Stoess, and Frank Bellosa. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*. ACM, 153–166.

[36] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *ASPLOS*.

[37] N. Mishra, J. D. Lafferty, and H. Hoffmann. 2017. ESP: A Machine Learning Approach to Predicting Application Interference. In *ICAC*.

[38] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. 2015. A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints. In *ASPLOS*.

[39] Ripal Nathuji and Karsten Schwan. 2007. VirtualPower: coordinated power management in virtualized enterprise systems. In *SOSP*.

[40] Owen O?Malley. 2008. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)* (2008), 1–3.

[41] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F. Wenisch, and Jack Underwood. 2010. Power routing: dynamic power provisioning in the data center. In *ASPLOS*.

[42] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No "power" struggles: coordinated multi-level power management for the data center. In *ASPLOS*.

[43] Parthasarathy Ranganathan, Phil Leech, David E. Irwin, and Jeffrey S. Chase. 2006. Ensemble-level Power Management for Dense Blade Servers. In *ISCA*.

[44] S. Reda, R. Cochran, and A.K. Coskun. 2012. Adaptive Power Capping for Servers with Multithreaded Workloads. *Micro, IEEE* 32, 5 (2012).

[45] Haris Ribic and Yu David Liu. 2016. AEQUITAS: Coordinated Energy Management Across Parallel Applications. In *ICS*.

[46] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. 2011. Energy Management in Mobile Devices with the Cinder Operating System. In *EuroSys*.

[47] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kale. 2014. Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget. In *SC*.

[48] L. Savoie, D. K. Lowenthal, B. R. d. Supinski, T. Islam, K. Mohror, B. Rountree, and M. Schulz. 2016. I/O Aware Power Shifting. In *IPDPS*. https://doi.org/10.1109/IPDPS.2016.15

[49] SLURM. [n. d.]. The SLURM Workload Manager. Online document, https://slurm.schedmd.com/.

[50] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. 2009. Koala: A Platform for OS-level Power Management. In *EuroSys*.

[51] Volker Springel. 2005. The cosmological simulation code GADGET-2. *Monthly notices of the royal astronomical society* 364, 4 (2005), 1105–1134.

[52] ExaOSR Team. [n. d.]. Key Challenges for Exascale OS/R. Online document, https://collab.mcs.anl.gov/display/exaosr/Challengesl.

[53] Xiaorui Wang, Ming Chen, and Xing Fu. 2010. MIMO Power Control for High-Density Servers in an Enclosure. *IEEE Transactions on Parallel and Distributed Systems* 21, 10 (2010).

[54] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2485922.2485974

[55] Wanghong Yuan and Klara Nahrstedt. 2003. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *SOSP*.

[56] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *ASPLOS*.