Penelope: Peer-to-peer Power Management

Tapan Srivastava
University of Chicago
Chicago, IL, USA
tapansriv@uchicago.edu

Huazhe Zhang
Meta
Freemont, CA, USA
huazhe@fb.com

Henry Hoffmann
University of Chicago
Chicago, IL, USA
hankhoffmann@uchicago.edu

ABSTRACT

Large scale distributed computing setups rely on power management systems to enforce tight power budgets. Existing systems use a central authority that redistributes excess power to power-hungry nodes. This central authority, however, is both a single point of failure and a critical bottleneck—especially at large scale. To address these limitations we propose *Penelope*, a distributed power management system which shifts power through peer-to-peer transactions, ensuring that it remains robust in faulty environments and at large scale. We implement *Penelope* and compare its achieved performance to *SLURM*, a centralized power manager, under a variety of power budgets. We find that under normal conditions *SLURM* and *Penelope* achieve almost equivalent performance; however in faulty environments, *Penelope* achieves 8–15% mean application performance gains over *SLURM*. At large scale and with increasing frequency of messages, *Penelope* maintains its performance in contrast to centralized approaches which degrade and become unusable.

CCS CONCEPTS

• **Hardware → Enterprise level and data centers power issues**.

KEYWORDS

Power Management, Adaptive Systems

ACM Reference Format:
Tapan Srivastava, Huazhe Zhang, and Henry Hoffmann. 2022. *Penelope*: Peer-to-peer Power Management. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545047

1 INTRODUCTION

One of the major problems facing the growth of exascale computing setups is operating under power constraints [5, 21, 30]. The United States Department of Energy identifies power as a key challenge, citing the need for exascale systems to stay within a tight power budget of 20-30 MW [3, 21, 30]. This tight power budget stems in part from the monetary cost of power delivery and cooling capacity [35]. Given that power is a limited resource, existing work illustrates how overprovisioning the system—having more nodes than can run simultaneously under the power budget—can greatly improve application performance [33].

Overprovisioning, however, creates the possibility of violating system-wide power budgets and causing serious damage to the cluster, so some systematic approach is needed to maintain these caps. A simple and widely-used method of system-wide powercap enforcement is a fair, static allocation of power. Each node receives an equal portion of the system-wide cap regardless of its usage. While this approach trivially enforces the power budget with no overhead, it fails to take advantage of the fact that a node will likely have different power consumption patterns over the course of its lifetime. Nodes running high CPU workloads will consume more power, whereas nodes running heavy I/O workloads or simply idling will consume less power, far below their assigned static cap.

Dynamic systems take advantage of the differences in power consumption across workloads to achieve greater application performance while maintaining the system-wide cap. Figure 1 illustrates how dynamic systems shift unused power from nodes operating under their cap to nodes operating at their cap. These transactions are zero-sum: one node increases its cap by the exact amount that is freed by the other node. As long as the initial assignment of node-level powercaps is valid, power shifting will not violate the system-wide power budget. The current state-of-the-art takes advantage of these patterns by launching a local decider on each node in the cluster and establishing a central server to coordinate all power management [17].

Each local decider operates in a classic feedback loop: it observes its environment, chooses an appropriate response, implements this choice, and repeats [23]. Every T seconds, it compares its actual power consumption (since the last iteration) against its assigned node-level cap. If a node is consuming less than its cap, the local decider sends the unused powercap to the server. If a node is consuming power close to its powercap, the decider requests power from the server. The server then collects excess power from those nodes operating under their cap and redistributes that excess to nodes operating near their cap. This approach maintains system-wide powercaps and allocates the total budget more efficiently in order to improve application performance.

However, the reliance on a central server has two key limitations:

• **Fault-Tolerance**—The central server is a single point of failure for the whole power management system. A node-level failure or network partition would fully halt any power shifting for the duration of the outage. In a large-scale, distributed environment where failures are common occurrences [3, 21, 30], a centralized approach cannot provide robust performance guarantees.

• **Scalability**—At high scale the number of clients simultaneously connecting to this server will cause performance degradation. The server will become a bottleneck, as the time to process all the

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545047

incoming events will, at some scale, surpass the T second period that local deciders wait between iterations. Likewise as T decreases, the problems at high scale amplify, resulting in an even greater message load on an already overburdened server, causing suboptimal power distribution and workload performance degradation.

To address these issues, we propose *Penelope*, a fault-tolerant and scalable peer-to-peer power management system. Rather than relying on a central server to coordinate power distribution, all power shifting occurs through ad hoc transactions between nodes. Each node holds a local cache of excess power, and instead of querying the server for power, power-hungry nodes randomly request power from other nodes in the system. Because *Penelope* does not have a central coordinator it limits the size of transactions to fairly spread out excess power in the system rather than having a large amount of excess power accumulate on one node while others starve. Additionally, *Penelope* implements a novel, distributed version of urgency, which allows unfairly disadvantaged nodes who are both power-hungry and operating below their initial assignment to bypass the transaction limit and quickly return to their initial powercap.

This approach has several benefits: (1) it does not have a single point of failure, as power can still be shifted even if any given node fails, (2) although the number of messages increases at scale, these will be split among a growing number of nodes, ensuring that the load on any one node is bounded, and (3) it does not require withholding node(s) from the computing setup in order to operate the central server.

We implement *Penelope* on a 21 node system and evaluate it against two alternate power management systems: *Fair* and *SLURM*. *Fair* statically allocates power evenly among all nodes in the system, and *SLURM* implements a power management system that relies on a central server to coordinate power distribution [43, 46]. All results are normalized to *Fair*. We find that:

- In nominal environments, *SLURM* and *Penelope* yield nearly the same mean performance gain over *Fair*, with *SLURM* achieving only a 1.8% speedup over *Penelope* on average.
- In faulty environments *Penelope* improves mean application performance by 8–15% over *SLURM*.
- In our simulations of large scale, *Penelope* quickly responds to queries and effectively shifts power with respect to scale or iteration frequency, while *SLURM* observes a linear increase in server response time and is unable to effectively shift power as frequency increases. We can estimate that at sufficient scale *SLURM's* server will be overburdened by the number of requests, slowing the rate of power distribution in the system.

The key contribution of this paper is *Penelope*, a fully distributed power management system. *Penelope's* design and reliance on peer-to-peer transactions allows it to be robust in faulty environments, a common problem in distributed systems, especially at large scale [3, 8, 11]. At large scale it will not bottleneck, and as local deciders iterate faster it will not degrade, unlike centralized approaches. And even without either of these conditions, *Penelope* performs equivalently to *SLURM*. *Penelope* efficiently manages power at low scale and without node faults, and it provides more robust performance guarantees than existing work in high scale and faulty systems.

The aim of this paper and Penelope as a system is to evaluate the feasibility and efficacy of peer-to-peer power management versus centralized approaches. Although studied in several other contexts, to the best of our knowledge this paper is the first to study peer-to-peer design in power management. Peer-to-peer designs present well-studied benefits over centralized approaches. The paper establishes that for dynamic power allocation a peer-to-peer approach successfully and efficiently allocates power, results in as good or better application performance versus centralized approaches, and has benefits in fault tolerance and scalability that derive from its decentralized design. Our goal was not to provide a survey of many possible designs, but rather to show that a simple, initial peer-to-peer approach could be successful. We feel that the simplicity of its design, coupled with the performance and design benefits, is a strength of this system. We hope that this leads to follow-on work in peer-to-peer power management.

2 BACKGROUND

This section proposes the concepts of *power assignment* and *power discovery* as identifying and distinguishing characteristics of different power management systems. It then covers case studies of a few such management systems in terms of these concepts to properly contextualize *Penelope*. However, it first discusses power management at a deeper level, covering the necessary constraints that power management systems must enforce while optimizing for application performance.

2.1 Power Management

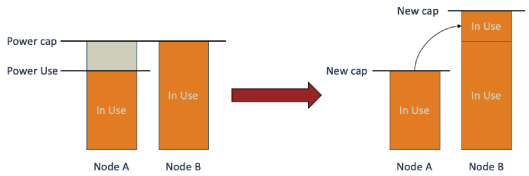

Figure 1: Shifting unused power to a power-hungry node.

Large computing setups, like exascale systems, will need to operate on a tight power budget between 20-30MW [3]. While the enforcement of this cap is vital to prevent damage and degradation, powercaps have a proportional, albeit non-linear relationship to application performance [19, 37]. Furthermore, each node has a maximum power setting, over which we could observe damage to the processor. Thus, we have two values to coordinate: *system-wide powercaps* and *node-level powercaps*. We have two constraints on these values: the sum of all node-level powercaps in the system must be equal to or less than the system-wide powercap, and each node-level powercap must be within a safe range for the node. As long as these two constraints are met, we prevent power-related damage and degradation.

While static allocation is the simplest way to enforce these constraints, dynamic approaches improve application performance by modifying node-level caps in real time to respond to workload behavior. However, a major challenge facing exascale computing is resilience, as tighter power budgets with more components will cause more faults [3]; static methods have no overhead, and so trivially overcome the challenges of fault-tolerance and scalability.

A power management system that can be robust in these conditions while improving application performance over static approaches would be greatly beneficial to the throughput and energy efficiency of large scale systems.

2.2 Power Assignment and Discovery

The processes of (1) locating and (2) redistributing excess power is the key contribution of dynamic power management over simpler static solutions. *Power discovery*—the location of excess power—and *power assignment*—the redistribution of excess power and the modification of node-level caps—can serve to uniquely identify different power allocation systems.

2.2.1 Power Assignment. We define power assignment as the mechanism by which node-level caps are set and potentially changed. Under static systems, this is clear: caps are set at system launch by some heuristic or central authority and then never altered.

Dynamic systems must define (1) how node-level caps are initially set and (2) how nodes modify their own cap. To the best of our knowledge, all prior dynamic power management systems establish a central authority. This authority has global knowledge and takes responsibility for the initial assignment of node-level caps as well as the collection and redistribution of excess power in the system.

2.2.2 Power Discovery. Power management systems do not need to fully utilize the system-wide powercap. In other words, if C_i is the cap of node i, and C_{system} is the system-wide cap, it is perfectly acceptable for $\Delta := C_{system} - \sum_{i=0}^{N} C_i > 0$. Dynamic systems aim to locate this excess power in the cluster. Power discovery is the means by which that location of excess power occurs. For example, centralized approaches have a global cache of excess power with a static address, meaning that every node knows where to find excess power.

2.3 Case Studies

We now survey three existing power management systems. *Fair* is a trivial static power capping solution which evenly splits a system-wide cap among all the nodes in a cluster. *SLURM* is a state-of-the-art job scheduler which implements a centralized power management system [43, 46]. *PoDD* is a hierarchical power management system designed for a specific class of workloads called coupled workloads [51].

2.3.1 Fair. *Fair* is a static power allocation system. Under *Fair*, each node is assigned the same cap: $\frac{C_{system}}{N}$, where N is the number of nodes. *Fair* does not attempt to locate or redistribute excess power, so it handles the notions of power assignment and power discovery trivially. In our experimental evaluation, *Fair* is used as a baseline model.

2.3.2 SLURM. *SLURM* is a state-of-the-art job scheduler for distributed systems with a dynamic power management system. *SLURM* assigns each node an initial cap of $\frac{C_{system}}{N}$. It then launches a local decider process on each node and a central server to handle all requests. Each local decider monitors local power consumption and sends information to the server based on a simple heuristic: if the current power consumption, P_i, is within ϵ of its node-level powercap, C_i, i.e. $P_i > C_i - \epsilon$, where ϵ is a fixed power margin, the local decider classifies the node as power-hungry, and it informs the server of its state. Otherwise, if $P_i \leq C_i - \epsilon$, then the node is classified as having excess power. In this case, the local decider reduces its cap, setting $C_i = P_i$, and sends its state and the excess power, $\Delta_i := (C_i - P_i)$, to the server.

We observe first that power discovery is handled by the server. The server is a global cache of all excess power, so power-hungry nodes know where any excess power is held. A centralized solution easily handles power discovery. Power assignment is similarly centralized—the central server holds the excess and then takes a percentage of the total excess and gives it to each requesting node. Power shifting from one node to another must be proxied through the server.

2.3.3 PoDD. *PoDD* is a hierarchical power management system for coupled workloads: workloads in exascale systems that run simultaneously rather than serially. These workloads are dependent on one another, so these pairs of applications are only as fast as their slowest member. As a result, *PoDD* observes that it is optimal for both applications in the couple to finish at the same time, as the runtime of the pair is most important.

PoDD approaches power management in a hierarchical way. It runs each application in the couple for a few iterations, learns the optimal initial node-level powercaps, and assigns these—a centralized process. It then launches a centralized power management system to coordinate node-level power shifting similarly to *SLURM*.

PoDD has a centralized solution to power discovery. Excess power is held and redistributed by the central server. *PoDD* has a hierarchical solution to power assignment. It performs a centralized, top-level powercap assignment and then allows for local refinement.

3 *PENELOPE* DESIGN AND IMPLEMENTATION

Penelope is a distributed power management system which relies on peer-to-peer transactions to shift power between nodes. Whereas existing work relies on a central server to coordinate power shifting, *Penelope* overcomes the challenges of fault-tolerance and scalability in power management by splitting the responsibilities of a central power management server among all the nodes in the cluster.

There are two components to *Penelope* on each node: a local decider and a local power pool. The local decider classifies a node as either being power-hungry or having excess. If power-hungry, it queries a power pool searching for excess power. Otherwise, it adds the excess power it has to its local power pool. The local power pool is a cache of excess power and acts as a server, giving power from its cache to power-hungry nodes.

We define a *transaction* as an exchange of power between a local decider and a power pool. Power housed on a power pool has already been freed and transactions are atomic, so we can ensure that no transaction increases the system's total power usage beyond the system-wide cap. Additionally, local deciders have information about safe power ranges for the node on which they are running and can ensure that nodes do not exceed that safe range. Because power shifts through these atomic transactions we can make sure

that *Penelope* meets both necessary requirements of a power management system: (1) it enforces system-wide powercaps, and (2) it makes sure nodes operate within safe power ranges.

We additionally integrate into *Penelope* a modified idea of *urgency* proposed by Zhang and Hoffmann [50]. We discuss the implementation of this concept in subsequent sections, but here we describe it at a high-level. The intuition behind urgency is as follows: over the uptime of a system, a node, call it node A, may lower its cap dramatically because it did not need power. Now suppose, either due to a changed workload or to a different phase of its current workload, node A suddenly becomes power-hungry. It is capped far below its initial assignment, and if there is not any excess power in the system this node will be unfairly throttled due to its previous power needs while other nodes are operating above their initial cap, having accessed the excess power that was previously released by node A. To provide a means of recourse for node A to at least return to its initial power level, we say that any node that (1) *Penelope* classifies as power-hungry and (2) has a powercap below its initial cap has an *urgent* state, and if it sends power requests we say that these are *urgent requests*. Non-urgent requests are subject to limitations, where the power pool restricts the maximum amount of power that the request can receive—an idea that will be discussed at length in Section 3.2. Urgent requests, however, bypass this restriction and are allowed access to as much excess power as they can locate until the urgent node reaches its initial cap. Additionally, if there is no excess power in the system, urgent requests induce nodes to release power down to their initial cap, even if they are power-hungry, thus artificially creating excess power which the urgent node can access. These mechanisms allow urgent nodes to rapidly return to their initial cap, ensuring that if there is no excess power in a system, one node will not be unfairly throttled.

Zhang and Hoffmann use offline profiles to determine optimal power assignments specifically for coupled workloads and use this as the threshold for urgency rather than initial power cap [50]. We adapt this idea for both a generalized application setup where we do not have knowledge *a priori* of any workload profiles and for a distributed environment.

3.1 Local Decider

The local decider is initialized with a few parameters: a pointer to the local power pool that is running on the same node, the initial powercap for the node, and a power margin ϵ.

As illustrated in Algorithm 1, the local decider operates in a control loop at distinct time steps t, with T seconds between each subsequent time step. The local decider first reads the average power dissipated since the previous time step $t-1$. The local decider compares this read power P_t to C_t, the powercap at time step t. If P_t is more than ϵ below C_t, the decider classifies the node as having excess power. If P is within the power margin ϵ of C_t, the decider classifies the node as power-hungry.

If the decider classifies the node as having excess, it calculates the amount of excess it has (Δ), lowers its cap for the next time step by Δ, and adds the corresponding amount to the local power pool. Adding this excess to the local pool exposes it to other nodes, so the local decider must lower its cap prior to adding to the local power pool to maintain the the system-wide cap.

Algorithm 1: Local Decider Pseudocode

```
Function PenelopeLocalDecider(pool, initialCap, ϵ):
    C_0 = initialCap;
    t = 0 ;
    while True do
        P = getPowerReading();
        if P < C_t − ϵ then
            // Current power reading is under cap
            Δ = C_t − P;
            C_{t+1} = C_t − Δ;
            Pool = Pool + Δ;
        else if P > C_t − ϵ then
            if Pool > 0 then
                // getMaxSize defined in Algorithm 2
                Δ = min(Pool, getMaxSize(Pool));
                Pool = Pool − Δ;
                C_{t+1} = C_t + Δ;
            else
                server = chooseRandomNode();
                if C_t < initialCap then
                    urgency = 1;
                    α = initialCap − C_t;
                    Δ = sendUrgentRequest(server, α);
                else
                    urgency = 0;
                    Δ = sendRequest(server);
                C_{t+1} = C_t + Δ;
            if urgency == 0 ∧ localUrgency == 1 then
                Δ = C_{t+1} − initialCap;
                C_{t+1} = C_{t+1} − Δ;
                Pool = Pool + Δ;
                localUrgency = 0;
        t = t + 1;
```

If the decider classifies the node as power-hungry, it first checks the local power pool before querying other nodes. This allows nodes to discover excess power whether it resides locally or externally.

If the local power pool yields nothing, the decider prepares to query a different power pool. This peer-to-peer query is how power assignment occurs in *Penelope*. Local deciders receive power through transactions with power pools, so power assignment occurs in a distributed, peer-to-peer manner. It chooses which node to query at random. This random query is how *Penelope* handles power discovery. Without prior knowledge of where power resides, local deciders choose a node uniformly at random to locate power.

Before sending a request, the local decider determines if the node is in an urgent state, defined as being both power-hungry and operating below the initial cap. If urgent, the decider calculates how much power is necessary for its cap to reach its initial cap and sends this value to a power pool in an urgent request. If not urgent, the decider simply sends a standard request. The power pool handles urgent requests differently, which will be discussed at greater length in Section 3.2.

If the decider receives any power in the transaction, it increases its cap C_{t+1} by the corresponding amount. If it receives nothing, we simply set $C_{t+1} = C_t$. Finally, the decider checks its *localUrgency* flag. This flag is set when the local power pool receives an urgent request. If this flag is set, and the node is not itself already in an urgent state, the decider lowers its cap to the initial assigned cap,

and adds the excess to the power pool. As discussed earlier, this mechanism frees up power that urgent nodes can access, allowing them to reach their initial cap more quickly.

3.2 Power Pool

Algorithm 2: Power Pool Pseudocode

```
Function getMaxSize(Pool):
    size = TEN_PERCENT * Pool ;
    if size > UPPER_LIMIT then
        return UPPER_LIMIT;
    else if size < LOWER_LIMIT then
        return LOWER_LIMIT;
    else
        return size;
Function PowerPool(Pool):
    while True do
        request = getIncomingRequest();
        if request.urgency == True then
            α = getNecessaryPower(request);
            Δ = min(Pool, α);
        else
            maxSize = getMaxSize(Pool);
            Δ = min(Pool, maxSize);
        Pool = Pool − Δ;
        response = Δ;
        replyToRequest(request, response);
        localUrgency = request.urgency;
```

In addition to the local decider, each node contains a local power pool. The power pool acts as both a local cache of excess power as well as a server, responding to requests for power from local deciders operating on other nodes. The power pool operates in a simple loop, illustrated in pseudocode in Algorithm 2. When the power pool receives a request, it first checks the urgency attached.

If the request is urgent, the requesting node will have also sent a value α which represents how much power is needed for that node to return to its initial cap. For urgent requests, the power pool tries to give α in response unless the size of the pool is too small, in which case it will give all excess power it has stored.

If the request is not urgent, the power pool uses a simple heuristic to determine the amount of power it will respond to this request with. The algorithm calculates 10% of the total size of the pool, capped above by UPPER_LIMIT and below by LOWER_LIMIT as shown in 2. This provides a hard upper and lower bound for extreme sized power pools while allowing for a gradual scaling of transaction size when the pool has moderate size. Our system sets UPPER_LIMIT to 30 watts and LOWER_LIMIT to 1 watt. So if the pool size is over 300 it returns 30, and if below 10 it returns 1.

The power pool limits the rate of power distribution to ensure roughly equal distribution and to prevent *power oscillation*. If transactions are too large, it is possible that one node unfairly hoards all excess power in the system. Limiting the rate allows multiple requesters to be served, each receiving a smaller, equal amount of power per transaction.

Large transaction sizes can also cause power oscillation. If too much power is given in one transaction, the power-hungry node will increase its cap by a large amount and may not be able to use up its entire cap by the next time step. In the next iteration the node will be classified as having excess and will lower its cap. In the following iteration it will again be power-hungry and request power. If it receives too much power again this process will repeat. When factoring in the changing needs of the workload, this can cause the powercap on a node to oscillate wildly. Limiting the amount of power receivable in a transaction can dampen this oscillation, allowing for the node to gradually increase its cap. This limit scales with the size of the pool, so more power can be given out if there is more excess, and the limits at 1 and 30 ensure that we always give a nonzero amount of excess power and that even if the pool becomes massive the size of transactions will be bounded.

At this point, based on urgency, the power pool has calculated Δ, the amount of power it will respond to the request with. It reduces the size of its pool by this amount and replies to the request with Δ. Finally, it sets the *localUrgency* flag based on the urgency of the request. As noted in Section 3.1, this flag induces the local decider to release power down to its initial cap.

The power pool and local decider encompass the entirety of Penelope's high level design. The local decider makes real-time choices based on the current power consumption of the node. If more power is needed, it queries first its local cache, and then into the system to search for available power. The power pool acts as a server to field these incoming requests and process them, and it provides special privileges to nodes who urgently need power to allow them to quickly return to their initial state.

3.3 Limitations and Assumptions

Penelope uses Intel's Running Average Power Limiting (RAPL) scheme to read and manipulate power and powercaps [12]. However, *Penelope* only requires an interface through which power can be read and node-level powercaps can be set. Therefore, *Penelope* easily be adapted to work with any power capping interface.

The local decider and local power pool both manipulate the amount of excess power available. As a result, some care is needed to ensure that changes to this value are atomic, otherwise system-wide caps could be violated. *Penelope* guarantees this through the use of a simple lock, but any form of synchronization would suffice.

Finally, we acknowledge that while *Penelope* is more robust in faulty environments and at scale, centralized approaches will converge faster than peer-to-peer power management systems at low scale or when the central server is not a bottleneck because they have a global server with total knowledge. *Penelope* effectively shifts power without requiring a central server or a separate node to host that server, and in the absence of faults or scale provides comparable application performance speedup to *SLURM*, data which is discussed in Section 4. However, there may be certain systems with more specific guarantees in terms of integrity or scale than we assume here, and on those systems a centralized approach may be feasible and more efficient than a peer-to-peer approach.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We use the NAS Parallel Benchmark (NPB) suite version 3.4 [4]. This includes 10 applications, from which we omit Integer Sort (IS). The applications compile with different classes (A–F) corresponding

to different test sizes. All benchmarks are compiled to class D to get an appropriately long test runtime. IS does not compile past level C, and at level C finishes significantly faster than the other applications (under five seconds, whereas each other application takes at least 40 seconds and all but one take at two minutes).

This benchmark contains a set of programs that are intended to test the performance of parallel supercomputers. The applications contain 5 kernels and 3 pseudo-applications, as well as benchmarks for unstructured adaptive mesh and parallel I/O. These applications have varying runtimes with different resource usage and power needs. We test every unique combination of these 9 applications, yielding 36 pairs. Our setup divides the cluster in half, running one application on the first half and the other on the second. We define the runtime of an experiment as the time necessary for all nodes to complete their workloads. All applications run with 48 threads per node, the maximum number of virtual cores on our test servers.

Additionally, we adapt the idea of prioritization proposed by Zhang and Hoffmann [50] into a model of urgency as discussed in Section 3, and we implement *SLURM* along with this centralized version of urgency. Power-hungry nodes operating below their initial cap send urgent requests to the central server. The server gives power greedily to these urgent nodes until they reach their initial cap. If the urgent cannot reach their initial cap, the central server indicates this to other, non-urgent nodes and induces them to release power down to their initial cap. Adapting this concept for a distributed system is non-trivial, so we implement *SLURM* in this way in order to compare *Penelope's* distributed version of urgency with the original, centralized version.

Our test system has 21 nodes. 20 of these are client nodes that run actual applications, and 1 is used to host the server for *SLURM*. *Penelope* and *Fair* use only the 20 client nodes. Each node is a dual-socket server, with 2 Intel Skylake Xeon Gold 6126 CPUs. The nominal clockspeed is 2.60 GHz. Each node has 256 GB of RAM divided between dual memory controllers. All nodes support Intel RAPL technology, which we utilize to set powercaps and read power from hardware. Each processor has 12 physical cores with hyperthreading, allowing for 48 virtual cores.

We measure runtime as the time necessary for all applications to complete, and we use $1/runtime$ as our performance metric. We test each combination of applications under *Fair*, *SLURM*, and *Penelope*, and we normalize the performance of *SLURM* and *Penelope* to *Fair*.

We first evaluate the overhead of *Penelope*. We then test *Fair*, *SLURM*, and *Penelope* under nominal conditions, specifically in the absence of node failures or heavy network traffic. Next we test these three systems under failing environments, when we induce a failure to *SLURM's* central server to observe the impact of node-level failures. Finally, we simulate the impact of large scale on *SLURM* and *Penelope*. We define two metrics, *power redistribution time* and *turnaround time*. We plot these metrics versus the frequency at which local deciders operate and versus simulated scale for both *SLURM* and *Penelope*. We omit *Fair* because it does not redistribute power nor cause any network traffic.

Under both normal and failing conditions, we compute the performance under *Fair*, *SLURM*, and *Penelope* for each pair of applications under an initial powercap. We normalize the performance of *Penelope* and *SLURM* to *Fair*, and we plot the geometric mean for *Penelope* and *SLURM* across all pairs of applications under a given initial cap, as well as the geometric mean across all application pairs and initial powercaps.

4.2 *Penelope* Overhead

We first measure the overhead *Penelope* incurs on each node. To evaluate this, we measure the runtime of each workload in the NAS Parallel Benchmark [4] on a single node under a static cap. We then run all the workloads again, but this time launching *Penelope* on this node. This is a one node system, so no power is being shared between nodes. We define overhead as the percent slowdown of running with *Penelope* versus under a static cap. We observe an average of 1.3% overhead across all workloads, indicating that *Penelope* minimally impacts workload runtimes. We note that all subsequent results include this overhead.

4.3 Performance Under Nominal Conditions

We start by running all 3 systems on our 21-node system in a fault-free environment. This means that during the test there are no node-level failures or severe network latency, so the integrity of any node, including the one running *SLURM's* server, is not tested or compromised. All 3 systems begin by dividing the system-wide cap evenly among all nodes in our system. We thus test 5 different initial settings: 60, 70, 80, 90, and 100W per socket, with 2 sockets per node. Each system enforces the system-wide cap constraint, but the application performance varies.

Figure 2 shows that *SLURM* on average outperforms *Penelope* by only 1.8% and never outperforms *Penelope* by more than 3% across all choices of initial powercap. Because it is centralized *SLURM* can make decisions based on total information, so at small scale we expect it to have an advantage over a distributed approach. However, this advantage is clearly limited.

Penelope addresses the limitations of centralized approaches under failing conditions or at large scale, but even in the absence of either of these factors we see that *Penelope* provides nearly identical mean application performance to *SLURM*. We believe this shows that *Penelope* can be used as a general purpose power management system.

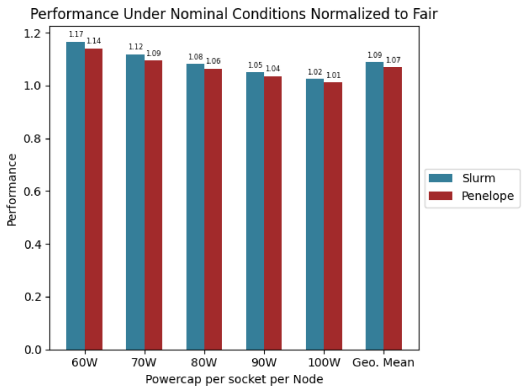

Figure 2: Performance Under Nominal Conditions

4.4 Performance with Faulty Power Management

We next run the same test as above, but we induce a failure to *SLURM's* server partway through execution for each application pair to observe the impact of a node-failure on *SLURM's* performance. As expected we see a sharp degradation, with *Penelope* observing an 8–15% improvement in mean application performance over *SLURM*. Without the server node, even though every client is properly functioning, there is no central authority that can shift power. Effectively, this means that the assignment of powercaps at the time of failure becomes a static assignment. However, these node-level powercaps are uneven across the cluster, which could throttle application performance worse than evenly divided assignments. Under our experimental setup, only one application runs on every node during a single test, but in a generalized environment multiple workloads would run on the same hardware back to back. If these workloads have drastically different power consumption patterns, a failure to *SLURM's* server could throttle application performance even more than is indicated by our data.

Figure 3 shows the geometric mean of performance for each system across all application pairs for each choice of initial cap. *SLURM* performs on average worse than even the trivial solution, *Fair*. Additionally, even though the server has died, the local decider running on each client node is still functioning, which requires overhead. In a real system, a fault to the server node would result in a similar situation, where the overhead of running local decider clients is still paid, but without any of the benefits of power shifting, on top of the overhead of requiring an additional node to run the coordinating server. In contrast, *Penelope* is not significantly perturbed by a client-node failure because *Penelope* relies exclusively on peer-to-peer transactions to shift power and does not introduce a single point of failure. As a result it is resilient to node-level faults. After a critical mass of nodes have failed, *Penelope's* performance will begin to degrade, but this would be true for any dynamic power management system, centralized or distributed.

A peer-to-peer design inherently benefits from increased fault-tolerance and scalability due to the lack of central authority and distributed nature of work in the system. Centralized designs do not share these inherent benefits, and we have shown the impact that this lack of implicit fault-tolerance can have on application performance.

While centralized systems can use fallback servers to improve their fault-tolerance, our goal is to evaluate a peer-to-peer design in contrast to a centralized design. We show that a fully centralized design is vulnerable to coordinator failures while a peer-to-peer design remains effective. We acknowledge that we do not explore all possible failure scenarios; however, we note that we induce a failure in each combination of applications, providing some variance. We leave a comprehensive study of fault tolerance in centralized systems for future work.

4.5 Scalability Analysis

The scalability of either *SLURM* or *Penelope* is dictated not just by the number of nodes in the system but also by the frequency at which local deciders iterate. Both *SLURM* and *Penelope* are implemented such that their local deciders iterate once every second, but this frequency is constrained in large part by how quickly RAPL can enforce an assigned node-level cap. RAPL already converges on average in under 0.5 seconds [48], and as power management becomes increasingly important we expect RAPL and competing technologies to reduce their overhead, so there is every reason to expect that local decider frequency will increase in the future.

Because we do not have sufficient hardware, we simulate *SLURM* and *Penelope* running at large scale. To implement this, we run multiple local deciders, up to the number of cores to make sure they can all run truly in parallel, on a single physical node simultaneously. All simulated deciders can share with each other, whether or not they reside on the same physical node. Local deciders no longer interact with hardware, and instead use curated profiles of power consumption over time for each application in the NPB benchmark. Our simulation uses real power profiles but allows individual cores to act as nodes would in a real system. We have a 45 node cluster with 48 cores per node (with hyperthreading). We withhold 1 node to operate the *SLURM* server. Each *SLURM* decider requires 1 core, but each *Penelope* instance utilizes two threads to simultaneously run a power pool and local decider. We can simulate 24 *Penelope* instances per node, so on our 44 compute nodes, we can simulate 1056 total nodes.

Our setup, aside from using NPB application profiles, is analogous to the prior experiments: we iterate over all possible pairs of applications, running one on the first half the cluster and the other on the second half. We modify the profiles to look at a shorter continuous set of power readings that occur around when one application completes, allowing us to observe how our systems behave when a large amount of power enters the system, as power should move from the now idle nodes to those still running.

We define two metrics in our scaling analysis: *power redistribution time* and *turnaround time*.

Power Redistribution Time: This is the time necessary for some percentage of excess power to be redistributed to power-hungry nodes. Because we are unable to directly measure application performance, we use power redistribution time to approximate the efficacy of a power management system, as this metric represents the speed at which power is shifted in the system.

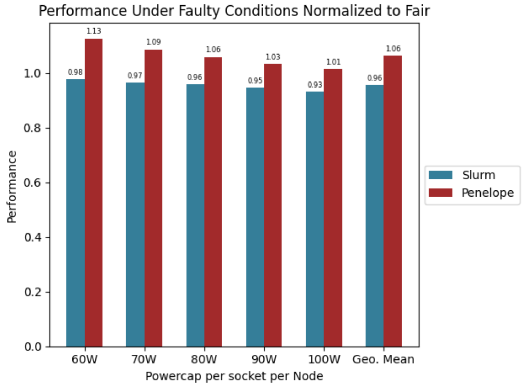

Figure 3: Performance Under Faulty Conditions

Turnaround Time: This is the average time a local decider spends waiting for a response from a power pool. For *SLURM* this is the server's average response time. For *Penelope* this is the average time needed to complete a transaction in the system. This metric allows us to quantify potential bottleneck effects at scale.

We run two tests. First, we fix scale at the maximum we can simulate reliably, 1056 nodes, and increase the frequency at which local deciders iterate. Second, we fix the frequency at 1 iteration per second and vary the scale, from 44 nodes to 1056. In all experiments, we compute the value in question under all 36 pairs of applications and plot the distribution of that value over these 36 combinations.

We note that in order to mitigate the increased power oscillation at scale because of the significantly higher amount of excess power available, we modify *SLURM's* rate limiting scheme to account for scale. We discuss power oscillation and its ramifications at greater length in Section 3.2

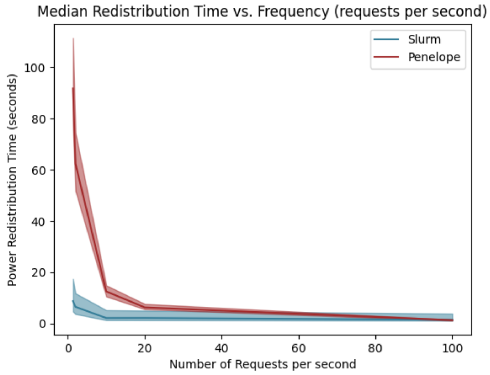

Figure 4: Median redistribution time (time to shift 50% of available power) versus local decider frequency

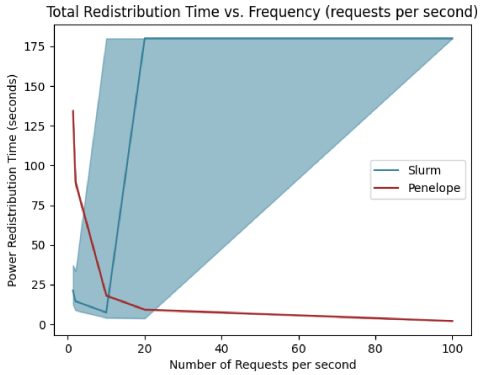

Figure 5: Total redistribution time (time to shift 100% of available power) versus local decider frequency

4.5.1 Power Redistribution Time versus Frequency and versus Scale. We first observe the impact of increased frequency and increased scale on power redistribution time.

In Figure 5, near 20 requests per second, *SLURM's* total redistribution time shoots up. This occurs because at this frequency *SLURM's* server begins dropping packets and is unable to redistribute all the excess power, so we define its total redistribution time as the total runtime of the experiment. However, if we look at the median redistribution time—time to redistribute 50% of the available power—in Figure 4, we see that *Penelope* rapidly improves its redistribution time and converges to that of *SLURM* as frequency increases. A relatively small increase in frequency causes a major reduction in redistribution time, both in total and median, for *Penelope*, whereas *SLURM's* server degrades as frequency increases. *Penelope* remains effective as frequency increases, as it quickly and fully redistributes available power in the system.

We present in Figure 6 power redistribution versus scale for both systems. From Figure 6, we observe that the trends for both systems are unchanged as scale increases from 44 nodes to 1056. What this indicates first is that at 1056 nodes with a one second period, *SLURM* does not degrade; however, *Penelope* does not either. As scale increases, we see that the gap in redistribution time remains essentially unchanged, and as we have seen on real world data *Penelope* and *SLURM* have extremely similar mean application performances on 20 nodes when running real workloads.

4.5.2 Turnaround Time versus Frequency and versus Scale. We see that frequency has a significant effect on turnaround time for *SLURM*. Figure 7 shows how *SLURM's* turnaround time approaches nearly 25ms before leveling off and slightly declining—which occurs at the point where the server begins dropping packets. Dropping packets will absolutely prevent turnaround time from increasing at the same pace. We note that the standard deviation increases as frequency increases, even as mean turnaround time remains flatter.

When varying scale, Figure 8 shows how the response time for the server is sharply increasing, but is still operating in the range of tens of milliseconds, while clients are sending messages every second. Because the turnaround time is still a small percentage of the overall period, *SLURM's* power redistribution time did not vary versus scale.

However, the turnaround time stands to continue to increase as scale increases, and once the turnaround time becomes a more

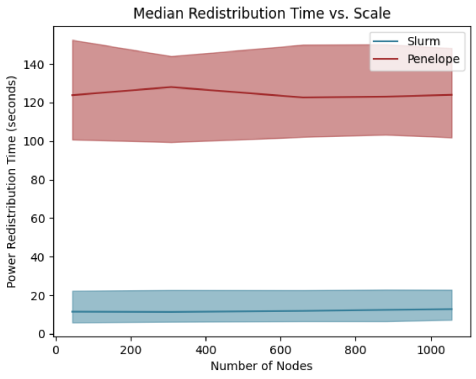

Figure 6: Median redistribution time (time to shift 50% of available power) versus scale

significant percentage of 1 second, *SLURM* will begin to observe degradation in power redistribution time. We further measure the average time needed to process a request by the server, which was about 80-100 microseconds. The server processes requests serially, so we can extrapolate that even at 80 microseconds, a system of 12,500 nodes sending messages every second would force the server to take 1 second to process all incoming requests, causing compounding delays in the processing for each local decider.

As the iterating frequency increases, a smaller number of clients would be able to overwhelm the server in this same way. At 1056 nodes, a frequency of about 11.8 iterations per second would be enough to cause *SLURM's* turnaround time to exceed the period for the local decider. As scale increases, the frequency necessary to reach this point will only decrease.

As frequency increases, *SLURM's* server degrades both in redistribution time and turnaround time, while *Penelope's* remains robust and efficient. At large simulated scale, *Penelope* is able to shift power just as well as at low scale, and has consistent and low turnaround times. *SLURM* has asymptotically worse turnaround times, and we can easily extrapolate the scale thresholds that would turn the central server into a bottleneck for the whole system, greatly

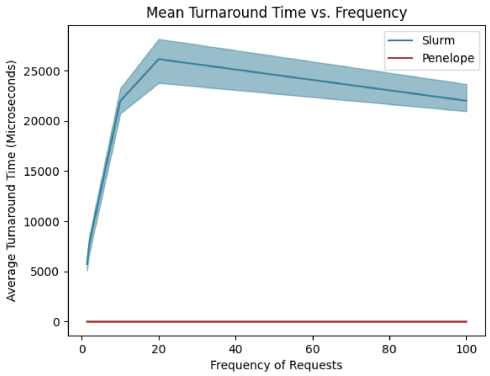

Figure 7: Mean turnaround time (time spent waiting for a server response) versus local decider frequency

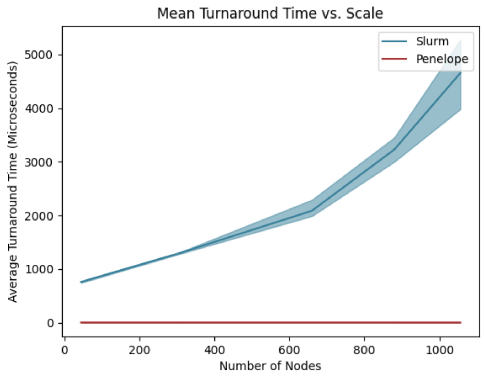

Figure 8: Mean turnaround time (time spent waiting for a server response) versus scale

increasing its power redistribution time. The trend in *SLURM's* turnaround time illustrates that it is on an untenable trajectory.

5 RELATED WORK

There is a great deal of work developing systems that control power, either through hardware or software. Early work controls one component, like DRAM, disk, or DVFS [15, 28, 29], while later work coordinates multiple components [10, 13, 14] or arbitrary components to optimize performance by utilizing offline profiles [24–26]. Intel's RAPL tool and PUPiL provide a hardware and flexible software approach respectively to read power and set powercaps [12, 49]. More recent work expands the breadth of power-capping work, specializing in GPUs, modulating uncore frequency, or focusing on certain systems like the Xen hypervisor [7, 21, 31]. However, to the best of our knowledge, these approaches focus on power management on a single node rather than on a larger cluster.

There has been a great deal of work concerning job scheduling in HPC systems [2, 20, 36]. More recent work focuses on scheduling for exascale [1] as well as on the effect of power and network factors on application performance [32]. These works show the continued importance of power management systems, but to the best of our knowledge these prior approaches rely on a centralized power management system.

More recently, because it is more efficient for large systems to be hardware overprovisioned [33, 41], existing work proposes a scheme to manage power and efficiently allocate resources and nodes to workloads in overprovisioned systems [22, 34, 38, 40]. The number of power management systems motivated the creation of a simulator for different power scheduling strategies [16], and a recently survey investigated these questions within data visualization on supercomputers [6].

Fault-tolerance in HPC and exascale systems is a major problem. Canal et al. survey the problem and some existing solutions, such as predicting node failures in real-time [8, 11]. While these works aim to mitigate the impact of node failures in large distributed systems, *Penelope's* distributed design grants fault-tolerance inherently, ensuring that in the face of these node failures the system can continue to shift power without any extra cost.

Works most similar to *Penelope* are those which aim to dynamically reallocate node-level powercaps to improve performance. Ellsworth et al. propose a centralized solution to power shifting [17, 18]. Other work detects low-power I/O phases and leveraging those phases to shift power via a central server towards applications in computation phases [42]. Some related work specializes in power shifting on coupled workloads. These are workloads which previously ran serially, but now run in parallel and communicate over a network instead of through a file interface [9, 27, 45]. Zhang and Hoffmann propose management schemes specifically targeting coupled workloads [50, 51]. While these systems all dynamically shift power to achieve better application performance, their reliance on a central authority provides key limitations which *Penelope* addresses through its distributed design.

Finally, we note that distributed power discovery is a hard problem, and we cite related work that has to do with other distributed discovery problems and their solutions. Zarrin et al. survey existing approaches to general resource discovery on distributed systems

[47], and certain approaches target resource discovery in edge systems and large-scale IoT systems [39, 44]. Although a major component for *Penelope* is the discovery of excess power, the need for urgency; i.e. the need to take a resource away from nodes operating without excess, is an added factor for power management system that is not addressed by prior resource discovery algorithms. To the best of our knowledge, *Penelope*'s distributed urgency is a novel contribution.

6 CONCLUSION

With the increased use of large-scale distributed computing setups, power management systems that can both maintain the integrity of system-wide powercaps and improve mean application performance are increasingly valuable. Static allocation methods such as *Fair* are simple and widely used, but they fail to account for the variable resource demands of workloads. Dynamic systems aim to shift excess power in the system to power-hungry nodes, enforcing system-wide caps while improving application performance. To the best of our knowledge, all existing dynamic systems rely on a central server to coordinate this power shifting, which means these systems have a single point of failure and will develop a bottleneck at scale. We present *Penelope*, a distributed power management system which relies instead on peer-to-peer transactions to shift power. *Penelope* does not have a single point of failure, as excess power is stored on each client node rather than on a central server. At large scale, a central server will become a bottleneck, delaying the iteration timing of local deciders running on client nodes and causing the system to more slowly shift power. *Penelope's* power redistribution time does not vary with scale and neither does the average response time of its servers. This is because, although under the same load at scale, this load is split across all nodes, ensuring that no single power pool is overburdened with requests and guaranteeing that no single area of the network will be flooded with messages. As frequency increases at scale the central server becomes overburdened, as its mean server response time sharply increases, whereas *Penelope* remains robust. In real world systems, *Penelope* performs nearly as well as *SLURM* under nominal conditions, and outperforms *SLURM* by 8–15% in situations where there is a fault to the server node. *Penelope* remains robust in faulty environments and at scale and performs well in the absence of these conditions, indicating that it can be used as a general purpose power management system for computing setups of varying size and integrity. We believe that this work illustrates the value and feasibility of relying exclusively on peer-to-peer power management, and we hope that this inspires further work into such approaches.

ACKNOWLEDGMENTS

This work was supported by NSF (grants CCF-2119184, CNS-1956180, CNS-1952050, CCF-1823032, CNS-1764039), a DOE Early Career Award (grant DESC0014195 0003) , and ARO (grant W911NF1920321).

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

REFERENCES

[1] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I. IngÃ³lfsson, Joseph Koning, Tapasya Patki, Thomas R.W. Scogland, Becky Springmeyer, and Michela Taufer. 2020. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems* 110 (2020), 202–213. https://doi.org/10.1016/j.future.2020.04.006

[2] Peter E Bailey, Aniruddha Marathe, David K Lowenthal, Barry Rountree, and Martin Schulz. 2015. Finding the limits of power-constrained application performance. In *SC*. ACM, Austin Texas, 1–12. https://doi.org/10.1145/2807591.2807637

[3] Pete Beckman, Ron Brightwell, Maya Gokhale, Bronis R. de Supinski, Steven Hofmeyr, Sriram Krishnamoorthy, Mike Lang, Barney Maccabe, John Shalf, and Marc Snir. 2012. Exascale Operating Systems and Runtime Software Report. (12 2012). https://doi.org/10.2172/1471119

[4] NAS Parallel Benchmark. [n.d.]. https://www.nas.nasa.gov/publications/npb.html.

[5] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. 2008. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA IPTO, Tech. Rep* 15 (2008).

[6] Stephanie Brink, Matthew Larsen, Hank Childs, and Barry Rountree. 2021. Evaluating adaptive and predictive power management strategies for optimizing visualization performance on supercomputers. *Parallel Comput.* 104-105 (2021), 102782. https://doi.org/10.1016/j.parco.2021.102782

[7] Rolando Brondolin, Marco Arnaboldi, and Marco D. Santambrogio. 2020. Power Consumption Management under a Low-Level Performance Constraint in the Xen Hypervisor. *SIGBED Rev.* 17, 1 (July 2020), 42–48. https://doi.org/10.1145/3412821.3412828

[8] Ramon Canal, Carles Hernandez, Rafa Tornero, Alessandro Cilardo, Giuseppe Massari, Federico Reghenzani, William Fornaciari, Marina Zapater, David Atienza, Ariel Oleksiak, Wojciech Piundefinedtek, and Jaume Abella. 2020. Predictive Reliability and Fault Management in Exascale Systems: State of the Art and Perspectives. *ACM Comput. Surv.* 53, 5, Article 95 (Sept. 2020), 32 pages. https://doi.org/10.1145/3403956

[9] J Chen, Alok Choudhary, S Feldman, B Hendrickson, CR Johnson, R Mount, V Sarkar, V White, and D Williams. 2013. *Synergistic Challenges in Data-Intensive Science and Exascale Computing: DOE ASCAC Data Subcommittee Report.* Department of Energy Office of Science. Type: Report.

[10] Jian Chen and Lizy Kurian John. 2011. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *ICS '11*. ACM Press, Tucson, Arizona, USA, 192–201. https://doi.org/10.1145/1995896.1995927

[11] Anwesha Das, Frank Mueller, and Barry Rountree. 2020. Aarohi: Making Real-Time Node Failure Prediction Feasible. In *2020 IPDPS*. 1092–1101. https://doi.org/10.1109/IPDPS47924.2020.00115

[12] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. 2010. RAPL: Memory power estimation and capping. In *2010 ACM/IEEE ISLPED*. 189–194. https://doi.org/10.1145/1840845.1840883

[13] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. 2012. CoScale: Coordinating CPU and memory system DVFS in server systems. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 143–154. https://doi.org/10.1109/MICRO.2012.22

[14] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. 2012. MultiScale: memory system DVFS with multiple memory controllers. In *ISLPED '12*. ACM Press, Redondo Beach, California, USA, 297–302. https://doi.org/10.1145/2333660.2333727

[15] Bruno Diniz, Dorgival Guedes, Wagner Meira Jr, and Ricardo Bianchini. 2007. Limiting the power consumption of main memory. In *ISCA '07*. ACM Press, San Diego, California, USA, 290–301. https://doi.org/10.1145/1250662.1250699

[16] Daniel Ellsworth, Tapasya Patki, Martin Schulz, Barry Rountree, and Allen Malony. 2017. Simulating Power Scheduling at Scale *(E2SC'17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3149412.3149414

[17] Daniel A Ellsworth, Allen D Malony, Barry Rountree, and Martin Schulz. 2015. Dynamic power sharing for higher job throughput. In *SC'15*. IEEE, 1–11. https://doi.org/10.1145/2807591.2807643

[18] Daniel A Ellsworth, Allen D Malony, Barry Rountree, and Martin Schulz. 2015. POW: System-wide Dynamic Reallocation of Limited Power in HPC. In *HPDC*. ACM, Portland Oregon USA, 145–148. https://doi.org/10.1145/2749246.2749277

[19] Keiichiro Fukazawa, Masatsugu Ueda, Mutsumi Aoyagi, Tomonori Tsuhata, Kyohei Yoshida, Aruta Uehara, Masakazu Kuze, Yuichi Inadomi, and Koji Inoue. 2014. Power consumption evaluation of an mhd simulation with cpu power capping. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 612–617. https://doi.org/10.1109/CCGrid.2014.47

[20] Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power tuning HPC jobs on power-constrained systems. In *2016 PACT*. IEEE, 179–190. https://doi.org/10.1145/2967938.2967961

[21] Neha Gholkar, Frank Mueller, and Barry Rountree. 2019. Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 27, 23 pages. https://doi.org/10.1145/3295500.3356150

[22] Neha Gholkar, Frank Mueller, Barry Rountree, and Aniruddha Marathe. 2018. PShifter: feedback-based dynamic power shifting within HPC jobs for performance. In *HPDC*. ACM, Tempe Arizona, 106–117. https://doi.org/10.1145/3208040.

3208047
[23] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E Miller, Sabrina M Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, et al. 2012. Self-aware computing in the Angstrom processor. In *DAC '12*. ACM Press, 259–264. https://doi.org/10.1145/2228360.2228409
[24] Henry Hoffmann and Martina Maggio. 2014. PCP: A Generalized Approach to Optimizing Performance Under Power Constraints through Resource Management. In *ICAC '14*. 241–247.
[25] Connor Imes and Henry Hoffmann. 2016. Bard: A unified framework for managing soft timing and power constraints. In *AMOS*. IEEE, 31–38. https://doi.org/10.1109/SAMOS.2016.7818328
[26] Connor Imes, Steven Hofmeyr, and Henry Hoffmann. 2018. Energy-efficient Application Resource Scheduling using Machine Learning Classifiers. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, Eugene OR USA, 1–11. https://doi.org/10.1145/3225058.3225088
[27] David E Keyes, Lois C McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, et al. 2013. Multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications* 27, 1 (2013), 4–83. https://doi.org/10.1177/1094342012468181 arXiv:https://doi.org/10.1177/1094342012468181
[28] Mohammed G Khatib and Zvonimir Bandic. 2016. PCAP: Performance-aware Power Capping for the Disk Drive in the Cloud. In *FAST*. USENIX Association, Santa Clara, CA, 227–240. https://www.usenix.org/conference/fast16/technical-sessions/presentation/khatib
[29] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. 2008. Power capping: a prelude to power shifting. *Cluster Computing* 11, 2 (June 2008), 183–195. https://doi.org/10.1007/s10586-007-0045-4
[30] Matthias Maiterth, Torsten Wilde, David Lowenthal, Barry Rountree, Martin Schulz, Jonathan Eastep, and Dieter Kranzlmüller. 2017. Power Aware High Performance Computing: Challenges and Opportunities for Application and System Developers — Survey Tutorial. In *HPCS*. 3–10. https://doi.org/10.1109/HPCS.2017.11
[31] Tapasya Patki, Zachary Frye, Harsh Bhatia, Francesco Di Natale, James Glosli, Helgi Ingolfsson, and Barry Rountree. 2019. Comparing GPU Power and Frequency Capping: A Case Study with the MuMMI Workflow. In *WORKS*. 31–39. https://doi.org/10.1109/WORKS49585.2019.00009
[32] Tapasya Patki, Zachary Frye, Harsh Bhatia, Francesco Di Natale, James Glosli, Helgi Ingolfsson, and Barry Rountree. 2019. Comparing GPU Power and Frequency Capping: A Case Study with the MuMMI Workflow. In *WORKS*. IEEE, 31–39.
[33] Tapasya Patki, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R De Supinski. 2013. Exploring hardware overprovisioning in power-constrained, high performance computing. In *ICS*. ACM Press, 173–182. https://doi.org/10.1145/2464996.2465009
[34] Tapasya Patki, David K Lowenthal, Anjana Sasidharan, Matthias Maiterth, Barry L Rountree, Martin Schulz, and Bronis R De Supinski. 2015. Practical Resource Management in Power-Constrained, High Performance Computing. In *HPDC*. ACM, Portland Oregon USA, 121–132. https://doi.org/10.1145/2749246.2749262
[35] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No "power" struggles: coordinated multi-level power management for the data center. *ACM SIGARCH Computer Architecture News* 36, 1 (March 2008), 48–59. https://doi.org/10.1145/1353534.1346289
[36] Haris Ribic and Yu David Liu. 2016. AEQUITAS: Coordinated Energy Management Across Parallel Applications. In *ICS*. ACM, Istanbul Turkey, 1–12. https://doi.org/10.1145/2925426.2926260
[37] Barry Rountree, Dong H Ahn, Bronis R De Supinski, David K Lowenthal, and Martin Schulz. 2012. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 947–953. https://doi.org/10.1109/IPDPSW.2012.116
[38] Ryuichi Sakamoto, Tapasya Patki, Thang Cao, Masaaki Kondo, Koji Inoue, Masatsugu Ueda, Daniel Ellsworth, Barry Rountree, and Martin Schulz. 2018. Analyzing Resource Trade-offs in Hardware Overprovisioned Supercomputers. In *2018 IPDPS*. 526–535. https://doi.org/10.1109/IPDPS.2018.00062
[39] Ahmed Salem, Theodoros Salonidis, Nirmit Desai, and Tamer Nadeem. 2017. Kinaara: Distributed discovery and allocation of mobile edge resources. In *MASS*. IEEE, 153–161. https://doi.org/10.1109/MASS.2017.10
[40] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant Kale. 2014. Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget. In *SC '14*. IEEE, 807–818. https://doi.org/10.1109/SC.2014.71
[41] Osman Sarood, Akhil Langer, Laxmikant Kalé, Barry Rountree, and Bronis De Supinski. 2013. Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems. In *CLUSTER*. IEEE, 1–8. https://doi.org/10.1109/CLUSTER.2013.6702684
[42] Lee Savoie, David K. Lowenthal, Bronis R. De Supinski, Tanzima Islam, Kathryn Mohror, Barry Rountree, and Martin Schulz. 2016. I/O Aware Power Shifting. In *IPDPS*. IEEE, Chicago, IL, 740–749. https://doi.org/10.1109/IPDPS.2016.15
[43] SLURM. [n.d.]. The SLURM Workload Manager. https://slurm.schedmd.com.
[44] Giacomo Tanganelli, Carlo Vallati, and Enzo Mingozzi. 2017. Edge-Centric Distributed Discovery and Access in the Internet of Things. *IEEE Internet of Things Journal* 5, 1 (2017), 425–438. https://doi.org/10.1109/JIOT.2017.2767381
[45] ExaOSR Team. [n.d.]. Key Challenges for Exascale OS/R. https://collab.cels.anl.gov/display/exaosr/Challenges.
[46] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.
[47] Javad Zarrin, Rui L Aguiar, and João Paulo Barraca. 2018. Resource discovery for distributed computing systems: A comprehensive survey. *J. Parallel and Distrib. Comput.* 113 (2018), 127–166. https://doi.org/10.1016/j.jpdc.2017.11.010
[48] Huazhe Zhang. [n.d.]. A quantitative evaluation of the RAPL power control system. ([n. d.]).
[49] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. *ACM SIGPLAN Notices* 51, 4 (June 2016), 545–559. https://doi.org/10.1145/2954679.2872375
[50] Huazhe Zhang and Henry Hoffmann. 2018. Performance & Energy Tradeoffs for Dependent Distributed Applications Under System-wide Power Caps. In *ICPP*. ACM, Eugene OR USA, 1–11. https://doi.org/10.1145/3225058.3225098
[51] Huazhe Zhang and Henry Hoffmann. 2019. *PoDD*: power-capping dependent distributed applications. In *SC*. ACM, Denver Colorado, 1–23. https://doi.org/10.1145/3295500.3356174