Leveraging Stored Energy for Handling Power Emergencies in Aggressively Provisioned Datacenters

Sriram Govindan
Datacenter Compute Infrastructure Team, Microsoft,
Redmond, WA, USA
srgovin@microsoft.com

Di Wang   Anand Sivasubramaniam
Bhuvan Urgaonkar
Department of Computer Science and Engineering,
The Pennsylvania State University, PA 16802
{diw5108,anand,bhuvan}@cse.psu.edu

Abstract

Datacenters spend $10-25 per watt in provisioning their power infrastructure, regardless of the watts actually consumed. Since peak power needs arise rarely, provisioning power infrastructure for them can be expensive. One can, thus, aggressively under-provision infrastructure assuming that simultaneous peak draw across all equipment will happen rarely. The resulting non-zero probability of emergency events where power needs exceed provisioned capacity, however small, mandates graceful reaction mechanisms to cap the power draw instead of leaving it to disruptive circuit breakers/fuses. Existing strategies for power capping use temporal knobs local to a server that throttle the rate of execution (using power modes), and/or spatial knobs that redirect/migrate excess load to regions of the datacenter with more power headroom. We show these mechanisms to have performance degrading ramifications, and propose an entirely orthogonal solution that leverages existing UPS batteries to temporarily augment the utility supply during emergencies. We build an experimental prototype to demonstrate such power capping on a cluster of 8 servers, each with an individual battery, and implement several online heuristics in the context of different datacenter workloads to evaluate their effectiveness in handling power emergencies. We show that: (i) our battery-based solution can handle emergencies of short duration on its own, (ii) supplement existing reaction mechanisms to enhance their efficacy for longer emergencies, and (iii) battery even provide feasible options when other knobs do not suffice.

Categories and Subject Descriptors C.0 [*Computer Systems Organization*]: General

General Terms Design, Experimentation, Measurement, Performance

Keywords UPS, Batteries, Data center, Peak power, Stored energy, Provisioning, Cap-ex, Peak shaving

1. Introduction

Datacenters incur capital expenditure (cap-ex) of $10-25 per watt of provisioned power capacity, regardless of whether this watt is actually consumed [4, 23]. The power delivery infrastructure amounts to several million dollars in cap-ex [5], contributing over a third of the amortized monthly datacenter costs [4]. Although not all of this may go towards the power delivery for IT equipment, since there is other supporting infrastructure including HVACs, fans, etc., the IT-related power network is still a substantial component that warrants meticulous planning and provisioning. Provisioning for the theoretical peak (using face-plate ratings of equipment) that may never happen, or even the occasional peak (which requires all equipment to simultaneously exercise their maximum draw), can prove very expensive. A study of power consumption in a Google datacenter, depicted in Figure 1, reiterates this observation showing a very low probability of reaching close to the provisioned peak (probability of exceeding 90% of the potential peak is less than 1%), with a profile that is highly skewed around the average. Aggressive provisioning of power infrastructure can thus yield substantial cap-ex savings. The goal of this paper is to deal with the rare ("black swan") power emergencies (when the draw exceeds provisioned power) that arise in such aggressively provisioned datacenters in a seamless and graceful fashion.

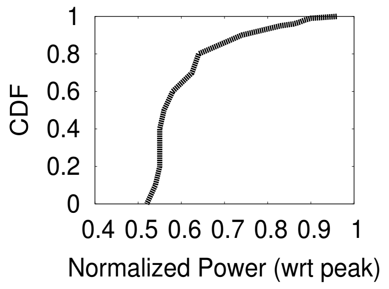


Figure 1. Rack Power profile of a Google datacenter [12].

The temporal and spatial load variations, and resulting power draw, in a datacenter offer a strong reason for leveraging statistical multiplexing to under-provision the power infrastructure. Circuit-breakers are used to ensure equipment safety and handle power spikes lasting several seconds [9], abruptly cutting off power supply upon an overdraw. *Can we employ alternate solutions that handle emergencies more gracefully by removing/easing such spikes, thereby reducing the probability of hitting these safety limits?* Circuit-breakers would continue to be used, but only as the last line of defense. Also, datacenters have power upgrade cycles to handle growing IT load. The probability of emergencies can increase as we get closer to the next upgrade, and such solutions can help relieve disruptions during these periods. Gracefully handling emergencies, while meeting performance SLAs, would allow more aggressive power provisioning [1].

Software mechanisms for power capping have been examined to some depth [12, 13, 16, 19, 25, 37, 38, 47]. Broadly, we can classify existing solutions into two categories. The first set of knobs are local to servers, where temporal load-shifting/demand-shaping is used to control the rate of workload execution (and thereby control power consumption which grows with utilization). Apart from scheduling the load (temporally spacing it out using CPU scheduling), hardware power mode control - clock throttling states (including server shutdown [7, 36]) and/or dynamic voltage-frequency modulation states (DVFS) - is also used to control the rate of workload execution (with different trade-offs between performance and power) [16, 47]. The second set of knobs exploits spatial non-uniformity in power profiles across (groups of) servers, and dynamically redirects/migrates load to regions of the datacenter with more headroom. While local knobs are agile and effective for handling short-lived emergencies, prolonged throttling can impact performance. While spatial migration is better for longer emergencies, there can be overheads during and after (load imbalance on some nodes, loss of locality, etc.) migration, in addition to the need for headroom elsewhere in the datacenter. The load may also not be "move-able" to other servers in some cases. Regardless, both these strategies can have substantial performance repercussions.

This problem is analogous to supply-demand mismatch issues on the electrical grid, and the solutions above are analogous to "demand-response" mechanisms in that context. However, one solution - energy storage - that is used in normal grids to amplify the efficacy of demand-response has been little explored in the datacenter context. Energy storage can be used to (i) deal with short/small power spikes without even requiring other reaction mechanisms (temporal/spatial) which have performance consequences, (ii) supplement existing mechanisms to improve their efficacy in meeting application SLAs, and (iii) offer remedial solutions, even if they are temporary, when other mechanisms may not find feasible options to meet application SLAs. Further, unlike in grids where energy storage may be a costly proposition, datacenters already have storage built-in in the form of UPS units to handle power disruptions - *can we tap into these for emergency handling without impinging on the availability mandates?*

There are different choices for UPS placement - from a centralized (usually redundant) configuration, to a distributed version at each server as in Google datacenters [18], as well as at intermediate levels (e.g., per-rack) similar to the ones in Microsoft [33] and Facebook [11] datacenters. Server-level UPS, when incorporated with the power supply, can help eliminate double-conversion needs and the associated energy losses. We assume a server-level UPS unit, although many of our ideas will also apply to other configurations . In general, UPS batteries help reduce/cap power draw at and above the level in the power hierarchy where they are placed. When using batteries beyond their original role, we must ensure that the datacenter's power availability (upon utility outages), as well as the batteries' normal lifetime (3-5 years for lead-acid) are not compromised. Since UPS units are mainly transitionary devices to temporarily handle load until diesel generators are started (which takes around 15-30 seconds), we are able to utilize them for a couple of minutes for our purposes while still leaving enough charge for availability purposes. To facilitate aggressive power provisioning, this paper makes the following contributions:

- We present an offline theoretical framework combining all knobs - batteries, power states, migration (both within and across datacenter clusters) - to find the performance optimal way of handling an emergency for a given level of power underprovisioning. This serves as a reference point for how well we can do and offers insights on what knobs/combinations work for given workload-emergency combinations.
- Since emergencies are inherently unpredictable (when? how severe? how long?), we develop several on-line heuristics employing existing temporal and spatial knobs, and combinations thereof. We also introduce battery-based emergency handling techniques, accommodating different drain rates (slow and fast), and combine them with existing knobs in interesting ways.
- We develop an experimental prototype of 8 servers with individual UPS units, and implement seven control heuristics. We examine emergencies in the context of several representative datacenter workloads with unique characteristics: (i) TPC-W [40] and Specjbb [41] server-based workloads with emergencies arising from load spikes (e.g., flash crowds), (ii) a MapReduce [22] application where the map phase introduces power spikes substantially higher than the reduce phase, (iii) a streaming media server where handling the power spike caused by a surge of new connections leads to jitters in existing streams, and (iv) a multiprogrammed GPU + Virus scan application pair that is tied to a particular set of servers (migration is not an option) which when run together introduces a high power draw (requiring them to be temporally spaced out).
- We consider different degrees of under-provisioning, and show that battery by itself can sustain short emergencies (around 10-20 minutes on our prototype). Longer emergencies (30 minutes or higher), require migration since sustained local throttling hurts performance. On the other hand, performance overheads of migration makes it less attractive for shorter emergencies. Our battery solution, in conjunction with migration, provides a seamless bridge across the spectrum of these duration, and is able to reduce the performance impact of these intermediate duration. In most cases, battery-based heuristics, with these knobs, help us get within 10-20% of the theoretical bounds.

2. Related Work

Tighter Power Provisioning: Statistical multiplexing-based overbooking/underprovisioning of resources, a widely used yield management technique in different domains, including IT resources in datacenters [43, 46], is now being suggested for the power infrastructure. This ranges from components within servers [13, 25], to groups/ensembles of servers and other equipment [19, 26, 35, 39, 48]. The basic underlying idea is to exploit the low likelihood of simultaneous peak power needs of all components/servers [12, 13, 25]. Such provisioning requires agile reactive techniques to ensure the power consumption stays below capacity to allow safe operation with satisfactory workload performance. These techniques include device power state control [10, 14, 16, 32, 37, 38, 47, 49] and workload scheduling or migration within or even outside the datacenter [2, 17, 19, 30, 34, 45].

Battery-based power management: While battery management has been studied in the mobile/embedded domains [15, 50, 51] (e.g., drain rate adjustment for longevity), their use in datacenters has been limited to mere transition devices during utility failure. Recent work has looked at using UPS batteries to reduce electricity operational costs [20, 44], but their role in tighter provisioning for reducing cap-ex is entirely novel. To our knowledge, we are the first to explore the use of stored energy (UPS batteries) for tighter provisioning.

[1] As Taleb says in his book [42]: "Black Swans being unpredictable, we need to adjust to their existence rather than naively try to predict them." Even if the datacenter power profile may follow bell-curve behavior as in Figure 1, the extreme consequences of ignoring the "highly improbable" tail makes emergency handling a critical problem.

3. Problem Details and Solutions

3.1 The Problem

Underprovisioning of power (equipment) does not have to be restricted to the highest level in the hierarchy. As we aggressively push such underprovisioning to equipment at lower levels, there is scope for additional cap-ex savings. However, going deeper lessens the potential for exploiting statistical multiplexing of power demands across underlying servers. The higher burstiness (variance) at deeper levels can result in higher probability for emergencies/violations. A more rigorous cost-benefit analysis is necessary to find out how much to underprovision at each level of the hierarchy. We consider this beyond the scope of this paper, and focus here on the following consequent problem: given a level c aggressively underprovisioned at P_{budget} to accommodate N servers under it, how do we cap the net power draw of these servers to P_{budget} at all times? We will consider different values of P_{budget} relative to the maximum possible peak draw by the N servers, henceforth referred to as a *cluster* (c).

If $P_j(t)$ represents power draw of server j at time t, then underprovisioning power capacity exploits the statistical property $Probability(\sum_{j=1}^{N} P_j(t) > P_{budget}) \leq \epsilon, \forall t$. Just as in other domains where underprovisioning is used, there are consequences to be carefully traded-off against its benefits. Since ϵ is non-zero, despite being small, there would inevitably be situations when the aggregate power draw of the N servers cannot be safely accommodated by the infrastructure at level c. Since ϵ may not be identified precisely (Section 1), we need reactive mechanisms to deal with the power overdraw situations. Normally, during such an episode of power overdraw, circuit-breakers/fuses at level c would kick in to ensure safety from fire hazards and overdraw-induced equipment burnout. Such workload-oblivious reaction mechanisms are highly undesirable in the datacenter context since they can lead to lost computation and inconsistent states or even impact IT equipment reliability (e.g., hard disk failures [52]). Therefore, when underprovisioning, it is essential to additionally employ workload-aware reaction mechanisms to operate gracefully under such emergencies, while continuing to leave circuit-breakers/fuses as the last line of defense. Such mechanisms should also be agile to quickly control the power draw. Fortunately, existing work has shown that such agile power capping can be realized well within the time limits (typically sub-second to a few seconds) imposed by circuit-breakers [19, 25, 48].

3.2 Current Solution Strategies

We classify existing workload-aware reaction mechanisms into two broad categories: temporal and spatial. Temporal mechanisms adjust the rate of resource usage within a server. Since the rate of resource usage (particularly the CPU) affects power consumption, it can be used as a knob for demand response. Two common temporal mechanisms include (i) scheduling to defer some of the peak load to a non-emergency period, and (ii) employing power states - dynamic voltage and frequency scaling (DVFS), clock throttling, and even sleep/shutdown states.

Spatial mechanisms direct/migrate excess load to servers and/or regions of the datacenter that have headroom in their power budgets. Here again, we consider two broad strategies: (i) load redirection/migration/consolidation to one or more servers within the hierarchy at level c, allowing some servers in c to be shut/slowed down. (ii) load movement/migration to one or more servers elsewhere in the datacenter (outside c), with headroom for the increased load. Many existing techniques [3, 6, 27, 36] on load balancing/unbalancing for energy/power reduction fall into this category. Such load movement could be achieved by (a) request redirection as in many network-based services [36], (b) "fault-tolerant" applications that detect server unavailability and automatically rebalance themselves amongst the available servers [22], and (c) dynamic process/VM migration [8].

All these mechanisms can have performance consequences with different pros and cons in the options that they offer. Temporal mechanisms are local to a server, and do not require headroom elsewhere in the datacenter. These mechanisms are also quite agile, since power mode control and scheduling decisions can be performed at a fine time granularity with the effects materializing quickly. The downside to these mechanisms is their performance degrading effect, especially when the applications do not provide sufficient slack in their offered load and SLA specifications. Finally, since the load is not moved/migrated elsewhere, the duration of emergency is dictated by the application load, making these mechanisms less desirable for handling long duration emergencies. Temporal dampening mechanisms have to be continuously applied as long as the high load persists.

On the other hand, spatial techniques can work well with long periods of high load since they can move demands to regions of the datacenter with sufficient headroom. However, these techniques have their limitations: (i) they are less agile than temporal mechanisms, implying their inadequacy in handling emergencies on their own. Reacting to an emergency can take a long time, during which a temporal mechanism must be relied upon to cap the power draw, making them only suitable for long-duration emergencies; (ii) migration can be expensive, depending on the application state that needs to be migrated. This can not only impact performance during migration (which can take a few minutes) and after migration (due to loss of locality), but can also temporarily increase power draw of concerned servers; (iii) they require headroom elsewhere in the datacenter which may not always be available (and sometimes undesirable due to administrative boundaries); (iv) not all applications are amenable to migration, since they may require resources (a graphics application requiring a GPU card, virus scan requiring the local disk), on specific servers.

In general, a combination of these techniques can be used depending on the duration of emergency and SLAs of involved workloads. Further, these decisions may not necessarily be static (i.e., determined completely at the beginning of the emergency), with possibly an online algorithm that starts with a temporal mechanism which optimistically looks for a purely local solution, and then adaptively migrates load if needed. *Regardless, all such algorithms have performance degrading consequences either by throttling using power states/shutdown, and/or due to the overheads of load migration (during the movement, loss of locality after the movement, etc.).*

3.3 Our Proposal: Employing Batteries

We now present an entirely orthogonal and novel solution of using battery-based energy storage for handling emergencies that can avoid/reduce the performance consequences of existing techniques. Batteries are analogous to buffers that are used in networks to smoothen out spikes and alleviate mismatches between workload demand and capacity availability. Rather than additional batteries, we propose to use UPS batteries that are already present in datacenters for handling power outages. While most current datacenters employ centralized UPS, we consider a distributed server-level UPS, similar to that in Google datacenters [18]. Our ideas are also applicable to a shared cluster-level UPS (with N times the capacity of a server-level UPS), where we only need to enforce how much draw comes from utility versus from these UPSes. Evaluating such a configuration is part of our future work.

Battery Capacity: We denote by t_b the duration for which a UPS can sustain the associated server's peak power needs. Typically, provisioned UPS battery capacities are for t_b ranging up to a few minutes. There can also be redundancy in the UPS units to accommodate UPS failures, implying higher gross battery capacity. In addition to availability criteria, these capacities are also determined by the discrete units of capacities that different vendors provide. For instance, many APC battery offerings are in discrete capacities of 4, 8, 16, 24 minutes, all rated at a certain peak power draw. In our setup, we consider a UPS with $t_b = 4$ minutes, which is in line with (in fact, at the low-end of) existing capacities. There are cost-benefit trade-offs with investing in higher battery capacities for additional cap-ex savings achieved by aggressive power infrastructure underprovisioning. A preliminary study suggests that the costs of extra battery capacity is worthwhile (see section 7.6), though a detailed study of such economics is beyond the scope of this paper.

Sourcing Power: While a server could potentially draw current simultaneously from the UPS and the power line (one source for each of its dual power supplies), we do not consider this option since our experimental platform does not allow this. Consequently, at any time, a server either draws all of its current from its UPS or from the power line. It is possible to meet a power budget over an extended period of time (beyond what a single battery offers), since servers could take turns sourcing power from their batteries at different times. E.g., if reducing the power draw of 1 server using its UPS meets the overall power budget for 2 minutes, then a cluster of 8 servers can stay within the power budget for up to 16 minutes.

Battery Runtime: Batteries (including lead-acid used in most UPS units) are characterized by their runtime chart as approximated by Peukert's Law [28], which shows the time to drain a certain capacity for different power draws. Runtime of $t_b = 4$ minutes is at the maximum power draw of a server, and the duration of possible battery draw is much higher for lower power draws. For instance, a draw of 50% of the peak, allows the battery to last 11 minutes. We will exploit this property to force a lower draw for longer battery operation.

Availability and Lifetime Concerns: Since we are using UPS battery beyond its normal purpose (handling power outages), we need to ensure that we do not compromise on datacenter availability. In recent work [21], we have modeled datacenter availability as a function of UPS battery capacities for different placement strategies across the layers of the power delivery hierarchy. Across these strategies, we have shown that leaving a 2 minute residual capacity suffices to ensure a high availability of up to five nines. Consequently, in all our experiments in this paper, we always leave 2 minutes of reserve capacity (to operate at full load) in each UPS battery.

Another concern is the lifetime (reliability) of the battery itself: the normal lead-acid battery typically lasts for about 3-5 years [31]. Since charge-discharge cycles can reduce its effective lifetime, we have to ensure that our approach does not result in replacement of batteries sooner than its expected lifetime. We leverage of our recent work [20] to address this concern, where we have conducted an extensive analysis of the impact of battery discharges cycles on its lifetime. Using this analysis, we find that one can handle these relatively rare power emergencies well within the expected battery lifetime.

Efficiency and Additional Power of Batteries: Each charge-discharge cycle has energy loss, which we find experimentally to be 28% of the overall energy drawn in the worst case. Since such loss is restricted to "rare" emergencies, we do not expect it to significantly impact operational costs. However, battery charging itself adds to instantaneous power draw and this must be considered for adhering to the budget. There are different strategies for dealing with this issue - restricting charging to non-emergency periods, slow-charging so that the instantaneous draw is not significant, and/or compensating with aggressive spatio-temporal workload throttling mechanisms. In this work, we employ the first strategy.

4. Handling Emergencies: An Offline Theoretical Framework

Having discussed the pros and cons of various strategies, we now turn our attention to combining their best features.

Is migration (spatial) of one or more workloads on the N servers under c even an option? There are different situations where a spatial knob may not be applicable, even if workloads have sufficient slack in their SLAs. First, there must be headroom elsewhere in the datacenter to accommodate the migrated load, which may not happen if the entire datacenter is highly utilized. Second, certain workloads may not be amenable to migration: consider a workload that needs a resource only available locally, e.g., a graphics application requiring a GPU card.

Even if migration is an option, can local (temporal + battery) knobs alone handle the emergency while meeting application SLAs? If yes, there is no reason to look for headroom elsewhere in the datacenter and incur additional migration costs. The agility of temporal knobs makes them more attractive over spatial knobs in this case.

When local (temporal + battery) knobs do not suffice because of the emergency duration and stringency of application SLAs, which workloads should we migrate and at what time? These decisions should be determined based on what impacts the application SLAs the least while meeting the power budget. We develop a simple framework to conduct such decision-making. Let us denote the remaining time for an application to complete [2], at the beginning of an emergency, as t_f. Let t_m denote the time (from the emergency beginning) when the application is migrated, if at all. t_m would be a function of migration overhead: $t_m = 0$ implies immediate migration (for relatively stateless applications with little migration overhead), and a larger t_m implies local knobs will be employed as much as possible until they become infeasible and migration is necessary. If t_m is larger than emergency duration, then it implies that local knobs suffice to handle the emergency without requiring any migration. To determine t_m, we denote the relative speed at which the application runs on the destination server(s) with respect to the source server(s) as s_r (the slowdown). We assume that there is sufficient power capacity at the destination, and the slowdown can be approximated using a simple slowdown factor s_r. However, the application runs until time t_m locally before migration, and would have been subjected to local temporal knobs which would have also slowed it down. This slowdown, is dependent on t_m itself (i.e., local knobs affect execution time) and we denote it as $s_l(t_m)$. Since $t_f - t_m * s_l(t_m)$ is the remaining time for the application at migration time t_m, its remaining execution time on the remote server(s) needs to be scaled as $(t_f - t_m * s_l(t_m)) * s_r$. The resulting total execution time of the application with migration (as depicted in Figure 2), which needs to be minimized, can now be expressed as

$$\min_{t_m=0}^{t_m=t_f} (t_m + (t_f - t_m * s_l(t_m)) * s_r)$$

The relative impact of the slowdown with local (temporal + battery) and migration (spatial) knobs is captured by $s_l(t_m)$ and s_r (both lie between 0 and 1). Of these two, s_r is mainly application governed by its locality properties, since we assume the destination has enough headroom to run it without any power-related dampening. However, s_l is crucially dependent on t_m, and how the local

[2] Even for applications without an explicit notion of remaining time (e.g., a Web server running forever), an equivalent framework, say based on execution rate, can be developed.

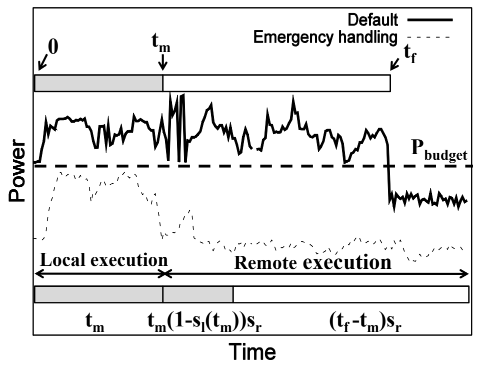


Figure 2. Timeline — above for default, and below for emergency handling with local and spatial knobs. Migration happens at t_m instantaneously in this illustration. Note the elongation in execution time due to s_l and s_r. Resulting lower power draw for the emergency between 0 and t_f, compared to default is also shown.

knobs were employed to meet the power budget over t_m. This, in turn, leads us to the following optimization problem: given a t_m, how should the local (battery + temporal) knobs be used to meet performance SLAs?

We use a generic metric R, whose minimization corresponds to meeting the application's SLA, to cast this optimization problem. R is general enough to capture a wide range of application metrics, e.g., response time (as in TPC-W used in our evaluation), reciprocal of throughput (in transaction-oriented applications such as Specjbb), time-to-finish (for long running applications such as Map-Reduce and virus scan), rate of playback discontinuities for a streaming media server, etc. We view the duration t_m as being divided into W equal-sized intervals. We use R_{ij} to denote average response time offered by server j in time interval i. We can now express the problem of minimizing R (i.e., minimizing $s_l(t_m)$) for a given t_m using purely local knobs (battery + power states)). This can then be iterated over different values of t_m between 0 to W to determine when to migrate.

Minimizing R Using Only Local Knobs: Let the battery on server j have upto B_j joules of energy that it can safely provide for this emergency, accounting for any residual energy that needs to be maintained for availability. Each server $j = 1, \ldots, N$, can operate during an interval within this emergency in a particular power mode (server off, DVFS and Clock throttling states), ordered as D_0 where the server is off, to D_d which is the highest power consuming state and best in performance. We denote the intensity of a workload during this emergency as L, discretized for the spectrum of intensities between the minimum and maximum for a given workload as $L_1, \ldots, L_l$. For instance, in Specjbb, the transaction rate specifies the intensity, and we can histogram this rate between a minimum and maximum into l buckets.

The response time $R_{ij}(L_{ij}, D_{ij})$ offered by server j during interval i depends on the load L_{ij} that is imposed on that server during i and its power mode D_{ij}, where $L_{ij} \in [L_1, \ldots, L_l]$ and $D_{ij} \in [D_0, D_1, \ldots, D_d]$. The power consumption of this server during i can be specified as $P_{ij}(L_{ij}, Dij)$, and this can also be calculated a priori and made available to the optimizer for different (L, D) combinations. We can then phrase our objective function of minimizing average response time over W by employing purely local (battery + power state) knobs as:

$$minimize \qquad \sum_{i=1}^{W}\sum_{j=1}^{N} R_{ij}(L_{ij}, Dij).$$

Let b_{ij} denote whether server j sources its power needs in the i-th interval from its battery ($b_{ij} = 1$) or the power line ($b_{ij} = 0$). Since we cannot drain more than B_j for this peak, we have: $\Sigma_{i=1}^{W} b_{ij} * P_{ij}(L_{ij}, Dij) \leq B_j, \forall j$. The resulting total power draw on the line, which has to adhere to the specified budget P_{budget}, is given by: $\Sigma_{j=1}^{N} \overline{b_{ij}} * P_{ij}(L_{ij}, Dij) \leq P_{budget}, \forall i$.

5. Online Heuristics

Our theoretical framework is impossible to use in practice since it requires a priori knowledge of the emergency duration and intensity. Even if such knowledge were available, it may be computationally prohibitive. *However, we still use it as a baseline for comparison with the practical solutions we develop next.* We refer to the solution offered by our framework above as `Opt`, and the solution it offers solely using local knobs as `Opt-local`.

	Heuristics	Description
Local to server	Throt	Use only power states
	BattFast+Throt	Drain batt. first before throttling
	BattSlow+Throt	Drain batt. slowly while throttling
Local to cluster	cMig	Mig. within cluster and turnoff servers
	BattSlow+Throt+cMig	Delay cMig as far as possible
Across datacenter	dMig	Mig. to elsewhere in datacenter
	BattSlow+Throt+dMig	Delay dMig as far as possible

Table 1. Summary of our online heuristics

We consider seven online heuristics (Table 1) based on whether the knob of adhering to P_{budget} under c is (a) local to a server, (b) local to the cluster of servers under c, or (c) pertains to the entire datacenter. As before, we assume a priori knowledge of performance ($R_{ij}(l, d)$) and power ($P_{ij}(l, d)$) for different loads (l) and power states (d) of a server for the application. Further, even though the above theoretical framework allows a different power state (d) for each server at a given time, our heuristics only consider a restricted version that employs the same state across all N servers at any time. The resulting detrimental effects (if any), would materialize in our results.

Heuristics Local to a Server: One would like to preferably use knobs local to a server, namely battery, power state modulation, and/or temporal deferring of the load to deal with the emergency, because of their agility and less disruption in the datacenter. Our first set of 3 heuristics - `Throt`, `BattFast+Throt`, and `BattSlow+Throt` - employ only such local knobs. `Throt` employs only power state modulation (DVFS and/or clock throttling) and temporal scheduling, and is representative of the power throttling mechanisms available today. With knowledge of the power consumption in different states for the current load, this heuristic picks the least performance impacting power state for all N servers, ensuring adherence to P_{budget}. `BattFast+Throt` and `BattSlow+Throt` supplement temporal knobs with battery to reduce performance impact, and thereby sustain longer emergency handling. The two differ in battery drain rate. `BattFast+Throt` is relatively optimistic about the emergency duration, and drains the battery fully before resorting to power mode knobs, thereby not requiring the latter if the battery can sustain the entire load for short emergencies. `BattSlow+Throt` is conservative, and tries to prolong the battery usage. Recall that we are using the same power state d across all N servers at any time, and our experimental setup allows the drain rate from the battery at only server granularity (i.e., a single server cannot source partly from battery and partly from the power line, which is possible in dual power supply servers). Consequently, we require $(N-1) * P(l,d) \leq P_{budget}$, since the remaining server, by sourcing its power from battery, would help reduce the overall draw from power line to adhere to the budget. Hence, in BattSlow+Throt, a single server draws power from the battery at a power state d, while the other servers (also operating at state d) draw their power from the normal supply. d is the highest power state (i.e., least performance impacting) that obeys the above conditions. This scheme can thus sustain a longer duration of battery operation than BattFast+Throt, though the performance consequences can be felt even earlier (where the battery alone may have been sufficient to handle the emergency in BattFast+Throt).

Heuristics Local to Cluster c: With longer emergencies, local knobs may not suffice to meet application SLAs within the stipulated power budget. One option may be to migrate the load. It may sometimes be desirable to simply re-arrange load within c during an emergency, since (a) migration outside of c may not be possible (either there is no headroom or administrative reasons force the application to be tied to nodes within c), and/or (b) the locality of application needs (frequent communication, data stored locally, etc.) may get impacted when parts of the application are forced out of c. The downside to migrating (redistributing the load) within the cluster is that performance may be impacted if the existing load is already pushing individual servers to high resource utilization (which is usually what leads to the emergency). When migrating within c, we consider two heuristics: cMig and BattSlow+Throt+cMig. In cMig, the load is immediately migrated at the beginning of the emergency, from one or more servers and these servers are subsequently shut down. The number of servers from which the load is to be migrated depends on how many need to be taken down to get the total subsequent power consumption within P_{budget}. Since migration is typically intended to be the option of last resort, we do not consider local knobs (either battery or power modes) after the migration. Using the same rationale, BattSlow+Throt+cMig defers the migration point to a time using the above-mentioned BattSlow+Throt strategy until the battery capacity reaches residual capacity needed for availability (2 minutes), and then employs cMig within c. While there are numerous ways of performing load migration (see Section 3) we restrict our evaluations to virtual-machine based (live) migration [8], which is a convenient vehicle for performing this task at the infrastructure level without any application-level knowledge/mechanism. We can explore other strategies in future work.

Heuristics Across Clusters: Handling an extended emergency without substantial performance repercussions may need moving or migrating load outside of c to parts of the datacenter with sufficient headroom. In this paper, we do not consider the problem of where to move this load, and simply assume that it can be accommodated elsewhere. As explained above, the loss in locality (communication and storage) after migration can impact subsequent performance, and we again explore two strategies - dMig, which performs the migration right at the beginning of the emergency, and BattSlow+Throt+dMig, which delays the migration as much as possible.

6. Implementation and Experimental Setup

We use a scaled-down experimental prototype to evaluate our heuristics and compare them with Opt-local and Opt. Our prototype uses a cluster c (Figure 3) of N=8 DELL PowerEdge servers with two Intel Xeon 3.4GHz processors each, running RedHat Linux 5.5. The face-plate rating of these servers is 450W. Their

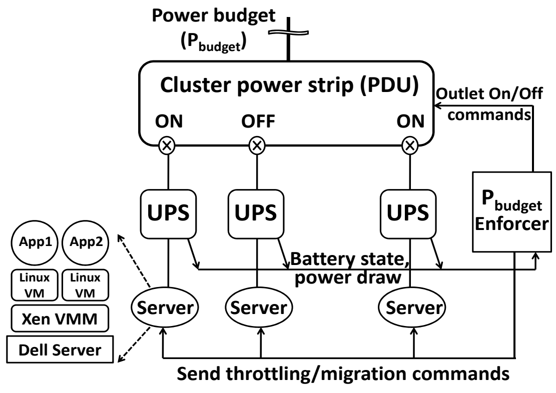


Figure 3. Experimental prototype.

idle power consumption is around 120W and the peak power that we can push the server to across our workloads is 320W. The dynamic power consumption can be modulated with 4 DVFS states (P-states: 3.4GHz, 3.2GHz, 3.0GHz, and 2.8GHz) and 8 clock throttling states (T-states: 12.5%, 25%, ..., 100%). To change power states, we write custom drivers using the $IA32\_PERF\_CTL$ and $IA32\_CLOCK\_MODULATION$ MSR registers. Each server is directly connected to a 1000W APC UPS [1] which, in turn, is connected to an outlet of a 30Amp Raritan PDU.

Although we have a 1000W UPS unit connected to each server, for all our experiments we only assume a 330W UPS (close to the maximum power consumed by our server) and drain the UPS using a corresponding scaled-down runtime chart [3]. We consider a 4-minute battery per server which is relatively on the lower end, of which we leave a residual capacity of 2 minutes, required for availability guarantees. The UPS is capable of reporting its load, power draw and remaining battery runtime over an RS232 serial interface. The PDU is capable of dynamically switching ON/OFF the supply to individual UPS units with SNMP commands over Ethernet. By turning on/off individual outlets, we can selectively have a server source power from either the battery or the power line. We use a separate machine ("Power Budget Enforcer") to implement the heuristics - send throttling, migration and PDU turn on/off commands. Our cluster has a shared NAS box which is mounted as a NFS storage volume by all the servers. We use another cluster (not shown in the figure) of 8 servers as the destination for migrating workloads in dMig and dMig+BattSlow+Throt. All our applications are hosted as VMs under Xen on each server.

7. Evaluation

We use four case studies involving six different applications to evaluate the efficacy of our online heuristics. In each, we present (i) salient workload properties, (ii) emergencies lasting a range of durations and corresponding to degrees (as 10-30% of potential peak) of underprovisioning, (iii) the remedial actions corresponding to our heuristics, and (iv) a comparison of their efficacy in alleviating the emergency with respect to Opt-local and Opt. Note that we are only concerned with performance during an emergency in this work. The remedial actions for emergencies lasting several hours will typically mandate migration and there are no further insights to be gained by studying such long durations. As discussed earlier, we have profiled the performance ($R(l,d)$) and power ($P(l,d)$) of

[3] The minimum capacity of UPS units available from APC is 500W. Although UPS units are typically over-provisioned, we assume this conservative tight provisioning of 330W for our experiments.

each workload a priori. In general, enterprise/internet applications typically undergo extensive profiling for right-sizing of datacenter IT resources. Even for cloud-hosted third-party applications, resource usage can be determined via offline/online profiling. Such profiling, with readily available power meters [25] on those platforms, or in combination with well-understood power models [16] that are based on resource utilization, can be used to determine $R_{ij}(l,d)$ and $P_{ij}(l,d)$. Since, we already have plenty of ground to cover in this paper, for the purposes of this work we assume that R_{ij} and P_{ij} for a given workload, as a function of different power states is made available. A more detailed treatment of these issues can be considered in future work.

7.1 TPC-W and SPECjbb

We study two well-known server benchmarks: TPC-W [40] and SPECjbb [41]. TPC-W emulates a 3-tiered (Apache, Tomcat, and MySQL) transactional Web-based eCommerce bookstore. The Apache front-end runs on a dedicated server, while the other two tiers run on a set of servers whose size is chosen to accommodate the workload intensity. Apache employs a request distribution module to balance requests among replicas of Tomcat. We use the clustered MySQL database engine that provides a replicable, shared-nothing database tier. Each Tomcat and MySQL instance runs in its own Xen domain. TPC-W services a specified number of clients over persistent HTTP sessions. We use the average client response time during the emergency as the performance metric (R) for TPC-W. SPECjbb is a 3-tiered server-side Java warehouse management application. We use the average transactions/second (tps) as our performance metric (R) for SPECjbb.

Emergency Handling for TPC-W: Applications like TPC-W are known to experience significant temporal variations in the load. Many such variations can be predicted (e.g., time-of-day behavior) to ensure that enough power capacity is provisioned. However, there are other variations not amenable to such prediction (e.g., flash crowds) which can cause emergencies when underprovisioning. Responding to the growing workload, the datacenter incrementally adds replicas of Tomcat and MySQL on new servers till all 8 servers are utilized. This can cause an emergency since the aggregate draw of these servers can exceed the power budget of the underprovisioned cluster. For instance, when the workload saturates all 8 servers, the aggregate power consumption hits 1630W. If the infrastructure is underprovisioned by 10%, 20% and 30%, then the corresponding P_{budget} limits are 1470 W, 1300 W, and 1140 W respectively. We inject load to introduce emergency durations of 2, 8, 15, 30, and 60 minutes, over these provisioned limits.

Figures 4(a) and (b) present the degradation in average response times of TPC-W with 10 and 30% underprovisioning. `Throt` chooses (2.8Ghz, 100%Clk) and (2.8Ghz, 25%Clk) for 10% and 30% underprovisioing, respectively, and severely degrades performance (going from 30% degradation to over 500% in the more underprovisioned case) in these high utilization regimes. On the other hand, `BattFast+Throt` is better, particularly for short to moderate emergency durations (upto 15 minutes). For such emergencies, it is able to completely source the excess power from batteries without throttling. For longer durations (30 minutes and more), the batteries run out, mandating throttling which degrades performance. The high sensitivity of performance to even small CPU rate modulation makes `BattSlow+Throt` mostly ineffectual: for short emergencies (upto 15 minutes), it unnecessarily throttles due to its conservative (slow) drain from batteries; for moderate durations (upto 30 minutes), it stretches the battery runtime, but the accompanying throttling hurts performance; for long durations (more than 30 minutes), batteries run out with subsequent consequences similar to `BattFast+Throt`.

Next we consider the cluster-level migration, `cMig`. `cMig` employs Xen's live migration facility [8] to seamlessly migrate a subset of (let us denote its size as n) Tomcat/MySQL replicas from their original servers and co-locates them with those on the remaining $8 - n$ servers - which in turn become overloaded. These n unoccupied servers are now turned off. We find n to be 1 and 3 for underprovisioning degrees of 10% and 30%, respectively. For the 30% underprovisioning, all 5 active servers continue to operate at their highest power states. Since the TPC-W components were operating at their peak requirements, the components that are co-located experience significant resource shortage, causing response time to nearly double. In fact, `cMig` fares worse than the earlier two heuristics involving the battery. The idea behind `BattSlow+Throt+cMig` is to improve upon `cMig` by postponing migration as much as possible; it starts as `BattSlow+Throt` and switches to `cMig` when the batteries run out. For small/moderate emergencies, this defaults to `BattSlow+Throt` with the same pros and cons - essentially undesirable for TPC-W. It is worse for longer emergencies since it will end up switching to `cMig`, which we have already found to hurt TPC-W.

Finally, let us discuss the efficacy of `dMig`. We migrate 1 and 3 Tomcat/MySQL VM pairs for 10% and 30% underprovisioning respectively. Live migration can be carried out relatively quickly for TPC-W (about 2 minutes) [4]. Furthermore, loss of data locality suffered by migrated VMs is negligible in this case (i.e. small s_r), implying little performance consequence after the migration. Consequently, `dMig` turns out to be the most effective heuristic and is able to handle the entire range of emergencies with little performance consequence (though requiring the battery to temporarily handle the power spike during the act of migration). The only situations where `dMig` is an unwise choice is when the emergency lasts less than the migration duration of 2 minutes. `BattSlow+Throt+dMig` attempts to postpone the remote migration invocation and ends up offering worse performance than `dMig` due to TPC-W's sensitivity to even small degrees of throttling.

`BattFast+Throt` comes close to `Opt-local` for emergencies less than 15 minutes, while `BattSlow+Throt` is closer to `Opt-local` for longer emergencies. Incidentally, `Opt-local` never chooses `Throt` when we examine (post-mortem) decisions reached by the offline algorithm for durations less than 15 minutes, and in longer durations it chooses heterogeneous power states across the servers, while our battery based heuristics choose the same state across all servers at any time. Specifically, `Opt-local` uses a lower power state on servers that source from batteries, stretching the battery runtime. In this case study, `Opt` first drains the battery fully, and then immediately migrates the load to a remote node since there is little performance impact after migration.

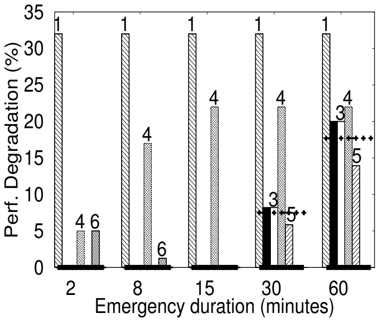
Emergency Handling for SPECjbb: Emergency handling in SPECjbb is similar to TPC-W, with similar results. The 8 servers housing SPECjbb replicas end up operating close to saturation, and their aggregate power consumption is found to be 1875W. Figures 4(c) and (d) present the degradation in average tps during various emergencies.

Key Insights: (i) CPU throttling is undesirable for this class of applications, even for short or moderate durations. (ii) batteries are helpful for a wide range of emergencies (upto 30 minutes). (iii) batteries can offer lower power draws than even the deepest

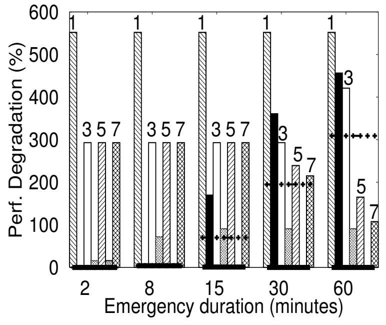
[4] Incidentally, though we do not explicitly present those details, we wish to point out that migration itself does introduce a spike in power consumption (of 10%) for this workload, and the battery is still needed (in solutions such as `cMig` and `dMig`) to temporarily get the power draw under control during the act of migration.

Legends for Bar Charts: Note that the legend numbers are given above some of the bars for easier readability. Opt and Opt-local lines are also drawn as reference lower bounds.

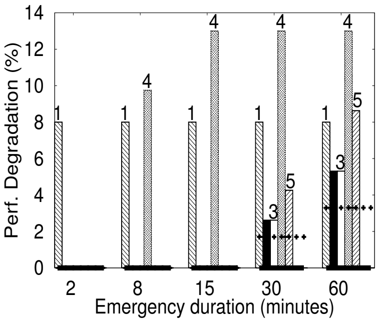
1. ▨ Throt 2. ■ BattFast+Throt 3. □ BattSlow+Throt
4. ▩ cMig 5. ▨ BattSlow+Throt+cMig 6. ▨ dMig
7. ▦ BattSlow+Throt+dMig ✦✦✦✦ Opt-local ▬ Opt



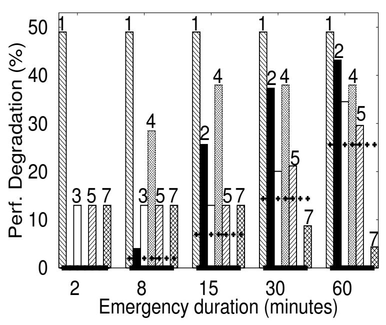
(a) TPC-W, 10%



(b) TPC-W, 30%

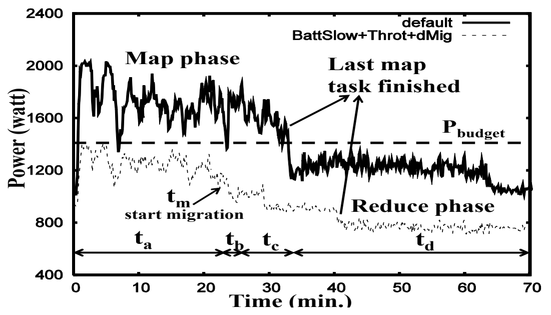


(c) Specjbb, 10%

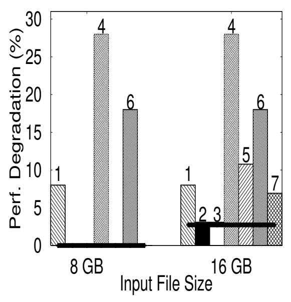


(d) Specjbb, 30%

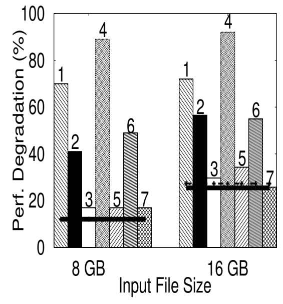
Figure 4. TPC-W and Specjbb results



(a) MapReduce profile: Default, 30% with `BattSlow+Throt+dMig`. Reduce phase has been truncated for clarity.



(b) MapReduce, 10%



(c) MapReduce, 30%

Figure 5. MapReduce results

power states. (iv) the low/zero slowdown after migration (s_r), favors `dMig` (coming close to `Opt`), though the battery is still needed to handle the initial power spike during migration (which takes 1-2 minutes). (v) `Opt-local` and `Opt` point towards the possibility of *heterogeneous* power state assignment to servers to achieve better power/performance trade-offs than offered by our heuristics.

7.2 MapReduce

This workload represents a growing and important class of parallel applications used in domains such as search engines. We run a word count application using Hadoop [22]. It schedules the mapper and reducer tasks across a set of specified servers and reports the application finish time as its performance metric. We consider 8 GB and 16 GB for input file sizes. MapReduce inherently uses distributed storage, placing computations closer to the data that it needs. If one considers a server's storage volume to be part of the VM, then `cMig` and `dMig` based on infrastructure-level VM migration would require moving all of this data to the destination node (which can be quite large), rather than just the data needed by subsequent computation. While one could consider this in future work, in this set of experiments, we assume storage to be decoupled from the VM image (by implementing the local storage as a NFS-mounted server), and only move the VM image for migration, with the subsequent computation making NFS calls to get the specific data that it needs from the source cluster c (where the data is replicated for availability even if servers go down).

Emergency Handling for MapReduce: Figure 5(a) shows evolution of the aggregated power of the 8 servers running MapReduce (labeled as *default*) with the 16GB input. We see high power variation over time, suggesting under-utilization if we provision for the peak. Specifically, the mapper phase (upto 30 min.) consumes significantly higher power that reaches up to 2020W. Consequently, a surge of mapper activities across the cluster can sometimes lead to emergencies in an aggressively underprovisioned system. Delaying the mappers and/or spreading them temporally/spatially can delay the application and possibly impact its locality.

Figures 5(b) and (c) compare the performance of our heuristics for underprovisioning of 10% and 30% (with respect to 2020W), respectively. For the 8GB input, we find battery-based techniques are able to handle the emergency with little/no performance consequences for 10% underprovisioning (comparable to `Opt`). The emergency duration for this input is roughly 15 minutes, which can be easily sustained by each of the 8 servers (only 1 server needs to not source from power line to meet power bounds) taking turns sourcing power from the battery for roughly 2.3 minutes each. In fact, the power state modulation and migration mechanisms do not even kick in when the battery is complemented with these knobs. However, for 30% underprovisioning, the other knobs are also employed, and the performance progressively degrades. Still, the battery-supplemented techniques do better than without this knob. Further, while `BattFast+Throt` does better than `BattSlow+Throt` in the 20% underprovisioning case (not shown in figure), the results are reversed in the 30% underprovisioning case (shown in Figure 5 (c)) where the emergency mandates a higher power shaving, causing the battery to run out faster if that is the first knob of choice. Sustaining a longer period of operation with the battery (a slow drain rate achieved with simultaneous power state control) is a better option in such cases of high underprovisioning.

The 16GB input extends the emergency duration to roughly 30 minutes, making it necessary to supplement battery with power state modulation and/or migration. This degrades performance even in the 10% underprovisioning case. In both 8GB and 16GB experiments, we find migration does relatively worse than using local knobs alone. This is because of migration overheads, where doing it locally (`cMig`) results in increasing the load on one or more slave nodes within the cluster, delaying the progress of the application. `dMig` for MapReduce, has a different problem - loss of data locality requiring considerable data movement across clusters - which tremendously impacts performance (though this is still better performing than `cMig` which overloads servers). MapReduce, thus, depicts a spectrum of workloads not as conducive to migration for emergency reaction (unlike the stateless applications in previous subsection) impacting performance not just during the migration but also subsequently. Like before, supplementing migration with local knobs helps defer t_m (to about 23 minutes in the 16GB experiment), to lessen the subsequent performance slowdown (s_r) after migration in both `cMig` and `dMig` cases. Migration is more competitive for longer emergency durations - for instance, consider the 16GB degradation results in 30% underprovisioning for `cMig` (92%) and `dMig` (55%) with respect to `BattSlow+Throt` (34%), and compare them with those for the 8GB degradation where `BattSlow+Throt` suffers only 16% degradation while `cMig` and `dMig` still suffer 89% and 49% degradation. The local knobs supplemented with migration can thus help bridge the emergency duration gap when migration becomes more competitive.

Figure 5(a) presents salient decisions made by our heuristic, `BattSlow+Throt+dMig` (the closest heuristic to `Opt`) during the emergency with 30% under-provisioning: (i) *Duration t_a:* one server at a time is sourced from its battery and all 8 servers are operated at 2.8GHz DVFS till $t = 23$ minutes; (ii) *Duration t_b:* migrating 3 mapper VMs to another cluster which takes 1-2 minutes; (iii) *Duration t_c:* 3 machines are shut down after migration, and the remaining 5 servers operate at the highest 3.4GHz DVFS state. (iv) *Duration t_d:* which is the "reduce" phase where there is no longer an emergency, though we do not consider the option of moving back the VMs to c in our experiments. Note that we adhere to the 1420W cap (30% underprovisioning) over the entire duration.

Performance degradation of `Opt` closely matches our heuristics that only use the server-local knobs for 10% underprovisioning where migration is less desirable (`Opt` does not choose migration in this case). We observe the same behavior for the 8GB input at 30% underprovisioning where `Opt` only uses local knobs for the entire emergency. One interesting observation is that for the 16GB input at 30% underprovisioning, `Throt` becomes very expensive - about 72% decrease in throughput. `Opt` resorts to `dMig` at $t_m = 24$ minutes in this case, which closely matches the decision by `BattSlow+Throt+dMig` where the battery runs out of charge at approximately the 23^{rd} minute and migration is initiated. It is important to note that MapReduce is very sensitive to the value of t_m and migrating before or after 24 minutes results in poor performance.

Key Insights: (i) Longer emergencies may mandate migration, but this slows down the application either due to overload of some servers in the cluster (`cMig`) or poor data locality (`dMig`); (ii) `BattSlow+Throt` postpones migration, and allows it to become more competitive; (iii) when migration is expensive, `Opt` refrains from migration for short emergencies (`BattFast+Throt` comes close) and defers migration as long as possible for long emergencies (`BattSlow+Throt+dMig` comes close).

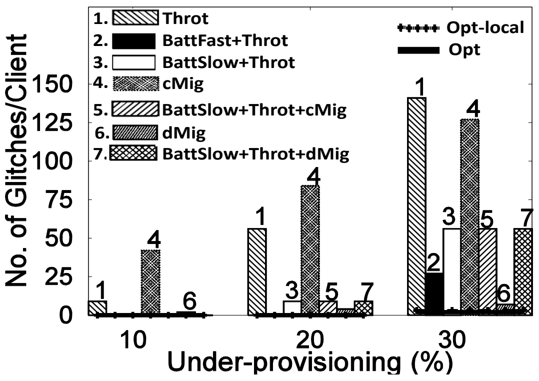


Figure 6. Media Server Results.

7.3 Streaming Media Server

Our third case study uses a multi threaded 1.5 Mbps streaming MPEG media server that services several Java clients. It spawns a separate thread for each client, which does password authentication before it starts streaming. A 4.5MB buffer is used at the client to smoothen traffic variations. There could still be playback discontinuities ("glitches") when the buffer becomes empty. The server runs on all 8 machines, streaming a 60-minute long video for a total of 2400 clients.

Emergency Handling: During the initial phase (say first 10 minutes), when the clients try to connect to the server, we observe a power spike of about 1700W, compared to the steady state draw of 1200W when there is only subsequent streaming. This initial connection/authorization phase can be viewed as the emergency duration and we evaluate our heuristics over this duration for 10%, 20% and 30% underprovisioning in Figure 7.2.

`Throt` degrades with aggressive underprovisioning (from 8 glitches per client for 10% underprovisioning to over 100 for 30%), since it is not able to sustain the streaming needs. Due to the short emergency (about 10 minutes), `BattFast+Throt` is able to handle it without throttling and hence performs much better than `BattSlow+ Throt`. In fact, `BattFast+Throt` performs very well even at 30% underprovisioning, incurring only one-fifth of the glitches compared to `Throt`. We note that `cMig` hurts performance even for small underprovisioning, since each server is already saturated. `BattSlow+Throt+cMig` is able to delay (actually avoids migration completely because the battery is able to fully sustain the emergency) this migration, defaulting to `BattSlow+Throt`. It is interesting that `dMig`, which had poor performance for MapReduce does very well since there is almost no performance impact after the migration (s_r is negligible). It takes only about 30 seconds for the media server VM to migrate and we find that the client buffer size is good enough to sustain the performance impact for most of this duration. While the battery serves as a buffer to allow temporary power spikes at the media servers (as in earlier case studies), a similar effect is achieved by the client buffer which allow the media servers to slowdown (either by power state transition or migration) temporarily - thus reducing their power draw. We also observe that unlike MapReduce, it does not make sense to delay migration (`BattSlow+Throt+dMig`) since migration overheads are low, and throttling has severe performance impact.

`Opt` chooses to use the battery alone for 10% and 20% underprovisioning, equivalent to `BattFast+Throt` and does not incur any degradation. For 30% underprovisioning, battery cannot fully sustain the peak and Opt uses datacenter-level migration which results in just 2 glitches per client. In this application, we find $t_m = 7$ minutes for Opt (unlike 23 minutes in MapReduce), since migration overhead is negligible. Interestingly, when our optimization (Opt-local) is run with just the local knobs where we see that for 30% underprovisioning it is comparable to Opt and achieves as few as 3 glitches per client compared to the best local knob heuristic - BattFast+Throt - which incurs 27 glitches/client. When we analyzed the Opt-local results, we noticed that it sources battery from 2 servers at any time and uses throttling to shave only the remaining 10%. This behavior is somewhere in-between our battery-aggressive, BattFast+Throt and battery-conservative, BattSlow+Throt heuristics which source from 3 servers and 1 server respectively from battery and shave the rest by throttling. This shows the importance of dynamically adjusting battery drain rate, which we plan to investigate in future work.

Key Insights: (i) Battery-based heuristics perform very well due to the smaller emergency duration. (ii) The battery and client buffer, in combination, provide a seamless strategy for offloading the work to elsewhere in the datacenter, and hide migration cost.

7.4 Graphics Application and Virus Scan

This case study involves two applications: a GPU application in CUDA implementing the Black-Scholes financial model using a NVIDIA card, and VirusScan (Linux AVG [29]) which needs to run on the local machine to scan its 40GB hard drive. Not all datacenter servers may offer the required graphics support (e.g., Amazon EC2 offers separate GPU cluster instances), and moving the VirusScan elsewhere is not an option. VirusScan is a strawman we use to illustrate applications, that (i) are tied to a specific server, (ii) are fairly flexible in their processing rate (low-priority), and (iii) have some kind of deadline (24 hours in this case). Other examples include back-ups, search-engine indexing, etc. Even though the deadlines are "soft" in these examples, we will use a *"hard" deadline for more general illustration*. We vary the rate at which GPU applications arrive which can, in turn impact the schedulability of VirusScan, and report throughput (GPU ops/sec) subject to the 24 hour deadline for VirusScan that needs to be met. In our servers, the GPU application running alone consumes 250W, the VirusScan at full-throttle runs at 208W taking 45 minutes to complete, and the two together at full-throttle hit 315W.

Emergency Handling: We define an emergency in this case as an out-of-the-ordinary day wherein the VirusScan does not get sufficient bandwidth (because of the power budget) to run until the last x minutes before the 24 hour deadline, and we vary x between 240 to 45 minutes (beyond 240 minutes there is sufficient bandwidth for our considered load). This setup is different from the earlier case studies, since we have multiple applications. We adapt Throt, BattFast+Throt and BattSlow+Throt for this scenario since migration is not an option. We do not explicitly discuss results for BattSlow+Throt whose behavior is not very different from BattFast+Throt. Throt does power state modulation continuously over the last x minutes for VirusScan to finish its remaining execution before the 24 hour deadline. CPU throttling states have little impact on the progress or power consumption of the GPU application (as it depends mainly on the GPU). Hence, if there is not enough slack at a given time to meet the VirusScan deadline, the GPU application is put on hold (suspended), and VirusScan is run at full-throttle until there is slack again. BattFast+Throt employs battery as long as possible from the beginning of the x minutes, until it is drained to residual capacity of 2 minutes (required for availability), with VirusScan running at full-throttle. Subsequently, Throt is employed.

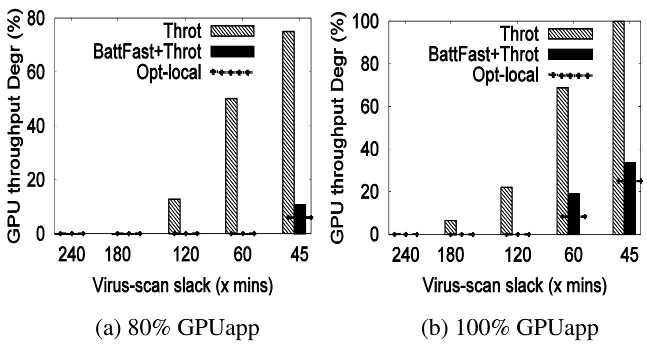


Figure 7. Performance of GPU application during various emergencies caused by VirusScan's deadline. (10% underprovisioning)

A representative result for 10% power underprovisioning (with respect to the maximum possible draw of 2520W) is given in Figure 7. The emergency is captured by different values of x on the x-axis, and we present the percentage degradation in GPU ops on the y-axis. This degradation is shown for different imposed loads by GPU application, with intensities depicted as 80% and 100%, i.e. percentage of time that the GPU is active. Throt by itself suffices when there is sufficient notice ($x > 240$) given to run these algorithms. Getting closer to the deadline to react does not allow much room for Throt to run both applications without affecting the GPU application. There is 50-60% slowdown in GPU ops during the last hour with Throt. Higher GPU application intensity leaves even less room for Throt to allocate sufficient bandwidth for the two, thereby impacting throughput. The battery-based solution, on the other hand, is able to do much better across the entire spectrum, not suffering any loss for $x \geq 120$. It is only at very high GPU application intensities, and very little reaction time (in the last hour) that it results in performance loss, and even then does significantly better than Throt. Further, the battery based solution comes quite close to Opt, with the latter doing slightly better because of more leeway in what gets scheduled (GPU or VirusScan) at any time on different servers, and the possible heterogeneous power states assignment across the servers at a given time.

Key Insights: (i) Application intensity, slack in scheduling and reaction window, all impact throughput. (ii) Power state modulation alone is not able to handle high intensities when sufficient slack is absent. (iii) Battery provides substantial leeway in handling emergencies for scheduling workloads with both temporal (deadlines) and spatial (need to run on specific servers) constraints.

7.5 Summary of Observations

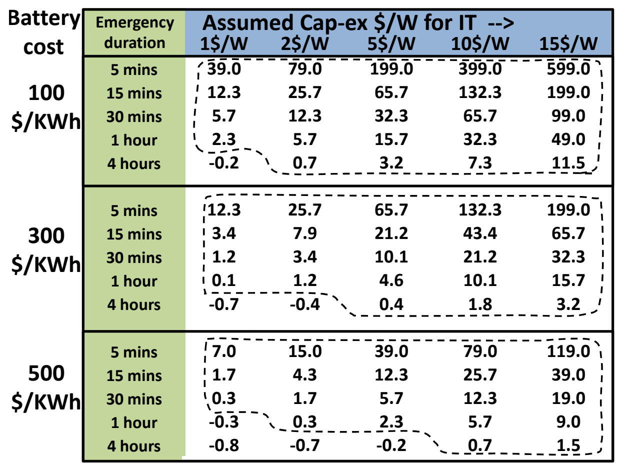
Table 2 summarizes the results from the six different application case-studies evaluated above, pointing out the efficacy of the knob(s) towards handling the different performance-power characteristics.

7.6 Invest in additional battery capacity?

Even though we have considered a conservative 4-minute battery capacity in our experiments, one may ask whether the cost of additional battery capacity can be justified by the potential reduction in datacenter power infrastructure costs that we can gain by underprovisioning. Let us denote the cost of procuring additional battery capacity to sustain e hours of emergency as $e \cdot c_{bat}$ \$/watt, and the cap-ex cost of the power infrastructure to under-provision by c_{cap} \$/watt. Lead-acid battery costs (c_{bat}) reported in literature (see DOE/Sandia data [24]) are in the 100-300 \$/kWh range, and we consider conservative values as high as 500 \$/kWh. Datacenter

Application characteristics	Emergency handling (knob selection)
Short peak widths	Batteries are self-sufficient in handling peaks up to 30 mins without requiring expensive throttling
Medium peak widths	Batteries supplement power state throttling to reduce performance overheads
Long peak widths	Immediate migration is benefitial, Batteries hide power spike during migration
Sensitive to data locality	Migration may impact performance, Batteries help postpone migration
Slack-based workloads	Shift peak temporally via flexible workload scheduling, Batteries help create more slack

Table 2. Summary of Results



Battery cost	Emergency duration	Assumed Cap-ex $/W for IT --> 1$/W	2$/W	5$/W	10$/W	15$/W
100 $/KWh	5 mins	39.0	79.0	199.0	399.0	599.0
	15 mins	12.3	25.7	65.7	132.3	199.0
	30 mins	5.7	12.3	32.3	65.7	99.0
	1 hour	2.3	5.7	15.7	32.3	49.0
	4 hours	-0.2	0.7	3.2	7.3	11.5
300 $/KWh	5 mins	12.3	25.7	65.7	132.3	199.0
	15 mins	3.4	7.9	21.2	43.4	65.7
	30 mins	1.2	3.4	10.1	21.2	32.3
	1 hour	0.1	1.2	4.6	10.1	15.7
	4 hours	-0.7	-0.4	0.4	1.8	3.2
500 $/KWh	5 mins	7.0	15.0	39.0	79.0	119.0
	15 mins	1.7	4.3	12.3	25.7	39.0
	30 mins	0.3	1.7	5.7	12.3	19.0
	1 hour	-0.3	0.3	2.3	5.7	9.0
	4 hours	-0.8	-0.7	-0.2	0.7	1.5

Figure 8. ROI with additional battery capacities. Positive values indicates investment is worthwhile (magnitude indicates higher returns) while negative values suggest not investing in higher capacities.

power infrastructure cap-ex is reported to grow by \$10-25 for every provisioned watt [4, 23]. Since this includes costs for cooling, Diesel Generators, UPS, etc., it is non-trivial to isolate the \$/W for the IT power infrastructure. Consequently, we study a wide range for c_{cap} starting from as low as 1 \$/W going all the way to 15 \$/W. We can then calculate the Return-On-Investment (ROI) for additional battery capacity as: $\frac{c_{cap} - e \cdot c_{bat}}{e \cdot c_{bat}}$ and show these in Figure 8 for emergency durations ranging from 5 minutes to 4 hours. In these calculations, we ensure that battery costs are amortized over 4 year lifetimes while infrastructure costs are amortized over 12 year lifetimes. We find a positive ROI (very high ROI in many cases) for most of the operating regions, despite considering pessimistic scenarios. This suggests that investing in additional battery capacity may be worthwhile, although a more in-depth economic analysis is warranted in future work.

8. Concluding Remarks

We have presented a framework for dealing with emergencies arising from aggressive underprovisioning of the power infrastructure. Rather than disruptive fuses/circuit-breakers, IT controlled techniques such as power state modulation, and workload migration can be supplemented with our new proposal of leveraging already existing UPS batteries to gracefully deal with emergencies. We demonstrate using an experimental prototype, with several interesting use cases, that the battery based approaches are (i) self-sufficient to deal with short duration emergencies, (ii) supplement existing solutions to enhance their efficacy over a wider range of operating conditions, and (iii) create opportunities where other options are infeasible. We have also presented an offline theoretical framework to find bounds on how well we can perform under these emergencies, and presented several online heuristics that can adapt themselves to work under these unpredictable black swan events (when? how severe? how long?). Since these emergencies are typically a consequence of load surges, existing solutions - power state throttling and migration - by themselves can have serious performance implications. Overall, we find that when migration/load-redirection overhead is low, a fast drain of the battery locally to control the power surge until migration is complete, works quite well. At the other extreme, when migration costs are high, delaying the migration with a combination of slow battery drain and power mode control is the better option. The battery is thus an agile and useful stand-alone and/or complementary solution to address both short and long duration emergencies.

Acknowledgments

This work was supported, in part, by NSF grants CNS-0720456, CNS-0615097, CAREER award 0953541, and research awards from Google and HP. We would like to thank Luiz Barroso from Google, who provided valuable inputs on motivating and shaping the work presented in this paper.

References

[1] 1 KW APC UPS - SURTA1500RMXL2U. `http://www.apc.com/products/`.

[2] F. Ahmad and T. N. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. In *Proceedings of the Architectural support for programming languages and operating systems (ASPLOS)*, 2010.

[3] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *Proceedings of the ACM Symposium On Cloud Computing (SOCC)*, 2010.

[4] L. A. Barroso and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[5] D. Bhandarkar. Watt Matters in Energy Efficiency, Server Design Summit, 2010.

[6] J. Chase, D. Anderson, P. Thakur, and A. Vahdat. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2001.

[7] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing Server Energy and Operational Costs in Hosting Centers. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2005.

[8] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[9] Commercial circuit breakers, 2008. `http://circuit-breakers.carlingtech.com/all_circuits.asp`.

[10] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: active low-power modes for main memory. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[11] Facebook Open Compute Project, 2011. `http://opencompute.org`.

[12] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.

[13] W. Felter, K. Rajamani, C. Rusu, and T. Keller. A Performance-Conserving Approach for Reducing Peak Power Consumption in Server Systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2005.

[14] M. E. Femal and V. W. Freeh. Safe overprovisioning: Using power limits to increase aggregate throughput. In *Workshop on Power-Aware Computer Systems (PACS)*, 2004.

[15] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *Transaction On Computer Systems (TOCS)*, 2004.

[16] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal power allocation in server farms. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[17] L. Ganesh, J. Liu, S. Nath, G. Reeves, and F. Zhao. Unleash Stranded Power in Data Centers with RackPacker. In *Workshop on Energy-Efficient Design (WEED)*, 2009.

[18] Google Server-level UPS for improved efficiency. `http://news.cnet.com/8301-1001_3-10209580-92.html`.

[19] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A.Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proceedings of the International European Conference on Computer Systems (EUROSYS)*, 2009.

[20] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and Limitations of Tapping into Stored Energy For Datacenters. In *Proceedings of the International Symposium of Computer Architecture (ISCA)*, 2011.

[21] S. Govindan, D. Wang, L. Chen, A. Sivasubramaniam, and B. Urgaonkar. Towards Realizing a Low Cost and Highly Available Datacenter Power Infrastructure. In *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower)*, 2011.

[22] Hadoop Map Reduce. `http://hadoop.apache.org/mapreduce/`.

[23] J. Hamilton. Internet-scale Service Infrastructure Efficiency, ISCA Keynote 2009.

[24] Lead-acid battery cost. `http://photovoltaics.sandia.gov/Pubs_2010/PV%20Website%20Publications%20Folder_09/Hanley_PVSC09%5B1%5D.pdf`.

[25] C. Lefurgy, X. Wang, and M. Ware. Server-Level Power Control. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, 2007.

[26] H. Lim, A. Kansal, and J. Liu. Power budgeting for virtualized data centers. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIX)*, 2011.

[27] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOMM)*, 2011.

[28] D. Linden and T. B. Reddy. *Handbook of Batteries*. McGraw Hill Handbooks, 2002.

[29] Linux AVG Anti Virus. `http://free.avg.com/`.

[30] K. Ma, X. Li, M. Chen, and X. Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[31] S. McCluer. APC White paper 30 (Revision 11): Battery Technology for Data Centers and Network Rooms: Lead-acid Battery Options, 2005.

[32] D. Meisner, C. M. Sadler, L. A. Barroso, W. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[33] Microsoft Reveals its Speciality Servers, Racks, 2011. `http://www.datacenterknowledge.com/archives/2011/04/25/microsoft-reveals-its-speciality-servers-racks/`.

[34] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling cool: Temperature-aware workload placement in data centers. In *Proceedings of the Usenix Annual Technical Conference (USENIX)*, 2005.

[35] S. Pelley, D. Meisner, P. Zandevakili, T. F. Wenisch, and J. Underwood. Power Routing: Dynamic Power Provisioning in the Data Center. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[36] E. Pinheiro, R. Bianchini, E.Carrera, and T. Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In *Workshop on Compilers and Operating Systems for Low Power (COLP)*, 2001.

[37] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No Power Struggles: Coordinated multi-level power management for the data center. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[38] L. Ramos and R. Bianchini. C-Oracle: Predictive thermal management for data centers. In *proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[39] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2006.

[40] W. Smith. TPC-W: Benchmarking An Ecommerce Solution. `http://www.tpc.org/information/other/techarticles.asp`.

[41] SPEC JBB2005: Java Business Benchmark. `http://www.spec.org/jbb2005/`.

[42] N. N. Taleb. *The Black Swan: The Impact of the Highly Improbable*. Random House, 2007.

[43] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[44] R. Urgaonkar, B. Urgaonkar, M. J. Neely, and A. Sivasubramaniam. Optimal Power Cost Management Using Stored Energy in Data Centers. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2011.

[45] A. Verma, P. De, V. Mann, T. Nayak, A. Purohit, G. Dasgupta, and R. Kothari. Brownmap: Enforcing power budget in shared data centers. In *Proceedings of the Conference on Middleware (MIDDLEWARE)*, 2010.

[46] C. Waldspurger. Memory Resource Management in VMWare ESX Server. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[47] X. Wang and M. Chen. Cluster-level feedback power control for performance optimization. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[48] X. Wang, M. Chen, and C. Lefurgy. How much power oversubscription is safe and allowed in data centers? In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2011.

[49] A. Weisel and F. Bellosa. Process cruise control-event-driven clock scaling for dynamic power management. In *Proceedings of Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2002.

[50] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[51] F. Zhang, Z. Shi, and W. Wolf. A dynamic battery model for co-design in cyber-physical systems. In *Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2009.

[52] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2005.