Blink: Managing Server Clusters on Intermittent Power

Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy

Department of Computer Science
University of Massachusetts, Amherst
{nksharma,sbarker,irwin,shenoy}@cs.umass.edu

Abstract

Reducing the energy footprint of data centers continues to receive significant attention due to both its financial and environmental impact. There are numerous methods that limit the impact of both factors, such as expanding the use of renewable energy or participating in automated demand-response programs. To take advantage of these methods, servers and applications must gracefully handle intermittent constraints in their power supply. In this paper, we propose blinking—metered transitions between a high-power active state and a low-power inactive state—as the primary abstraction for conforming to intermittent power constraints. We design Blink, an application-independent hardware-software platform for developing and evaluating blinking applications, and define multiple types of blinking policies. We then use Blink to design BlinkCache, a blinking version of memcached, to demonstrate the effect of blinking on an example application. Our results show that a load-proportional blinking policy combines the advantages of both activation and synchronous blinking for realistic Zipf-like popularity distributions and wind/solar power signals by achieving near optimal hit rates (within 15% of an activation policy), while also providing fairer access to the cache (within 2% of a synchronous policy) for equally popular objects.

Categories and Subject Descriptors C.5.0 [*Computer System Implementation*]: General
General Terms Design, Management, Performance
Keywords Power, Intermittent, Renewable Energy, Blink

1. Introduction

Energy-related costs have become a significant fraction of total cost of ownership (TCO) in modern data centers. Recent estimates attribute 31% of TCO to both purchasing power and building and maintaining the power distribution and cooling infrastructure [15]. Consequently, techniques for reducing the energy footprint of data centers continue to receive significant attention in both industry and the research community. We categorize these techniques broadly as being either primarily *workload-driven* or *power-driven*. Workload-driven systems reconfigure applications as their workload demands vary to use the least possible amount of power to satisfy demand. Examples include consolidating load onto a small number of servers, e.g., using request redirection [8, 17] or VM migration, and powering down the remaining servers during off-peak hours, or balancing load to mitigate thermal hotspots and reduce cooling costs [3, 23, 24]. In contrast, power-driven systems reconfigure applications as their power supply varies to achieve the best performance possible given the power constraints.

While prior work has largely emphasized workload-driven systems, power-driven systems are becoming increasingly important. For instance, data centers are beginning to rely on intermittent renewable energy sources, such as solar and wind, to partially power their operations [14, 35]. Intermittent power constraints are also common in developing regions that experience “brownouts” where the electric grid temporarily reduces its supply under high load [8, 38]. Price-driven optimizations, due to either demand-response incentives or market-based pricing, introduce intermittent constraints as well, e.g., if multiple data centers coordinate to reduce power at locations with high spot prices and increase power at locations with low spot prices [29]. Variable pricing is an important tool for demand-side power management of future smart electric grids. The key challenge in power-driven systems is optimizing application performance in the presence of power constraints that may vary significantly and frequently over time. Importantly, these power and resource consumption constraints are *independent of workload demands*.

In this paper, we present *Blink*, a new energy abstraction for gracefully handling intermittent power constraints. Blinking applies a duty cycle to servers that controls the fraction of time they are in the active state, e.g., by activating and deactivating them in succession, to gracefully vary their energy footprint. For example, a system that blinks every 30 seconds, i.e., is on for 30 seconds and then off for 30 seconds, consumes half the energy, modulo overheads, of an always-on system. Blinking generalizes the extremes of either keeping a server active (a 100% duty cycle) or inactive (a 0% duty cycle) by providing a spectrum of intermediate possibilities. Blinking builds on prior work in energy-aware design. First, several studies have shown that turning a server off when not in use is the most effective method for saving energy in server clusters. Second, blinking extends the PowerNap [21] concept, which advocates frequent transitions to a low-power sleep state, as an effective means of reducing idle power waste.

An application’s *blinking policy* decides when each node is active or inactive at any instant based on both its workload characteristics and energy constraints. Clearly, blinking impacts application performance, since there may not always be enough energy to power the nodes necessary to meet demand. Hence, the goal of a blinking policy is to minimize performance degradation as power varies. In general, application modifications are necessary to adapt traditional server-based applications for blinking, since these applications implicitly assume always-on, or mostly-on, servers. Blinking forces them to handle regular disconnections more often associated with weakly connected [36] environments, e.g., mobile, where nodes are unreachable whenever they are off or out of range.

ASPLOS’11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03…$10.00

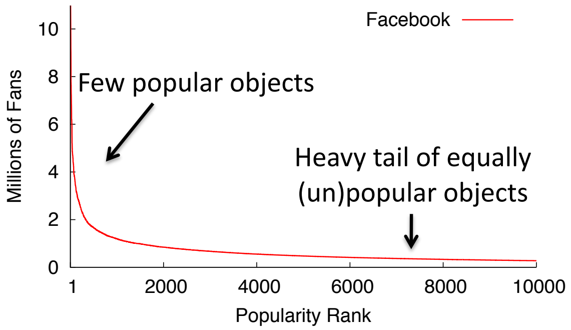


Figure 1: The popularity of web data often exhibits a long heavy tail of equally unpopular objects. This graph ranks the popularity of Facebook group pages by their number of fans.

1.1 Example: BlinkCache

To demonstrate how blinking impacts a common data center application, we explore the design of BlinkCache—a blinking version of *memcached* that gracefully handles intermittent power constraints—as a proof-of-concept example. Memcached is a distributed memory cache for storing key-value pairs that many prominent Internet sites, including LiveJournal, Facebook, Flikr, Twitter, YouTube, and others, use to improve their performance.

Memcached is a natural first application to optimize for variable power constraints for two reasons. First, a memcached cluster is energy-intensive, since it requires continuous operation of high-memory nodes to ensure instant access to in-memory state. Second, since memcached is a performance supplement that is not necessary for correctness, it does not preclude a constrained power source that may offer little or no power over some time periods. A blinking memcached cluster also exploits the increasing trend toward elevating memory in the storage hierarchy, and using it as the primary storage substrate in cloud data centers [26]. An important consequence of this trend is that applications increasingly use memory less like a cache that only stores a small set of popular objects, and more like a storage system that also stores a long heavy tail of unpopular objects. However, unpopular objects are not unimportant.

For Internet services that store user-generated content, the typical user is often interested in the relatively unpopular objects in the heavy tail, since these objects represent either their personal content or the content of close friends and associates. As one example, Figure 1 depicts a popularity distribution for Facebook group pages in terms of their number of fans. While the figure only shows the popularity rank of the top 10,000 pages, Facebook has over 20 million group pages in total. Most of these pages are nearly equally unpopular. For these equally unpopular objects, blinking nodes synchronously to handle variable power constraints results in fairer access to the cache. While fair cache access is important, maximizing memcached's hit rate requires prioritizing access to the most popular objects. We explore these performance tradeoffs in-depth for a memcached cluster with intermittent power constraints.

1.2 Contributions

In designing, implementing, and evaluating BlinkCache as a proof-of-concept example, this paper makes the following contributions.

- **Make the Case for Blinking Systems**. We propose blinking systems to deal with variable power constraints in server clusters. We motivate why blinking is a beneficial abstraction for dealing with intermittent power constraints, define different types of blinking policies, and discuss its potential impact on a range of distributed applications.
- **Design a Blinking Hardware/Software Platform**. We design Blink, an application-independent hardware/software platform to develop and evaluate blinking applications. Our small-scale prototype uses a cluster of 10 low-power motherboards connected to a programmable power meter that replays custom power traces and variable power traces from a solar and wind energy harvesting deployment.
- **Design, Implement, and Evaluate BlinkCache**. We use Blink to experiment with blinking policies for BlinkCache, a variant of memcached we optimize for intermittent power constraints. Our hypothesis is that a load-proportional blinking policy, which keeps nodes active in proportion to the popularity of the data they store, combined with object migration to group together objects with similar popularities, results in near optimal cache hit rates, as well as fairness for equally unpopular objects. To validate our hypothesis, we compare the performance of activation, synchronous, and load-proportional policies for realistic Zipf-like popularity distributions. We show that a load-proportional policy is significantly more fair than an optimal activation policy for equally popular objects (4X at low power) while achieving a comparable hit rate (over 60% at low power).

Section 2 provides an overview of blinking systems and potential blinking policies. Section 3 presents Blink's hardware and software architecture in detail, while Section 4 presents design alternatives for BlinkCache, a blinking version of memcached. Section 5 then evaluates BlinkCache using our Blink prototype. Finally, Section 6 discusses related work and Section 7 concludes.

2. Blink: Rationale and Overview

Today's computing systems are not energy-proportional [6]—a key factor that hinders data centers from effectively varying their power consumption by controlling their utilization. Designing energy-proportional systems is challenging, in part, since a variety of server components, including the CPU, memory, disk, motherboard, and power supply, now consume significant amounts of power. Thus, any power optimization that targets only a single component is not sufficient, since it reduces only a fraction of the total power consumption [6, 19]. As one example, due to the power consumption of non-CPU components, a modern server that uses dynamic voltage and frequency scaling in the CPU at low utilization may still operate at over 50% of its peak power [5, 37]. Thus, deactivating entire servers, including most of their components, remains the most effective technique for controlling energy consumption in server farms, especially at low power levels that necessitate operating servers well below 50% peak power on average.

However, data centers must be able to rapidly activate servers whenever workload demand increases. PowerNap [21] proposes to eliminate idle power waste and approximate an energy-proportional server by rapidly transitioning the entire server between a high-power active state and a low-power inactive state. PowerNap uses the ACPI S3 state, which places the CPU and peripheral devices in sleep mode but preserves DRAM memory state, to implement inactivity. Transition latencies at millisecond-scale, or even lower, may be possible between ACPI's fully active S0 state and its S3 state. By using S3 to emulate the inactive "off" state. [1] PowerNap is able to consume minimal energy while sleeping. Typical high-end servers draw as much as 40x less power in S3.

Blink extends PowerNap in important ways. First, PowerNap is a workload-driven technique that eliminates idle server power waste—it uses rapid transitions in a workload-driven fashion to activate each server when work arrives and deactivate it when idle.

[1] We use "active" and "on" interchangeably to reference ACPI's S0 state, and "inactive" and "off" interchangeably to represent ACPI's S3 state.

Type	Model	S3 Transition Time (s)
Desktop	Optiplex 745	13.8
Desktop	Dimension 4600	12.0
Laptop	Lenovo X60	11.7
Laptop	Lenovo T60	9.7
Laptop	Toshiba M400	9.1
Laptop	OLPC-XO (w/ NIC)	1.6
Laptop	OLPC-XO (no NIC)	0.2

Table 1: Latencies for several desktop and laptop models to perform a complete S3 cycle (suspend and resume). Data from both [2] and our own measurements of Blink's OLPC-X0.

In contrast, Blink is a *power-driven* technique that regulates average node power consumption independent of workload demands. Second, the PowerNap mechanism applies to each server independently, while Blink applies to collections of servers. Blinking policies, which we formally define next, are able to capture, and potentially exploit, cross-server dependencies and correlations in distributed applications. Finally, unlike workload-driven transitions, blinking provides benefits even for the non-ideal S3 transition latencies on the order of seconds that are common in practice, as we show in Section 5.1.3. [2] Table 1 shows S3 transition latencies for a variety of platforms, as reported in [2], with the addition of Blink's OLPC-X0 nodes. The latencies include both hardware transitions as well as the time to restart the OS and reset its IP address.

DEFINITION 1. *The* blink state *of each node i is defined by two parameters that determine its duty cycle d_i,* (i) *length of the ON interval t_{on} and* (ii) *length of the OFF interval t_{off}, such that* $d_i = \frac{t_{on}}{t_{on}+t_{off}} \cdot 100\%$

DEFINITION 2. *A* blink policy *defines the* blink state *of each node in a cluster, as well as a* blink schedule *for each node.*

The blink schedule defines the clock time at which a specified node transitions its blink state to active, which in turn dictates the time at which the node turns on and goes off. The schedule allows nodes to synchronize their blinking with one another, where appropriate. For example, if node A frequently accesses disk files stored on node B, the blink policy should specify a schedule such that the nodes synchronize their active intervals. To illustrate how a data center employs blinking to regulate its aggregate energy usage, consider a scenario where the energy supply is initially plentiful and there is sufficient workload demand for all nodes. In this case, a feasible policy is to keep all nodes continuously on.

Next assume that the power supply drops by 10%, and hence, the data center must reduce its aggregate energy use by 10%. There are several blink policies that are able to satisfy this 10% drop. In the simplest case, 10% of the nodes are turned off, while the remaining nodes continue to stay on. Alternatively, another blink policy may specify a duty cycle of $d_i = 90\%$ for every node i. There are also many ways to achieve a per-server duty cycle of 90% by setting different t_{on} and t_{off} intervals, e.g., $t_{on} = 9s$ and $t_{off} = 1s$ or $t_{on} = 900ms$ and $t_{off} = 100ms$. Yet another policy may assign different blink states to different nodes, e.g., depending on their loads, such that aggregate usage decreases by 10%.

We refer to the first policy in our example above as an *activation* policy. An activation policy only varies the number of active servers at each power level [8, 37] such that some servers are active, while others are inactive; the energy supply dictates the size of the active server set. In contrast, *synchronous policies* toggle all nodes between the active and inactive state in tandem. In this case, all servers are active for t_{on} seconds and then inactive for t_{off} seconds, such that total power usage over each duty cycle matches the available power. Of course, since a synchronous policy toggles all servers to active at the same time, it does not reduce peak power, which has a significant impact on the cost of energy production. An *asynchronous* policy may randomize the start of each node's active interval to decrease peak power without changing the average power consumption across all nodes. Finally, an *asymmetric* policy may blink different nodes at different rates, while ensuring the necessary change in the energy footprint. For example, an asymmetric policy may be *load-proportional* and choose per-node blink states that are a function of current load.

All of the policies above are equally effective at capping the average power consumption for a variable power signal over any time interval. However, the choice of the blink policy greatly impacts application performance. To see why, consider two common applications: a simple cluster-based web server [8, 17] and an Hadoop cluster [20]. An activation policy is well-suited for a cluster-based web server where each node serves static content replicated on each node—by turning off a subset of nodes and evenly redirecting incoming web requests to active servers, the application is able to regulate its energy usage to match its supply. Since any node is able to service any request, transitioning a subset of the nodes to the inactive state does not pose a problem. In this case, the application requires only minimal changes to accommodate dynamic changes in the set of active servers.

However, an activation policy presents a problem for applications that maintain memory or disk state on specific nodes and exhibit inter-node dependencies. In this case, deactivating nodes will render some application-specific state unavailable. Hadoop is one example, since Hadoop's HDFS file system [31] replicates and stores data chunks of each file across many different nodes to improve both performance and data availability. As a result, simply powering down some nodes for long periods is not effective, since inactive nodes may store data necessary for a job to continue execution [4, 16, 20]. One, potentially non-optimal, option for addressing the problem without incurring the overhead of changing the data layout or migrating state prior to deactivating nodes is to leverage a synchronous blinking policy, where all nodes have the same duty cycle. Since all nodes blink between the active and inactive state in tandem, all disk state is accessible during the active periods. While the synchronous policy is not necessarily optimal for all applications and maximizes peak power demand, it does eliminate the complexities of dealing with application-specific communication patterns when determining the blink schedule. However, dealing with application-specific complexities may improve performance: an asynchronous blinking policy may reduce costs by reducing peak power or an asymmetric blinking policy may improve performance by prioritizing nodes that store heavily-accessed data.

In general, distributed applications that store per-node state require application modifications to gracefully handle blinking, and ensure the state is available at the proper times. However, since intermittent power scenarios may yield little or no power during certain periods, it is not appropriate in all application scenarios. For instance, intermittent power is not appropriate for applications that make performance guarantees. The approach is applicable in many scenarios, such as for best-effort batch jobs, including Hadoop/MapReduce jobs, or for performance optimizations that augment an always-on infrastructure. In this paper, we focus on the latter example by developing a version of memcached, called BlinkCache, that gracefully handles intermittent power constraints via blinking. While memcached's design explicitly avoids the type of inter-node dependencies present in Hadoop, and other tightly-coupled distributed applications, it is able to benefit from an

[2] PowerNap's on-demand transitions show little benefit once latencies exceed 100 milliseconds [21].

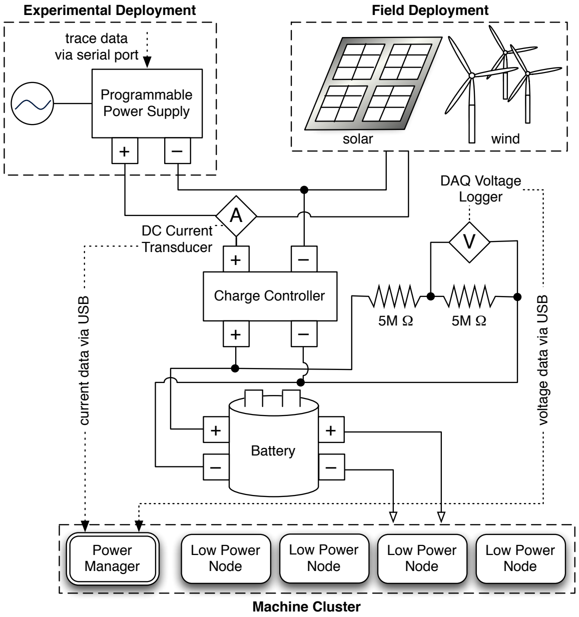


Figure 2: Hardware architecture of the Blink prototype.

asymmetric load-proportional policy, as we describe in Section 4. Additionally, nodes dedicated to memcached deployments do not require disks. Mechanical disks pose an additional constraint for blinking due to both performance and reliability concerns with frequently spinning disks up and down [41]. The effect of power cycling on the lifetime of CPUs, memory, and motherboards is an open question; we know of no prior work that addresses the issue.

3. Blink Prototype

Blink is a combined hardware/software platform for developing and evaluating blinking applications. This section describes our prototype's hardware and software architecture in detail.

3.1 Blink Hardware Platform

Blink's current hardware platform consists of two primary components: (i) a low-power server cluster that executes Blink-aware applications and (ii) a variable energy source constructed using an array of micro wind turbines and solar panels. We use renewable energy to expose the cluster to intermittent power constraints.

3.1.1 Energy Sources

We deployed an array of two wind turbines and two solar panels to power Blink. Each wind turbine is a SunForce Air-X micro-turbine designed for home rooftop deployment, and rated to produce up to 400 watts in steady 28 mph winds. However, in our measurements, each turbine generates approximately 40 watts of power on windy days. Our solar energy source uses Kyocera polycrystalline solar panels that are rated to produce a maximum of 65 watts at 17.4 volts under full sunlight. Although polycrystalline panels are known for their efficiency, our measurements show that each panel only generates around 30 watts of power in full sunlight and much less in cloudy conditions.

We assume blinking systems use batteries for short-term energy storage and power buffering. Modern data centers and racks already include UPS arrays to condition power and tolerate short-term grid disruptions. We connect both renewable energy sources in our deployment to a battery array that includes two rechargeable deep-cycle ResourcePower Marine batteries with an aggregate capacity of 1320 watt-hours at 12V, which is capable of powering our entire cluster continuously for over 14 hours. However, in this paper we focus on energy-neutral operation over short time intervals, and thus use the battery array only as a small 5-minute buffer. We connect the energy sources to the battery pack using a TriStar T-60 charge controller that provides over-charging circuitry. We deployed our renewable energy sources on the roof of a campus building in September 2009 and used a HOBO U30 data logger to gather detailed traces of current and voltage over a period of several months under a variety of different weather conditions.

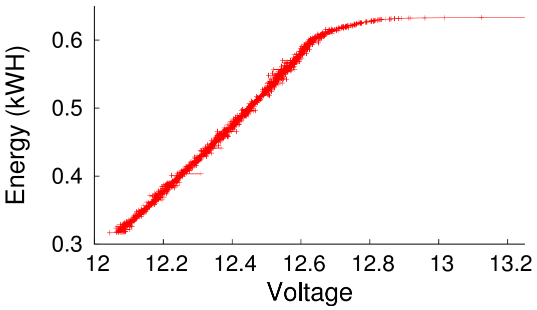


Figure 3: Empirically-measured battery capacity as a function of voltage for our deep-cycle battery. We consider the battery empty below 12V, since using it beyond this level will reduce its lifetime.

While our energy harvesting deployment is capable of directly powering Blink's server cluster, to enable controlled and repeatable experiments we leverage two Extech programmable power supplies. We use the programmable power supplies, instead of the harvesting deployment, to conduct repeatable experiments by replaying harvesting traces, or emulating other intermittent power constraints, to charge our battery array. [3]

Since the battery's voltage level indicates its current energy capacity, we require sensors to measure and report it. We use a data acquisition device (DAQ) from National Instruments to facilitate voltage measurement. As shown in Figure 2, the prototype includes two high-precision 5MOhm resistors between the battery terminals and employs the DAQ to measure voltage across each resistor. We then use the value to compute the instantaneous battery voltage, and hence, capacity. Figure 3 shows the empirically-derived capacity of our prototype's battery as a function of its voltage level. In addition to battery voltage, we use DC current transducers to measure the current flowing from the energy source into the battery, and the current flowing from the battery to the cluster. The configuration allows Blink to accurately measure these values every second.

3.1.2 Low-power Server Cluster

Our Blink prototype uses a cluster of low-power server nodes. Since our energy harvesting deployment is only capable of producing 100-140 watts of power, a cluster of traditional high-power servers, such as Xeon servers that consume roughly 500W each, is not feasible. As a result, we construct our prototype from low-power nodes that use AMD Geode processor motherboards. Each motherboard, which we scavenge from OLPC-XO laptops, consists of a 433 MHz AMD Geode LX CPU, 256 MB RAM, a 1GB solid-state flash disk, and a Linksys USB Ethernet NIC. Each node runs the Fedora Linux distribution with kernel version 2.6.25. We con-

[3] We are able to set the initial battery level for each experiment using a separate charge controller in load-control mode.

Blinking Interface
`setDutyCycle(int nodeId, int onPercentage)`
`setBlinkInterval(int nodeId, int interval)`
`syncActiveTime(int node, long currentTime)`
`forceSleep(int nodeId, int duration)`

Table 2: Blink APIs for setting per-node blinking schedules.

Measurement Interface
`getBatteryCapacity()`
`getBatteryEnergy()`
`getChargeRate(int lastInterval)`
`getDischargeRate(int lastInterval)`
`getServerLoadStats(int nodeId)`

Table 3: Blink's measurement APIs that applications use to inform their blinking decisions.

nect our 10 node cluster together using 2 energy-efficient 8-port Rosewill 100 Mbps switches. Each low-power node consumes a maximum of 8.6W, and together with the switch, the 10 node cluster has a total energy footprint of under 100 watts, which closely matches the energy generated from our renewable energy sources.

Similar clusters of low-power nodes, e.g., using Intel Atom processors, are currently being considered by data centers for energy-efficient processing of I/O-intensive workloads [5]. Our low-power Blink design should also scale to traditional Xeon-class servers for appropriately sized energy sources, although, as we discuss in Section 5, an application's performance may differ for higher-power nodes. An advantage of using XO motherboards is that they are specifically optimized for rapid S3 transitions that are useful for blinking. Further, the motherboards use only 0.1W in S3 and 8.6W in S0 at full processor and network utilization. The wide power range in these two states combined with the relatively low power usage in S3 makes these nodes an ideal platform for demonstrating the efficacy of Blink's energy optimizations.

3.2 Blink Software Architecture

Blink's software architecture consists of an application-independent control plane that combines a power management service with per-node access to energy and node-level statistics. Blink-aware applications interact with the control plane using Blink APIs to regulate their power consumption. The power management service consists of a power manager daemon that runs on a gateway node and a power client daemon that runs on each cluster node. The architecture separates mechanism from policy by exposing a single simple interface for applications to control blinking for each cluster node.

The power manager daemon has access to the hardware sensors, described above, that monitor the battery voltage and current flow. Each Blink power client also monitors host-level metrics on each cluster node and reports them to the power manager. These metrics include CPU utilization, network bandwidth, and the length of the current active period. The power client exposes an internal RPC interface to the power manager that allows it to set a node's blinking pattern. To set the blinking pattern, the power client uses the timer of the node's real-time clock (RTC) to automatically sleep and wake up, i.e., transition back to S0, at specific intervals. Thus, the power client is able to set repetitive active and inactive durations. For example, the power manager may set a node to repeatedly be active for 50 seconds and inactive for 10 seconds. In this case, the blink interval is 60 seconds with the node being active 83% of the time and inactive 17% of the time. We assume that nodes synchronize clocks using a protocol such as NTP to enable policies that coordinate blink schedules across cluster nodes.

The impact of clock synchronization is negligible for our blink intervals at the granularity of seconds, but may become an issue for blink intervals at the granularity of milliseconds or less. Note that clock synchronization is not an issue for applications, such as memcached, that do not perform inter-node communication. Transitioning between S0 and S3 incurs a latency that limits the length of the blink interval. Shorter blink intervals are preferable since they allow each node to more closely match the available power, more rapidly respond to changes in supply, and reduces the battery capacity necessary for short term buffering. The XO motherboard yields S3 sleep latencies that range from roughly 200 milliseconds to 2 seconds depending on the set of active devices and drivers (see Table 1). For instance, since our USB NIC driver does not implement the ACPI `reset_resume` function, we must unload and load its driver when transitioning to and from S3. As a result, the latency for our experiments is near 2 seconds. Unfortunately, inefficient and incorrect device drivers are commonplace, and represent one of the current drawbacks to blinking in practice.

The Blink control plane exposes an RPC interface to integrate with external applications as shown in Tables 2 and 3. Applications use these APIs to monitor input/output current flow, battery voltage, host-level metrics and control per-node blinking patterns. Since Blink is application-independent, the prototype does not report application-level metrics. In such cases, an application must monitor itself. For instance, for some policies in our blinking version of memcached, a proxy monitors per-key hit rates, as described in Section 4.3.1.

4. Blinking Memcached

Memcached is a distributed in-memory cache for storing key-value pairs that significantly reduces both the latency to access data objects and the load on persistent disk-backed storage. Memcached has become a core component in Internet services that store vast amounts of user-generated content, with services maintaining dedicated clusters with 100s to 1000s of nodes [26]. Since end users interact with these services in real-time through web portals, low-latency access to data is critical. High page load latencies frustrate users and may cause them to stop generating new content [25], which is undesirable since these services' primary source of revenue derives from their content, e.g., by selling targeted ads.

4.1 Memcached Overview

Memcached's design uses a simple and scalable client-server architecture, where clients request a key value directly from a single candidate memcached server with the potential to store it. Clients use a built-in mapping function to determine the IP address of this candidate server. Initial versions of memcached determined the server using the function `Hash(Key)%NumServers`, while the latest versions use the same consistent hashing approach popularized in DHTs, such as Chord [34]. In either case, the key values randomly map to nodes without regard to their temporal locality, i.e., popularity. Since all clients use the same mapping function, they need not communicate with other clients or servers to compute which server to check for a given key. Likewise, Memcached servers respond to client requests (gets and sets) without communicating with other clients or servers. This lack of inter-node communication enables Memcached to scale to large clusters.

Importantly, clients maintain the state of the cache, including its consistency with persistent storage. As a result, applications are explicitly written to use memcached by (i) checking whether an object is resident in the cache before issuing any subsequent queries, (ii) inserting a newly referenced object into the cache if it is not already resident, and (iii) updating a cached object to reflect

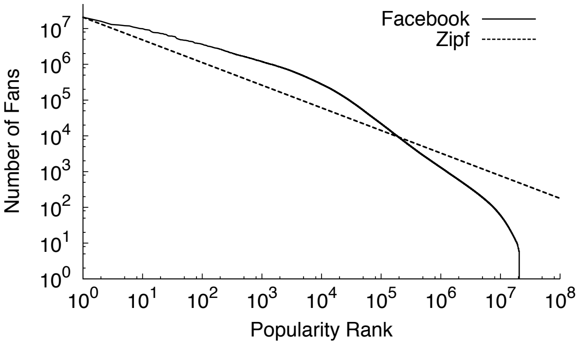


Figure 4: The popularity rank, by number of fans, for all 20 million public group pages on Facebook follows a Zipf-like distribution with $\alpha = 0.6$.

a corresponding update in persistent storage. Each memcached server uses the Least Recently Used (LRU) replacement policy to evict objects. One common example of a cached object is an HTML fragment generated from the results of multiple queries to a relational database and other services. Since a single HTTP request for many Internet services can result in over 100 internal, and potentially sequential, requests to other services [10, 26], the cache significantly decreases the latency to generate the HTML.

4.2 Access Patterns and Performance Metrics

The popularity of web sites has long been known to follow a Zipf-like distribution [7, 39], where the fraction of all requests for the i-th most popular document is proportional to $1/i^\alpha$ for some constant α. Previous studies [7, 39] have shown that α is typically less than one for web site popularity. The key characteristic of a Zipf-like distribution is its heavy tail, where a significant fraction of requests are for relatively unpopular objects. We expect the popularity of user-generated content for an Internet service to be similar to the broader web, since, while some content may be highly popular, such as a celebrity's Facebook page, most users are primarily interested in either their own content or the content of close friends and associates.

As a test of our expectation, we rank all 20 million user-generated fan pages on Facebook by their number of fans. We use the size of each page's fan base as a rough approximation of the popularity of its underlying data objects. Figure 4 confirms that the distribution is Zipf-like with α approximately 0.6. Recent work also states that Facebook must store a significant fraction of their data set in massive memcached cluster, i.e., on the order of 2000 nodes, to achieve high hit rates, e.g., 25% of the entire data set to achieve a 96.5% hit rate [26]. This characteristic is common for Zipf-like distributions with low α values, since many requests for unpopular objects are inside the heavy tail. Thus, the distribution roughly divides objects into two categories: the few highly popular objects and the many relatively unpopular objects. As cache size increases, it stores a significant fraction of objects that are unpopular compared to the few popular objects, but nearly uniformly popular compared to each other. These mega-caches resemble a separate high-performance storage tier [26] for all data objects, rather than a small cache for only the most popular data objects.

Before discussing different designs alternatives for BlinkCache, we define our performance metrics. The primary cache performance metric is hit ratio, or hit rate, which represents the percentage of object requests that the cache services. A higher hit rate indicates both a lower average latency per request, as well as lower load on the back-end storage system. In addition to hit rate, we argue that fairness should be a secondary performance metric for large memcached clusters that store many objects of equal popularity. A fair cache distributes its benefits—low average request latency—equally across objects. Caches are usually unfair, since their primary purpose is to achieve high hit rates by storing more popular data at the expense of less popular data. However, fairness increases in importance when there are many objects with a similar level of popularity, as in today's large memcached clusters storing data that follows a Zipf-like popularity distribution. An unfair cache results in a wide disparity in the average access latency for these similarly popular objects, which ultimately translates to end-users receiving vastly different levels of performance. We use the standard deviation of average request latency per object as our measure of fairness. The lower the standard deviation the more fair the policy, since this indicates that objects have average latencies that are closer to the mean.

4.3 BlinkCache Design Alternatives

We compare variants of three basic memcached policies for variable power constraints: an activation policy, a synchronous policy, and an asymmetric load-proportional policy. In all cases, any get request to an inactive server always registers as a cache miss, while any set request is deferred until the node becomes active. We defer a discussion of the implementation details using Blink to the next section.

- **Activation Policy**. An activation policy ranks servers $1...N$ and always keeps the top M servers active, where M is the maximum number of active servers the current power level supports. We discuss multiple activation variants, including a *static* variant that does not change the set of available servers in each client's built-in mapping function to reflect the current set of active servers, and a *dynamic* variant that does change the set. We also discuss a *key migration* variant that continuously ranks the popularity of objects and migrates them to servers $1...N$ in rank order.
- **Synchronous Policy**. A synchronous policy keeps all servers active for time t and inactive for time $T - t$ for every interval T, where t is the maximum duration the current power level supports and T is short enough to respond to power changes but long enough to mitigate blink overhead. The policy does not change the set of available servers in each client's built-in mapping function, since all servers are active every interval.
- **Load-Proportional Policy**. A load-proportional policy monitors the aggregate popularity of objects P_i that each server i stores and keeps each server active for time t_i and inactive for time $T - t_i$ for every interval T. The policy computes each t_i by distributing the available power in the same proportion as the aggregate popularity P_i of the servers. The load-proportional policy also migrates similarly popular objects to the same server.

4.3.1 Activation Policy

A straightforward approach to scaling memcached as power varies is to activate servers when power is plentiful and deactivate servers when power is scarce. One simple method for choosing which servers to activate is to rank them $1...N$ and activate and deactivate them in order. Since, by default, memcached maps key values randomly to servers, our policy for ranking servers and keys is random. In this case, a static policy variant that does not change each client's built-in mapping function to reflect the active server set arbitrarily favors keys that happen to map to higher ranked servers, regardless of their popularity. As a result, requests for objects that map to the top-ranked server will see a significantly lower average latency than requests for objects that happen to map to the bottom-ranked

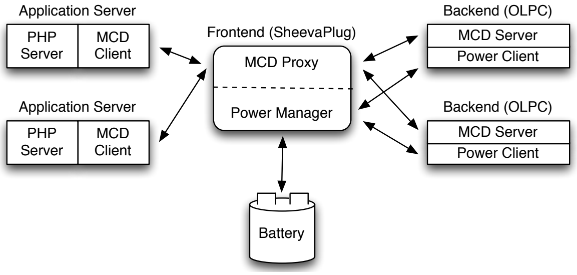


Figure 5: To explicitly control the mapping of keys to servers, we interpose always-active request proxies between memcached clients and servers. The proxies are able to monitor per-key hit rates and migrate similarly popular objects to the same nodes.

server. One way to correct the problem is to dynamically change the built-in client mapping function to only reflect the current set of active servers. With constant power, dynamically changing the mapping function will result in a higher hit rate since the most popular objects naturally shift to the current set of active servers.

Hash-based Key Mapping. Memcached recently added support for consistent hashing to reduce the disruption from changing a cluster's size. The original function (`Hash(Key)%NumServers`) invalidates nearly every key when adding or removing a single server from a cluster of size n. Only the keys that have both the old `NumServers` and the new `NumServers` as common factors do not change mappings. The approach is clearly not scalable, since each change, regardless of size, flushes nearly the entire cache, which abruptly increases load on the back-end storage system, as well as request latency, until the cache re-populates itself.

Consistent hashing, originally popularized [34] by DHTs, significantly improves the situation, since adding or removing each server only invalidates $1/n^{th}$ of the keys for a cluster of size n. The approach scales gracefully, since the percentage of keys the cache invalidates by changing a single server in the active set decreases as cluster size increases. However, consistent hashing is not a panacea when power varies either frequently or significantly. For instance, if the available power doubles, thereby doubling the number of active servers, a consistent hashing approach will still invalidate 50% of its resident objects. With a dynamic approach, frequent variations in power repeatedly incur this steep invalidation penalty.

Thus, while our dynamic variant results in a higher hit rate than a static variant under constant power, the opposite is true, due to invalidation penalties, for power that varies frequently or significantly. One option for eliminating invalidation penalties entirely is to explicitly control the mapping of individual keys to servers, and pro-actively migrate the most popular objects to the most popular servers. Figure 5 illustrates a memcached design that interposes an always-active proxy between memcached clients and servers to control the mapping. In this design, clients issue requests to the proxy, which maintains a hash table that stores the current mapping of keys to servers, issues requests to the appropriate back-end server, and returns the result to the client.

Table-based Key Mapping. Since all requests pass through the proxy, it is able to continuously monitor and sort the popularity of objects in the background and dynamically change the server mappings as popularities change. Note that to migrate an object the proxy need only change its key→server mapping. After the change, the next key request for the object will incur one cache miss on the new server, which results in application-level code re-generating and re-inserting the object at the new location. The proxy may either pro-actively evict the object from the old server or simply allow LRU replacement to evict the object. This strategy eliminates invalidation penalties, since popularity-based migration always places the most popular objects on the highest-ranked servers. The design requires no changes to either memcached clients or servers.

Deactivating lower-ranked servers invalidates objects that are already less popular than objects on the higher-ranked active servers, while activating additional servers grows the size of the cache without invalidating the more popular objects that are already resident. The approach does introduce the overhead of monitoring and sorting keys by popularity, as well as proxying requests through an intermediary server. However, these overhead costs are independent of power variations and amortized over time, rather than imposed abruptly at each change in the power level. The proxy approach scales to multiple proxies by allowing memcached clients to use their original built-in mapping function to map keys to a set of proxies instead of a set of servers, such that each proxy is responsible for a random subset of keys.

In a multi-proxy design, the proxies periodically send popularity statistics to a central server that sorts them in the background and distributes each proxy a new server mapping. The period between re-mappings is a function of both how fast key popularity is changing and the number of keys in the cache, since drastic popularity changes require re-mappings and more keys increase the time to gather, sort, and distribute new mappings. Changes to server mappings need not be highly synchronized across all proxies, though, since memcached provides no guarantees that data is present or consistent with a back-end database. A multi-proxy design allows the proxies to share the bandwidth and processing load of issuing client requests to servers, while maintaining the ability to explicitly control key mappings.

We expect a few-to-many relationship between proxies and memcached servers, although highly-loaded memcached clusters may require a mapping as low as one-to-one. In this case, the argument for using proxies instead of the nodes themselves mirrors Somniloquy's argument for using low-power, augmented network interfaces to perform simple application functions on behalf of sleeping nodes [2]. Another consideration for a proxy-based migration approach is the load imposed on the highest-ranked memcached servers. If the highest-ranked servers become overloaded, the re-mapping process may need to place a few less popular keys on high-ranked nodes to ensure they are not overloaded. The technique is analogous to Popularity-based Data Concentration (PDC) for disk arrays that must be careful not to deactivate too many disks and then overload the remaining active disks [28].

4.3.2 Synchronous Policy

The migration-enabled activation policy, described above, approaches the optimal policy for maximizing the cache's hit rate, since ranking servers and mapping objects to them according to popularity rank makes the distributed cache operate like a centralized cache that simply stores the most popular objects regardless of the cache's size. We define optimal as the hit rate for a centralized cache of the same size as the distributed Memcached instance under the same workload. However, the policy is unfair for servers that store similarly popular objects, since these servers should have equal rankings. The activation policy is forced to arbitrarily choose a subset of these equally ranked servers to deactivate. In this case, a synchronous policy is significantly more fair and results in nearly the same hit rate as the optimal activation policy. To see why, consider the simple 4-node memcached cluster in Figure 6 with enough available power to currently activate half the cluster. There is enough power to support either (i) our activation policy with migration that keeps two nodes continuously active or (ii) a

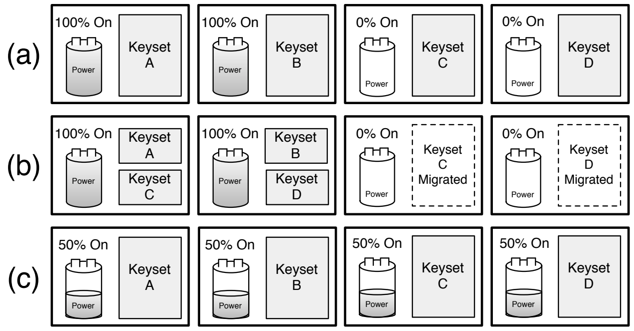


Figure 6: Graphical depiction of a static/dynamic activation blinking policy (a), an activation blinking policy with key migration (b), and a synchronous blinking policy (c).

synchronous policy that keeps four nodes active half the time but synchronously blinks them between the active and inactive state.

For now we assume that all objects are equally popular, and compare the expected hit rate and standard deviation of average latency across objects for both policies, assuming a full cache can store all objects at full power on the 4 nodes. For the activation policy, the hit rate is 50%, since it keeps two servers active and these servers store 50% of the objects. Since all objects are equally popular, migration does not significantly change the results. In this case, the standard deviation is 47.5ms, assuming an estimate of 5ms to access the cache and 100ms to regenerate the object from persistent storage. For a synchronous policy, the hit rate is also 50%, since all 4 nodes are active half the time and these nodes store 100% of the objects. However, the synchronous policy has a standard deviation of 0ms, since all objects have a 50% hit probability, if the access occurs when a node is active, and a 50% miss probability, if the access occurs when a node is inactive. Rather than half the objects having a 5ms average latency and half having a 100ms average latency, as with activation, a synchronous policy ensures an average latency of 52.5ms across all objects.

Note that the synchronous policy is ideal for a normal memcached cluster with a mapping function that randomly maps keys to servers, since the aggregate popularity of objects on each server will always be roughly equal. Further, unlike an activation policy that uses the dynamic mapping function, the synchronous policy does not incur invalidation penalties and is not arbitrarily unfair to keys on lower-ranked servers.

4.3.3 Load-Proportional Policy

A synchronous policy has the same hit rate as an activation policy when keys have the same popularity, but is significantly more fair. However, an activation policy with migration is capable of a significantly higher hit rate for highly skewed popularity distributions. A proportional policy combines the advantages of both approaches for Zipf-like distributions, where a few key values are highly popular but there is a heavy, but significant, tail of similarly unpopular key values. As with our activation policy, a proportional policy ranks servers and uses a proxy to monitor object popularity and migrate objects to servers in rank order. However, the policy distributes the available power to servers in the same proportion as the aggregate popularity of their keys.

For example, assume that in our 4 server cluster after key migration the percentage of total hits that go to the first server is 70%, the second server is 12%, the third server is 10%, and the fourth server is 8%. If there is currently 100W of available power then the first server ideally receives 70W, the second server 12W, the third server 10W, and the fourth server 8W. These power levels then translate directly to active durations over each interval T. In practice, if the first server's maximum power is 50W, then it will be active the entire interval, since its maximum power is 70W. The extra 20W is distributed to the remaining servers proportionally. If all servers have a maximum power of 50W, the first server receives 50W, the second server receives 20W, i.e., 40% of the remaining 50W, the third server receives 16.7W, and the fourth server receives 13.3W. These power levels translate into the following active durations for a 60 second blink interval: 60 seconds, 24 seconds, 20 seconds, and 16 seconds, respectively.

Metric	*Workload*	*Best Policy*
Hit Rate	Uniform	Synchronous
Hit Rate	Zipf	Activation (Migration)
Fairness	Uniform/Zipf	Synchronous
Fairness + Hit Rate	Zipf	Load-Proportional

Table 4: Summary of the best policy for a given performance metric and workload combination.

The hit rate from a proportional policy is only slightly worse than the hit rate from the optimal activation policy. In this example, we expect the hit rate from an activation policy to be 85% of the maximum hit rate from a fully powered cluster, while we expect the hit rate from a proportional policy to be 80.2%. However, the policy is more fair to the 3 servers—12%, 10%, and 8%—with similar popularities, since each server receives a similar total active duration. The Zipf distribution for a large memcached cluster has similar attributes. A few servers store highly popular objects and will be active nearly 100% of the time, while a large majority of the servers will store equally unpopular objects and blink in proportion to their overall unpopularity.

4.4 Summary

Table 4 provides a summary of the best policy for each performance metric and workload combination. In essence, an activation policy with key migration will always have the highest hit rate. However, for distributions with equally popular objects, the synchronous policy achieves a similar hit rate and is more fair. A load-proportional policy combines the best attributes of both for Zipf-like distributions, which include a few popular objects but many similarly unpopular objects.

5. Implementation and Evaluation

We implement and evaluate the BlinkCache design alternatives from the previous section using our small-scale Blink prototype. The purpose of our evaluation is not to maximize the performance of our particular memcached deployment or improve on the performance of the custom memcached server deployments common in industry. Instead, our goal is to explore the effects of churn on memcached caused by power fluctuations for different BlinkCache designs. Our results will differ across platforms according to the specific blink interval, CPU speed, and network latency and bandwidth of the servers and the network. Since our prototype uses low-power CPUs and motherboards, the request latencies we observe in our prototype are not representative of those found in high performance servers.

Each node in Blink connects to a low-power (2.4W/switch) 100 Mbps switch and runs an instance of Blink's power client and an unmodified memcached server. We wrote a memcached client workload generator to issue key requests at a configurable, but steady, rate according to either a Zipf popularity distribution, parameterized by α, or a uniform popularity distribution. As in a typical application, the workload generator fetches any data not resi-

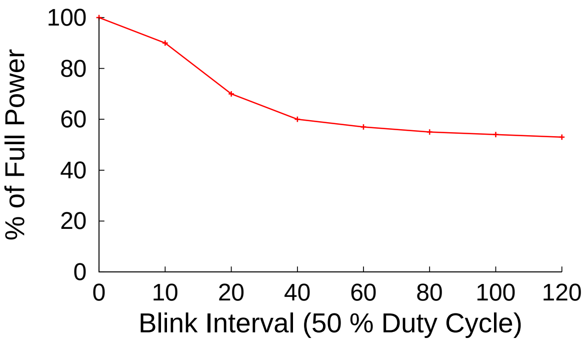


Figure 7: The near 2 second latency to transition into and out of S3 in our prototype discourages blinking intervals shorter than roughly 40 seconds. With a 50% duty cycle we expect to operate at 50% full power, but with a blink interval of less than 10 seconds we operate near 100% full power.

dent in the cache from a MySQL database and places it in the cache. Since we assume the MySQL server provides always-available persistent storage, it runs off the power grid and not variable power. Note that for our benchmarks we take the conservative approach of simply fetching key values directly from MySQL with a single query, and do not imitate the sequential, multi-query nature of production web applications. While we do not model or evaluate the impact of workloads that include multi-get memcached requests that issue gets for multiple keys in parallel, blinking with key migration should not impact the performance of multi-get requests for keys with similar popularities, e.g., part of the same HTML page, since our proxy will migrate these keys to the same server.

Unless otherwise noted, in our experiments, we use moderate-size objects of 10 kilobytes, Facebook-like Zipf α values of 0.6, and memcached's consistent hashing mapping function. Each experiment represents a half-hour trace, we configure each memcached server with a 100MB cache to provide an aggregate cache size of 1GB, and we use our programmable power supply to drive each power trace. Since each node has only 256MB of memory, we scale our workloads appropriately for evaluation. We modify magent, a publicly available memcached proxy, [4] to implement the design alternatives in the previous section, including table-based key mapping and popularity-based key migration. Our modifications are not complex: we added or changed only 300 lines of code to implement all of the BlinkCache design variants from Section 3. Since all requests pass through the proxy, it is able to monitor key popularity. The proxy controls blinking by interacting with Blink's power manager, which in our setup runs on the same node, to monitor the available power and battery level and set per-node blinking patterns. We also use the proxy for experiments with memcached's default hash-based key mappings, rather than modifying the memcached client. Since our always-on proxy is also subject to intermittent power constraints, we run it on a low-power (5W) embedded SheevaPlug with a 1.2 GHz ARM CPU and 512 MB of memory.

We first use our workload generator to benchmark the performance of each blinking policy for both Zipf-like and uniform popularity distributions at multiple power levels with varying levels of oscillation. We then demonstrate the performance for an example web application—tag clouds in GlassFish—using realistic traces from our energy harvesting deployment that have varying power and oscillation levels.

5.1 Benchmarks

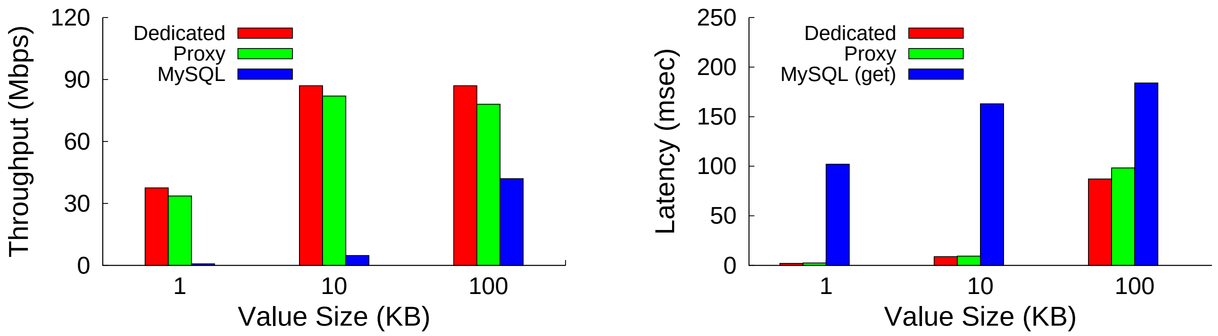
We measure the maximum power of each node, at 100% CPU and network utilization, in S0 to be 8.6W and its minimum power in S3 to be 0.2W. We use these values in the proxy to compute the length of active and inactive periods to cap power consumption at a specific level. We also measure the impact of our node's near 2 second transition latency for blink intervals T between 10 seconds and 2 minutes. Figure 7 shows the results for a duty cycle of 50%. In this case, the blinking interval must be over 40 seconds before average power over the interval falls below 55% of the node's maximum power, as we expect. The result shows that on-demand transitions that occur whenever work arrives or departs are not practical in our prototype. Further, even blinking intervals as high as 10 seconds impose significant power inefficiencies. As a result, we use a blinking interval of 60 seconds for our experiments. Our 60 second blink interval is due solely to limitations in the Blink prototype. Note that there is an opportunity to significantly reduce blink intervals through both hardware and software optimizations. Since server clusters do not typically leverage ACPI's S3 state, there has been little incentive to optimize its transition latency.

Next, we determine a baseline workload intensity for memcached, since, for certain request rates and key sizes, the proxy or the switch becomes a bottleneck. In our experiments, we use a steady request rate (1000 get requests/sec) that is less than the maximum request rate possible once the proxy or switch becomes a bottleneck. Note that our results, which focus on hit rates, are a function of the popularity of objects rather than the distribution of request inter-arrival times. Our goal is to evaluate how blinking affects the relative hit rates between the policies, and not the performance limitations of our particular set of low-power components. Figure 8 demonstrates the maximum performance, in terms of total throughput and request latency for different key values sizes, of an unmodified memcached server, our memcached proxy, and a MySQL server. As expected, the memcached server provides an order of magnitude higher throughput and lower request latency than MySQL. Further, our proxy implementation imposes only a modest overhead to both throughput and latency, although the latency impact of proxy-based redirections will be greater on faster CPUs since less relative request time is spent in the OS and network. Our subsequent experiments focus on request hit rates rather than request latencies, since latencies vary significantly across platforms and workloads. Further, the wide disparity in latency between serving a request from memory and serving it from disk would show a larger, and potentially unfair, advantage for a blinking system. Thus, we consider hit rate a better metric than latency for evaluating a blinking memcached instance.

5.1.1 Activation Blinking and Thrashing

An activation policy for an unmodified version of memcached must choose whether or not to alter the hash-based mapping function as it activates and deactivates servers. For constant power, a dynamic mapping function that always reflects the currently active set of servers should provides the best hit rate, regardless of the popularity distribution, since applications will be able to insert the most popular keys on one of the active servers. Figure 9 demonstrates this point for a workload with a Zipf popularity distribution ($\alpha = 0.6$), and shows the hit rates for both static and dynamic activation variants at multiple constant power levels. While at high power levels the approaches have similar hit rates, as power level decreases, we see that the static variant incurs a higher penalty under constant power. However, Figure 10 demonstrates that the opposite is true for highly variable power. The figure reports hit rates for different levels of power oscillation, where the average power for each experiment is 45% of the power necessary to run all nodes concurrently. The x-axis indicates oscillation level as a percentage,

[4] http://code.google.com/p/memagent/



(a) Throughput

(b) Latency

Figure 8: Maximum throughput (a) and latency (b) for a dedicated memcached server, our memcached proxy, and a MySQL server. Our proxy imposes only a modest overhead compared with a dedicated memcached server.

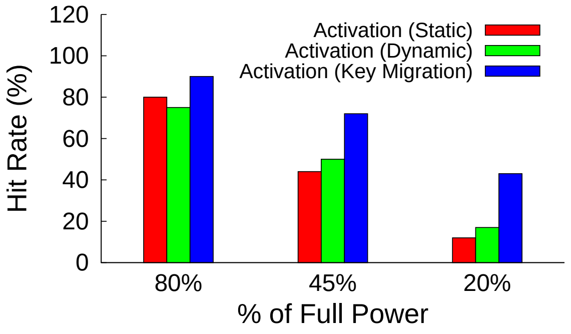


Figure 9: Under constant power for a Zipf popularity distribution, the dynamic variant of the activation policy performs better than the static variant as power decreases. However, the activation policy with key migration outperforms the other variants.

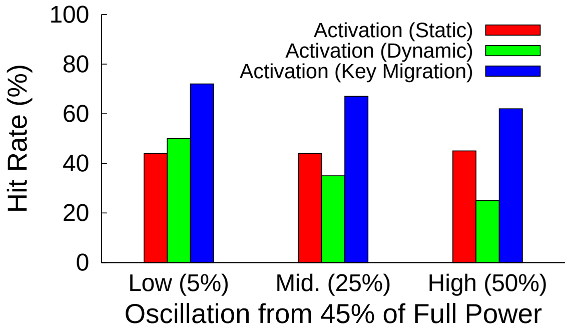


Figure 10: Under oscillating power for a Zipf popularity distribution, the static variant of the activation policy performs better than the dynamic variant as the oscillation increases. Again, the activation policy with key migration outperforms the other variants.

such that 0% oscillation holds power steady throughout the experiment and $N\%$ oscillation varies power between $(45 + 0.45N)\%$ and $(45 - 0.45N)\%$ every 5 minutes.

We see that dynamic changes in the active server set of memcached's hash-based mapping function incur an invalidation penalty. Since the invalidation penalty does not occur when memcached does not change the mapping function, the static variant provides a significantly better hit rate as the oscillations increase. Although not shown here, the difference with the original modulo approach is much greater, since each change flushes nearly the entire cache. The hash-based mapping function forces a choice between performing well under constant power or performing well under variable power. A table-based approach that uses our proxy to explicitly map keys to servers and uses key migration to increase the priority of popular keys performs better in both scenarios. That is, the approach does not incur invalidation penalties under continuously variable power, or result in low hit rates under constant power, as also shown in Figure 9 and Figure 10. Note that oscillation has no impact on other policies, e.g. those using key migration or the synchronous policy.

5.1.2 Synchronous Blinking and Fairness

While the activation policy with key migration results in the highest hit rate overall, it is unfair when many servers store equally popular objects since the policy must choose some subset of equally popular servers to deactivate. Figure 11 quantifies the fairness of the dynamic activation policy, the activation policy with key migration, and the synchronous policy, as a function of standard deviation in average per-object latency, at multiple constant power levels for a uniform popularity distribution where all objects are equally popular. Note that for distributions where all objects are equally popular, key migration is not necessary and is equivalent to using the static variant of hash-based mapping.

The synchronous policy is roughly 2X more fair than the activation policy with key migration at all power levels. While the dynamic hash-based mapping is nearly as fair as the synchronous policy, it has a worse hit rate, especially in high-power scenarios, as shown in Figure 12. Thus, the synchronous policy, which is more fair and provides lower average latency, is a better choice than any variant of the activation policy for uniform popularity distributions. Note that the key popularity distribution across servers in *every* memcached cluster that uses a hash-based mapping function is uniform, since keys map to servers randomly. Thus, the synchronous policy is the best choice for a heavily-loaded memcached cluster that cannot tolerate the throughput penalty of using proxies.

5.1.3 Balancing Performance and Fairness

Activation with key migration results in the maximum hit rate for skewed popularity distributions where some objects are significantly more popular than others, while the synchronous policy re-

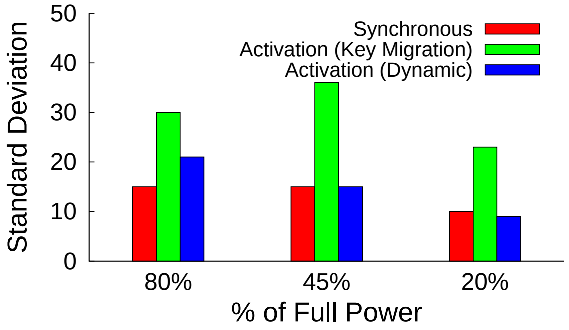


Figure 11: For a uniform popularity distribution, both the synchronous policy and the dynamic variant of the activation policy are significantly more fair, i.e., lower standard deviation of average per-object latency, than the activation policy with key migration.

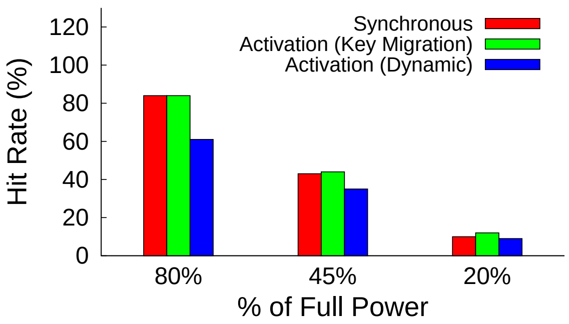


Figure 12: For a uniform popularity distribution, the synchronous policy and the activation policy with key migration achieve a similar hit rate under different power levels. Both policies achieve a better hit rate than the dynamic variant of the activation policy.

sults in the best overall performance, in terms of both hit rate and fairness, for uniform popularity distributions. The proportional policy combines the advantages of both and works well for Zipf-like distributions with a few popular objects but a long tail of similarly (un)popular objects, since the long heavy tail in isolation is similar to the uniform distribution. Figure 14 shows the hit rate for the proportional policy, the activation policy with migration, and the synchronous policy for a Zipf popularity distribution with $\alpha = 0.6$ at different power levels. The synchronous policy performs poorly, especially at low power levels, in this experiment, since it does not treat popular objects different than unpopular objects.

However, the proportional policy attains nearly the same hit rate as the activation policy at high power levels, since it also prioritizes popular objects over unpopular objects. Even at low power levels its hit rate is over 60% of the activation policy's hit rate. Further, the proportional policy is significantly more fair to the many unpopular objects in the distribution. Figure 13 reports fairness, in terms of the standard deviation in per-object latency, at different power levels for the unpopular keys, i.e., keys ranked in the bottom 80th percentile of the distribution. The activation policy's unfairness is nearly 4X worse at low power levels. Thus, the proportional policy strikes a balance between performance and fairness when compared against both the synchronous and activation policies.

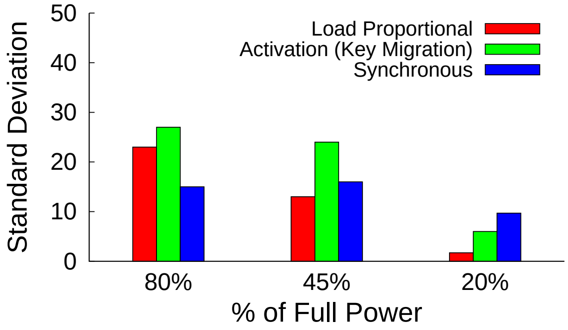


Figure 13: The load-proportional policy is more fair to the unpopular objects, i.e. bottom 80% in popularity, than the activation policy with key migration for Zip popularity distributions, especially in low-power scenarios.

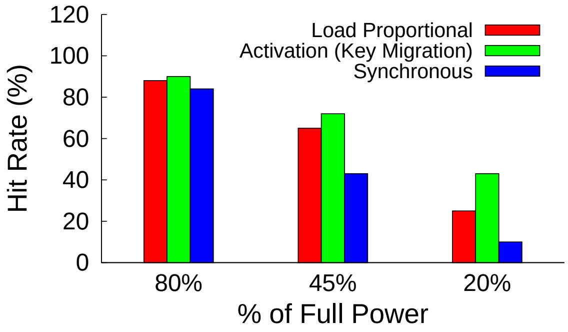


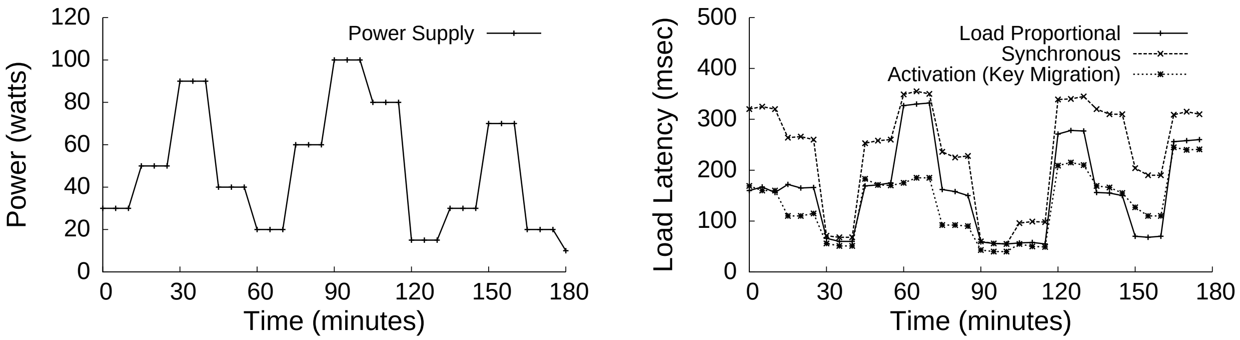
Figure 14: The load-proportional policy has a slightly lower hit rate than the activation policy with key migration.

Finally, Figure 15 shows how the S3 transition overhead affects our results at a moderate power level. The figure shows that the overhead has only a modest effect on the load-proportional policy's hit rate. The overhead does not affect the relative fairness of the policies. Note that all of our previous experiments use our prototype's 2 second transition overhead. A shorter transition overhead would improve our results, and even a longer transition would show some, albeit lesser, benefits.

5.2 Case Study: Tag Clouds in GlassFish

While our prior experiments compare our blinking policies for different power and oscillation levels, we also conduct an application case study using traces from our energy harvesting deployment. The experiment provides a glimpse of the performance tradeoffs for realistic power signals. GlassFish is an open source Java application server from Sun, which includes a simple example application that reuses parts of the Java PetStore multi-tier web application, used in prior research, e.g., [9], to create tag clouds for pets. Tag clouds are a set of weighted tags that visually represent the most popular words on a web page. We modify the default web application to generate HTML for per-user tag cloud pages and cache them in memcached. The data to construct each HTML page comes from a series of 20 sequential requests to a MySQL database.

For these experiments, we measure the latency to load user tag cloud pages, which incorporates MySQL and HTML regeneration latencies whenever HTML pages are not resident in the cache.



(a) Power

(b) Page Load Latency

Figure 16: Power signal from a combined wind/solar deployment (a) and average page load latency for that power signal (b).

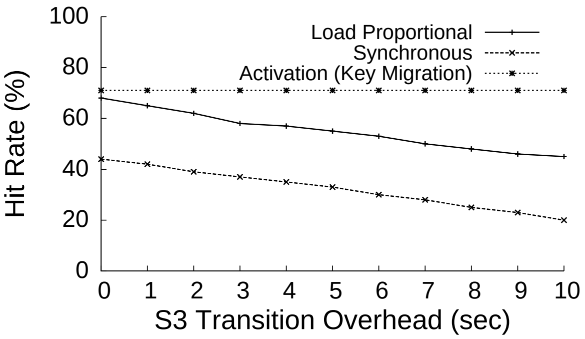


Figure 15: As S3 transition overhead increases, the hit rate from the load-proportional policy decreases relative to the activation policy with key migration for a Zipf distribution at a moderate power level.

The MySQL latency for our simple table-based data is typically 30ms per database query. While page load latency follows the same trend as hit rate, it provides a better application-level view of the impact of different policies. Figure 16(b) shows the average latency to load user web pages across 40,000 users for our three different policies—activation with key migration, proportional, and synchronous—for a combined solar and wind trace, assuming the popularity of each user's tag cloud page follows a Zipf distribution with $\alpha = 0.6$. We derive the power signal, shown in Figure 16(a), by compressing a 3-day energy harvesting trace to 3 hours.

As expected, the activation policy with key migration and the load-proportional policy exhibit comparable page load latencies at most points in the trace. For this trace, the load-proportional policy is within 15% of the activation policy's hit rate. The activation policy is slightly better at low energy levels, since it tends to strictly ensure that more popular content is always cached. Also as expected, the synchronous policy tends to perform poorly across all power levels. Also as expected, we measure the standard deviation of page load latencies for the load-proportional policy to be within 2% to the synchronous policy for the vast majority, i.e., bottom 80%, of the equally unpopular objects.

6. Related Work

The sensor network community has studied strategies for dealing with variable sources of renewable power, since these systems often do not have access to the power grid. However, since sensor networks are geographically distributed, each node must harvest its own energy, resulting in network-wide energy imbalances [11], whereas data center nodes share a common power delivery infrastructure. Further, the primary performance metric for a sensor network is the amount of data the network collects. As a result, much of the energy harvesting work is not directly applicable to data centers. Similarly, mobile computing generally focuses on extending battery life by regulating power consumption [40], rather than modulating performance to match energy production.

The increasing energy consumption of data centers [1] has led companies to invest heavily in renewable energy sources [22, 35]. For example, the goal of Google's RE<C initiative is to make large-scale renewable power generation cheaper than coal-based production. As a result, researchers have started to study how to incorporate renewables into a data center's power delivery infrastructure [33]. As one example, Lee et al. [18] use request redirection to control the carbon footprint of data centers by redirecting load to servers powered by renewable energy sources. While not directly related to energy harvesting, Power Routing [27] proposes shuffled power delivery topologies that allow data centers to control how much power each rack receives. While the topologies are well-suited for delivering variable amounts of power to racks based on aggregate demand, they are also useful for flexible routing of a variable power supply. Prior research on workload-driven approaches to improve data center energy efficiency is orthogonal to our work. Examples include designing platforms that balance CPU and I/O capacity [5, 32], routing requests to locations with the cheapest energy [29], and dynamically activating and deactivating nodes as demand rises and falls [8, 17, 37]. PowerNap's node-level energy proportional technique has also been viewed as a workload-driven optimization [21]. We show that a similar technique is useful for controlling per-node power consumption in a power-driven system.

Power capping has also been studied previously in data centers to ensure collections of nodes do not exceed a worst-case power budget [12, 30]. However, the work assumes exceeding the power budget is a rare transient event that does not warrant application-specific modifications, and that traditional power management techniques, e.g., DVFS, are capable of enforcing the budget. These assumptions may not hold in many scenarios with intermittent power constraints, as with our renewable energy power source. Gandhi et al. cap CPU power by forcing CPU idle periods [13]. While similar, blinking focuses on capping per-node power where the CPU is only one component of the total power draw. Improving the energy-efficiency of storage is also a related research area. While Memcached does not offer persistent storage, our modifications for blinking adapt similar ideas from prior storage research, such as migrating popular objects to more active nodes [28, 41]. Additionally, power-aware caching algorithms focus on maximizing the idle time between disk accesses to reduce disk power consumption, while our work focus on controlling the power consumption of the cache itself [42].

Blinking introduces regulated churn into data center applications as nodes switch from the active to inactive state. Churn has been well-studied in decentralized, self-organizing distributed hash tables [34]. However, the type of churn experienced by DHTs is different than the churn caused by blinking, which motivates our different approach to the problem. In the former case, nodes arrive and depart unexpectedly based on autonomous user behavior and network conditions, while in the latter case, nodes switch between the active and inactive states in a regular and controllable fashion. Finally, RAMCloud [26] proposes using memory for low-latency persistent storage, and cites as motivation the increasingly large memcached clusters used in production data centers. The size of these clusters motivates our observation that fairness for the large number of equally unpopular objects, in addition to hit rate, is an important performance metric.

7. Applicability of Blinking

While we apply blinking to a memcached cluster powered by renewable energy in this paper, we believe blinking is applicable to other applications with intermittent power constraints. There are a range of scenarios beyond renewable energy where imposing intermittent constraints may be attractive. For example, data centers may wish to participate in automated demand-response programs with the electric grid. Automated demand-response, which is a cornerstone of a future smart electric grid, decreases power levels at participating consumers when the electric grid is under stress in exchange for lower power rates. Data centers are well-positioned to benefit from automated demand-response, since servers, as opposed to other types of electrical appliances, already include sophisticated power management mechanisms and are remotely programmable. Blink simply uses these pre-existing mechanisms to gracefully scale application performance as power varies. Additionally, data centers consume significant quantities of power, and demand-response programs typically target large power consumers first. Thus, addressing intermittent constraints in data centers may contribute to a more flexible and efficient electric grid.

In addition to automated demand-response programs, data center operators may wish to cap energy bills or power consumption at a fixed level for a variety of reasons, which also imposes intermittent power constraints. For instance, capping energy bills imposes variable power constraints when energy prices vary, as with wholesale energy prices which vary at intervals as low as every 5 minutes [29]. Thus, as market prices vary, the amount of power a fixed budget purchases will also vary. Capping power is also necessary during “brownout” scenarios, more common in developing countries, where the electric grid is not always able to fully meet demand. Further, Ranganathan et al. [30], as well as others [12], point out the benefits of oversubscribing a data center’s power delivery infrastructure, including the possibility of using dense clusters of lower-cost, but higher-power, components and then capping power to prevent damage.

Finally, we believe blinking is applicable to applications beyond memcached. As with memcached, applying blinking will likely require application-level modifications to handle regular and periodic disconnections. One particularly interesting case is leveraging blinking to run distributed storage systems under intermittent power constraints, such as in “brownout” scenarios. Persistent storage presents a different problem than memcached, since there is not an alternative always-on option to fallback on to retrieve data. The problems memcached presents stem from its process of mapping keys to nodes that is sensitive to power-level changes, which necessitate changing the active set of servers. Since distributed storage systems explicitly control the mapping of data to nodes they do not present the same problem. However, while we measure memcached’s performance primarily as a function of hit rate and fairness, a blinking storage system’s performance is primarily a measure of data availability, including both the latency and throughput to access data. As a result, a blinking storage system may need to judiciously replicate data to increase availability and ensure consistency across replicas, despite regular and frequent node transitions between the active and inactive states.

8. Conclusion

In this paper, we focus on managing server clusters running on intermittent power. We propose blinking as the primary abstraction for handling intermittent power constraints, and define multiple types of blinking policies. We then design an application-independent platform for developing and evaluating blinking applications, and use it to perform an in-depth study of the effects of blinking on one particular application and power source: memcached operating off renewable energy. We find that while an activation policy with key migration results in the best hit rates, it does not distribute the benefits of the cache equally among equally popular objects. As in-memory caches continue grow in size, they will store a greater fraction of equally popular objects for Zipf-like object popularity distributions. We then propose and evaluate an asymmetric load-proportional policy to increase fairness without significantly sacrificing the cache’s hit rate. We are currently studying how blinking applies to other types of data center applications, including distributed storage layers and data-intensive batch systems.

Acknowledgements. We would like to thank our shepherd Thomas Wenisch and the anonymous reviewers for their insightful comments that improved this paper. This work was supported in part by NSF grants CNS-0855128, CNS-0834243, CNS-0916577, and EEC-0313747.

References

[1] U.S. Environmental Protection Agency. Report To Congress On Server And Data Center Energy Efficiency. August 2nd 2007.

[2] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces To Reduce PC Energy Usage. In *Proceedings of the Conference on Networked Systems Design and Implementation*, pages 365-380, April 2009.

[3] F. Ahmad and T. Vijaykumar. Joint Optimization of Idle and Cooling Power in Data Centers while Maintaining Response Time. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243-256, March 2010.

[4] H. Amur, J. Cipar, V. Gupta, M. Kozuch, G. Ganger, and K. Schwan. Robust and Flexible Power-Proportional Storage. In *Proceedings of the Symposium on Cloud Computing*, June 2010.

[5] D. Anderson, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array Of Wimpy Nodes. In *Proceedings of the Symposium on Operating Systems Principles*, pages 1-14, October 2009.

[6] L. Barroso and U. Hölzle. The Case For Energy-proportional Computing. In *Computer*, 40(12):33-37, December 2007.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching And Zipf-like Distributions: Evidence And Implications. In *Proceedings of the International Conference on Computer Communications*, pages 126-134, June 1999.

[8] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy And Server Resources In Hosting Centres. In *Proceedings*

of the Symposium on Operating Systems Principles, pages 103-116, October 2001.

[9] I. Cohen, J. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data To System States: A Building Block For Automated Diagnosis And Control. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 231-234, December 2004.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the Symposium on Operating Systems Principles*, pages 205-220, October 2007.

[11] K. Fan, Z. Zheng, and P. Sinha. Steady And Fair Rate Allocation For Rechargeable Sensors In Perpetual Sensor Networks. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, pages 239-252, November 2008.

[12] X. Fan, W. Weber, and L. Barroso. Power Provisioning for a Warehouse-Sized Computer In *Proceedings of the International Symposium on Computer Architecture*, pages 13-23, June 2007.

[13] A. Gandhi, M. Harchol-Balter, R. Das, J. Kephart, and C. Lefurgy. Power Capping via Forced Idleness. In *Proceedings of the Workshop on Energy-efficient Design*, June 2009.

[14] P. Gupta. Google To Use Wind Energy To Power Data Centers. In *New York Times*, July 20th 2010.

[15] J. Hamilton. Overall Data Center Costs. In *Perspectives* at http://perspectives.mvdirona.com/. September 18, 2010.

[16] R. Kaushik and M. Bhandarkar. GreenHDFS: Towards an Energy-Conserving Storage-Efficient, Hybrid Hadoop Compute Cluster. In *Proceedings of the USENIX Annual Technical Conference*, June 2010.

[17] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, R. Katz. NapSAC: Design And Implementation Of A Power-Proportional Web Cluster. In *Proceedings of the Workshop on Green Networking*, August 2010.

[18] K. Lee, O. Bilgir, R. Bianchini, M. Martonosi and T. Nguyen Managing the Cost, Energy Consumption, and Carbon Footprint of Internet Services. In *Proceedings of the SIGMETRICS Conference*, June 2010.

[19] E. Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the Workshop on Power Aware Computing and Systems*, October 2010.

[20] J. Leverich and C. Kozyrakis. On The Energy (In)efficiency Of Hadoop Clusters. In *ACM SIGOPS Operating Systems Review*, 44(1):61-65, January 2010.

[21] D. Meisner, B. Gold, and T. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205-216, March 2009.

[22] R. Miller. Microsoft To Use Solar Panels In New Data Center. In *Data Center Knowledge*, September 24th 2008.

[23] J. Moore, J. Chase, and P. Ranganathan. Weatherman: Automated, Online, And Predictive Thermal Mapping And Management For Data Centers. In *Proceedings of the International Conference on Autonomic Computing*, pages 155-164, June 2006.

[24] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making Scheduling "Cool": Temperature-Aware Resource Assignment In Data Centers. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.

[25] F. Nah. A Study On Tolerable Waiting Time: How Long Are Web Users Willing To Wait? In *Behaviour and Information Technology*, 23(3), May 2004.

[26] J. Ousterhout, P. Agarwal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazieres, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. Rumble, E. Stratmann, and R. Stutsman. The Case For RAMClouds: Scalable High-performance Storage Entirely In DRAM. In *ACM SIGOPS Operating Systems Review*, 43(5):92-105, December 2009.

[27] S. Pelley, D. Meisner, P. Zandevakili, T. Wenisch, and J. Underwood. Power Routing: Dynamic Power Provisioning In The Data Center. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 231-242, March 2010.

[28] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-based Servers. In *Proceedings of the International Conference on Supercomputing*, pages 68-78, July 2004.

[29] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting The Electric Bill For Internet-scale Systems. In *Proceedings of the SIGCOMM Conference*, pages 123-134, August 2009.

[30] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proceedings of the International Symposium on Computer Architecture*, pages 66-77, June 2006.

[31] S. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the Symposium on Mass Storage Systems and Technologies*, pages 1-10, May 2010.

[32] S. Rivoire, M. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: A Balanced Energy-efficiency Benchmark. In *Proceedings of the SIGMOD Conference*, pages 365-376, June 2008.

[33] C. Stewart and K. Shen. Some Joules Are More Precious Than Others: Managing Renewable Energy In The Datacenter. In *Proceedings of the Workshop on Power-Aware Computer Systems*, October 2009.

[34] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service For Internet Applications. In *Proceedings of the SIGCOMM Conference*, pages 149-160, August 2001.

[35] B. Stone. Google's Next Frontier: Renewable Energy. In *New York Times*, November 28th 2007.

[36] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts In Bayou, A Weakly Connected Replicated Storage System. In *Proceedings of the Symposium on Operating Systems Principles*, pages 172-183, December 1995.

[37] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering Energy Proportionality With Non Energy-proportional Systems: Optimizing The Ensemble. In *Proceedings of the Workshop on Power-Aware Computer Systems*, San Diego, California, December 2008.

[38] A. Verma, P. De, V. Mann, T. Nayak, A. Purohit, G. Dasgupta, and R. Kothari. BrownMap: Enforcing Power Budget In Shared Data Centers. IBM, Technical Report RI09016, December 2009.

[39] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On The Scale And Performance Of Cooperative Web Proxy Caching. In *Proceedings of the Symposium on Operating Systems Principles*, pages 16-31, December 1999.

[40] H. Zeng, 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*, pages 123-132, October 2002.

[41] 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*, pages 177-190, October 2005.

[42] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton and J. Wilkes. Power-Aware Storage Cache Management. In *IEEE Transactions on Computers*, 54(5):587-602, May 2005.