BlinkFS: A Distributed File System for Intermittent Power

Navin Sharma[a,*], David Irwin[b], Prashant Shenoy[a]

[a] *CS Department, University of Massachusetts Amherst, MA 01003*
[b] *ECE Department, University of Massachusetts, Amherst, MA 01003*

Abstract

The ability to use intermittent power in data centers introduces numerous new opportunities for optimization, including i) using real-time markets to buy more power when it is cheap, ii) increasing the fraction of clean, but intermittent, renewable power, iii) capping power for long periods to extend UPS lifetime during blackouts, and iv) fully utilizing a data center's power delivery infrastructure. The capability to run off intermittent power also moves us closer to the vision of a net-zero data center that consumes no net energy from the electric grid and has a small carbon footprint. However, designing systems to operate under intermittent power is challenging, since applications often access persistent distributed state, where power fluctuations can impact data availability and I/O performance. To address the problem, we design and implement BlinkFS, which combines blinking with a power-balanced data layout and popularity-based replication/reclamation to optimize I/O throughput and latency as power varies. Our experiments show that BlinkFS outperforms approaches that co-opt existing energy-proportional distributed file systems (DFSs) for intermittent power, particularly at low steady power levels and high levels of intermittency. For example, we show that BlinkFS reduces completion time for MapReduce-style jobs by 42% at 50% full power compared to existing energy-proportional DFSs.

Keywords:
Distributed File System, Intermittent Power

1. Introduction

The growth of cloud-based services continues to fuel a rapid expansion in the size and number of data centers. The trend only exacerbates the environmental and cost concerns associated with data center power usage, which a recent report estimates at 1.7-2.2% of U.S. consumption [25]. Excessive energy consumption also has serious environmental ramifications, since 83% of U.S. electricity derives from burning "dirty" fossil fuels [26]. Energy costs are also on a long-term upward trend, due to a combination of government regulations to limit carbon emissions, a continuing rise in global

*Corresponding author
Email addresses: nksharma@cs.umass.edu (Navin Sharma), irwin@ecs.umass.edu (David Irwin), shenoy@cs.umass.edu (Prashant Shenoy)

energy demand, and dwindling supplies. Even with today's "cheap" power, a data center's energy-related costs represent a significant fraction ($\sim$31% [10]) of its total cost of ownership. Prior research on green data centers often assumes that grid energy is always available in unlimited quantities [7, 8], and focuses largely on optimizing applications to use less energy without impacting performance. By comparison, there has been little research on optimizing applications for intermittent power that fluctuates over time. Operating off intermittent power introduces new opportunities for optimizing a data center to be both cheaper and greener.

For instance, companies are highly interested in utilizing more intermittent renewable energy sources in data centers, both from a cost and environmental perspective [9, 29]. Both Microsoft (at the recent Rio+20 summit) [2] and HP [6] have announced bold initiatives to design net-zero data centers that consume no net energy from the electric grid and include substantial use of on-site renewable energy sources. Google has also pledged to reduce its carbon footprint to zero [3]. Additionally, startups, such as AISO.net [1], have formed around the idea of green hosting using only renewables. While today's energy prices do not strongly motivate renewable power from an economic perspective, companies are concerned about continued future price increases and the environmental (and public relations) consequences of fossil fuel-based energy. Since long-term battery-based storage is prohibitively expensive, increasing renewable penetration requires closely matching power consumption to generation. Data centers are particularly well-positioned to benefit from renewables, since unlike household and industrial loads, many workloads, including delay-tolerant batch jobs, may permit some performance degradation due to varying power. As the fraction of renewables increases in data centers, the data centers must be designed to gracefully handle significant and frequent variations in the available power, and even sustained low power scenarios, e.g., during extended cloudy periods.

The ability to use intermittent power introduces other opportunities, beyond increasing use of renewable energy, for optimizing a data center to be cheaper, greener, and more reliable. We argue that designing systems to exploit these optimizations will move us closer to the vision of a net-zero data center.

- **Market-based Electricity Pricing**. Electricity prices vary continuously based on supply and demand. Many utilities now offer customers access to market-based rates that vary every five minutes to an hour [4]. As a result, the power data centers are able to purchase for a fixed price varies considerably and frequently over time. For instance, in the New England hourly wholesale market in 2011, maintaining a fixed \$55/hour budget, rather than a fixed per-hour power consumption, purchases 16% more power for the same price (Figure 1). The example demonstrates that data centers that execute delay-tolerant workloads, such as data-intensive batch jobs, have an opportunity to reduce their electric bill by varying their power usage based on price.

- **Unexpected Blackouts or Brownouts**. Data centers often use UPSs for backup power during unexpected blackouts. An extended blackout may force a data center to limit power consumption at a low level to extend UPS lifetime. While low power levels impact performance, it may be critical for certain applications

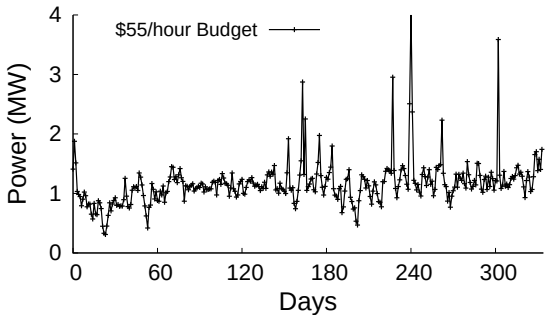


Figure 1: Electricity prices vary every five minutes to an hour in wholesale markets, resulting in the power available for a fixed monetary budget varying considerably over time.

to maintain some, even low, level of availability, e.g., disaster response applications. As we discuss, maintaining availability at low power levels is challenging if applications access distributed state. Further, in many developing countries, the electric grid is highly unstable with voltage rising and falling unexpectedly based on changing demands. These "brownouts" may also affect the power available to data centers over time.

- **100% Power Infrastructure Utilization**. Another compelling use of intermittent power is continuously operating a data center's power delivery infrastructure at 100%. Since data center capital costs are enormous, maximizing the power delivery infrastructure's utilization by operating as many servers as possible is important. However, data centers typically provision power for peak demands, resulting in low utilization [12, 24]. In this case, intermittent power is useful to continuously run a background workload on a set of servers—designed explicitly for intermittent power—that always consume the excess power PDUs are capable of delivering. Since the utilization (and power usage) of a data center's foreground workload may vary rapidly, the background servers must be capable of quickly varying power usage to not exceed the power delivery infrastructure's limits.

In this paper, we present the design of a distributed file system (DFS) for intermittent power. Since DFSs now serve as the foundation for a wide range of data-intensive applications run in today's data centers, taking advantage of any of the opportunities above necessitates a DFS optimized for intermittent power. As we discuss, designing such a DFS poses a significant research challenge, since periods of scarce power may render data inaccessible, while periods of plentiful power may require costly data layout adjustments to scale up I/O throughput. Our approach leverages a recently proposed blinking abstraction [31], which rapidly, e.g., once a minute, "blinks" servers between a high-power active state and a low-power inactive state, that has been shown to improve performance for stateless applications, e.g., memcached, running on intermittent power. Whether or not the technique applies to stateful applications is an open

problem, since both disk and memory state become unavailable whenever a blinking node is inactive. Researchers must address the problem to make blinking a practical option for a wide range of stateful applications.

Our work is the first to address the problem by designing a blink-aware stateful DFS optimized for intermittent power. Our system, called BlinkFS, represents a dramatic departure from all prior techniques used in energy-efficient storage systems, which generally rely on powering down large sets of servers for long periods of time to reduce overall energy consumption. Our goal is to design a DFS that performs well across a wide range of intermittent power scenarios – ranging from large and rapid power variations (+/- 90%) to sustained low power periods ($\sim$20%). In all cases, BlinkFS's goal is to utilize intermittent power as efficiently as possible, rather than guarantee a specific amount of work finishes within some time period. Since intermittent power may not be available to satisfy workloads with strict performance requirements or deadlines, it is not appropriate in these cases. Since we focus on using intermittent power, optimizations that use grid energy in combination with renewables to provide SLA guarantees are beyond the scope of this paper. As we describe in Section 2, the dynamics of intermittent power, where changes in available power may be significant, frequent, and unpredictable, warrant our new approach. Below, we highlight the advantages of our DFS designed for intermittent power, called BlinkFS, over co-opting prior energy-efficient storage techniques, e.g., [16, 23, 27, 30, 34, 35].

- **Low Amortized Overhead**. Blinking every node at regular intervals prevents costly and abrupt data migrations—common in many systems—whenever power decreases—to concentrate data on a small set of active nodes—or increases—to spread data out and increase I/O throughput. Instead, blinking ensures that each node is active, and its data is accessible, for some period of time each blink interval, at the expense of a modest overhead to transition each node between a high-power active and low-power inactive state.

- **Bounded Replica Inconsistency**. Deactivating nodes for long periods requires write off-loading to temporarily cache writes destined for inactive or overloaded nodes [15]. The technique requires excessive writes whenever nodes activate or deactivate to either apply or migrate off-loaded writes, respectively, while impacting reliability if off-loaded writes are lost due to a node failure. In contrast, BlinkFS ensures all replicas are consistent within one blink interval of any write, regardless of the power level.

- **No Capacity Limitations**. Since migrating to a new data layout is expensive, a goal of BlinkFS is to decouple I/O performance at any power level from the data layout: the same layout should perform well at all power levels. To ensure such a data layout, Rabbit [16] severely limits the capacity of nodes storing secondary, tertiary, etc. replicas. Blinking enables a power-independent layout without such limitations.

- **Minimally Disruptive**. DFSs support higher-level applications designed assuming fully active nodes with stable data layouts. Frequently changing the set of active nodes or the data layout disrupts scheduling and placement algorithms for

applications, such as MapReduce, that co-locate computation with DFS storage. BlinkFS is less disruptive, since it keeps every node active for the same duration every blink interval and does not change the data layout as power varies.

- **Always-accessible Data**. Prior systems render data completely inaccessible if there is not enough power to store all data on the set of active nodes. In contrast, BlinkFS ensures all data is accessible, with latency bounded by the blink interval, even at low power levels where the set of active nodes is unable to store the entire data set.

Since each node's data is inaccessible for some period each blink interval, BlinkFS's goal is to gain the advantages above without significantly degrading access latency. In achieving this goal, our work makes the following contributions.

Blinking-aware File System Design. We detail BlinkFS's design and its advantages over co-opting existing energy-proportional DFSs for intermittent power. The design leverages a few always-active proxies to absorb file system operations, e.g., reads and writes, while masking BlinkFS's complexity from applications.

Latency Reduction Techniques. We discuss techniques for mitigating blinking's latency penalty. Our approach combines staggered node active intervals with a power-balanced data layout to ensure replicas stored on different nodes are active for the maximum duration each blink interval. BlinkFS also uses popularity-based data replication and reclamation to further decrease latency for frequently-accessed data blocks.

Implementation and Evaluation. We implement BlinkFS on a small-scale prototype using 10 Mac minis connected to a programmable power supply that drives variable power traces. We then benchmark BlinkFS's performance and overheads at different (fixed and oscillating) power levels. We also compare BlinkFS with prior energy-efficient DFSs in two intermittent power scenarios—maintaining a fixed-budget despite variable prices and using intermittent wind/solar energy— using three different applications: a MapReduce-style batch system, the MemcacheDB key-value store, and file system traces from a search engine. As an example of our results, BlinkFS improves MapReduce job completion time by 42% at 50% power compared to an existing energy-proportional DFS. At 20% power, BlinkFS still finishes jobs, while existing approaches stall completely due to inaccessible data.

2. DFSs and Intermittent Power

Reducing data center power consumption is an active research area. Much prior work focuses on *energy-proportional* systems, where power usage scales linearly with workload demands [22]. The goal of energy-proportional systems is to not impact performance: if demands increase, these systems increase power consumption to maintain performance. Energy-proportional distributed applications typically vary power consumption by activating and deactivating nodes as workload demands change. An obvious approach for addressing intermittent power is to co-opt existing energy-proportional approaches, but vary the number of active nodes *in response to changes in available power rather than workload demands*. Unfortunately, as we discuss below, the

approach does not work well for DFSs using intermittent power, since power variations may be significant, frequent, and unpredictable, e.g., from changing prices, explicit demand response signal sent by the electric grid, or wind, solar, geothermal, etc. power. While energy-proportional systems optimize energy consumption to satisfy workload demands, designing for intermittent power requires systems to optimize performance as power varies. Below, we summarize how intermittent power affects energy-proportional DFSs, and then discuss two specific approaches.

2.1. Energy-Proportional DFSs

DFSs, such as the Google File System (GFS) [21] or the Hadoop Distributed File System (HDFS) [32], distribute file system data across multiple nodes. Designing energy-proportional DFSs is challenging, in part, since naïvely deactivating nodes to reduce energy usage has the potential to render data inaccessible [27]. One simple way to prevent data on inactive nodes from becoming inaccessible is by storing replicas on active nodes. Replication is already used to increase read throughput and reliability in DFSs, and is effective if the fraction of inactive nodes is small. For example, with HDFS's random placement policy for replicas, the probability that any block is inaccessible is $\frac{m!(n-k)!}{n!(m-k)!}$ for n nodes, m inactive nodes, and k replicas per block. Figure 2 plots the fraction of inaccessible data as a function of the fraction of inactive nodes, and shows that nearly all data is accessible for small numbers of inactive nodes. However, the fraction of inaccessible data rises dramatically once half the nodes are inactive, even for aggressive replication factors, such as k=7. Further, even a few inactive nodes, where the expected percentage of inaccessible data is small, may pose problems, e.g., by stalling batch jobs dependent on a small portion of the inaccessible data.

Thus, a popular approach for designing energy-efficient storage systems is to use *concentrated data layouts*, which deactivate nodes without causing inaccessible data. The layouts often store primary replicas on one subset of nodes, secondary replicas on another mutually-exclusive subset, tertiary replicas on another subset, etc., to safely deactivate non-primary nodes [16, 27]. Other systems concentrate data to optimize for skewed access patterns, by storing only popular data on a small subset of active nodes [14, 20, 23, 30, 35]. Unfortunately, concentrated layouts cause problems if available power varies independently of workload demands. Below, we highlight three problems with approaches that use concentrated data layout to deactivate nodes for long periods.

Inaccessible Data. If there is not enough power available to activate the nodes necessary to store all data, then some data will become inaccessible at low power levels. As we mention in Section 1, sustained low power periods are common in many intermittent power scenarios, such as during extended blackout or brownout periods, when using on-site solar generation on a cloudy day, or maintaining a fixed power budget as energy prices rise. Thus, gracefully degrading throughput and latency down to extremely low power levels is important. With concentrated data layouts, as data size increases, the number of nodes, and hence minimum power level, required to store all data and keep it accessible increases.

Write Off-loading Overhead. Energy-proportional systems leverage write off-loading to temporarily cache writes on currently active nodes, since clients cannot apply writes

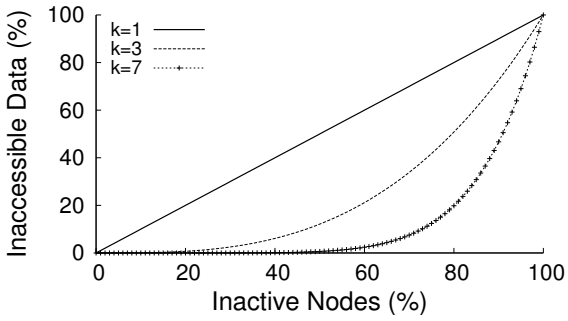


Figure 2: Inaccessible data rises with the fraction of inactive nodes using a random replica placement policy.

to inactive nodes, e.g., [15, 16, 34]. Write off-loading is also useful for deferring writes to overloaded nodes, which are common when only a small number of active nodes store all data. While a small number of active primary nodes decreases the minimum power level necessary to keep data accessible, it overloads primaries by requiring them to process all writes. The approach also imposes abrupt overheads when activating or deactivating nodes, either to apply off-loaded writes to newly active nodes or overloaded primary nodes, respectively. Further, intermittent sources, e.g., wind power, that exhibit abrupt power variations require near immediate node deactivations, precluding the completion of time-consuming operations. While using a large battery array as a buffer mitigates the impact of sudden variations, it is prohibitively expensive [18]. Further, deferring writes to replicas on inactive nodes degrades reliability in the event of node failure. Failure's consequences are worse during low power periods, by increasing the number of off-loaded writes on active nodes, and the time replicas on inactive nodes remain in an inconsistent state.

Disrupts Higher-level Application. A common paradigm for DFSs in data centers is to distribute file system data across compute nodes that host a variety of distributed applications. These applications, e.g., MapReduce, have their own, often highly optimized, algorithms to schedule and place computation to minimize data transfer overheads. Thus, activating and deactivating nodes or changing data layouts as power varies often requires significant modifications to higher-level applications.

Below, we outline two approaches to energy-proportional DFSs that use concentrated data layouts and vary power by activating and deactivating nodes. We highlight the additional problems these DFSs encounter if power variations are significant and frequent.

2.2. Migration-based Approach

We classify any approach that varies power consumption by migrating data to concentrate it on a set of active nodes, and then deactivating the remaining nodes, as a *migration-based approach*. With this approach, power variations trigger changes to number of nodes storing either the most popular data or primary, secondary, tertiary, etc. replicas. In either case, data layout changes require migrations to spread data out to provide higher I/O throughput (as nodes become active) or to concentrate data and

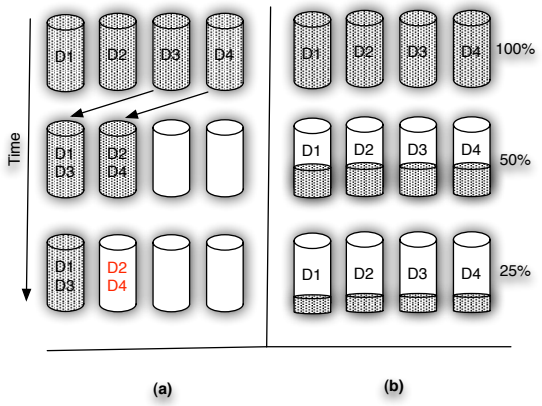


Figure 3: Simple example using a migration-based approach (a) and blinking (b) to deal with power variations.

keep it accessible (as nodes become inactive). Thus, mitigating migration overheads is a focus of prior work on energy-efficient storage [23, 30, 35].

To highlight the problems with this approach, consider the simple example in Figure 3(a), where there is enough power to operate four nodes storing primary replicas and the data fills two nodes' storage capacity. A sudden and unexpected drop in power by 2X, leaving only two active nodes, may not afford enough time for the necessary migrations, leaving some data inaccessible. Even with sufficient time for migration, an additional 2X power drop, leaving only one active node, forces at least 50% of the data to become inaccessible. Note that we focus on regulating power consumption within a single data center. Another way to handle power variations is to migrate applications and their data to remote data centers with ample or cheap power [17]. The technique is infeasible for large storage systems. Even assuming dedicated high-bandwidth network links, we view frequent transfers of large, e.g., multi-petabyte, storage volumes as impractical.

2.3. Equal-Work Approach

Amur et al. propose an energy-proportional DFS, called Rabbit, that eliminates migration-related thrashing using an *equal-work* data layout [16]. The layout uses progressively larger replica sets, e.g., more nodes store $(n+1)$-ary replicas than n-ary replicas. Specifically, the layout orders nodes $1 \ldots i$ and stores $b_i = \frac{B}{i}$ blocks on the ith node, where $i > p$ and p nodes store primary replicas (assuming a data size of B). The layout ensures that any $1 \ldots k$ active nodes (for $k < i$ total nodes) are capable of servicing $\frac{B}{k}$ blocks, since $\frac{B}{i} < \frac{B}{k}$. Since the approach is able to spread load equally across any subset of nodes in the ideal case of reading all data, it ensures energy-proportionality with no migrations.

Amur et al. provide details of the approach in prior work [16], including its performance for workloads that diverge from the ideal. Rabbit's primary constraint is its storage capacity limitations as $i \to \infty$, since $\frac{B}{i}$ defines the capacity for node i. Thus, for

N homogeneous nodes capable of each storing M blocks, the nodes' aggregate storage capacity is MN, while Rabbit's storage capacity is $pM + \sum_{i=p+1}^{N} \frac{pM}{i} = O(logN)$. For example, for N=500 nodes and M=2^{14}=16384 64MB blocks, the aggregate storage capacity across all nodes is MN=500 terabytes, while Rabbit's capacity is less than 15 terabytes, or 3% of total capacity, when p=2.

The relationships above show that the fraction of unused capacity increases linearly with N. Thus, the total storage capacity is capable of accommodating significantly more replicas than Rabbit uses as N increases. To reduce capacity limitations, Rabbit is able to individually apply the layout to multiple distinct data sets, by using a different $1 \ldots i$ node ordering for each data set. However, multiplexing the approach between data sets trades off desirable energy-efficient properties, e.g., few nodes storing primary replicas and ideal energy-proportionality. Thus, Rabbit's design presents issues for large clusters of nodes with similar storage capacities.

3. Applying Blinking to DFSs

The systems in the previous section use *activation* policies that vary power consumption only by varying the number of active nodes. As discussed by Sharma et al. [31], the blinking abstraction supports many other types of *blinking policies*. As we discuss in Section 4, BlinkFS uses an asynchronous staggered blinking policy. Below, we provide a brief, high-level summary of blinking. A detailed description of the abstraction and its implementation is available in Blink [31]. Blinking builds on PowerNap [28], which enables rapid server transitions between the active and inactive states.

The blinking abstraction permits an external controller to remotely set a blink interval t and an active interval t_{active} on each node, such that for every interval t the node is active for time t_{active} and inactive for time $t - t_{\text{active}}$. ACPI's S3 (Suspend-to-RAM) state is currently a good choice for the inactive state, since it combines the capability for fast millisecond-scale transitions with low power consumption (<5% peak power). In contrast, techniques that target individual components, such as DVFS in processors, are much less effective at satisfying steep drops in available power, since they are often unable to reduce consumption below 50% peak power [33]. To control inter-node blinking patterns, the abstraction also enables a controller to specify when a blink interval starts, as well as when within a blink interval the active interval starts.

3.1. Advantages for DFSs

To see the advantages of blinking for DFSs, recall the previous section's example (Figure 3(b)), where there is initially enough power to operate four nodes that each provide storage for a fraction of the data. If the available power decreases by 2X, with blinking we have the option of keeping all four nodes active for time $t_{\text{active}} = \frac{t}{2}$ every blink interval t. In this case, instead of migrating data and concentrating it on two active nodes, we are able to keep the same data layout as before without changing our aggregate I/O throughput over each blink interval, assuming each node has the same I/O throughput when active. Thus, at any fixed power level, blinking is able to provide the same I/O throughput, assuming negligible transition overheads, as an activation

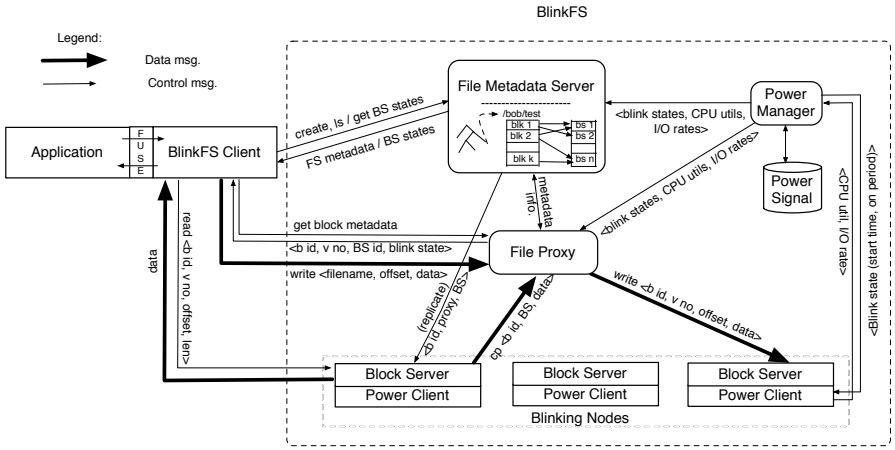


Figure 4: BlinkFS Architecture

approach. However, blinking has a distinct advantage over a migration-based approach if the available power changes, since it is possible to alter node active intervals nearly instantly to match the available power without the overhead of migration. Additionally, in contrast to Rabbit, the blinking approach does not require severe capacity limitations on nodes to maintain throughput. Finally, the approach is beneficial at low power levels if not enough nodes are active to store all data, since data is accessible for some period each blink interval.

3.2. Mitigating Reliability Concerns

We are not aware of any work that addresses the reliability impact of frequently transitioning a platform's *electric* components between ACPI's S0 and S3 state. In fact, related work on PowerNap [28] advocates even more rapid transitions ($\sim$100ms) than our prototype ($\sim$60s). Anecdotally, we have blinked our prototype tens of thousands of times without any failures.

Prior work on energy-efficient storage has likely not considered blinking due to the reliability concerns of frequently transitioning mechanical components, such as magnetic disks and exhaust fans, to and from their low-power standby state. For instance, prior work estimates a disk reaches its rated limit (estimated at $50,000$ start-stop cycles) in five years when transitioning only 28 times per day [35]. Since our prototype blinks nodes once a minute, it would reach the same limit in only 35 days. However, a node's mechanical components typically comprise only a small percentage of overall power consumption. For example, prior work estimates that consumer disks use roughly 10W when active and 5W when idle [11], while electric components may consume more than 150W. Thus, introducing a new low-power state that is similar to ACPI's S3 state, but decouples the power state of mechanical components would permit blinking a node's high-power electric components, without wearing out the mechanical components.

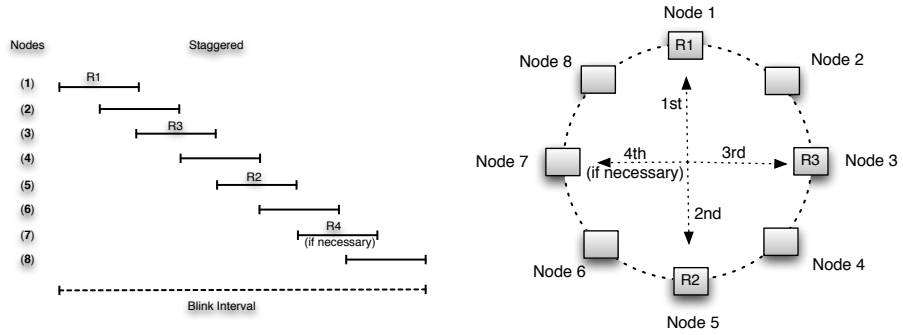
Of course, flash-based Solid State Drives (SSDs) are reducing the reliance on disks, and do not have the reliability concerns associated with rapid blinking. SSDs are increasingly popular, since they support higher I/O rates and are more energy-efficient than disks for a range of seek- and scan-intensive workloads [11, 33]. Since today's nodes do not decouple the power state of the electric and mechanical components, our prototype uses only SSDs.

4. BlinkFS Design

Figure 4 depicts BlinkFS's architecture, which resembles other recent DFSs, including GFS [21], HDFS [32], Rabbit [16], etc., that use a *master meta-data server* to coordinate access to each node's data via a *block server*. The master also maintains the file system namespace, tree-based directory structure, file name $\rightarrow$ blocks mapping, and block $\rightarrow$ node mapping, as well as enforces the access control and block placement and replication policy. As in prior systems, files consist of multiple fixed-size blocks replicated on zero or more nodes. To mitigate the impact of node failure, the master may recover from meta-data information stored at one or more proxies, described below, or maintain an up-to-date copy of its meta-data on backup nodes.

BlinkFS also includes a *power manager* that monitors available power, as well as any energy stored in batteries, using hardware sensors. The power manager implements a blinking policy that continuously alters per-node blinking patterns to match power consumption with available power. Specifically, the power manager communicates with a *power client* on each node to set the blink interval duration t, as well as its start time and active interval (t_{active}). The power client also acts as an interface for accessing other resource utilization statistics, including CPU utilization, I/O accesses, etc. The power manager informs the master and proxies, described below, of the current blinking policy, i.e., when and how long each node is active every blink interval, and per-node resource utilization statistics. To access the file system, higher-level applications interact with BlinkFS *clients* through well-known file system APIs. Our prototype uses the POSIX API's file system calls.

We do not assume that BlinkFS clients are always active, since clients may run on blinking nodes themselves, e.g., in clusters that co-locate computation and DFS storage. Thus, to enable clients to read or write blocks on inactive nodes, BlinkFS utilizes one or more always-active *proxies* to intercept read and write requests if a client and block server are not concurrently active, and issue them to the appropriate node when it next becomes active. Each proxy maintains a copy (loaded on startup by querying the master) of the meta-data information necessary to access a specific group of files (each file is handled by a single proxy), and ensures replica consistency every blink interval. The proxy propagates any file system operations that change meta-data information to the master before committing the changes. The power manager also maintains an up-to-date view of each node's power state, since each power client sends it a status message when transitioning to or from the inactive state. The messages also serve as heartbeats: if the power manager does not receive any status messages from a power client within some interval, e.g., 5 minutes, it checks if its block server has failed. A failure prompts the master to initiate recovery actions.



(a) Staggered Blinking (b) Power-balanced Data Layout

Figure 5: Combining staggered blinking (a) with a power-balanced data layout (b) maximizes block availability.

Similar to a set of always-active nodes storing primary replicas, proxies consume power that increases the minimum threshold required to operate the cluster. Importantly, however, proxies only serve as intermediaries, and do not store data. As a result, the data set size does not dictate the number of proxies. However, proxies do limit I/O throughput by redirecting communication between many clients and block servers through a few points. However, as we discuss below, mostly-active clients may often bypass the proxies when accessing data. Further, proxies are most useful at low power levels, where available power, rather than proxy performance, limits I/O throughput. Below we discuss the details of how BlinkFS's components facilitate reading and writing files, and then present techniques for mitigating BlinkFS's high latency penalty.

4.1. Reading and Writing Files

Proxies mask the complexity of interacting with blinking nodes from applications. The master and each client use a well-known hash function to map a file's absolute path to a specific proxy. To read or write a file, clients either issue requests to the proxy directly, or use an optimization, discussed below, that bypasses a file's proxy if the client is active at the same time as the file's block servers.

Handling Reads. The meta-data necessary to read a file includes its block IDs and their version numbers, as well as the (IP) address and blinking information of the block servers storing replicas of the file's blocks. The proxy holds read requests until a node storing the block becomes active, issues the request to the block server, receives the data, and then proxies it to the client. If multiple block servers storing the block's replicas are active, the proxy issues the request to the node with the longest remaining active interval, assuming the remaining active time exceeds a minimum threshold necessary to read and transmit the block. Using a proxy to transfer data is necessary when executing both clients and block servers on blinking nodes, since the client may not be active at the same time as the block server storing the requested data.

To optimize reads, mostly-active clients may directly request from the proxy the block information—IDs and version numbers—and blinking policy for each block server holding a replica, and then access block servers directly when they become

active. The optimization significantly reduces the proxy load for read-intensive workloads. To ensure the proxy applies all previous client writes to a block before any subsequent reads, the proxy includes a version number for each block, incremented on every update, in its response to the client. If the version number for the block stored at the block server is lower than the requested version number, then the proxy holds pending writes that it has not yet applied. In this case, the read stalls until the proxy applies the writes and the version numbers match. If the block server has an equivalent or higher version number, it sends back the data immediately. In either case, a block server ensures that a client never gets stale data, i.e., a block of version number lower than the requested version number. Like a Unix file system, application-level file locking might be necessary to ensure the atomicity of cross-block reads, e.g., as in the case of concurrent producers and consumers.

Handling Writes. The proxy performs a similar sequence for writes. All writes flow through a file's proxy, which serializes concurrent writes and ensures all block replicas are consistent each blink interval. The proxy may also return to the client before applying the write to every block replica, since subsequent reads either flow through the proxy or match version numbers at the block server, as described above. The proxy maintains an in-memory write-ahead log to track pending off-loaded writes from clients. Since the log is small, the proxy stores in-memory backups on one or more nodes (updated on each write before returning to the client), which it recovers from after a failure. When the client issues the write, the proxy first records the request in its log, increments the version number of the updated blocks, updates the master metadata and its own metadata, and returns to the client; next it then propagates the write to all replicas as the block servers become active; finally, when all replicas successfully apply the write, it removes the request from its log of pending writes.

Since all block servers are active for a period each blink interval, all replicas are consistent within one blink interval from when the write is issued, and the maximum time a write remains pending in the proxy's log is one blink interval. Of course, the proxy does have a fixed-size log for pending writes. After filling the log, further write requests stall until the proxy propagates at least one of its queued writes to each replica. Based on available power and the CPU and network utilization of block servers, the proxy limits write throughput to ensure all pending writes are applied within a blink interval, e.g., by stalling additional writes.

As with reads, mostly-active clients could also interact directly with block servers, as long as the client and block server are both active at the same time. In this case, the proxy maintains an intermediate version number for each block, not visible to read requests, to handle concurrent writes. An intermediate version number is always greater than or equal to the real version number for any block. An intermediate version number greater than the real version number indicates that one or more writes are pending for the block. Below we describe the complete flow of a direct or bypass write:

1. To write data directly to block servers a client first sends the filename, offset, and data size (in bytes) to the proxy.
2. The proxy increments the intermediate version number of the blocks to be updated, and sends back the meta-data to the client. The meta-data includes block IDs and their intermediate version numbers, as well as the address and blinking

information of block servers storing any replicas.

3. The client pushes the data to all replicas as the block servers become active. Each block server keeps the data from the client in an internal cache until it is directed by the proxy to apply the write or delete it from the cache. The client can push data in any arbitrary order.
4. Once the data is successfully pushed to all the block replicas, the client sends a request to the proxy. The request describes the update (block IDs, version numbers, offsets, data sizes) sent to the replicas. Note that the version number in a block update is same as the intermediate version number assigned by the proxy for the block.
5. The proxy updates the metadata, including the version number, of the blocks and the file metadata, updates the master metadata, and finishes the write operation by returning back to the client. The client then returns back to the application.
6. The proxy notifies block servers to apply writes to blocks.
7. If the client could not finish the write operation within a time threshold set by the proxy, based on the blink and I/O rates of the block servers, the proxy aborts the write and directs the block servers to remove any writes from their caches.

A write operation could span several blocks. To ensure consistency and allow concurrent writes the proxy imposes two restrictions. First, the proxy cannot finish a bypass write operation (steps 5 and 6) until all previous overlapping write operations are already finished or aborted. Second, the proxy stalls a via-proxy write until all previous overlapping bypass writes are either completed or aborted. Two write operations are overlapping if they have at least one block in common. Since versioning and metadata updates are serialized by the proxy, all replicas apply concurrent writes in the same serial order, although the data could arrive in any order. Finally, by applying the restrictions above the proxy also ensures the atomicity of cross-block write operations.

Since a write call in an application returns success only after the proxy updates the metadata information, as described above, a subsequent read call from the same application will always see the written data or a more recent version. Likewise, a read request never gets inconsistent data since it cannot see intermediate versions and all stable versions are already consistent.

Proxy Overhead and Scalability. As BlinkFS scales, it requires more proxies to increase its maximum workload, especially at moderate power levels. Note that the workload, and not data size, dictates the number of proxies. At high power, since clients can bypass proxies, proxies are not a bottleneck. At low power, the lack of node availability is the constraint, and not the proxies. For moderate power levels, our experiments (Section 6) show a proxy-to-block server ratio of 1:10 performs well, and also suggests that for some workloads a higher ratio may be acceptable. Thus, we expect the power overhead of proxies (and the minimum power necessary for operation) to be less than or equal to 10% in today's clusters.

4.2. Reducing the Latency Penalty

While migration-based approaches incur high overheads when power levels change, they ensure data is accessible, i.e., stored on an active node, as long as there is power to activate nodes necessary to store all data. In contrast, naïve blinking incurs a high

latency penalty, since each node is inactive for some time each blink interval. BlinkFS combines three techniques to reduce latency: an asynchronous staggered blinking policy, a power-balanced data layout, and popularity-aware replication and reclamation.

Asynchronous Staggered Blinking. Staggered blinking's goal is to minimize the overlap in node active intervals by staggering start times equally across each blink interval. Figure 5(a) depicts an example of staggered blinking. To perform well at both high and low power levels, the policy assigns equal-sized active intervals to all nodes, while varying the size of this interval to adjust to changes in available power. Thus, at any power level all nodes are active for the same amount of time. In contrast, while activating all nodes in tandem (akin to co-scheduling) may exhibit slightly lower latencies at high power levels (especially for read requests issued during an active interval that span multiple blocks stored on multiple nodes), it performs much worse at moderate-to-low power since it does not take advantage of replication to reduce latency.

Formally, for available power $p_{\text{available}}$, total power p_{total} necessary to activate all nodes, total power p_{inactive} required to keep all nodes in the inactive state, blink interval duration t, and N nodes, the duration of each node's active interval is $t_{\text{active}} = t * \frac{p_{\text{available}} - p_{\text{inactive}}}{p_{\text{total}} - p_{\text{inactive}}}$, and the blink start time (within each interval) for the ith node (where i=$0 \ldots N-1$) is b_{start}=$(t - t_{\text{active}}) * \frac{i}{N-1}$. Next we discuss how combining staggered blinking with a data layout that spreads replicas across nodes, maximizes the time at least one block replica is stored on an active node each blink interval. Importantly, the approach maximizes this time at *all* power levels.

Power-balanced Data Layout. A power-balanced data layout spreads replicas for each block across nodes, such that any set of nodes storing the block's replicas have minimum overlapping active intervals using the staggered blinking policy above. To place replicas in such a layout, we order all N nodes in a circular chain from $0 \ldots N-1$ and choose a random node to store the first replica of each block. We then place the second replica on the node opposite the first replica in the circle, the third replica on one of the nodes half-way between the first and second replicas, the fourth replica on the other node between the first and second replicas, etc. To delete replicas, we reverse the process. Figure 5(b) depicts an example for three replicas using staggered blinking from 5(a).

The layout policy above is optimal, i.e., maximizes the time each block is available on an active node each blink interval, if the number of replicas is a power of two. Maintaining an optimal placement for any number of replicas requires migrating all replicas each time we add or remove a single one. Our layout policy does not always maintain an optimal placement of replicas – placement is optimal only when the number of replicas is a power of two. However, the layout does perform well without requiring expensive migrations each time the number of replicas for a block changes. Note that for blocks with stable access patterns, where the number of replicas rarely changes, we evenly distribute replicas around the chain. Our layout is more resilient to failures than concentrated data layouts, since it spreads replicas evenly across nodes, rather than concentrating them on small subsets of nodes.

Popularity-aware Replication and Reclamation. Replication in DFSs is common to tolerate node failures and improve read throughput. Likewise, migrating popular replicas to active nodes is common in energy-efficient DFSs [23, 30, 34, 35]. BlinkFS also

uses replication to mitigate its latency penalty as power varies by employing popularity-aware replication and reclamation to reduce the latency for popular blocks. Note that our replication strategy is independent of the power level, since replicating at low power levels may be infeasible. In this case, a modest amount of battery-based storage may be necessary to spawn the appropriate replicas to satisfy performance demands [18]. By default, BlinkFS maintains three replicas per block, and uses any remaining capacity to potentially store additional latency-improving replicas.

As clients create new files or blocks become less popular, BlinkFS lazily reclaims replicas as needed. Using staggered blinking and a power-balanced data layout, the number of replicas r required to ensure a block is available 100% of each blink interval, based on the total nodes N, blink interval t, available power p, and active node power consumption p_{node}, is $r = \lceil \frac{N}{\lfloor \frac{(N-1)p}{N*p_{\text{node}}-p} \rfloor} \rceil$. At low enough power levels, i.e., where $1 > \frac{p}{p_{\text{node}}}$, there are periods within each blink interval where no nodes are active. In this case, the minimum possible fraction of each blink interval the block is unavailable is $1 - \frac{p}{p_{\text{node}}}$, assuming it is replicated across all nodes.

The master uses the relationships above to compute a block's access latency, given its replication factor and the current power level, assuming requests are uniformly distributed over each blink interval. There are many policies for spawning new replicas to satisfy application-specific latency requirements. In our prototype, the master tracks block popularity as an exponentially weighted moving average of a block's I/O (read) accesses, updated by the proxy every blink interval, and replicates blocks every period in proportion to their relative popularity, such that all replicas consumes a pre-set fraction of the unused capacity. For frequently updated blocks, BlinkFS caps the replication factor at three, since excessive replicas increase write overhead. To replicate a block, the master selects a source and a destination block server based on blinking patterns of the nodes, and directs the source node to send the data to the destination node either directly – if both nodes are active at the same time – or via the proxy, otherwise.

5. Implementation

We implement a BlinkFS prototype in C, including a master (∼3000LOC), proxy (∼1000LOC), client (∼1200LOC), power manager (∼100LOC), power client (∼50LOC), and block server (∼900LOC). The client uses the FUSE (Filesystem in Userspace) library in Linux to transfer file system-related system calls from kernel space to user space. Thus, BlinkFS clients expose the POSIX file system API to applications. BlinkFS also extends the API with a few blink-specific calls, as shown in Table 1. These system calls enable applications to inspect information about node blinking patterns to improve their data access patterns, job scheduling algorithms, etc., if necessary. All other BlinkFS components run in user space. While the master, proxy, and power manager are functionally separate and communicate via event-based APIs (using libevent), our prototype executes them on the same node. To experiment with a wide range of unmodified applications, we chose to implement our prototype in FUSE, rather than extend an existing file system implementation, such as HDFS.

Our prototype includes a full implementation of BlinkFS, including the staggered blinking policy, power-balanced data layout, and popularity-aware replication. Our

FUSE Functions
`getattr(path, struct stat *)`
`mkdir(path, mode)`
`rmdir(path)`
`rename(path, newpath)`
`chmod(path, mode)`
`chown(path, uid, gid)`
`truncate(path, offset)`
`open(path, struct fuse_file_info *)`
`read(path,buff,size,offset,fusefileinfo*)`
`write(path,buff,size,offset,fusefileinfo*)`
`release(path,fuse_file_info*)`
`create(path,mode,fuse_file_info*)`
`fgetattr(path, stat*,fuse_file_info*)`
BlinkFS-specific Functions
`getBlinkState(int nodeid)`
`getBlockInfo(int blockid)`
`getFileInfo(path)`
`getServerLoadStats(int nodeId)`

Table 1: POSIX-compliant API for BlinkFS

current implementation redirects all writes through the proxy, but permits clients to issue reads directly to block servers if both are concurrently active. Also, we maintain an in-memory log of writes in the proxy, but currently do not mirror it to a backup. Since our prototype has a modular implementation, we are able to insert other blinking policies and data layouts. We implement the migration-based approach and Rabbit from Section 2 to compare with BlinkFS. We also implement the *load-proportional blinking policy* proposed by Sharma et al. [31], which blinks nodes in proportion to the popularity of blocks they store. The policy is useful for access patterns with skewed popularity distributions, e.g., Zipf, and does not require migrations.

Hardware Prototype. We construct a small-scale hardware prototype (Figure 6) that uses intermittent power to experiment with BlinkFS in a realistic setting. Our prototype is similar to the Blink prototype used by Sharma et al. [31]. Unlike the Blink prototype, which uses OLPC nodes, our BlinkFS prototype is based on more powerful but energy-efficient Mac minis. We use a small cluster of ten Mac minis running Linux kernel 2.6.38 with 2.4Ghz Intel Core 2 Duo processors and 2GB of RAM connected together using an energy-efficient switch (Netgear GS116) that consumes 15W. Each Mac mini uses a flash-based SSD with a 40GB capacity. We also use a separate server to experiment with external always-on clients, not co-located with block servers. To minimize S3 transition times, we boot each Mac mini in text mode, and unload all unnecessary drivers. With the optimizations, the time to transition to and from ACPI's S3 state on the Mac mini is one second. Note that much faster sleep transition times, as low as a few milliseconds, are possible [31], and would further improve BlinkFS's performance. Unfortunately, manufacturers do not optimize sleep transition time in today's server-class nodes. Fast millisecond-scale transitions, as in PowerNap [28], significantly improve performance, especially access latency, by reducing the blink interval's length, but are not yet commercially available in today's servers.

We select a blink interval of one minute, resulting in a transition overhead of $\frac{1}{60}$=1.67% every blink interval. We measure the power of the Mac mini in S3 to be 1W and the power in S0 to be 25W. Thus, in S3, nodes operate at 4% peak power.

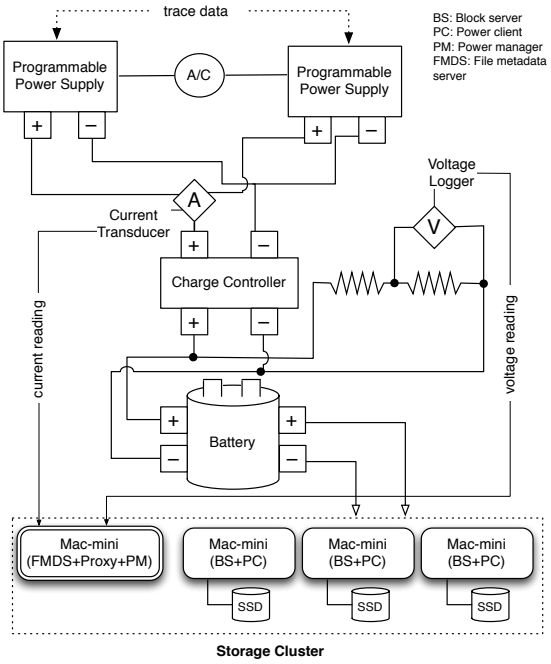


Figure 6: BlinkFS hardware prototype.

Since BlinkFS requires at least one node (to host the master, proxy, and power manager) and the switch to be active, its minimum power consumption is 40W, or 15% of its maximum power consumption. The remaining nine nodes each run a power client, block server, and BlinkFS client. We power the cluster from a battery that connects to four ExTech 382280 programmable power supplies, each capable of producing 80W, that replay the variable power traces below. To prevent the batteries from over and under-charging we connect the energy source to the battery using a TriStar T-60 charge controller. We also use two DC current transducers and a voltage logger from National Instruments to measure the current flowing in and out of the battery and the battery voltage, respectively. Our experiments use the battery as a short-term buffer of five minutes; optimizations that utilize substantial battery-based storage are outside the scope of this paper.

Power Signals. We program our power supplies to replay DC generation traces from our own small-scale solar panel and wind turbine deployment and 2) traces based on wholesale electricity prices. We also experiment with both multiple steady and oscillating power levels as a percentage, where 0% oscillation holds power steady throughout the experiment and $N\%$ oscillation varies power between $(45 + 0.45N)\%$ and $(45 - 0.45N)\%$ every five minutes.

For our renewable trace, we combine traces from our solar/wind deployment, and set a minimum power level equal to the power necessary to operate BlinkFS's switch and master node (40W). We compress our renewable power signal to execute three

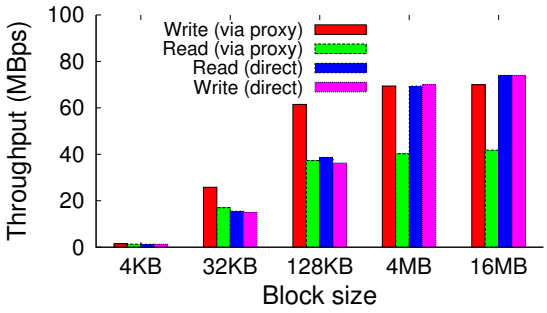
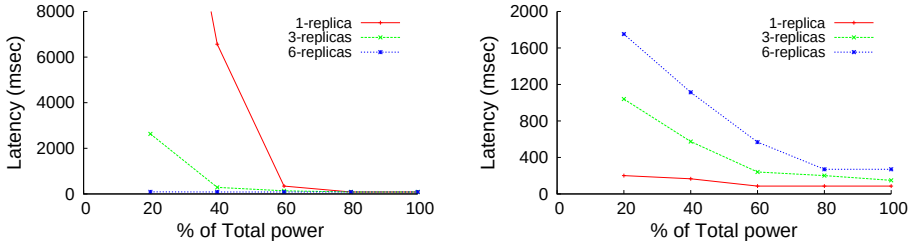


Figure 7: Maximum sequential read/write throughput for different block sizes with and without the proxy.



(a) Reads (b) Writes

Figure 8: Read and write latency in our BlinkFS prototype at different power levels and block replication factors.

days in three hours, and scale the average power to 50% of the cluster's maximum power. Note that the 24X compressed power signal is not unfair to the migration-based approach, since our data sets are relatively small (less than 20GB). We would expect large clusters to store more than 24X this much data, increasing the relative transfer time for migration. BlinkFS's performance is, by design, not dependent on the data set size. For our market-based electricity price trace, we use the New England ISO 5-minute spot price of energy for the 3-hour period from 7am to 10am on September 22, 2011, assuming a fixed monetary budget of 1¢/kWh; ISO's regulate wholesale electricity markets in the U.S. The average price in the trace is 4.5¢/kWh, the peak price is 5.2¢/kWh, and the minimum price is 3.5¢/kWh. We envision utilities increasing their use of market-based electricity pricing in the future, as electricity demands and prices increase. For example, Illinois already requires utilities to provide residential customers with access to hourly market-based electricity prices based directly on the wholesale price [4].

6. Evaluation

We first benchmark BlinkFS's overheads as a baseline for understanding its performance at different steady and oscillating power levels. We then evaluate BlinkFS

Latency (ms) ⇓		*Power (%)* *20*	*40*	*60*	*80*	*100*
Replication factor = 1						
Std Dev	W	1619	1069	1014	9	7
	R	15524	12701	1692	725	9
90^{th}per	W	60	60	61	62	65
	R	46058	33636	64	64	63
Replication factor = 3						
Std Dev	W	6017	4475	2089	22	22
	R	5476	322	309	9	7
90^{th}per	W	79	103	131	145	147
	R	13065	64	63	63	63
Replication factor = 6						
Std Dev	W	8883	5743	2467	703	372
	R	523	7	7	7	7
90^{th}per	W	127	183	257	258	263
	R	63	63	63	63	63

Table 2: Standard deviation and 90th percentile latency at different power levels and block replication factors.

for three different applications: a MapReduce-style application [13] (a data-intensive batch system), unmodified MemcacheDB [5] (a latency-sensitive key-value store), and file system traces from a search engine. Each application runs as a normal process with access to the BlinkFS mount point.

6.1. Benchmarks

To benchmark BlinkFS, we wrote a single-threaded application that issues blocking read/write requests to the client's interface, rather than through FUSE, to examine performance independent of FUSE overheads. One limitation of FUSE is that the maximum size of write and read requests are 4KB and 128KB, respectively, irrespective of BlinkFS's block size. The breakdown of the latency overhead at each component for a sample 128KB read is 2.5ms at the proxy, 0.57ms at the block server, 2.7ms at the client, and 0.33ms within FUSE for a total of 6.1ms. The results demonstrate that BlinkFS's overheads are modest. We also benchmark BlinkFS's maximum sequential read and write throughput (for a single replica) at full power for a range of block sizes. Figure 7 shows that, as expected, read and write throughput increase with increasing block size. However, once block size exceeds 4MB throughput improvements diminish, indicating that I/O transfer overheads begin to dominate processing overheads.

Read and write throughput via the proxy differ because clients off-load writes to proxies, which return before applying the writes to block servers. We also benchmark the throughput for reads sent directly to the proxy, which shows how much the proxy decreases maximum throughput ($\sim$40% for large block sizes). The overhead motivates our client optimization that issues reads directly to the block server, assuming both are concurrently active. The throughput of writes sent directly to block servers is similar to that of reads. We ran a similar experiment using 4MB blocks that scales the number of block servers, such that each block server continuously receives a stream of random I/O requests from multiple clients (using a block size of 4MB). As shown in Figure 9, write throughput reaches its maximum using three block servers, due to CPU overheads. The result shows that in the worst case a proxy-to-block server ratio larger than 1:3 does not

improve write throughput; for realistic workloads, each proxy is capable of supporting at least ten nodes, as our case studies demonstrate.

We also benchmark the read and write latency for different block replication factors for a range of power levels. Figure 8(a) shows that average read latency increases rapidly when using one replica if available power drops below 50%, increasing to more than 8 seconds. Additional replicas significantly reduce the latency using staggered blinking: in our prototype, all blocks are always available, i.e., stored on an active node, when using six replicas at 20% power. Write latency exhibits worse performance as we increase the number of replicas. In this benchmark, where clients issue writes as fast as possible, the proxy must apply writes to all replicas, since its log of pending writes becomes full (Figure 8(b)). Since the increase in the write latency is much less than the increase in read latency, the trade-off is acceptable for workloads that mostly read data. Table 2 shows the standard deviation and 90th percentile latency for read and write requests as the replication factor and power levels change.

Finally, we benchmark the overhead to migrate data as power oscillates, to show that significant data migration is not appropriate for intermittent power. For the benchmark, we implement a migration-based approach that equally distributes data across the active nodes. As power varies, the number of active nodes also varies, forcing migrations to the new set of active nodes. We oscillate available power every five minutes, as described in Section 5. We wrote a simple application that issues random (and blocking) read requests; note that the migration-based approach does not respond to requests while it is migrating data. Figure 10 shows that read throughput remains nearly constant for BlinkFS at different oscillation levels, whereas throughput decreases for the migration-based approach as oscillations increase. Further, the size of the data significantly impacts the migration-based approach. At high oscillation levels, migrations for a 20GB data set result in zero effective throughput. For smaller data sets, e.g., 10GB, the migration-based approach performs slightly better than BlinkFS at low oscillation levels, since the overhead to migrate the data is less than the overheads associated with BlinkFS.

Small power variations trigger large migrations in large clusters. If a 1000 node cluster with 500GB/node varies power by 2% (the average change in hourly spot prices), it must deactivate 20 nodes, causing a 10 terabyte migration. Even with a 10 gigabit link, the migration would take >2 hours, and prevents the cluster from performing useful work. We compare the migration-based approach with BlinkFS for these small power variations.

6.2. Case Studies

We experiment with a MapReduce-style application, MemcacheDB, and file system traces from a search engine using the traces discussed in Section 5 for three different approaches: BlinkFS, Rabbit, and Load-proportional. Since MapReduce executes batch jobs, it is well-suited for intermittent power if its jobs are tolerant to delays. We also experiment with interactive applications (MemcacheDB and file system traces) to demonstrate BlinkFS's flexibility. To fairly compare with Rabbit, we use an equal-work layout where the first two nodes store primary replicas, the next five nodes store secondary replicas, and the last two nodes store tertiary replicas.

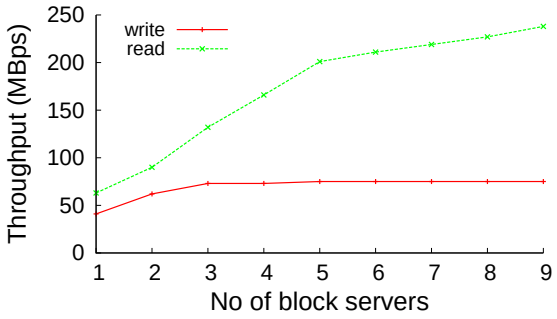


Figure 9: Maximum possible throughput in our BlinkFS prototype for different number of block servers.

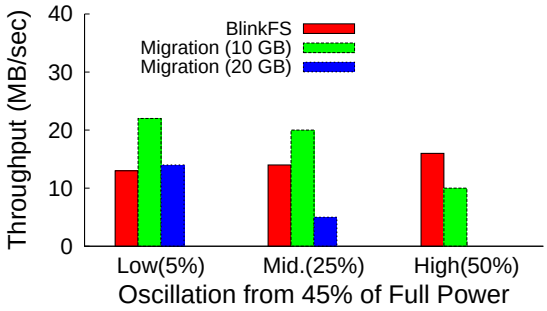
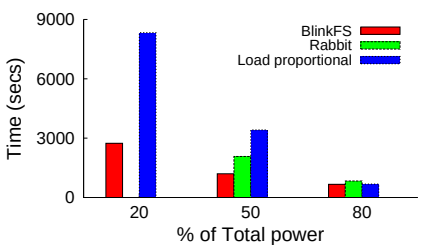


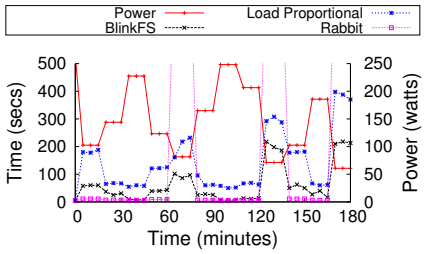
Figure 10: BlinkFS performs well as power oscillation increases.

Note that, while Rabbit performs well in some instances, it relies on severe capacity limitations, as described in Section 2, to avoid migrations. For BlinkFS's power-balanced data layout, we use $2/9$ths of the capacity to store one replica of each block, and the rest to store additional replicas. We set the default number of replicas to three, with a maximum replication factor of six for our popularity-aware replication policy. For load-proportional, we arrange blocks on nodes *a priori* based on popularity (from an initial run of the application) to eliminate data migrations, which provides an upper bound on load-proportional performance. Since MapReduce co-locates computation and data, the nodes execute both a client and a block server. For the other applications, we use an external, e.g., always-on, client. Finally, we use a 4MB block size.

MapReduce. For MapReduce, we create a data set based on the top 100 e-books over the last 30 days from Project Gutenberg (http://www.gutenberg.org/). We randomly merge these books to create 27 files between 100 and 200MB, and store them in our file system. We then write a small MapReduce scheduler in Python, based on BashReduce, that partitions the files into groups for each job, and sends each group to a MapReduce worker node, co-located on each block server. We execute the simple WordCount MapReduce application, which reads files on each node, counts the words in those files, and sends the results back to the scheduler. The scheduler then executes a reduce step to output a file containing all distinct words and their frequency.

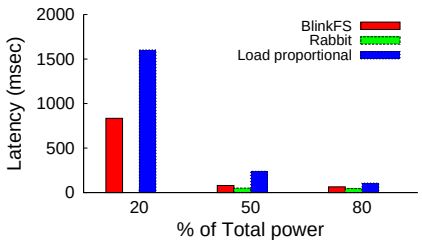


(a) Steady Power

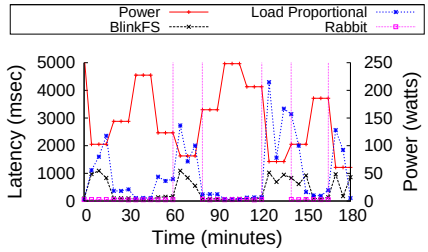


(b) Solar/Wind Power Signal

Figure 11: MapReduce completion time at steady power levels and using our combined wind/solar power trace.



(a) Steady Power



(b) Solar/Wind Power Signal

Figure 12: MemcacheDB average latency at steady power levels and using our combined wind/solar power signal.

We experiment with MapReduce using both constant and intermittent power. At constant power, Figure 11(a) shows that the completion time is nearly equal for all three policies at high power, but BlinkFS outperforms the others at both medium and low power. For instance, at 50% power BlinkFS improves completion time by 42% compared with Rabbit and 65% compared with load-proportional. Note that at low power (20%), MapReduce stalls indefinitely using Rabbit, since it requires at least two active nodes to ensure all data is accessible. Both Rabbit and Load-proportional also impact MapReduce computations by deactivating or reducing, respectively, the active time of cluster nodes as power decreases. BlinkFS does not affect the scheduling or placement policy as power varies.

For variable power, we execute a stream of smaller jobs, which process data sets that only consist of 27 e-books, to track the number of jobs we complete every five minutes. For this experiment, Figure 11(b) shows that BlinkFS outperforms Load-proportional at all power levels, since it does not skew the active periods of each node. While Rabbit performs better at high power levels, it stalls indefinitely whenever power is unable to keep all data accessible, i.e., two active nodes.

MemcacheDB Key-Value Store. MemcacheDB is a persistent version of memcached, a widely-used distributed key-value store for large clusters. MemcacheDB uses BerkeleyDB as its backend to store data. We installed MemcacheDB on our external node, and configured it to use BlinkFS to store its BerkeleyDB. To avoid any caching effects,

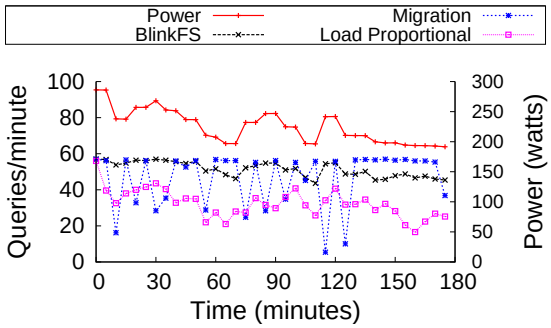


Figure 13: Search engine query rate with price signal from 5-minute spot price in New England market.

we configure MemcacheDB to use only 128 MB of RAM and set all caching-related configuration options to their minimum possible value. We then populated the DB with 10,000 100KB objects, and wrote a MemcacheDB workload generator to issue key requests at a steady rate according to a Zipf distribution.

Our results show that BlinkFS and Rabbit perform well at high and medium constant power levels (Figure 12(a)), while load-proportional performs slightly worse. Load-proportional does not benefit from replication, since replicas of popular blocks are inevitably stored on unpopular nodes. Thus, BlinkFS significantly outperforms load-proportional at low power levels. As with MapReduce, Rabbit has infinite latency at low power, since its data is inaccessible. Next, we run the same experiment using our wind/solar signal and observe the average request latency over each 5-minutes interval. As shown in Figure 12(b), BlinkFS performs better than load-proportional at nearly all power levels. The latency for BlinkFS scales up and down gracefully with the power signal. As in the MapReduce example, Rabbit performs better, except when available power is not sufficient to keep primaries active.

Search Engine. We emulate a search engine by replaying file system traces, and measuring the number of queries serviced each minute. The trace is available at http://www.storageperformance.org/specs/. To emulate the trace, we created a 30GB file divided into $491,520$ blocks of size 64KB and implemented an emulator in Python to issue I/O requests. We run the experiment with the power signals described in Section 5. As Figure 13 shows, BlinkFS outperforms load-proportional at all power levels. As expected, the migration-based approach performs slightly better than BlinkFS at steady power levels, but much worse for even slight ($\sim$10%) fluctuations in the available power. Since the available power is always more than the power required to run two nodes, Rabbit (not shown for clarity), outperforms (57 queries/minute) the others, since it stores primary replicas on these two nodes. Even for such a small dataset and power fluctuations, BlinkFS satisfies 12% and 55% more requests within a 3-hour period than the migration-based and load-proportional approaches.

7. Conclusion

We design BlinkFS to handle significant, frequent, and unpredictable changes in available power. Our design includes techniques to mitigate blinking's latency penalty. Recent work [19] also highlights the difficulty of designing latency-sensitive applications that are also energy-efficient. Intermittent power enables opportunities for optimizing data centers to be cheaper and greener, including incorporating more intermittent renewables, leveraging market-based electricity pricing, operating during extended blackouts, and fully utilizing a data center's power delivery infrastructure. Regulating power usage independent of workload demands enables data centers to leverage these optimizations. While today's energy prices do not strongly motivate intermittent power optimizations, such as increasing the fraction of renewable power in data centers, companies remain interested to be environmentally friendly and due to expectations of future increases in energy prices. We envision rising energy prices to incentivize data centers to design systems optimized for intermittent power in the future.

References

[1] AISO (Solar Powered Green Web Hosting). http://www.aiso.net/.

[2] United Nations Conference on Sustainable Development. http://www.uncsd2012.org/.

[3] Google's Green PPAs: What, How, and Why. http://www.google.com/green/pdfs/renewable-energy.pdf, April 2011.

[4] Dynamic Pricing and Smart Grid, 2011.

[5] MemcacheDB, 2011.

[6] M. Arlitt, C. Bash, S. Blagodurov, Y. Chen, T. Christian, D. Gmach, C. Hyser, N. Kumari, Z. Liu, M. Marwah, A. McReynolds, C. Patel, A. Shah, Z. Wang, and R. Zhou. Towards The Design and Operation of Net-zero Energy Data Centers. In *ITherm*, May 2012.

[7] I. Goiri, K. Le, M. Haque, R. Beauchea, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenSlot: Scheduling Energy Consumption in Green Datacenters. In *SC*, April 2011.

[8] I. Goiri, K. Le, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks. In *EuroSys*, April 2012.

[9] P. Gupta. Google to use Wind Energy to Power Data Centers. In *New York Times*, July 20th 2010.

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

[11] S. Rivoire and M. Shah and and P. Ranganathan. JouleSort: A Balanced Energy-Efficient Benchmark. In *SIGMOD*, June 2007.

[12] X. Fan and W. Weber and L. Barroso. Power Provisioning for a Warehouse-Sized Computer. In *ISCA*, June 2007.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, December 2004.

[14] L. Ganesh and H. Weatherspoon and M. Balakrishnan and K. Birman. Optimizing Power Consumption in Large Scale Storage Systems. In *HotOS*, May 2007.

[15] Dushyanth Narayanan and Austin Donnelly and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST*, February 2008.

[16] H. Amur and J. Cipar and V. Gupta and M. Kozuch and G. Ganger and K. Schwan. Robust and Flexible Power-Proportional Storage. In *SoCC*, June 2010.

[17] S. Akoush and R. Sohan and A. Rice and A. Moore and A. Hopper. Free Lunch: Exploiting Renewable Energy for Computing. In *HotOS*, May 2011.

[18] R. Urgaonkar and B. Urgaonkar and M. Neely and A. Sivasubramaniam. Optimal Power Cost Management Using Stored Energy in Data Centers. In *SIGMETRICS*, March 2011.

[19] D. Meisner and C. Sadler and L. Barroso and W. Weber and T. Wenisch. Power Management of Online Data-Intensive Services. In *ISCA*, June 2011.

[20] Dennis Colarelli and Dirk Grunwald. Massive Arrays of Idle Disks for Storage Archives. In *SC*, November 2002.

[21] S. Ghemawat and H. Gobioff and S. Leung. The Google File System. In *SOSP*, October 2003.

[22] L. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12), December 2007.

[23] R. Kaushik and M. Bhandarkar. GreenHDFS: Towards an Energy-Conserving Storage-Efficient, Hybrid Hadoop Compute Cluster. In *USENIX*, June 2010.

[24] V. Kontorinis, L. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. Tullsen, and T. Rosing. Managing Distributed UPS Energy for Effective Power Capping in Data Centers. In *ISCA*, June 2012.

[25] J. Koomey. Growth in Data Center Electricity Use 2005 to 2010. In *Analytics Press*, Oakland, California, August 2011.

[26] L. L. N. Laboratory. U.S. Energy Flowchart 2008, June 2011.

[27] J. Leverich and C. Kozyrakis. On the Energy (In)efficiency of Hadoop Clusters. In *HotPower*, October 2009.

[28] D. Meisner, B. Gold, and T. Wenisch. PowerNap: Eliminating Server Idle Power. In *ASPLOS*, March 2009.

[29] R. Miller. Microsoft to use Solar Panels in New Data Center. In *Data Center Knowledge*, September 24th 2008.

[30] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-based Servers. In *SC*, July 2004.

[31] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: Managing Server Clusters on Intermittent Power. In *ASPLOS*, March 2011.

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, May 2010.

[33] V. Vasudevan, D. Andersen, M. Kaminsky, L. Tan, J. Franklin, and I. Moraru. Energy-efficient Cluster Computing with FAWN: Workloads and Implications. In *e-Energy*, April 2010.

[34] A. Verma, R. Koller, L. Useche, and R. Rangaswami. SRCMap: Energy Proportional Storage Using Dynamic Consolidation. In *FAST*, February 2010.

[35] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping Disk Arrays Sleep Through the Winter. In *SOSP*, October 2005.