A Distributed File System for Intermittent Power

Navin Sharma, David Irwin, and Prashant Shenoy
University of Massachusetts Amherst
Amherst, Massachusetts 01003
{nksharma,irwin,shenoy}@cs.umass.edu

***Abstract*—Designing server clusters for intermittent power introduces new possibilities to make them cheaper, greener, and more reliable, including leveraging variable electricity prices to buy more power when it is cheap, increasing the use of clean renewable energy, and capping power at low levels to extend UPS lifetime during blackouts. However, regulating power usage to take advantage of these possibilities is challenging, since applications often access persistent distributed state, where power fluctuations impact I/O performance and data availability. To address the problem, we design and implement BlinkFS, which combines a blinking abstraction 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 existing approaches, particularly at low steady power levels and high levels of intermittency. As one example of our results, we show that BlinkFS reduces completion time for MapReduce-style jobs by 42% at 50% full power compared to an existing energy-proportional DFS.**

***Keywords*—Distributed file system, Intermittent power**

I. Introduction

The power consumption of information technology (IT)—estimated at 1.7-2.2% of U.S. consumption [1]—raises both environmental and cost concerns. For instance, excessive power consumption has serious environmental ramifications, since 83% of U.S. electricity derives from burning "dirty" fossil fuels. Prior research often assumes that grid energy is always available in unlimited quantities, and focuses largely on designing applications to best use the available energy without impacting performance [2], [3]. By comparison, there has been little research on designing server clusters to use intermittent power that fluctuates over time. Designing server clusters for intermittent power introduces new possibilities to make IT greener, cheaper, and more reliable, as outlined below.

Increasing Renewable Penetration. Environmental and cost concerns have led IT companies to experiment with using clean energy sources, such as wind [4] and solar [5], that are inherently intermittent. Since long-term battery-based storage is prohibitively expensive, increasing renewable penetration requires closely matching local power consumption to generation. IT is particularly well-positioned to leverage renewables, since unlike household and industrial loads, many workloads, including delay-tolerant batch jobs, may permit some amount of performance degradation due to periodic power shortages. We argue that reducing carbon emissions to zero (Google's explicit goal [6]) necessitates using intermittent power.

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. As a result, there are now opportunities to reduce the electricity costs of servers by using less power when prices are high, and more power when prices are low.

978-1-4799-0623-9/13/$31.00 © 2013 IEEE

Unexpected Blackouts or Brownouts. Servers often connect to Universal Power Supplies (UPSs) that supply backup power during unexpected blackouts. An extended blackout may force servers to limit their power consumption at a low level to extend UPS lifetime. As we discuss, maintaining the availability of distributed applications at low power levels is challenging if these applications access persistent distributed state.

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 distributed applications, 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 [7], 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. We show how the technique applies to applications that store persistent state, which remains an open problem since both disk and memory state become unavailable whenever a blinking node is inactive.

Our system, called BlinkFS, represents a departure from prior techniques that focus solely on energy-efficiency and are *workload-driven*—they assume a varying incoming workload and attempt to match the energy use of the system to varying demand, e.g., by powering down servers during periods of low demand. Instead, our techniques are *power-driven*—we assume a varying supply of power, due to renewables, and modulate the energy footprint of the system to match *available power*, while continuing to provide good performance. As we discuss, our DFS designed for intermittent power has numerous advantages over prior workload-driven energy-efficient storage techniques, e.g., [8], [9], [10], [11], [12], [13]. These advantages include low amortized overhead, bounded replica inconsistency, no capacity limitations, and always-accessible data. The primary research challenge with applying the blinking abstraction to a DFS is that each node's data is inaccessible for some period of time each blink interval. Thus, BlinkFS's goal is to achieve the advantages above without significantly degrading latency. In achieving this goal, we make the following contributions.

Blinking-aware File System Design. We detail BlinkFS's power-driven energy management and its advantages over co-opting existing energy-proportional DFSs for intermittent power. The design leverages a small number of always-active proxies to absorb file system operations, while masking BlinkFS's complexity from higher-level applications.

Latency Reduction Techniques. We design techniques to mitigate blinking's latency penalty. Our techniques combine

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. We also use popularity-based replication/reclamation to further decrease latency for frequently-accessed blocks of data.

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 storage systems in an intermittent power scenario using wind and solar energy for two real-world applications: a MapReduce-style batch system and the MemcacheDB key-value store. 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 due to inaccessible data.

II. DFSs and Intermittent Power

Much prior work on power consumption in server clusters focuses on *workload-driven energy-proportional* systems, where power usage increases or decreases with workload demands [14]. The goal of these systems is to not impact performance: if demands increase, these systems increase power consumption to maintain performance, and conversely, can reduce power usage during low demand periods without impacting performance. In contrast to these workload-driven techniques, the primary constraint in our system is a *varying supply of power*; our system must reduce power usage when the available power (rather than the workload) drops, e.g., when the output from a solar panel drops. Thus, while energy-proportional systems optimize energy consumption to satisfy workload demands, designing for intermittent power requires systems to optimize performance as power varies. A strawman approach to modulating the energy footprint of the system is to borrow ideas from workload-driven techniques and vary the number of active nodes *in response to changes in available power rather than workload demands*. As we discuss below, such an approach does not work well for DFSs using intermittent power due to the presence of disk state on nodes.

A. Energy-Proportional DFSs

DFSs, such as the Hadoop Distributed File System (HDFS) [15], are widely-used systems for storing data across multiple nodes in a server cluster. Designing energy-proportional DFSs is challenging, since naïvely deactivating nodes to reduce energy usage may render data inaccessible [10]. 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 1 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 serious problems, e.g., by stalling short batch jobs that are dependent on inaccessible data.

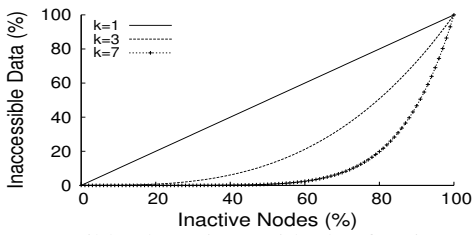


Fig. 1: Inaccessible data rises with the fraction of inactive nodes using a random replica placement policy.

Thus, one popular approach for designing energy-efficient DFSs 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 yet another subset, etc., to deactivate non-primary nodes [8], [10] without causing inaccessible data. Another approach is to concentrate data to optimize for skewed access patterns, by storing only popular data on a small subset of active nodes [9], [11], [13], [16], [17]. Concentrated layouts lead to two problems if available power varies, as we discuss below.

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 §1, sustained low power periods may occur during extended blackout or brownout scenarios. 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 DFSs leverage write off-loading to temporarily cache writes on currently active nodes, since clients cannot apply writes to inactive nodes, e.g., [8], [12], [18]. 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.

Below, we highlight two energy-proportional DFSs that use concentrated data layouts, and discuss additional problems they encounter if power variations are significant and frequent.

B. 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 the number of nodes storing either the most popular data or primary, secondary, tertiary, etc. replicas. In either case, data layout changes require expensive migrations to spread data out

to provide higher I/O throughput (as nodes become active) or to concentrate data and keep it accessible (as nodes become inactive). Thus, mitigating overheads is a focus of much prior work on energy-efficient storage [9], [11], [13].

C. Equal-Work Approach

Amur et al. propose an energy-proportional DFS, called Rabbit, that eliminates migration-related thrashing using an *equal-work* data layout [8]. 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 [8], 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 constraint 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. Thus, to achieve its benefits, Rabbit's design *wastes a significant amount of storage space*—97% in the example above—in homogeneous clusters where each node has a similar sized storage capacity. As we show, for intermittent power, BlinkFS achieves performance close to or better than Rabbit without wasting any storage capacity.

III. 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. The blinking abstraction supports many other types of *blinking policies*. As we discuss in §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 prior work [7]. Blinking builds on PowerNap [19], 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 a good choice for the inactive state, since it combines the capability for fast millisecond-scale transitions with low power consumption (<5% peak power). 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.

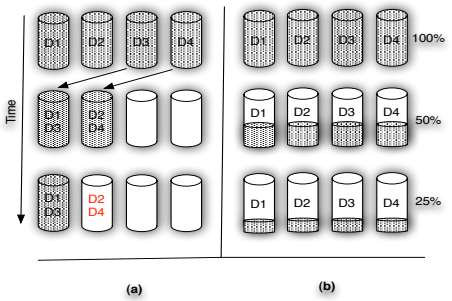


Fig. 2: Simple example using a migration-based approach (a) and blinking (b) to deal with power variations.

A. Advantages for DFSs

To see the advantages of blinking for DFSs over a migration-based approach, consider Figure 2, 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 (b) we have the option of keeping all four nodes active for time $t_{\text{active}} = \frac{t}{2}$ every blink interval t, rather than deactivating two nodes. In this case, instead of migrating data (a) 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 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. In essence, blinking migrates power to data, rather than migrating data to power, which is similar to designs that advocate migrating computation to data rather than migrating data to computation. 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.

B. Reliability Concerns

One potential drawback of frequently transitioning magnetic disks to and from their low-power standby state is its impact on the reliability of the stem. We are not aware of any work that directly addresses the reliability impact of frequently transitioning a server platform's *electric* components between ACPI's S0 and S3 state. In fact, recent work on PowerNap [19] advocates even more rapid transitions (∼100ms) than our prototype (∼60s) to eliminate server idle power. Anecdotally, we have blinked our prototype tens of thousands of times over the past year without any failures. Similarly commodity operating systems aggressively and frequently spin down laptop disks to save battery power without significantly impacting reliability.

Nevertheless sever disks, which are not designed for frequent transitions, may reach their rated limit (estimated at $50,000$ start/stop cycles) within 5 years when transitioning only 28 times per day [13]. More frequent transitions will proportionately reduce lifetimes. However flash-based Solid State

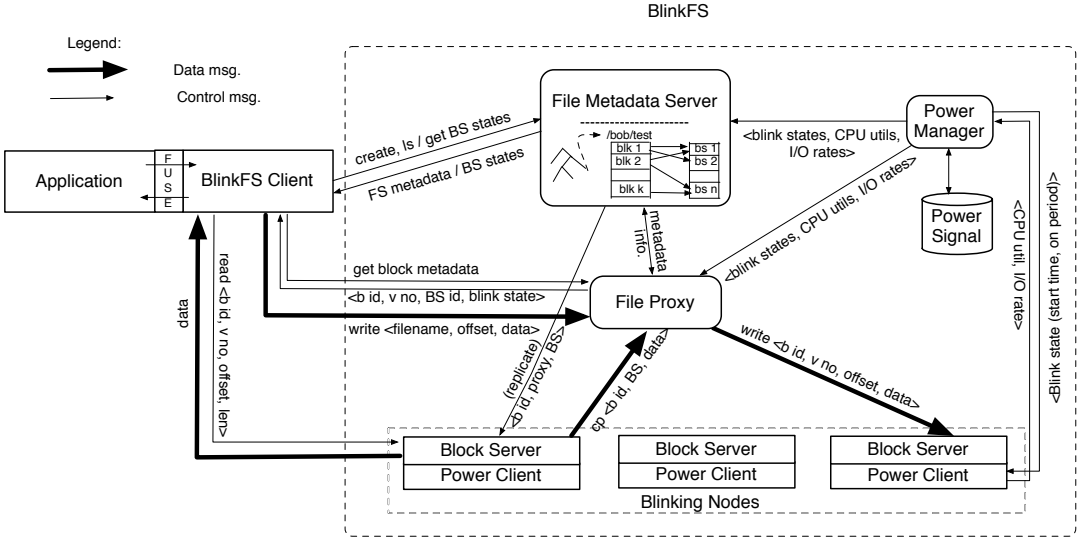


Fig. 3: Detailed architecture of BlinkFS, including its file metadata server, file proxy, client, block servers, and power manager.

Drives (SSDs) are reducing our reliance on magnetic disks, and are becoming 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 [20], [21]. SSDs do not have these reliability concerns since they do not contain mechanically moving parts. Further, SSD storage capacities continue to increase and the cost per bit of storage has been decreasing rapidly. Thus SSDs are likely to become common in small and medium-size servers in the coming years—just as they have become common in utralight laptops. Blinking and BlinkFS are well suited to exploit these trends.

Further, prior work estimates that consumer disks use roughly 10W when active and 5W when idle [20], while other non-mechanical components may consume more than 150W. Thus, introducing a new low-power state, similar to ACPI's S3 state, that decouples the power state of the mechanical components would still permit blinking only a node's high-power electric components, without power cycling other mechanical components such as fans and reducing their expected lifetime. Finally, servers highly optimized for data-intensive applications may operate many disks, e.g., 16 disks per server in TritonSort [22], making the power consumption of the disks a substantial fraction of the platform's power consumption. In this case, BlinkFS's approach may not be effective, since blinking only the electric components may have only a negligible impact on the cluster's overall power consumption. BlinkFS does not target these scenarios.

IV. BLINKFS DESIGN

Figure 3 depicts BlinkFS's architecture, which resembles other DFSs, including GFS [23], HDFS [15], Rabbit [8], etc., which uses 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/replication policy. Files consist of multiple fixed-size blocks replicated on zero or more nodes. The master may also recover from meta-data information stored proxies, as described below.

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 APIs. Our prototype uses the POSIX API's file system calls.

We do not assume that BlinkFS clients are always active, since the BlinkFS clients in the general mixed-use clusters, which co-locate computation and DFS storage, that BlinkFS targets may be running on blinking nodes themselves. 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 the appropriate recovery actions.

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, proxies only serve as intermediaries, and do not store data. As BlinkFS scales, it requires more proxies to increase its maximum throughput, especially at moderate power levels. Note that the target I/O throughput, and not data size, dictates the number of proxies. As we discuss below, to ensure proxies are not a bottleneck when power is plentiful, we employ an optimization that enables clients to bypass proxies at high power. At low power, the lack of node availability is the constraint, and not the proxies. At low power, proxies buffer client read and write operations until the necessary block servers are available, which in-turn blocks client applications, reducing the rate at which they issue new reads and writes. Thus, the use of a proxy is primarily an issue at moderate power levels ($\sim$50 full power), where the high-power optimization above is less effective but clients still issue requests at a high rate. For moderate power levels, our experiments (Section VI) show a proxy-to-block server ratio of 1:10 performs well, and suggest that for some workloads a proxy may be able to support more nodes. Below we discuss how BlinkFS facilitates consistent reading and writing of files, and then present techniques for mitigating BlinkFS's high latency penalty.

A. 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, while preserving consistency.

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.

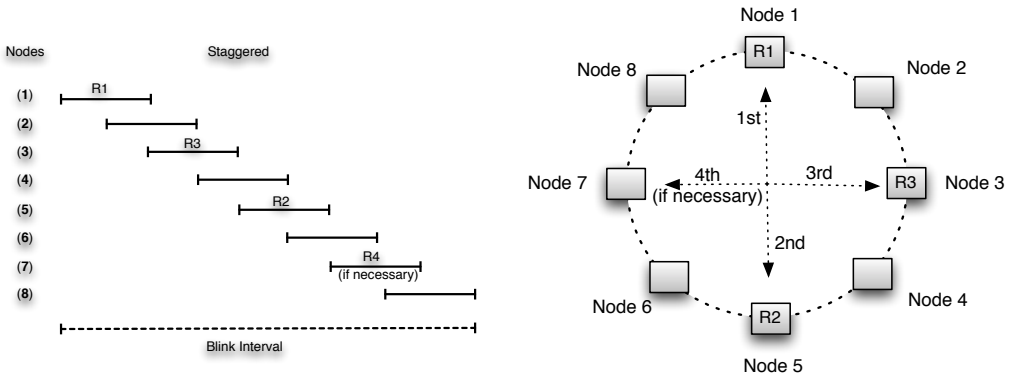
To optimize reads, mostly-active clients may directly request from the proxy the block information—IDs and version numbers—and blinking policy for each block server holding a replica, and then access block servers directly when they become active. The optimization significantly reduces the proxy load 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, it means 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 are necessary to ensure the atomicity of cross-block reads, e.g., as with 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 on 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 some time period each blink interval, all replicas are consistent within one blink interval from when the client issues the write, 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 can also write directly to block servers, while still preserving consistency as described below, 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; if it is strictly greater than the real version number, it indicates that one or more writes are pending for the block. We summarize the flow of a direct or bypass write below.

- To write data directly to block servers a client first sends the filename, offset, and data size to the proxy. The proxy increments the intermediate version number of the blocks to be updated, and then sends back the appropriate meta-data to the client.
- 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. The client can push data in any arbitrary order.
- Once the data is successfully pushed to all block replicas, the client sends the proxy a request that describes the update (block IDs, version numbers, offsets, data sizes) sent to the replicas. Note that the



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

Fig. 4: Combining staggered blinking (a) with a power-balanced data layout (b) maximizes block availability.

version number in an update is the same as the block's intermediate version number assigned by the proxy.

- The proxy updates the metadata, including the version number of the blocks and the file, updates the master metadata, and finishes the write by returning back to the client. Finally, the proxy notifies block servers to apply writes to blocks.
- If the client could not finish the write 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 clear their caches.

A write operation could span several blocks. To ensure consistency and allow concurrent multi-block writes, the proxy imposes two additional 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 consistent.

B. 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 enough power to activate the 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 I/O latency, as described below.

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 4(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 staggered blinking combined with a data layout that spreads replicas evenly across nodes, maximizes (at *any* power level) the time at least one block replica is available each blink interval.

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. However, maintaining an optimal placement for any number of replicas requires migrating all replicas each time we add or remove a single one. Thus, to avoid migrations, BlinkFS implements a policy that diverges slightly from optimal, but prevents migrations. In particular, to place replicas BlinkFS orders 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 halfway between the first and second replicas, the fourth replica on the other node between the first and second replicas, etc. Similarly, to delete replicas, we reverse the process.

Figure 4(b) depicts an example for the three replicas using staggered blinking from Figure 4(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. While BlinkFS's layout is not perfectly optimal, it is designed to be close to optimal 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 do evenly distribute replicas around the chain. In addition, our power-balanced layout is more resilient to correlated failures than concentrated data layouts, since it spreads replicas evenly across nodes, rather than concentrating them on small subsets.

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 [9], [11], [12], [13]. 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. By default, as with GFS and Hadoop, 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, assuming requests are uniformly distributed over a blink interval. There are many policies for spawning replicas to satisfy application 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 consume a target fraction of unused capacity. For frequently updated blocks, BlinkFS caps the replication factor at three, since excessive replicas increase write overhead.

V. Implementation

We implement a BlinkFS prototype in C, including a master, proxy, client, power manager, power client , and block server, totaling (∼6000LOC). The client uses FUSE (Filesystem in Userspace) to transfer system calls from kernel space to user space. Thus, BlinkFS clients expose the POSIX file system API to applications, including some blink-specific calls. To experiment with unmodified applications, we chose to implement our prototype in FUSE, rather than extend an existing file system implementation, such as HDFS.

Our prototype includes an implementation of BlinkFS, including asynchronous staggered blinking, a power-balanced data layout, and popularity-aware replication. Our current implementation redirects all writes through the proxy, but permits clients to issue reads directly to block servers if both are concurrently active. We also maintain an in-memory log of writes in the proxy, but currently do not mirror it to a backup. In addition, we implement both the migration-based approach and Rabbit from Section II for comparison in our evaluation. We also implement the *load-proportional blinking policy* from [7], which blinks nodes in proportion to block popularity. This blinking policy specifically targets skewed popularity distributions, e.g., Zipf, without requiring migrations.

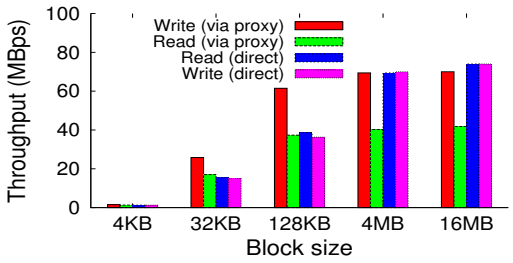


Fig. 5: Maximum sequential read/write throughput for different block sizes with and without the proxy.

Hardware Prototype. To conduct real experiments, we construct a small-scale hardware prototype that uses intermittent power to experiment with BlinkFS in a realistic setting. We chose to implement and experiment with a real-world, albeit small-scale, system because BlinkFS's design includes a number of components and complex interactions that are not conducive to accurate simulation. Our prototype is similar to the Blink prototype used by Sharma et al. [7], although instead of low-power OLPCs, we use 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. The time to transition to and from ACPI's S3 state on the Mac mini is one second. Much faster sleep transition times, as low as a few milliseconds, are possible [7]. Unfortunately, manufacturers do not optimize transition time in today's server-class nodes.

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. 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. Our experiments use the battery as only 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. We also experiment with multiple steady and oscillating power levels as a percentage, where

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

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 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 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.

VI. 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 for two different types of real-world applications: a MapReduce-style application [24] (a data-intensive batch system) and an unmodified MemcacheDB (a latency-sensitive key-value store). Each application runs as a normal process with access to the BlinkFS mount point via FUSE. Due to space constraints, we focus our evaluation on performance due to, potentially significant, variations in available power. Since BlinkFS builds on the design of existing DFSs, prior work has already evaluated other general aspects of its design. For example, prior DFSs have already shown that a centralized master meta-data server scales to large server clusters, e.g., thousands of nodes [15], [23].

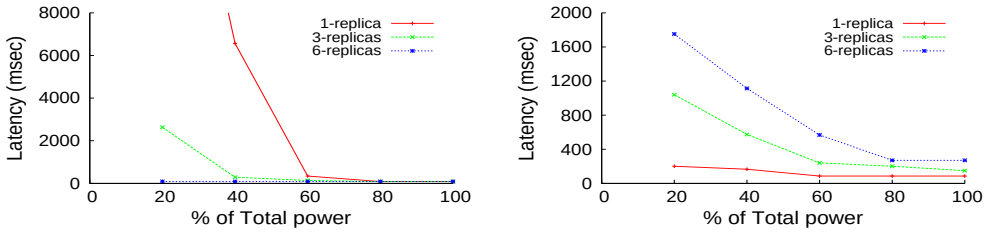
A. Benchmarks

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. 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 at full power for a range of block sizes. Figure 5 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). In this case, 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 6(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. As expected, 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 6(b)). Since the increase in the write latency is much less than the increase in read latency, the tradeoff is acceptable for I/O workloads that mostly read data. For our benchmarks, Table I shows the standard deviation and 90th percentile latency for reads and writes 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 §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 7 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%, it must deactivate 20 nodes, causing a 10 terabyte



(a) Reads (b) Writes

Fig. 6: Read and write latency in our BlinkFS prototype at different power levels and block replication factors.

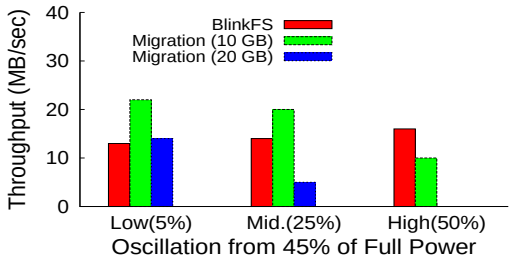


Fig. 7: BlinkFS performs well as power oscillation increases.

migration which takes hours with a gigabit link.

B. Case Studies

We experiment with a MapReduce-style application and MemcacheDB using the traces discussed in §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) 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.

Note that, while Rabbit performs well in some instances, it relies on severe capacity limitations, as described in §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 block size of 4 MB.

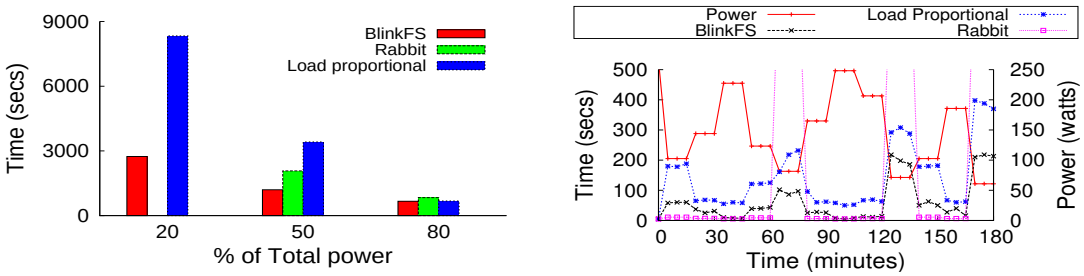
MapReduce. For MapReduce, we create a data set based on the top 100 e-books over the last 30 days from Project Gutenberg. We randomly merge these books to create 27 files between 100 and 200MB, and store them in our file system. We then write a small MapReduce scheduler in Python, based on BashReduce, 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.

We experiment with MapReduce using both constant and intermittent power. At constant power, Figure 8(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 8(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.

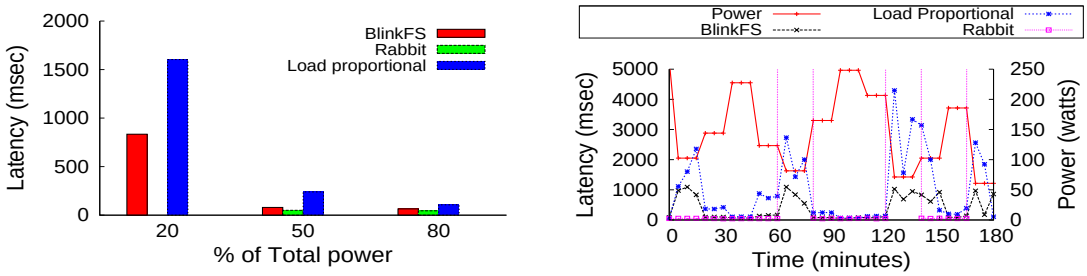
MemcacheDB Key-Value Store. MemcacheDB is a persistent version of the memcached key-value store that uses BerkeleyDB as its backend. We installed MemcacheDB on our external node, and configured it to use BlinkFS to store its BerkeleyDB. To avoid any caching effects, we configure MemcacheDB to use only 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 9(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 9(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 the available power is not sufficient to keep the primaries active.



(a) Steady Power (b) Solar/Wind Power Signal

Fig. 8: MapReduce completion time at steady power levels and using our combined wind/solar power trace.



(a) Steady Power (b) Solar/Wind Power Signal

Fig. 9: MemcacheDB average latency at steady power levels and using our combined wind/solar power signal.

VII. Conclusion

Intermittent power opens up new possibilities to make server clusters cheaper, greener, and more reliable, including leveraging market-based electricity pricing, incorporating clean renewable energy, or operating during extended blackouts. In this paper, we design BlinkFS, a DFS optimized for intermittent power, implement a real-world, albeit small-scale, prototype, and evaluate its performance compared to co-opting existing energy-proportional DFSs for two real applications.

Acknowledgements. This work was supported by NSF grants CNS-1253063, CNS-1117221, CNS-0916577, EEC-0313747, CNS-0855128, and CNS-0834243.

References

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

[2] 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.

[3] 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.

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

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

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

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

[8] 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.

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

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

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

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

[13] 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.

[14] L. Barroso and U. Hölzle, "The Case for Energy-Proportional Computing," *Computer*, vol. 40, no. 12, December 2007.

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

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

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

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

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

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

[21] 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.

[22] A. Rasmussen, G. Porter, M. Conley, H. Madhyasthay, R. Mysore, A. Pucher, and A. Vahdat, "Tritonsort: A balanced large-scale sorting system," in *NSDI*, April 2011.

[23] S. Ghemawat and H. Gobioff and S. Leung, "The Google File System," in *SOSP*, October 2003.

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