NVCache: Increasing the effectiveness of disk spin-down algorithms with caching

Timothy Bisson Scott A. Brandt Darrell D.E. Long *

University of California, Santa Cruz
Santa Cruz, CA

Abstract

Being one of the few mechanical components in a typical computer system, hard drives consume a significant amount of the overall power used by a computer. Spinning down a hard drive reduces its power consumption, but only works when no disk accesses occur, limiting overall effectiveness. We have designed and implemented a technique to extend disk spin-down times using a small non-volatile storage cache called NVCache, which contains a combination of caching techniques to service reads and writes while the hard disk is in low-power mode. We show that combining NVCache with an adaptive disk spin-down algorithm, a hard disk's power consumption can be reduced by up to 90%.

1 Introduction

Power consumption in computer systems is a problem in many industries, from mobile computing, where battery lifetime is a limiting factor, to large storage system installations, where monetary costs and machine room power limits become relevant issues. Previous research has shown that the disk subsystem can be responsible for 20-30% of the power consumed in a typical computer [13]. With current CPU architecture trends leaning toward multi-core versus single-core processors, the disk subsystem may consume a larger percentage of power in typical computers, as multiple cores are potentially more power efficient than higher-clock rates [15]. In larger storage systems, hard disks dominate power consumption. In an EMC Symmetrix 3000 storage system [1] and 2003 Dell PowerEdge 6650 system [10], hard disks represent 86% and 71% of the total power consumed, respectively.

There are two main approaches when trying to conserve power in hard disks. The first is to use disks which support multiple speeds by balancing energy consumption with performance [16, 5, 38, 39, 2]. Although multi-speed drives exist, such as the Hitachi Deskstar 7K400, they cannot yet service requests in LowRPM mode [19]. The other approach is to assume the use of a conventional disk, without multiple speeds, and alternate it between active and standby mode using a disk spin-down algorithm. In standby mode, the platter is not spinning and the heads are parked, making it consume less power than active mode. Adaptive spin-down algorithms [12, 18, 27, 14] are very efficient at minimizing a hard disk's power consumption. They are timeout driven, such that a timeout is set on each disk request and if no request occurs in the timeout duration, the disk is put in standby mode. The timeout value is recalculated after each disk request. An adaptive algorithm uses the performance of previous timeout values to compute the next timeout. They are very effective and approach the performance of an optimal offline algorithm, which knows the inter-arrival time of requests, *a priori*. Therefore, there is little benefit to be gained by introducing more complex and expensive adaptive algorithms with the hopes of gaining a few percent in energy conservation.

A largely untapped opportunity for improving the effectiveness of disk spin-down algorithms lies in lengthening disk idle periods. Out of the box operating systems do this to some extent via the page cache and/or buffer cache, which exists in some form in most operating systems. The goal of the page cache is to increase performance. It hides disk latency by reducing the number of actual disk requests. Writes are batched up in memory, in the hopes of coalescing multiple smaller operations into a small number of larger requests, and to avoid issuing multiple write requests to the same location in a short time window. Additionally, the page cache typically keeps recently used pages in-memory to reduce the number of read operations that must be serviced from disk. Thus, the page cache artificially affects idle periods as disk requests are often asynchronous with respect to their corresponding file system calls.

While the goal of the page-cache is performance, ours is power reduction. Dirty-page write-back timeouts in

*Supported in part by National Science Foundation Grant CCR-0310888

the tens of seconds are insufficient for hard disk power conservation—several minute timeouts are necessary. It is possible to set the dirty-page write-back timeout to several minutes. This will increase idle periods typically interrupted by write operations. However, because main-memory is volatile, caching several minutes worth of writes will be lost if a sudden power failure occurs before the cache contents are written back to disk.

To increase disk idle periods without reducing reliability, we have developed an integrated solution, NVCache, which uses a small low-power non-volatile storage device to absorb disk accesses while it is spun-down. To maximize the spin-down duration of a disk, NVCache is composed of two sub-caches: a write-cache and prefetch read-cache. The write-cache buffers write traffic while the disk is spun-down and lazily replays them to the disk after it becomes active again. Since the device is non-volatile, power outages during caching do not result in data loss. The prefetch read-cache contains copies of on-disk data with the goal of satisfying disk read requests while it is spun-down. We investigate two prefetch read-cache policies: LRU and LFU.

2 NVCache

We propose to increase the performance of any disk spin-down algorithm by servicing I/Os from a small low-power non-volatile storage device, NVCache, while the disk is in standby mode. By servicing I/O from the NVCache while the disk is in standby mode, a disk can remain spun-down longer, reducing its power consumption. Incorporating NVCache with a spin-down algorithm also decreases the effect a spin-down algorithm has on disk reliability and aggregate spin-up latency, by reducing the total number of cycle start-stop operations.

NVCache sits beneath the file system level at the block level. As a result, it is file system independent and compatible with any file system. NVCache also works on a per disk granularity, transparently supporting a disk with multiple partitions; NVCache has no notion of partition tables, so to the NVCache, redirecting disk I/O requests from different partitions is the same as redirecting requests from the same partition—they are just requests for the same disk with varying LBAs.

By using an NVCache device that is non-volatile, write requests redirected to the NVCache survive power failures. During reboot after an unexpected power failure, redirected writes in the NVCache can be written back to their intended disk locations, but before a file system verification operation, such as fsck. Note that if the system crashes, the data in main memory is still lost, which makes longer main-memory dirty-page write-back timeouts undesirable. NVCache does not try to improve reliability of a system that crashes, merely maintain the same reliability level had NVCache not been present.

NVCache has knowledge of the disk's power state and uses it to decide when to redirect requests between the disk and NVCache. While the disk is spun-down, write operations are redirected to the NVCache. NVCache also attempts to redirect read requests for a spun-down disk. If the NVCache has the requested data, it is returned and the disk remains spun-down. If not, the disk is spun-up and must service the request. In order to increase the likelihood that read operations are satisfied by the NVCache when the disk is spun-down, NVCache caches popular on-disk data while it is active.

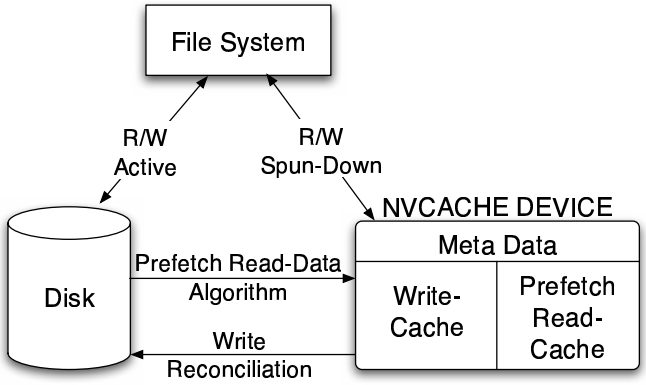


Figure 1. NVCache Organization

The NVCache organization is shown in Figure 1. There are two caches: a write-cache holding redirected writes while the disk is spun-down, and a prefetch read-cache. The prefetch-read cache is populated by the Prefetch Read-Data algorithm when disk is active. To determine what to prefetch, it records all disk read requests. Write reconciliation occurs to keep a disk consistent after a spin-down period. The metadata contains state statistics for the NVCache, such as disk power state, size, and spin-down iteration.

2.1 Read/Write Caching

The goal of NVCache is to extend spin-down periods by servicing as many disk I/O operations as possible while the disk is spun-down. To do this, we use write and read caching. The write-cache is straight forward—while the disk is spun-down writes are redirected to the write-cache. To the file system, redirected writes are committed operations. However, ensuring disk data consistency is more complicated. Our approach is to flush the write-cache contents back to disk when it is spun-up. To reduce the overhead of copying data from the NVCache to disk, we flush the smallest amount of data from the write-cache to disk

which doesn't cause write-cache capacity fills to initiate future spin-ups. We track the total amount of redirected write data for each spin-down period and compute an average from the most recent *N* spin-down periods to compute the amount of data to flush each spin-down period. To prevent the empty write-cache space from perpetually getting smaller, the amount of data flushed is artificially increased by a small percentage (25%) when a redirected write ends a spin-down period with a capacity fill.

The remaining write-cache data is lazily flushed to disk by letting the file system initiate consistency. We keep an in-memory list of data in the write-cache. While the disk is active, disk requests are first checked if they overlap with data from the NVCache. If the request is a write and overlaps with write-cache data, the associated in-memory list element is removed, and the request continues on to disk. If the request is a read and overlaps with all write-cache data, the request returns with write-cache data. However, if the request isn't completely covered by write-cache data, the overlapping region is still read from the write-cache, but the non-overlapping regions are read from disk. The two data sources are merged in memory and returned to the request. In both read cases, since write-cache data is already in memory, it is written back to its original location on disk, then removed from the in-memory list.

The read-cache is populated with copies of data from the disk that is likely to be read while the disk is spun-down. The prefetching algorithm that populates the NVCache only considers read operations as candidates for the read-cache. Since NVCache resides at the block level, we only utilize temporal information about the disk request pattern to drive the prefetching algorithm. In this work, we investigate the use of two traditional temporal caching algorithms: LRU/MRU and LFU/MFU, as well as a combination of the two, to prefetch data into the read-cache.

Justifying MRU/LRU is simple—data that is recently read is likely to be read in the near future. However, since operating system buffer caches typically use some variant of LRU, it is likely that prefetching the most recently used data will result in double caching, with the buffer cache satisfying requests of the doubled cached data. Yet, the MRU/LRU prefetching algorithm has several distinctions from a normal LRU buffer cache algorithm. First, only read requested data is prefetched to the NVCache while a buffer cache typically does not discriminate between reads and writes. Second, the prefetching algorithm for the NVCache is driven by disk accesses, not file system accesses, which hopefully yields different working sets. We also look at an MFU/LFU prefetching algorithm to avoid potential double caching effects. Finally, we investigate combining the two caching policies to exploit the benefits of each policy.

We choose to put the read-cache on the NVCache and not in main-memory for several reasons. First, populating the buffer cache with data from disk will cause evictions for data recently used by the file system, which are likely to be accessed again. Unfortunately, such requests must now go to disk, thus negating the benefit of prefetching into the buffer cache. Second, the file system should drive what is in the buffer cache, not the disk request pattern, as the disk request pattern is an effect of the file system request pattern, and buffer cache size and policy.

	Notebook drive	Compact Flash
Read/Write	2W	.17W
Idle	1.8W	2.5mW
Standby	.2W	2.5mW
Capacity	60-100GB	256MB-8GB

Table 1. Power requirements and capacity of a Hitachi Travelstar E7K100 notebook drive and a Sandisk Ultra II CompactFlash Memory Card

2.2 NVCache Device

Our focus is not on the type of NVCache device used but rather about the power saving potential of a generic NVCache, yet the design, implementation, and performance rely on specific properties of the NVCache device. Therefore, we will briefly discuss the type of device we designed NVCache for: flash memory. We chose flash memory for several reasons. It is readily available, has a large capacity, is non-volatile, and has a low-power requirement relative to notebook disk drives. Table 1 shows the requirements of a modern notebook disk drive and a compact flash memory card. Compact flash requires significantly less power than a notebook drive and has roughly 10% the capacity of a notebook drive.

Flash has several unique properties. It has a page allocation unit and a block erase unit, where a block size is several times larger than a page. Data cannot simply be overwritten in flash—it must first be erased, then written to. Therefore, Flash Translation Layers (FTL) are used to map LBAs to pages. Additionally, blocks have a limited number of erasures, 2 million is a current threshold [9], which means that wear-leveling techniques are necessary to distribute erase operations across the device. It is important to note that a failed block is still readable, just not writable. The FTL should remove the block from potential mappings, similar to a file system finding bad disk sectors and not using them. Some flash device manufacturers, such as Sandisk, provide both wear-leveling and an internal FTL, embedding the functionality in the controller of a compact flash device.

Although compact flash can be treated as a pure block device, it is still useful to keep its physical properties in mind when designing the NVCache data layout format, especially when wear-leveling implementations vary from manufacturer to manufacturer. For example, Sandisk has a patented regional wear-leveling algorithm, where a compact flash device is broken up into multiple logical regions [34]. With wear-leveling in mind, the obvious choice for the read and write cache layout is in a log format. By writing data out in a log format, erase cycles will be distributed across the entire device.

3 Implementation

NVCache is implemented in the Linux Kernel and as a simulator. Both implementations use the dynamic spin-down algorithm developed by Helmbold *et al.* [18]. Their spin-down algorithm is based on a machine learning class of algorithms known as Multiple Experts [6]. In the dynamic spin-down algorithm, each expert has a fixed time-out value and weight associated with it. The time-out value used is the weighted average of each expert's weight and time-out value. It is computed at the end of each idle period. After calculating the next time-out value, each expert's weight is decreased proportional to the performance of its timeout value.

3.1 Kernel

The kernel implementation with a focus on the write-cache is discussed in this section. We only describe the write-cache implementation as the read-cache is part of future work. As mentioned in the previous section, a log is a suitable format for a flash NVCache device, which we use. Figure 2 shows the data layout. A meta-block describes one or more data-blocks to its right. A meta-block contains the location on disk of the data-block(s), including LBA, offset, length, checksum, and status field. A meta-block is currently 512 bytes. Each redirected write contains one meta-block followed by one or more data-blocks(s).

Intercepting and redirecting requests occurs at the block level. When a write request comes in for a disk that is spun-down, it is intercepted and the I/O structure is modified. The I/O destination is redirected to the write-cache and a meta-block sector is prepeneded to the I/O structure's data field. The location of the the redirected I/O is the next set of available contiguous sectors large enough to hold the meta-block and data field. To make write-cache space allocation efficient, we don't write to the write-cache in page sized chunks (Linux's base allocation unit for I/O), but rather, only write actual data from within the page to the write-cache.

In order to make searching through the write-cache more efficient, we keep information about the data stored on the write-cache in memory. Currently, we use a list, sorted by time, but will switch to a hash table to yield faster search performance. The in-memory list is searched when a read operation occurs while the disk is spun-down, but also while the disk is active (for lazy consistency). Each element in the list contains: a meta-block number, disk sector, length, and status. The meta-block number is the logical block address of the meta-block that describes one or more data-block(s). When appending to the write-cache, the list is scanned to see if it overlaps with any previous writes. If so, the overlapped list entry is removed. Finally, the I/O entry is appended to the in-memory list.

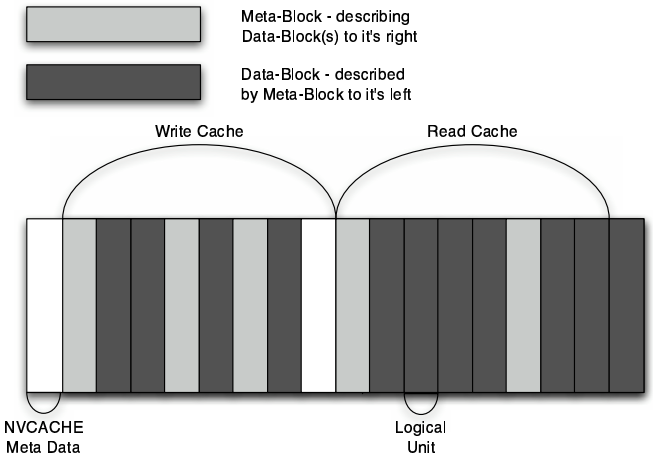


Figure 2. NVCache Data Layout

To reduce block erase operations, when an in-memory list element is removed, it isn't marked as clear on the actual device. As a result, all data in the write-cache will get written to disk, including stale data. However, since the write-cache is ordered by time, any stale data written to disk will eventually get overwritten by newer data. For example, when an older data block is written back to disk but the newer block isn't (due to lazy consistency), a read operation will still be redirected to newer data on the NVCache until it is written back to disk thus overwriting the old data.

On a read that is only covered by a portion of the data in the write-cache, overlapping data is still read into memory. The non-overlapping regions residing on disk are also read into memory. When all of the data is in memory, it is merged and returned to the caller. However, before finishing the request and discarding the data, the redirected request, which is now in memory, is reformatted to look like the original write request and written back to disk. The associated entry in the in-memory list is then removed.

	Seconds	MB/S
Flash Read	19.85	2.51
Disk Write	44.48	1.12
Total	64.60	0.77

Table 2. Flushing write-cache to Disk

3.2 Simulator

In addition to the kernel implementation, we implemented an NVCache simulator. The simulator has several run-time options: write-cache consistency model, prefetching algorithm for the prefetch read-cache, and the size for each cache. Both LRU/MRU and LFU/MFU prefetching algorithms are available. Any configuration of the two caches is possible simply by specifying the cache desired and size. When both prefetching algorithms are selected, they each have their own logical partition within the read-cache specified by its size. There are two options for the consistency model. They are full-flush and lazy consistency. Full flush flushes the entire write-cache back to disk on a spin-up. In our evaluation we use the lazy consistency technique.

The LRU/MRU prefetching algorithm is implemented by prefetching each read access (MRU) from the disk while it is active to the read-cache if it is not already in the read-cache and using LRU eviction policy. The LFU/MFU implementation is slightly more complex. We maintain a hash of all disk read requests. A request consists of its sector, offset, and length. The hash value is the frequency of that particular request. Every time the same request occurs, the frequency is incremented. If the request is not in the read-cache, its frequency is compared against the request with the lowest frequency from the read-cache, and replaces it if the request not in the read cache has a higher frequency.

4 Evaluation

We evaluate NVCache using the simulator implementation. However, we did test the overhead of full-flush with the kernel implementation. The results are in Table 2. The kernel implementation uses Linux kernel version 2.6.8.1 (Gentoo distribution) on a 3GHz Pentium 4 machine. The compact flash and disk used are described in Table 1. The interconnect is over USB 2.0. This experiment measures the total time to flush 50MB of non-sequential data back to the hard-disk. Reading from flash is significantly faster than writing to disk because of the log data layout format. We exploit this layout by reading 1MB of write-cache at a time, then reconstructing multiple original disk I/Os in-memory and resubmitting them to the disk queue.

4.1 Traces

To properly evaluate the simulator, we use three disk traces from real workloads: cello, an engineering workstation, and a personal web/mail server.

The Cello trace is from an HP-UX server in 1999 and records disk activity. We use a seven-day snapshot of the trace from 5/10/1999 through 5/16/1999.

We recorded disk accesses from an engineering workstation and web/mail server. The workstation was traced for 5 days, and the server trace for 71 days. Both systems were running Linux from a Gentoo Distribution with a 2.6 kernel. The engineering workstation was used primarily for software development, e-mail, web-browsing, and text preparation. Newer software packages were automatically updated each night from source on the system. The disk being traced held the root file system (ReiserFS).

The web/mail server used apache and postfix for serving web content and mail services, respectively. The disk traced contained the data directory for both the mail and web server applications, and the root file system (ReiserFS). It is important to note that both systems ran fcron, which updated the spool file every 15 minutes.

We recorded all I/O operations destined for the traced disk at the disk driver level and appended an entry describing each I/O request to an in-memory kernel data structure containing 8KB worth of entries. To avoid affecting the trace itself, when the data structure filled up, it was passed to a user-space process, which appended the data structure to a file on another disk. The format of a trace entry is:

```
struct trace_entry{
    unsigned long long time;
    unsigned int rw;
    unsigned long long sector;
    unsigned int size;
};
```

4.2 Experiments

In most figures, there is an ”ME SD” plot. This plot stands for the Multiple Experts Spin-Down algorithm. It represents using the Multiple Experts Spin-Down algorithm without an NVCache, which is why the plot is horizontal with respect to NVCache size. All other plots use an NVCache with the Multiple Experts Spin-Down algorithm.

There are four possible NVCache combinations: *write*, *write/LFU*, *write/LRU*, and *write/LRU/LFU*. *Write* represents the write-cache, while *LRU* and *LFU* represent the prefetching algorithm for the read-cache. We use equal cache size partitioning in our experiments when combining multiple read-cache policies. For example, in Figure 3(a), plot *write/LRU/LFU* with cache size 1MB, the actual size of the write-cache is 333KB while the read-cache is

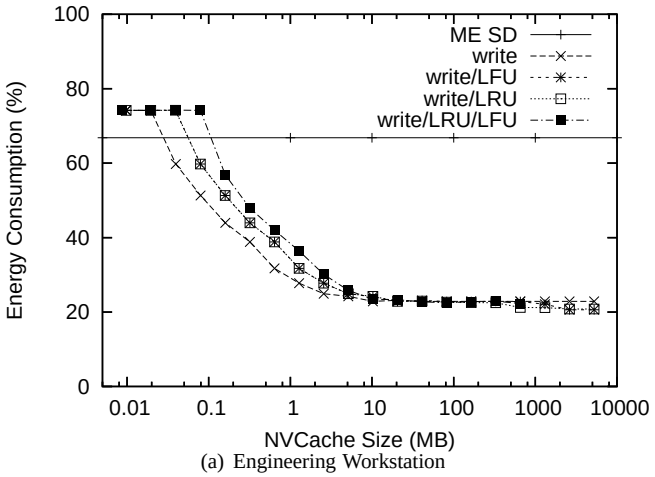
666.66KB, divided equally for the LRU and LFU prefetching algorithm. Similarly, if the total cache size is 1MB and the plot is *write/LRU*, the write-cache and read-cache are both 500KB.

Figure 3 shows the percentage of energy consumed relative to not spinning down the hard disk. We use the energy model for a hard disk and compact flash device described in Table 1. For NVCache sizes under 10MB, the write-cache alone performs better than any combination including a read-cache. This is primarily due to the fact that any combination effectively decreases the write-cache size by two or three times. This effect is more pronounced in the personal server and engineering workstation because writes occur more frequently between idle periods—fcron periodically updating the spool file.

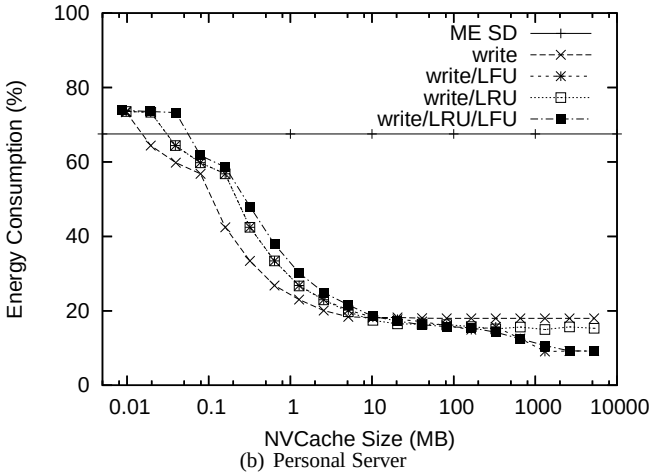
In plot *write*, the effectiveness of the write-cache alone plateaus between 1 and 10MB as read requests are not satisfied by write-cache content. With larger NVCache sizes, a read-cache is effective at increasing spin-down periods. Looking at the results for the Cello trace at a 2GB NVCache size, the energy consumed goes from 90% to 70% when partitioning the NVCache equally into a *write/LFU* cache from a *write* cache alone. Since the personal server and engineering workstation are write-dominated during idle periods, relative to Cello, a 10MB write-cache alone decreases their energy consumption down to 20%. Yet, adding an LFU read cache to the personal server when the NVCache device is greater than 2GB still decreases the energy consumption from 20% to 10%.

Figure 3 also shows how different prefetching algorithms affect NVCache performance. The *write/LFU* combination significantly outperforms the *write/LRU* combination. We speculate this is because double caching occurs between the LRU read-cache and buffer cache, which explains why the performance of the *write/LRU* configuration is not much better than the *write* only configuration. However, by prefetching only the most frequently used data to the read-cache we can avoid double caching effects. We also tried using both an LRU and LFU prefetching algorithm, but the results do not show a noticeable difference when compared to the *write/LFU* configuration.

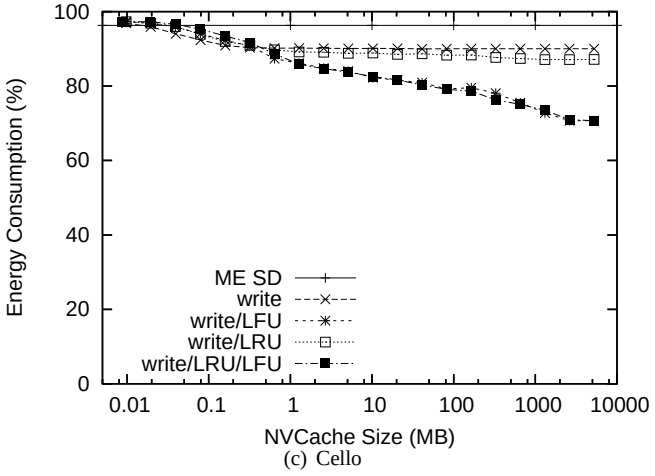
Figure 4 shows the average number of ops satisfied by the NVCache per spin-down period. This includes both read and write operations. The plot "ME SD" is missing from all three figures because no operations can be satisfied without an NVCache. Again, the *write/LRU* marginally increases the average number of satisfied operations over the *write* plot alone, probably because of the previously mentioned effects of double caching. However, Figure 4(a) *write/LRU* deviates from this pattern by increasing proportionally to the other two read-cache combinations. Neither LFU read-cache combinations in Figure 4(a) make the dramatic increase in satisfied operations that the personal



(a) Engineering Workstation

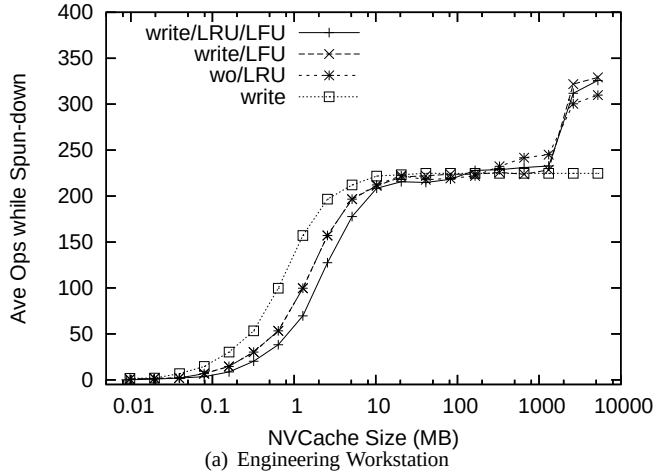


(b) Personal Server

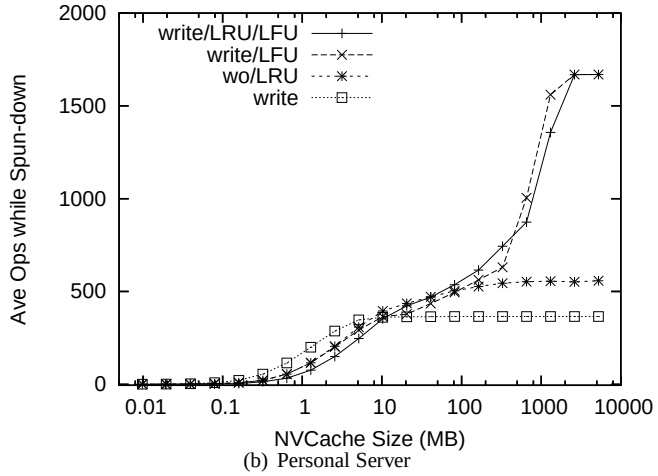


(c) Cello

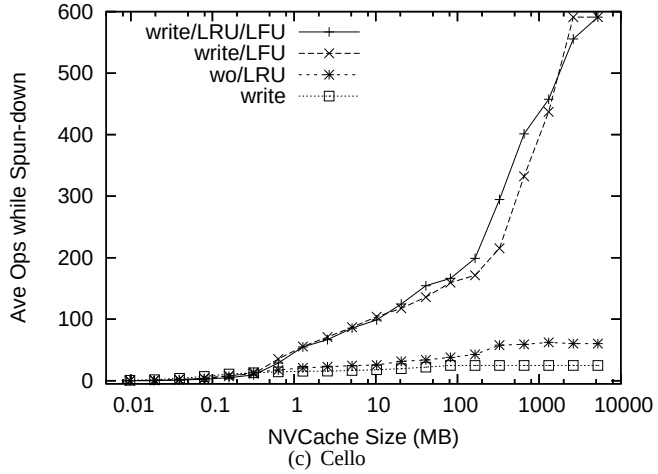
Figure 3. Energy Consumption: relative to an always on hard disk



(a) Engineering Workstation



(b) Personal Server



(c) Cello

Figure 4. Average # of I/Os satisfied per spin-down period

server or Cello make with an LFU read-combination at larger NVCache sizes, allowing the *write/LRU* plot in Figure 4(a) to increase proportionally to the LFU read-cache combinations. At NVCache size 2GB, cello has a fifteen times increase with *write/LFU* over *write*, while the engineering workstation only has a 50% increase.

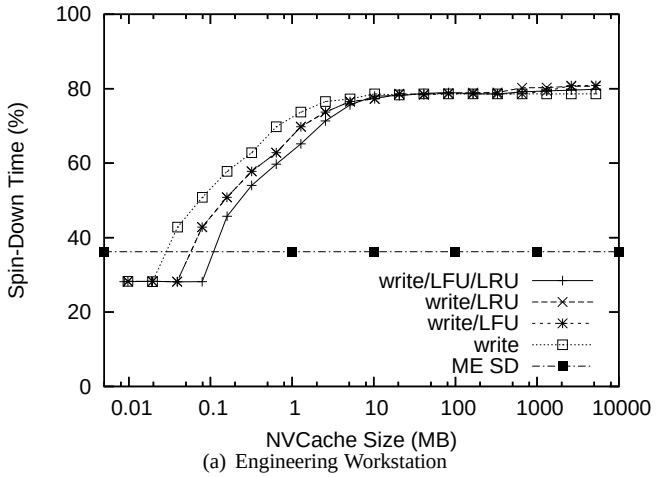
Figure 5 shows the percentage of time the hard disk was spun-down relative to the duration of the trace. For the Cello trace, the disk was spun-down for 4% of the trace duration without an NVCache. Adding a write-cache of size 2GB increased this percentage to 10%, and partitioning the NVCache into a *write/LFU* cache at NVCache size 2GB, the total spin-down percentage increased to 30% for an improvement of seven and half times over the spin-down algorithm alone. For the other two traces, the spin-down algorithm alone kept the disk spun down for just under 40% of the trace. By only adding the *write* cache, the spun-down percentage roughly doubled. Adding an *LRU* or *LRU* cache marginally increased their total spin-down time.

An interesting result in Figures 5(a) and 5(b) is that for cache sizes of 100KB or less, the actual time spent spun-down was actually less than the multiple experts algorithm alone. Although we expected this for energy consumption—buffering a few additional operations does not justify the additional power consumption of a flash device. However, seeing similar results in a time-only metric was unexpected. We discovered it is an artifact of the spin-down algorithm's time-out calculation. When the spin-down algorithm was used alone, it would generate small time-out values and therefore stay spun-down longer. With a small NVCache, a couple operations could be satisfied by the NVCache, which caused the spin-down algorithm to generate slightly longer time-out values keeping the disk in active mode longer.

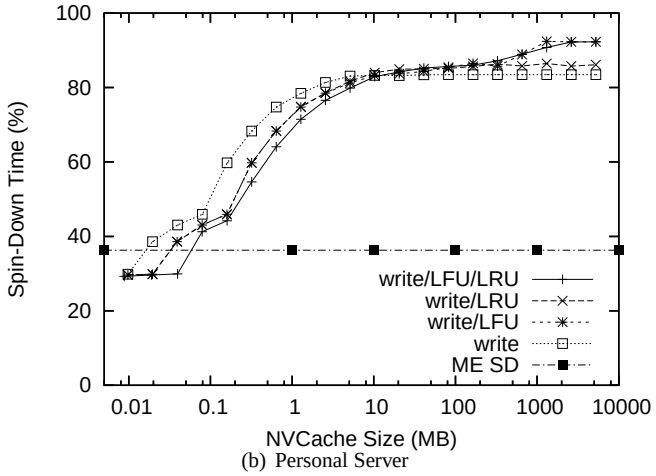
Figure 6 shows the overhead of adding a read-cache to the NVCache. These figures shows the percentage of read-updates to actual disk read operations. It includes updates where the data has gone stale in the read-cache due to a disk over-write operation. Fundamentally, they represent the total number of cache insertions, which impacts performance and NVCache device reliability. In (a), (b), and (c) the *write/LFU* read-cache incurs the least number of insertions. For all the traces, a 1.3GB LFU read-cache is sufficient to contain a large percentage of the working set.

5 Related work

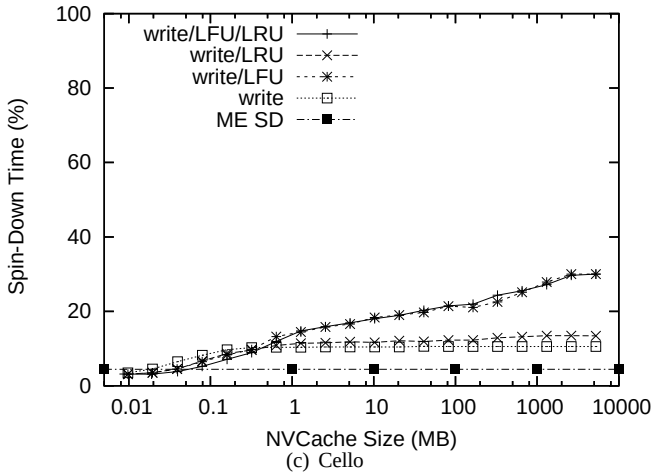
Power Management is traditionally performed in the Operating System. However, applications are responsible for generating the I/O which OS-level power management policies are designed for. By providing applications with power-aware interfaces or dynamically modifying an application to be power-conscious, OS-level power management



(a) Engineering Workstation

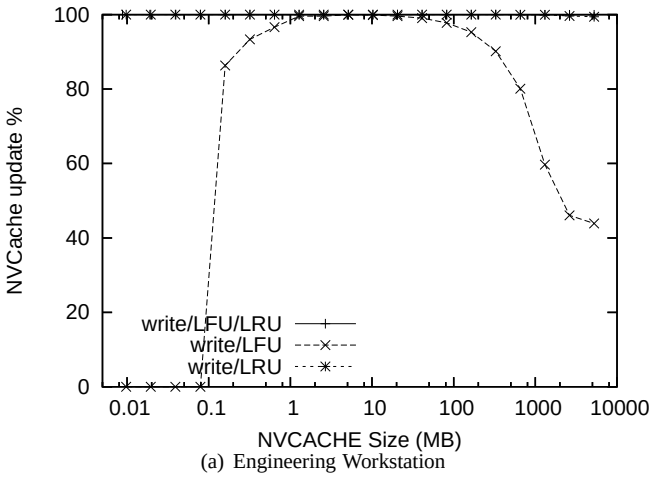


(b) Personal Server

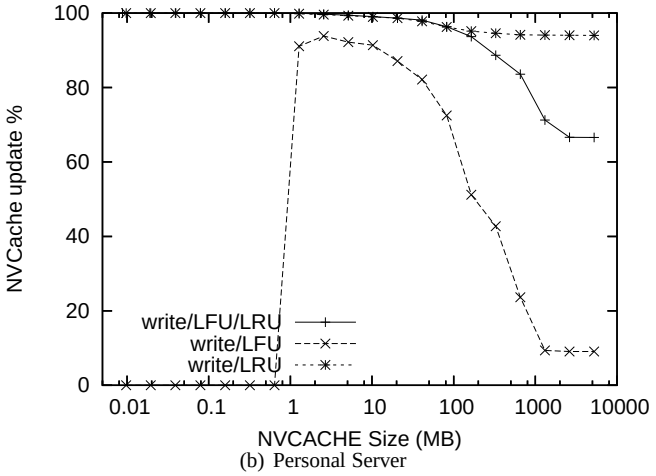


(c) Cello

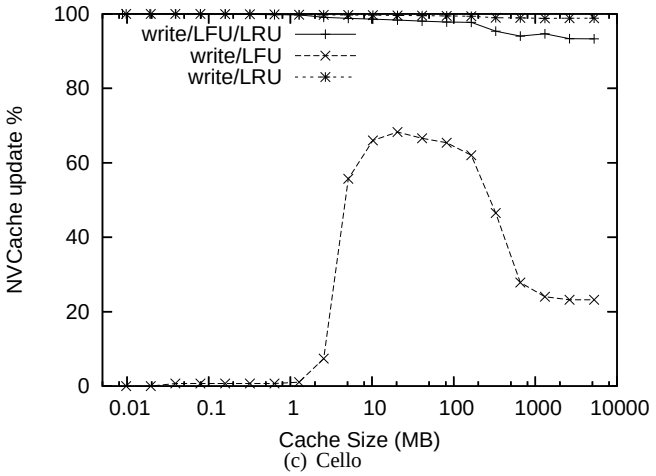
Figure 5. Spin-down time: relative to an always on hard disk



(a) Engineering Workstation



(b) Personal Server



(c) Cello

Figure 6. Read-cache overhead: read-cache updates vs. total read I/Os

schemes can perform better.

Cooperative I/O proposes to provide power-aware I/O interfaces to applications [36]. This work introduces three replacement I/O operations for open, read, and write. The main contribution of this work is that when these cooperative I/O interfaces are used, their work can be deferred till an opportune time, such as when the disk is back in active mode, from a low-power mode.

Heath *et al.* propose to increase disk idle periods [17]. Their approach is done within the compiler by profiling source code, then generating code that increases idle times. Their compiler also generates code that notifies the operating system of upcoming idle periods and their length.

5.1 Non-Volatile Memory File Systems

There are several file systems whose physical media is designed for solid-state memory. Examples of such file systems include JFFS2 [30] and YAFFS [8]. They are designed for raw flash and implement a Flash Translation Layer (FTL) to provide a mapping between flash and a block device interface. They also implement their own wear-leveling techniques. eNVy is another flash file system, but it uses SRAM to minimize block-erase operations [37]. Some flash devices, such as CompactFlash by Sandisk [33] build an FTL into the controller such that an operating system can treat the device as a normal block device.

Douglis *et al.* looked at storage solutions for Mobile computing [11]. They primarily investigate the trade-offs of using flash memory versus hard disks for primary storage. Additionally, they look at using SRAM and DRAM to function as a buffer cache. They found that flash memory can provide significant energy conservation while providing decent I/O performance.

Hybrid disk/non-volatile memory file systems aim to increase performance by using non-volatile memory to store data alongside the disk [35, 25]. The fundamental design question is, which data should be placed in non-volatile memory to provide faster I/O. Conquest proposes to use non-volatile memory to store small data files, all metadata, and shared libraries. HeRMES stores all meta-data and the first few bytes of files in non-volatile memory. HeRMES also uses the non-volatile memory as a persistent write cache.

5.2 Non-Volatile Memory as a write cache

After analyzing traces of several systems, Ruemmler and Wilkes concluded that using a small non-volatile memory write cache for each disk can significantly improve performance [31]. Baker *et al.* looked at using NVRAM as a file cache to reduce write traffic to a file server [3]. RAPID-Cache proposes to use two caches, of which one is required to be NVRAM, on top of a log-structured cache-disk residing on primary RAID storage [20]. By caching at so many hierarchies, disk latencies can be hidden.

Preliminary work has been performed by looking at flash as a cache to decrease power consumption of disks [24, 23]. "FLASHCACHE" is a read/write flash cache that exists between disk and memory. FLASHCACHE differs from our work because it directly sits in between RAM and disk, and services requests while the disk is in active mode, which is done for performance reasons. However, an artifact of this design is that FLASHCACHE's ability to absorb write-traffic while the disk is spun-down is diminished. FLASHCACHE only uses LRU as it's replacement policy, while we investigate other replacement and insertion policies. Additionally, FLASHCACHE does not describe meta-data management or consistency.

Microsoft has proposed an extension to hard disk drives to include a non-volatile cache, which a host can then manage through an extended set of ATA commands [26]. Coupling the non-volatile cache the disk drive reduces possible cache corruption, but restricts it's usefulness to a particular disk. Microsoft plans to use such a drive to reduce power consumption, but also decrease boot and resume times. Unfortunately, there are no details describing the non-volatile cache data layout or management policies.

5.3 Prefetching for Power

Papathanasiou and Scott show that prefetching is useful for power conservation as well as performance [28]. Their work focuses on speculative file prefetching into main memory using file access hints passed from the application. The hints include access sequentially, loop, and random access. The operating system uses file access information to perform intelligent prefetching and make more informed spin-down decisions. Additionally, they increase the dirty page write-back time-out from 30 seconds to 1 minute. Rybczynski *et al.* show that when prefetching to conserve energy, a single prefetching algorithm may not be sufficient [32]. A supervisor algorithm is useful to dynamically select the most energy efficient prefetching algorithm.

LaRosa and Bailey attempt to provide a new approach to reduce energy consumption for mobile devices [21]. Their approach uses non-volatile memory to reduce energy consumption during mobile use by prefetching likely to be used files to a non-volatile cache during plugged-in mode. When the laptop enters mobile-mode the hard disk is spun-down and files are accessed from the non-volatile memory cache.

5.4 Storage System Power Management

Power is becoming a design consideration for storage systems since a large portion of the TCO for storage system

comes from power consumption. Recent work into reducing the power consumption of disks in storage systems includes Hibernator, a storage system designed around multi-speed disks [38]. Hibernator tries to balance energy consumption with desired performance goals. MAID (massive arrays of idle disks) performs power management by treating a subset of disks in the storage system as as cache disks to absorb I/O traffic [7]. Pinheiro and Bianchini take the approach of data migration, rather than caching, by distinguishing between hot and cold data. Hot data is migrated to active disks and cold data is is migrated to disks which are spun-down [29].

5.5 I/O redirection

Redirecting I/O to another source while a disk has been spun-down has been done with RAID 1 schemes, in which reads to a spun-down disk are redirected to its mirror [22]. In previous work, we proposed I/O redirection for a distributed object-based file system with smart disks [4]. The approach redirects both reads and writes to other disks in the file system. Reads are redirected to locations containing replicas and writes are redirected temporarily to a cache on other active disks.

6 Conclusion

In this paper, we have presented the design, implementation, and evaluation of NVCache - a technique to decrease energy consumption by increasing idle periods of spun-down hard disks. NVCache buffers writes to a non-volatile low-power device while the disk is spun-down. NVCache also services reads on behalf of the spun-down disk from the write-cache as well as a read-cache, which is also located on the device. The read-cache contains cached copies of popular disk data, prefetched from the disk while it was active using using temporal caching algorithms. We investigated the use of two popular caching algorithms: LRU and LFU, and their combination. NVCache is complimentary to any disk spin-down algorithm. We have shown that by combining NVCache with an adaptive disk spin-down algorithm, a hard disk's energy consumption can be reduced by up to 90%.

References

[1] Symmetrix 3000 and 5000 enterprise storage systems product description guide. `http://www.emc.com`, 1999.

[2] X. L. , Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar. Performance-directed energy management for main memory and disks. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, October 2004.

[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 10–22, 1992.

[4] T. Bisson, J. Wu, and S. A. Brandt. A distributed spin-down algorithm for an object-based storage device with write redirection. In *Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS '06)*, January 2006.

[5] E. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th International Conference on Supercomputing*, June 2003.

[6] N. Cesa-Bianchi, Y. Freund, D. Haussler, and D. P. Helmbold. How to use expert advice. *Journal of the ACM*, 44(3):427–485, 1997.

[7] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[8] A. O. Company. Yet another flash file system. `www.aleph1.co.uk/yaffs`, 2004.

[9] A. Corporation. Executive summary: Flash quality. `http://www.adtron.com/pdf/AdtronFlashQual121103.pdf`, 2003.

[10] Dell. Dell poweredge 6650 executive summary. `http://www.tpc.org/results/individual_results/Dell/dell_6650_010603_es.pdf`, Mar 2003.

[11] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation*, pages 25–37, 1994.

[12] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, pages 130–142, 1996.

[13] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.

[14] Y. G. De Micheli. Adaptive hard disk power management on personal computers. In *IEEE Greate Lakes Symposium on VLSI*, pages 50–53, March 1999.

[15] D. Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[16] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Rpm: dynamic speed control for power management in server class disks. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-03)*, pages 169—181, June 2003.

[17] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Application transformations for energy and performance-aware device management. In *Proceedings of the Eleventh Conference on Parallel Architectures and Compilation Techniques (PACT'02)*, September 2002.

[18] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 130–142, 1996.

[19] Hitachi. Deskstar 7k400. `http://www.hitachigst.com/tech/techlib.nsf/products/Deskstar_7K400`, 2004.

[20] Y. Hu, T. Nightingale, and Q. Yang. Rapid-cache-a reliable and inexpensive write cache for high performance storage systems. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):290–307, 2002.

[21] C. R. LaRosa and M. W. Bailey. A docked-aware storage architecture for mobile computing. In *Proceedings of the first conference on computing frontiers on Computing frontiers*, pages 255–262, 2004.

[22] D. Li and J. Want. Conserving energy in conventional disk based raid systems.

[23] K. Li. Towards a low power file system. Technical Report UCB/CSD-94-814.

[24] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *To appare in Proceedings of the 27th Hawaii Conference on Systems Science*, 1994.

[25] E. L. Miller, S. A. Brandt, and D. D. E. Long. Hermes: High-performance reliable mram-enabled storage. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 95. IEEE Computer Society, 2001.

[26] N. Obr and F. Shu. Non volatile cache command proposal for ata8-acs. `http://t13.org`, 2005.

[27] J. V. P. Krishnam, P.M. Long. Adaptive disk spin-down via optimal rent-to-buy in probabilistic environments. In *Proceedings of the 12th annual Internation Conference on Machine Learning*, pages 322–330, July 1995.

[28] A. Papathanasiou and M. L. Scott. Energy efficient prefetching and caching. In *Usenix '04 Annual Technical Conference*, June 2004.

[29] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 68–78. ACM Press, 2004.

[30] I. Red Hat. Jffs2: The journalling flash file system. `http://sources.redhat.com/jffs2/jffs2.pdf`, 2001.

[31] C. Ruemmler and J. Wilkes. UNIX disk access patterns. Technical Report HPL-92-152, HP Laboratories, December 1992.

[32] J. P. Rybczynski, D. D. E. Long, and A. Amer. Expecting the unexpected: adaptation for predictive energy conservation. In *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 130–134, New York, NY, USA, 2005. ACM Press.

[33] Sandisk. Operation of cf host operation. `http://www.sandisk.com`.

[34] Sandisk. Sandisk flash memory cards wear leveling. `http://www.sandisk.com/Assets/File/OEM/WhitePapersAndBrochures/RS-MMC/WPaperWearLevelv1.0.pdf`, 2003.

[35] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2002.

[36] A. Weissel, B. Beutel, and F. Bellosa. Cooperative i/o: a novel i/o semantics for energy-aware applications. *SIGOPS Oper. Syst. Rev.*, pages 117–129, 2002.

[37] M. Wu and W. Zwaenepoel. envy: a non-volatile, main memory storage system. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 86–97, New York, NY, USA, 1994. ACM Press.

[38] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 177–190, New York, NY, USA, 2005. ACM Press.

[39] Q. Zhu and Y. Zhou. Power-aware storage cache management. *IEEE Transactions on Computers*, pages 587—602, May 2005.