EXCES: EXternal Caching in Energy Saving Storage Systems

Luis Useche, Jorge Guerra, Medha Bhadkamkar, Mauricio Alarcon and Raju Rangaswami
School of Computing and Information Sciences, Florida International University, Miami, FL.
{luis,jguerra,medha,malar002,raju}@cis.fiu.edu

Abstract

Power consumption within the disk-based storage subsystem forms a substantial portion of the overall energy footprint in commodity systems. Researchers have proposed external caching *on a persistent, low-power storage device, which we term* external caching device (ECD)*, to minimize disk activity and conserve energy. While recent simulation-based studies have argued in favor of this approach, the lack of an actual system implementation has precluded answering several key questions about external caching systems. We present the design and implementation of EXCES, an external caching system that employs prefetching, caching, and buffering of disk data for reducing disk activity. EXCES addresses important questions related to external caching, including the estimation of future data popularity, I/O indirection, continuous reconfiguration of the ECD contents, and data consistency. We evaluated EXCES with both micro- and macro- benchmarks that address idle, I/O intensive, and real-world workloads. Overall system energy savings was found to lie in the modest 2-14% range, depending on the workload, in somewhat of a contrast to the higher values predicted by earlier studies. Furthermore, while the CPU and memory overheads of EXCES were well within acceptable limits, we found that flash-based external caching can substantially degrade I/O performance. We believe that external caching systems hold promise. Further improvements in ECD technology, both in terms of their power consumption and performance characteristics can help realize the full potential of such systems.*

1 Introduction

The need for energy-efficient storage systems for both personal computing and data center environments has been well established in the research literature. The key argument is that the disk drive, the sole mechanical device in modern computers, is also one of its most power consuming [15]. The varied proposals for addressing this problem include adaptive disk spin down policies [10, 13, 17], exploiting multi-speed drives [12, 23, 29, 30], using data migration across drives [7, 9], and energy-aware prefetching and caching techniques [22, 25, 28].

A different approach, complementary to most of the above techniques, is *external caching*[1] on a non-volatile storage device, which we shall henceforth refer to as *external caching device* (ECD). An ECD can be any non-volatile device that consumes less power than a disk drive, such as [14, 26]. Recent technological advancements, adoption trends, and economy-of-scale benefits have brought the non-volatile flash-based storage devices into the mainstream. Recent work on external caching have presented the merits of such systems. While these studies serve to make the case for further research in external caching systems, they still leave several key questions unanswered. First, these studies do not evaluate the power consumption of the system as a whole, but only focus on the reduction in disk power consumption. It is important to refine this evaluation criteria since the ECD subsystem itself can consume a considerable amount of energy. Second, existing studies do not evaluate an important artifact of external caching, which is the impact on application performance. Flash-based devices handle random reads much better than disk drives, but perform slightly worse than disk drives for sequential accesses and substantially worse for random writes [2, 11]. Third, the existing approaches base their evaluation of external caching on simulation models [4, 8, 18]. While simulation-based evaluation may be well-suited for an approximate evaluation of a system, they also sidestep key design and implementation complexities as well as preclude evaluating the overhead contributed by the system itself.

In this paper, we present EXCES, an external caching system for energy savings, that comprehensively addresses the above questions and advances the state of our understanding of external caching systems. EXCES operates by utilizing an ECD for prefetching, caching, and buffering of disk data to enable the disk to be spun-down for large periods of time and saving power. EXCES is an online system — it adapts to the changing workload by identifying popu-

[1] We term this as "external caching" to primarily differentiate it from in-memory caching.

lar data continuously, reconfiguring the contents of the ECD (as and when appropriate) to maximize ECD hits on both read and write operations. Prediction of future popularity in EXCES is based on a novel technique that accounts for both the recency and frequency of accesses along the time line. To prefetch popular data which are not present in the ECD, EXCES opportunistically reconfigures the ECD contents, when the disk is woken up on an ECD *read miss*. EXCES always redirects writes to the ECD, regardless of whether the written blocks were prefetched/cached in the ECD; this is particularly important since most systems perform background write IO operations, even when idle [8, 24, 25]. All of the above optimizations minimize disk accesses and prolong disk idle periods, consequently conserving energy.

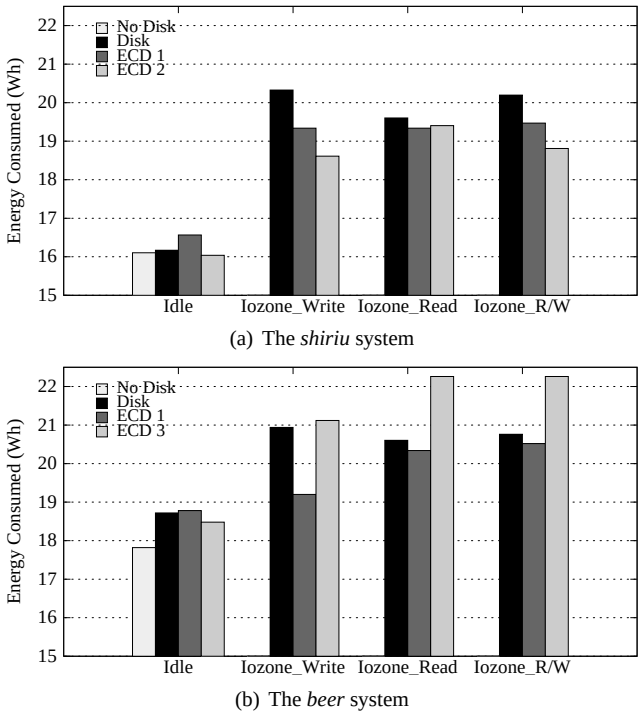
We implement EXCES as a Linux kernel module to demonstrate the suitability of external caching in production systems. EXCES operates between the file system and I/O scheduler layers in the storage stack, making it independent of the filesystem and availing kernel I/O scheduling automatically. EXCES provides strong block-layer data consistency for all blocks managed by upper layers, by maintaining a persistent page-level *indirection map*. It successfully addresses the challenges of page indirection, including partial/multiple block reads and writes, optimally flushing dirty pages to the disk drive during reconfiguration, correctly handling foreground accesses to pages that are undergoing reconfiguration, and ensuring "up-to-dateness" of the indirection map under all these conditions.

We evaluated EXCES for different workloads including both micro-benchmarks and laptop-specific benchmarks. In most cases, EXCES was able to save a reasonable amount of energy ($\sim$2-14%). However, we found that using a flash-based ECD can substantially degrade I/O performance and careful consideration is needed before deploying external caching, especially in performance-centric data center environments [11]. Finally, we measured the resource overheads incurred due to EXCES and found these well within acceptable limits.

The rest of this paper is organized as follows. In Section 2, we profile the power consumption of disk drives, ECDs and ECD interfaces, on two different systems. Section 3 presents the architecture of EXCES. Section 4 presents the detailed design and Section 5 overviews our Linux kernel implementation of EXCES. In Section 6, we conduct an extensive evaluation of EXCES. Related research is discussed in Section 7. We make concluding remarks and outline future work in Section 8.

2 Profiling Power Consumption

To understand the power consumption characteristics of ECD relative to disk drives, we experimented with two different NAND-flash ECDs and three different ECD interfaces on two laptop systems. Table 1 shows the different configurations of the devices used in the profiling experiments.[2] All ECD devices were 2GB in size. We measured the overall system power consumption for four states: when the system was idle with each device merely being active, and with the Iozone [20], an I/O intensive benchmark, generating a read intensive, write intensive, and read-write workload.



(a) The *shiriu* system

(b) The *beer* system

Figure 1. Power consumption profiles of various ECD types and interfaces.

Figure 1 depicts the individual power consumption profiles for each storage device on two different laptops: `shiriu` and `beer`. A detailed setup of each machine is given in Section 6 (Please see Table 2). During each experiment exactly one device is turned on. These experiments were conducted using a Knoppix Live CD to enable complete shutdown of the disk when not being tested.

It can be observed that each machine has a distinct behavior. On the `shiriu` system, the USB subsystem consumes substantially more energy than the disk subsystem when the system is idle; we believe this is partly due to an unoptimized driver for the Linux kernel [6]. However, both types of flash memory consume less power than the disk in all the Iozone benchmarks. On the `beer` system, the find-

[2] We also tried using an SD NAND flash device. Unfortunately its Linux driver is still under development and performs poorly for writes ($<$4 KB/s); consequently, we discontinued experiments with that device.

Configuration	Disk State	Iozone Data	ECD Specification	ECD Interface
No Disk	Standby	N/A	N/A	N/A
Disk	Active	On disk	N/A	N/A
ECD 1	Standby	On ECD	SanDisk Cruzer Micro USB	USB interface
ECD 2	Standby	On ECD	SanDisk Ultra CF Type II	eFilm Express Card 34 CF Adapter
ECD 3	Standby	On ECD	SanDisk Ultra CF Type II	SanDisk Ultra PC Card Adapter

Table 1. Various laptop configurations used in profiling experiments.

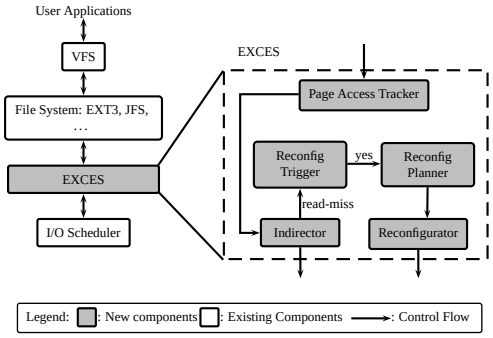


Figure 2. EXCES system architecture.

ings were somewhat surprising. Although the exact same flash device was used in the $ECD\ 2$ and $ECD\ 3$ configurations, the PC Card interface in the $ECD\ 3$ configuration negatively impacted power consumption in all the Iozone benchmarks. While we do not know the exact cause, we postulate this could be due to an unoptimized device driver.

More importantly, for both systems, even in configurations when the disk is powered down completely, we observe that the power savings are bound within 10% for an I/O intensive benchmark. Further, when the system is idle, the ECD subsystems consumes as much power as the disk drive. While the laptop workload would be somewhere in between idle and I/O intensive, these findings nevertheless call to question the effectiveness of *external caching* systems in saving power. Our goal in this study is to address this question comprehensively.

3 EXCES System Architecture

Figure 2 presents the architecture of EXCES in relation to the storage stack within the operating system. Positioning EXCES at the block layer is important for several reasons. First, this allows EXCES coarse-grained monitoring and control over system devices, at the *block device* abstraction. Additionally, the relatively simple block layer interface allows easy I/O interception and indirection, and also allows EXCES to be designed as a dynamically loadable kernel module, with no modifications to the kernel. Second, by operating at the block layer, EXCES becomes independent of the file system, and can thereby work seamlessly with any file system type, and support multiple active file systems and mount-points simultaneously. Third, internal I/Os generated by EXCES itself leverage the I/O scheduler, automatically addressing the complexities of block request *merging* and *reordering*.

EXCES consists of five major components as shown in Figure 2 . Every block I/O request issued by the upper layer to the disk drive is intercepted by EXCES. The *page access tracker* receives each request and maintains updated popularity information at a 4KB page granularity. Control subsequently passes to the *indirector* component which redirects the I/O request to the ECD as necessary. Read requests to ECD cached blocks and all write requests are indirected to the ECD. A *read-miss* occurs for blocks not present on the ECD and the read request is then indirected to the disk drive. The *reconfiguration trigger* module is invoked which decides if the state of the system necessitates a reconfiguration operation. If a reconfiguration is required, the *reconfiguration planner* component uses the page rank information maintained by the page access tracker to generate a new "reconfiguration plan" which contains the popular data based on recent activity. The *reconfigurator* uses this plan and performs the corresponding operations to achieve the desired state of the ECD. EXCES continuously iterates through this process until the EXCES module is unloaded from the kernel.

4 EXCES System Design

In designing EXCES, we used the following behavioral goals as guidelines: *(i)* increase disk inactivity periods through data prefetching, caching, and write buffering on the ECD, *(ii)* make more effective use of the ECD by continuously adapting to workload changes, *(iii)* ensure block-level data consistency under all system states, and *(iv)* minimize the system overhead introduced due to EXCES itself. In the rest of this section, we describe how the various architectural components of EXCES work towards realizing these design goals.

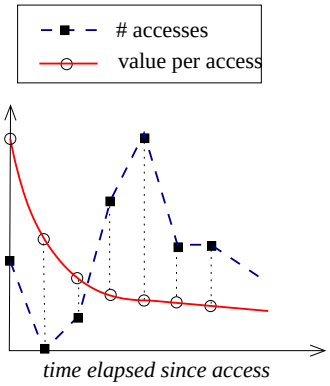


Figure 3. Page rank decay function

4.1 Page Access Tracker

The *page access tracker* continuously tracks the popularity of the pages accessed by applications. We track popularity at the *page granularity* (instead of block granularity, the unit of disk data access) to utilize the fact that file systems access the disk at the larger page granularity for most operations. This reduces the amount of EXCES metadata by a factor of 8X.

Page popularity is tracked by associating with each page a *page rank*. In our initial study we found that while accounting for recency of access was important for a high ECD hit ratio, there were certain pages that were accessed periodically. LRU-type main memory caching algorithms, tuned to minimize the total *number* of disk accesses, typically end up evicting such pages prematurely for large working-set sizes. In the case of external caching systems, if this page is not present in the ECD the disk will need to be woken up to service it periodically, thereby leaving little opportunity for energy savings. Consequently, the page ranking mechanism in EXCES provides importance to both recency and frequency of accesses to determine the rank of a page.

For each page P, the page ranking mechanism splits time into discrete quanta (t_i) and records the number of accesses to the page within each quantum (a_i^P). When the rank for a page must be updated, the page ranking mechanism weights page accesses over the time-line using an exponential decay function (f) as shown in Figure 3. The rank of a page P is obtained as $rank(P) = \sum a_i^P \cdot f(t_i)$. The page ranking mechanism thus awards a higher value for recent accesses, but also takes into account frequency of accesses, by retaining a non-trivial value for accesses in the past.

4.2 Indirector

The *indirector* is a central component of EXCES. Similar to the page access tracker, it gets activated upon each I/O request to appropriately redirect it to the ECD if required.

Require: Page Request: req, Indirection Map: map.
1: **if** req does not contain an entry in map **then**
2: **if** req is write **then**
3: **if** disk state is STANDBY **then**
4: find free (alternatively clean) page in ECD
5: **if** page in ECD is found **then**
6: add new entry in map (mark dirty)
7: change the req location as per map entry
8: **else**
9: change the req location as per map entry
10: send request req

Algorithm 1: Indirection Algorithm

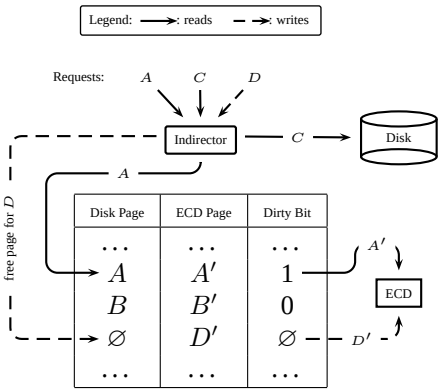


Figure 4. Indirection example.

The indirector maintains an *indirection map* data structure to keep track of disk pages that have been prefetched, cached, or buffered in the ECD. Each entry in the indirection map includes the *disk page* mapped, the corresponding *ECD page* where it is mapped to, and whether the copy in the ECD is *dirty* or not. The data structure is implemented so that we can find a specific entry, either given the page information on the ECD or the page on disk. EXCES uses native kernel data structures that allow constant time operations for the above.

For each I/O request, the indirector component first checks to see if it is larger than a page. If so, it splits it into multiple requests, one for each page. Each page request is handled based on four factors: *(i)* type of operation (read or write), *(ii)* the disk power state, *(iii)* indirection map entry, and *(iv)* presence of free/clean page in the ECD. Algorithm 4.1 shows the algorithm followed by the indirector for each page request. The algorithm attempts to keep the disk in idle state as long as possible, to maximize energy savings. This is feasible in two cases - if there is a free or clean page in the ECD (line 5) to absorb a page write request, or if the page is already mapped (line 9).

In the rest of the cases, the disk is either active or would have to be spun up owing to an ECD miss. In each such case, the ECD miss counter is incremented; this counter is

used by the *reconfiguration trigger* component of EXCES (described shortly). In the example of Figure 4, there are three page requests: A, C and D. In the case of A, there is an entry in the indirection map; consequently, it gets indirected to the corresponding page in the ECD. In the case of C, the page does not have an entry in the indirection map; the indirector lets the request continue to the disk. Finally, the write request D is handled differently than the above. There is no map entry for D. However, having found a free page in the ECD, the indirector creates a new map entry and redirects the request to the ECD, thereby avoiding spinning up the disk.

4.3 Reconfiguration Trigger

Upon each ECD miss, the indirector invokes the *reconfiguration trigger*, which determines if a reconfiguration of the ECD contents would be appropriate at the current time. If yes, it invokes the *reconfiguration planner* component (described next); otherwise, it does nothing.

The appropriateness of a reconfiguration operation depends on three necessary conditions: *(i)* the target state of the ECD contents is different than the current one; *(ii)* the current ECD miss rate (per unit of time) has exceeded a threshold, and *(iii)* a threshold amount of time has elapsed since the previous reconfiguration operation. If the above hold true, the reconfiguration trigger concludes that the current state of ECD contents is not favorable to energy saving, and consequently must be reconfigured to reflect recent changes in the workload.

4.4 Reconfiguration Planner

The *reconfiguration planner* creates a list of operations, which constitute the *reconfiguration plan*, to be performed during the next reconfiguration operation. To be able to create such a list whenever invoked, it continuously maintains a *top-k matrix* data structure, that holds the "top k" ranked pages. Choosing k as the size of the ECD in pages, this matrix can then be used to identify the target contents for the ECD that the reconfiguration operation must achieve.

The top-k matrix continuously incorporates the page rank updates provided by the page access tracker. The threshold for being inserted into the top-k matrix is set by its lowest ranked page. (We present and analyze this data structure in detail in Section 5).

The reconfiguration plan is constructed in two parts. The first are the "outgoing" pages which must be flushed to the disk; these are no longer popular enough to be in the ECD and are dirty. The second are the "incoming" pages which now have a sufficiently high rank to be in the ECD but are not currently in it. These constitute the pages to be *prefetched* to ensure a high ECD hit ratio in the future.

Construction of the reconfiguration plan occurs upon invocation by the reconfiguration trigger. The outgoing and incoming lists are then created based on the top-k matrix contents and the indirection map. The reconfiguration planner walks through each page of the ECD, creating an entry in the outgoing list for each page that is no longer in the top-k matrix. Next, it walks through each entry in the top-k matrix, creating an entry in the incoming list for each page that is currently not in the ECD. Once these two stages are completed, the new reconfiguration plan is obtained.

4.5 Reconfigurator

The *reconfigurator* component of EXCES performs the actual data movement between the disk and ECD. Broadly, the goal of each reconfiguration operation is to reorganize the ECD contents based on changes in the application I/O workload, so that disk idle periods are prolonged. This is done simply by following the reconfiguration plan as created by the planner component.

Require: Phase: $phase$, Origin: $orig$, Destination: $dest$, Indirection Map: map Table: map
1: **if** $phase = DISK\_TO\_ED$ **then**
2: add a new entry $[orig, dest]$ in the map
3: mark the new entry as "clean"
4: read from $orig$
5: write to $dest$
6: **if** $phase = ED\_TO\_DISK$ **then**
7: delete the entry for $dest$ from the map

Algorithm 2: Algorithm used for a single operation during reconfiguration.

The reconfiguration operation is managed in two distinct "phases": *ECD to Disk* and *Disk to ECD*. These two phases are treated differently, and are detailed in Algorithm 2. The first phase, *ECD to Disk*, addresses operations in the *outgoing* list of the reconfiguration plan. For each entry in the list, the data movement operation is *followed by* deleting the corresponding entry in the indirection map. The second *Disk to ECD* phase, handles the incoming list in a similar way, except that a new entry is added to the indirection map, *prior to* the actual data movement.

Indirection during reconfiguration.
Indirecting I/O requests issued by applications during the reconfiguration operation must be carefully handled due to implicit race conditions. A race condition arises if an application accesses a page currently being reconfigured. While it is perhaps simpler to postpone servicing the application I/O request until the reconfiguration operation for the page is completed (to ensure data consistence), this delay can be avoided. We designed separate policies to handle *read* and *write* operations issued by the application. If the application issued a *read* request, the indirector issues the read to

the $origin$ page location so it provides the most up-to-date data. For a *write* request by the application, the request is issued to the $dest$ location and the reconfiguration for the page is discontinued. These policies help to alleviate the overhead the reconfiguration causes to the user level applications by minimizing I/O wait time for foreground I/O operations.

4.6 Other Design Issues

Disk spin-down policy.
Researchers have proposed two classes of policies: *dynamic* and *static* [13, 17, 18]. In dynamic policies, the system dynamically varies the time the disk needs to stay idle before being put on standby. In EXCES we chose to use a static policy with a fixed timeout for spin-down. In the evaluation section, we experiment with various values for this timeout parameter.

Data consistency in EXCES.
Data consistency is always an important issue whenever multiple copies of the same information exist. In EXCES, data is replicated in the ECD. We need to ensure that the system reads up-to-date versions of data after rebooting the machine as well as in case of system crash or sudden power failure. We reserve the first portion of the ECD to maintain a persistent copy of the indirection map.

This persistent copy of the indirection map is updated each time the map is changed and gets invalidated if the EXCES kernel module is unloaded cleanly. In case of a power failure or system crash, all entries contained in the persistent indirection map are assumed to be dirty.

5 EXCES System Implementation

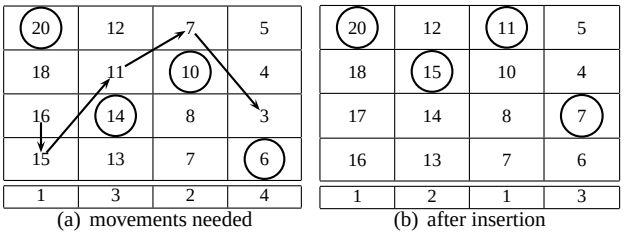
We implemented EXCES as a Linux kernel module that can be dynamically inserted and removed without any changes to the kernel source. Since the block layer interface of the Linux kernel is very stable, EXCES can run "out of the box" on the latest 2.6 series kernels. The current implementation of EXCES utilizes native kernel data structures such as *radix trees* and *red-black trees* which are very likely to be retained in the future kernel versions. In this section, we elaborate on key aspects of the EXCES system implementation that are novel and those which were particularly challenging to "get right".

5.1 Maintaining the Top-k Ranked Pages

The EXCES page ranking mechanism (described in Section 4.1) considers both recency and frequency of page accesses. Figure 5 shows the `page_ranker_t` structure

```
typedef struct {
  unsigned int rank;
  unsigned int tmp_rank;
  unsigned short last_acc[H_SIZE];
  unsigned int disk_lbn;
} page_ranker_t
```

Figure 5. The `page_ranker` structure



(a) movements needed (b) after insertion

Figure 6. The Top-k matrix. Figure 6(a) shows the matrix before the insertion and indicates the necessary movements and the resulting matrix after inserting the entry 17 is shown in Figure 6(b).

that is used to encapsulate the rank of a page. This structure allows us to efficiently capture the history of page rank values updated due to accesses over time. `disk_lbn` stores the starting on-disk *logical block number* of the page and `last_acc` array contains the timestamps of the last `H_SIZE` accesses (default is 4). Each time the `last_acc` array is filled up, it is passed to the ranking decay function (Figure 3); the resulting values are stored in `tmp_rank` using a compact representation and the `last_acc` array is reset. Before overwriting the `tmp_rank`, its previous value is decomposed and added to the historical rank of the page contained in `rank`. The *actual* rank of a page at any time is given by the sum of decomposed `tmp_rank` and `rank` values.

To be able to access the top-k ranked pages (whenever required by the reconfiguration planner), we implemented the *top-k matrix*, a novel matrix data structure of dimensions $(\sqrt{k}+1)\times(\sqrt{k})$, which stores the top-$k$ ranked pages. Since k can be large (as much as 10^8 for gigabyte-sized ECDs), operations on the top-k matrix must be highly efficient. While regular sorted matrices are good for lookups ($O(log(\sqrt{k}))$ using binary search in both columns and rows), insertions are expensive at $O(k)$ since all the lower (or upper) values must be shifted. To reduce the insertion cost, we use an extra row to store an offset that indicates where the maximum value of the column is located; all the elements of that column are also sorted according to that offset. (Please see Figure 6(b) for an example.) By maintaining this extra information we retain $O(log(\sqrt{k}))$

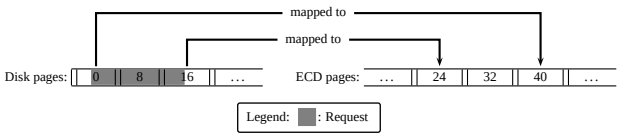


Figure 7. Alignment problem example

lookups and can also perform insertions in $O(\sqrt{k})$. This is because, in the worst case, we need to shift elements in exactly half a column and transfer its minimum to the next column where it becomes the maximum, and so on, until we reach the last column. A detailed example of the worst-case insertion process into the top-k matrix is presented in Figure 6.

Detecting if a page belongs in the top-k highest ranked pages is as easy as checking if its rank is greater than the minimum rank in the top=k matrix, in which case the page must be inserted, and marked for inclusion in the next round of reconfiguration.

5.2 Indirector Implementation Issues

As mentioned earlier, in EXCES we chose to maintain metadata about data popularity and data replication at the granularity of a *page*. While this optimization allows us to drastically cut down on metadata memory requirement (by 8X), it complicates the implementation of the indirector component. Since I/Os may be issued at the block granularity, the indirector component must carefully handle I/O requests whose sizes are not multiples of the page-size and/or which are not page-aligned to the beginning of the target partition. In EXCES, we address this issue via I/O request splitting and page-wise indirection.

Figure 7 shows an example of the alignment problem that the indirector must handle. Notice that two pages on the disk mapped to the ECD. The first page on the disk that starts at block 0, is mapped to the fifth page on the ECD that starts from block 40. Also, the third page in the disk (starting at block 16), is mapped to the fourth page of the ECD, starting at block 24 of the ECD. The second page in the disk is not mapped to the ECD at all.

Consider an application I/O request as represented by the shaded region. This request covers a part of the first, the entire second page, and a part of the third page on disk. The indirection operation is complicated because the I/O request is not page-aligned. The indirector must individually redirect each part of the request to their appropriate locations. The above can occur with both read and write I/O requests.

In EXCES, to address I/O splitting, we create new requests using the Linux kernel *block I/O* structure called `bio`, one per page. All attributes of the `bio` structure are automatically populated based on lookups to the *indirection map*, including the sector, offset, and length within the page that will be filled/emptied depending of the operation. After the splitting and issuing each "sub-I/O", the indirector waits for all sub-I/Os to complete before notifying the requester about the completion of the original I/O operation.

There is a special case while handling write requests that are not already mapped to ECD and that are not page-aligned. If EXCES buffers such writes in the ECD (as it does with other page-aligned writes), there will be inconsistency since a portion of the page will hold invalid data. For this special case, we let the request continue to disk.

5.3 Modularization and Consistency

EXCES utilizes the design of the block layer inside the Linux kernel to enable its operation as a dynamically loadable kernel module. Specifically, each instantiated block device registers a kernel function called `make_request` that is used to handle the requests to the device. EXCES is dynamically included in the I/O stack by substituting the `make_request` function of the disk device targeted for energy savings. This allows us to easily and directly modify any I/O requests before they are forwarded to the disk.

While module insertion is simple enough, module removal/unload must bear the additional responsibility of ensuring data consistency. Upon removal, EXCES must flush on-ECD dirty blocks to their original positions on disk. In EXCES, the I/O operations required to flush dirty pages upon module unload are handled using the reconfigurator through the *ECD to Disk* phase. In addition, EXCES must address race conditions caused when an application issues an I/O request to a page that is being flushed to disk at that exact instant. To handle such races, EXCES stalls (via `sleep`) the foreground I/O operation until the specific page(s) being flushed are committed to the disk. Since we expect module unload to be a rare event and the probability that a request for a page at the exact time it is being flushed to be low, the average response time for application I/O remains virtually not affected.

6 Evaluation

In our evaluation of the EXCES system, we answer the following questions in sequence: *(i)* What is an appropriate spin-down timeout for EXCES? *(ii)* How much energy does EXCES save? *(iii)* What is the impact on application I/O performance when EXCES is used? and *(iv)* what is the overhead of the EXCES system in terms of memory and computation?

To assess the above, we conducted experiments on two laptops, `shiriu` and `beer` (Table 2), both running Linux kernel 2.6.20. The experiments utilized the ECDs described

in Table 1. The hard drives on both laptops were running the Linux ext3 file system and the ECDs used ext2.

To quantify the system's power dissipation we used the battery information provided by ACPI, and took readings of the power level at 10 seconds intervals, the default ACPI update interval. The display brightness was reduced to its minimum visible level in all experiments.

For comparison purposes, in each experiment we set up various configurations including a default system with no optimizations, a system configured with the *laptop-mode* power saving solution [25], a system configured with EXCES, and a system configured with both laptop-mode and EXCES. Laptop-mode is a setting for the Linux kernel that forces much larger read-aheads (default 4MB) and holds off writes to the disk by buffering them in memory for a much longer time period. In all experiments, EXCES was configured to use an ECD miss rate threshold of 1000 misses-per-minute to trigger reconfiguration and a minimum duration of one minute between two reconfiguration operations.[3]

We used the BLTK (Linux Battery Life Tool Kit) [5] as our primary benchmark for system evaluation. This benchmark focuses specifically on laptop-specific workloads, targeted for evaluating battery life of laptop systems in realistic usage scenarios. Specifically, we use the BLTK Office and the BLTK Developer benchmarks in our experiments. Additionally, we use the Postmark [16] file system benchmark, designed to simulate small file workloads, typical of an email server. While we do not suggest the use of EXCES on the server-side (as yet), this benchmark allows us to evaluate the impact of I/O intensive workloads on external caching systems.

6.1 Choosing the Disk Spin-down Timeout

While researchers have suggested the benefits of using adaptive disk spin-down timeouts [10, 13, 17], the current version of EXCES uses a static disk spin-down timeout value. We used two benchmark workloads to determine the effect of the spin-down timeout on energy savings, using the `shiriu` system. We compared the system when configured with EXCES, laptop-mode [25], and a combination of EXCES with laptop-mode. We used the *ECD 2* configuration from Table 1 for this experiment. The BLTK Office benchmark, which automates the activities of opening and editing OpenOffice.org documents, spreadsheets, and drawings, was our first workload. The second workload used was the PostMark benchmark.

Figure 8 shows the results using timeout intervals of 5, 10, 15, 30 seconds and no timeout (∞ seconds). We used

[3]While we used these static values (based on preliminary experimentation) for simplicity, subsequent versions of EXCES will be able to dynamically adapt these thresholds based on application workload.

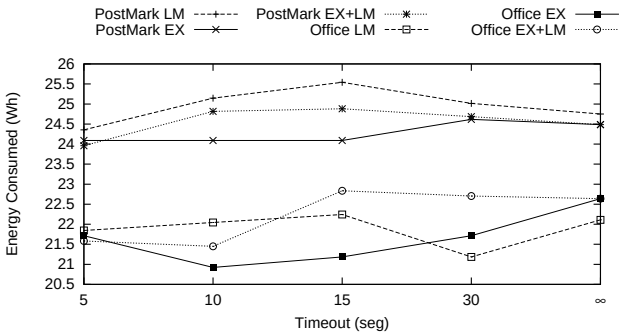


Figure 8. Effect of the disk spin-down timeout value on energy savings.

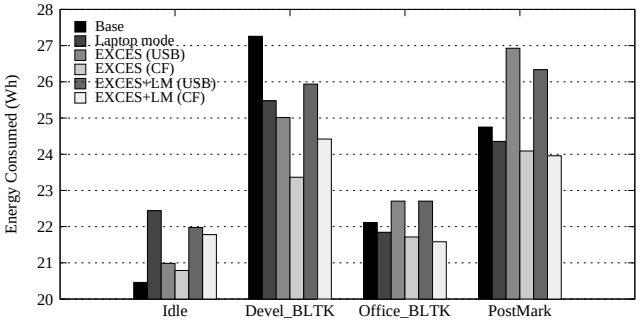


Figure 9. Power consumption with different workloads.

`hdparm` to set the timeout intervals in the disk's firmware, restricted to a minimum value of 5 seconds. A general trend observed when using EXCES (with and without laptop-mode) is that smaller timeout values allowed for greater energy savings, except for the BLTK Office workload which reaches its optimum at 10 seconds. All subsequent experiments use a 5 second disk spin-down timeout.

6.2 Energy Savings

To evaluate energy savings, we measure the power consumption with six different system configurations: the system "as is" with no power saving solution (Base), system running Laptop mode, EXCES using a USB and CF as ECD respectively, and EXCES with Laptop mode activated, using a USB and CF as ECD respectively. These experiments were conducted on the `shiriu` system, using configurations *ECD 1* for USB and *ECD 2* for CF (per Table 1).

We evaluated four different workloads: *(i)* an idle system, *(ii)* the BLTK Developer benchmark, *(iii)* the BLTK Office benchmark, and *(iv)* the Postmark benchmark. Figure 9 shows that on an idle system, all energy saving systems consume more power that the Base configuration. The

Name	Model	CPU	RAM	HD Specifications
shiriu	Dell E1505	Intel Core 2 @ 1.83 GHz	1 GB	Toshiba MK1234GSX (5400 RPM, 120 GB)
beer	Dell 600m	Intel Pentium M @ 1.6 GHz	512 MB	Western Digital WD400VE (5400 RPM, 40 GB)

Table 2. Specifications of the machines used in the experiments.

laptop-mode configuration uses additional power because of its aggressive prefetching mechanism, which ends up waking the disk for unnecessary data fetch operations. A similar behavior is observed when using EXCES in combination with laptop-mode. When EXCES is used by itself, since the disk is mostly spun-down anyway, any small disk energy savings is negated by the extra power consumed due to the ECD device itself.

The BLTK Developer benchmark performs a moderate amount of I/O. Its behavior mimics the operations of a developer who creates new files, edits files using the `vi` editor, and compiles the Linux kernel source tree, all of these interspersed with appropriate "human like" idle periods. We notice that moderate power savings can be obtained for all the power-optimized solutions. The configuration with EXCES alone provides the most power savings ($\sim$14% with CF and $\sim$8% with USB). The configurations that use laptop-mode deliver relatively lesser power savings; we attribute this to a fraction of the prefetching operations turning out to be ineffective.

For the BLTK Office benchmark, we note that there is no substantial energy saving in any of the configurations, and power consumption is somewhat increased when using a USB device as a ECD. We believe that this is due to the behavior of the benchmark which opens large executables from the OpenOffice suite, typically stored sequentially on the disk drive. These are subsequently cached in main memory, resulting in very few I/O operations after the executables have been loaded, reducing the opportunity for energy saving.

Finally, power consumption for the Postmark benchmark follows a similar trend to the BLTK Office. However, in this case, we believe the reasoning is different. Postmark is an I/O intensive workload, with a large fraction of sequential write operations. These sequentially written blocks to disk get absorbed as random writes in the ECD, owing to the current implementation of write buffering which does not attempt to sequentialize buffered writes. Random writes on flash-based storage are the least efficient, both in performance and power consumption [2]. We believe that a better implementation of write buffering in EXCES which improves the sequentiality of buffered writes, can result in better power savings for a write intensive workload.

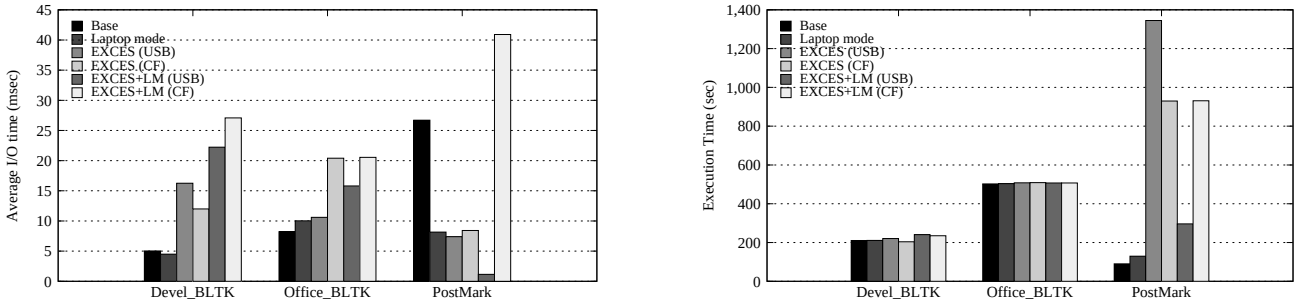
It is interesting to note that in almost all the cases using the USB as ECD makes the power saving system to consume more energy than the base case. On the other hand, using the CF leads to a better results for EXCES. Further, our findings point to a range of $\sim$2-14% for the cases when EXCES was indeed able to reduce power consumption, in somewhat of a contrast to earlier results from simulation studies [3, 8, 18] that predicted energy savings of $\sim$20-46% [4]. This difference is primarily because the power-consumption of the ECD was considered negligible and ignored in those studies.

6.3 Performance Impact of External Caching

While ECDs offer better performance than disk drives for random reads, they performs worse for other workloads. To evaluate performance, we focus on two metrics: *(i)* the average I/O (completion) time, and *(ii)* overall benchmark execution time. These provide complementary information and allow us insight into I/O performance. The average I/O time was obtained by using the Linux kernel tool *blktrace* [1]. The benchmark execution time was measured using the Bash *time* command. Each benchmark was run several times and the results averaged.

Figure 10 shows the results of these experiments. In both the BLTK benchmarks, the average I/O time increases substantially for the ECD based solutions, due to a large fraction of the I/O workload being sequential, allowing the disk drive to perform better. However, the increase in the overall benchmark execution time is negligible, due to substantial idle periods between I/O operations. This indicates that the impact to user perceived performance is minimal, thereby making the case for using external caching with these laptop-oriented workloads. On the other hand, an interesting anomaly is observed for the write intensive server-oriented Postmark benchmark. While the average I/O times with most of the ECD based solutions are lower, the total execution times are higher. We believe that this counter-intuitive result is because of writes being reported as completed by the ECD when they are written to the cache on the ECD but before they are actually committed to the flash medium. This reporting mechanism gives the false impression of fast individual write operations, when in reality the overall write I/O performance is severely throttled as the ECD commits these writes to the persistent flash with high latency.

[4] extrapolated to address whole system energy consumption



(a) Average I/O time

(b) Benchmark Execution time

Figure 10. Performance impact of EXCES with various workloads.

Pages/gigabyte	Dirty-bit array	Indirection map	Phase table	Top-k matrix	Page ranker
$2^{18} \cdot S$	$\frac{S}{2^5}$	S	$\frac{13 \cdot S}{2^2}$	$S + \frac{1}{2^{18}}$	$2^2 \cdot D$

Table 3. Size in megabytes for each EXCES data structure. S and D are the ECD and the disk sizes in GB respectively. The Phase Table is a temporary data structure used only during reconfiguration.

6.4 EXCES Resource Overhead

In EXCES, we paid special attention to how we used the scarce kernel-space memory. The memory usage for each of the EXCES's data structures is presented in the Table 3. The calculated sizes are for the worst case, i.e., when the ECD is completely filled of data. These formulas give us permanent memory usage of 0.1% and a temporary usage of 0.3% (during reconfiguration) relative to the ECD size. We believe these values are well-within acceptable limits.

To measure the CPU overhead due to EXCES, we used the following microbenchmark on the `beer` system. The microbenchmark issues a `grep` operation for a non-existent string on the EXCES source directory 100 times, that created a total of 21264 I/O operations. We divide the CPU overhead into two parts: the processing of the request before it is issued to the storage device (either ECD or disk), and the processing after the completion of the request. On an average, for each I/O operation, the corresponding numbers were 52 μs and 0.58 μs. Based on these, EXCES adds an average latency of 0.05 ms to the processing of each I/O request, which is relatively small compared to disk latency ($\geq$ 1 ms) and ECD latency ($\geq$ 0.5ms) [11]. While our current implementation of EXCES optimizes several operations, we believe that there is room for further improving this overhead time. Finally, we measured the reconfiguration overhead for the moderately I/O intensive BLTK developer benchmark. The average per-page reconfiguration time was measured to be 722 μs, an acceptable value for an infrequent operation.

7 Related Work

We classify research related to EXCES into three categories: energy-saving external caching techniques, energy-saving in-memory caching techniques, and other applications of external caching.

External caching for energy saving.
Early work on external caching was pioneered by Marsh *et. al* [18], who proposed incorporating an ECD as part of the memory stack between the disk and memory. They proposed that all I/O traffic to the disk drive be cached/buffered in the ECD before continuing on its normal path. This technique, while having the potential to reduce the number of disk accesses, does not effectively utilize the ECD space by choosing carefully what to cache/buffer. Much more recently, Chen *et. al* [8] also propose to use the ECD to buffer writes, as well as prefetch and cache popular data. Their solution divides the ECD into zones dedicated for each optimization, as opposed to the unified buffer/cache technique of EXCES. Additionally, since they propose using read-ahead values at the VFS layer to anticipate future accesses, their solution does not have a clear presence in the I/O stack, with both block- and file- level concerns. Similarly, Bisson and Brandt proposed NVCache, an external caching system for power savings [4]. While the design of EXCES has some similarities to both NVCache and SmartSaver, EXCES differs in its implementation-oriented techniques to efficiently ensure data consistency under all conditions, its use of a novel page-rank algorithm tailored for increasing disk inactivity periods, and continuous and timely reconfiguration capability. More importantly, while all of the above studies evaluate their techniques on sim-

ulated models of disk operation and power consumption, we evaluate an actual implementation of EXCES with real-world benchmarks that realistically demonstrate the extent of power-savings as well as impact to application performance.

In-memory caching for energy saving.
Weissel *et. al* [22] and Papathanasiou *et. al* [28] propose to use cooperation/hints between the applications and the operating system. While Weissel *et al.* propose hints at the system call API for read/write operations, Papathanasiou propose using high-level hints about application I/O semantics such as sequentiality/randomness of access inside the operating system. Researchers have also looked at adaptive disk spin-down policies to complement in-memory caching techniques [10, 13, 17]. We believe that all of the above can complement EXCES to further improve energy savings. Specifically, in this study, we compared EXCES against the open-source Laptop-mode tool [25], and demonstrate that the Laptop-mode techniques complement EXCES well for some workloads to improve energy savings.

Other applications of external caching.
External caching has been used to improve I/O performance and reliability. Researchers have long argued for utilizing battery-backed caching (providing similar functionality as an ECD) for improving both reliability and performance [21]. Wang *et al.* [27] suggest using a Disk-ECD hybrid file system for improving application I/O performance by partitioning file system data into two portions, one stored on disk and the other on an ECD. More recently, the ReadyBoost [19] feature in the Windows Vista operating system utilizes an ECD if available to cache data. Since its primary objective is performance improvement, ReadyBoost directs small random read requests to the ECD and all other operations to the disk drive.

8 Conclusions and Future Work

EXCES is an external caching system that reduces system power consumption by prefetching, caching, and buffering disk data on a less power consuming, persistent, external caching device. While external caching systems have been proposed in the past, EXCES is the first implementation and evaluation of such a system. We conducted a systematic evaluation of the EXCES system to determine overall energy savings and the impact on application performance. EXCES delivered overall system energy savings in the modest range of ∼2-14% across the BLTK and Postmark benchmarks. Further, we demonstrated that external caching systems can substantially impact application performance, especially for a write-intensive workload.

We believe that external caching systems offer a new direction for building energy saving storage systems. Improvements in ECD technology, especially in the performance dimension, can help accelerate the adoption of such systems. Our future work on EXCES will be directed towards the performance-sensitive server environment, where, in the absence of a display device, disk-drives would be the second highest power consuming component. Optimizations that address random write performance on the ECD will gain significant importance in such systems.

EXCES Software

The EXCES source code is available for download at: `http://dsrl.cs.fiu.edu/projects/exces/`

Acknowledgments

We thank the anonymous reviewers for their excellent feedback that helped us immensely in positioning and presenting our work, and specifically for recommending that we view EXCES results in a more positive light. We would also like to thank Zoran Dimitrijevič for early discussions on EXCES, and David Delgado, Ian De Felipe, and Dayanara Hernandez, for their initial investigations on EXCES feasibility.

This work was supported by the National Science Foundation grant IIS-0534530 and the Department of Energy grant DE-FG02-06ER25739. However, the views expressed in this paper do not necessarily reflect those of the above agencies.

References

[1] J. Axboe, A. D. Brunelle, and Others. blktrace user guide, February 2007.

[2] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, 2007.

[3] T. Bisson and S. A. Brandt. Reducing energy consumption with a non-volatile storage cache. In *International Workshop on Software Support for Portable Storage (IWSSPS)*, March 2005.

[4] T. Bisson, S. A. Brandt, and D. D. E. Long. Nvcache: Increasing the effectiveness of disk spin-down algorithms with caching. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 422–432, September 2006.

[5] L. Brown, K. A. Karasyov, V. P. Lebedev, A. Y. Starikovskiy, and R. P. Stanley. Linux laptop battery life: Measurement tools, techniques, and results, February 2007.

[6] D. Brownell. Linux usb "On-The-Go" (OTG) on OMAP H2, 2004.

[7] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th*

annual international conference on Supercomputing, pages 86–97, 2003.

[8] F. Chen, S. Jiang, and X. Zhang. Smartsaver: Turning flash drive into a disk energy saver for mobile computers. In *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*, pages 412–417, New York, NY, USA, 2006. ACM Press.

[9] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, 2002.

[10] F. Douglis, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, 1995.

[11] J. Gray and B. Fitzgerald. Flash disk opportunity for server-applications. *Online,, http://research.microsoft.com/∼Gray/papers/ FlashDiskPublic.doc*, January 2007.

[12] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DPRM: Dynamic speed control for power management in server class disks. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'03)*, June 200.

[13] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking*, pages 130–142, 1996.

[14] IBM. IBM Attempts to Reinvent Memory. Online, http://www.technologyreview.com/ Nanotech/19477/page1/.

[15] Intel Corporation. Intel® mobile platform vision guide for 2003, 2002.

[16] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

[17] K. Li, R. Kumpf, P. Horton, and T. E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the USENIX Winter Conference*, 1994.

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

[19] Microsoft Corporation. Windows Readyboost. Online, http://www.microsoft.com/windows/products/windowsvista/features/details/readyboost.mspx.

[20] W. D. Norcott and D. Capps. The Iozone File System Benchmark. http://www.iozone.org/.

[21] J. K. Ousterhout and F. Douglis. Beating the i/o bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, 1989.

[22] A. E. Papathanasiou and M. L. Scott. Energy efficient prefetching and caching. In *Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[23] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th annual International Conference on Supercomputing*, pages 68–78, 2004.

[24] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.

[25] B. Samwel. Kernel korner: extending battery life with laptop mode. *Linux J.*, 2004(125):10, 2004.

[26] Technical Review. Higher-Capacity Flash Memory. Online, http://www.technologyreview.com/.

[27] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H.Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[28] A. Weissel, B. Beutel, and F. Bellosa. Cooperative i/o - a novel i/o semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[29] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, October 2005.

[30] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing energy consumption of disk storage using power-aware cache management. In *The 10th International Conference on High-Performance Computer Architecture (HPCA-10)*, pages 118–129, February 2004.