Flash on Disk for Low-Power Multimedia Computing

Leo Singleton, Ripal Nathuji, Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{lcs,rnathuji,schwan}@cc.gatech.edu

ABSTRACT

Mobile multimedia computers require large amounts of data storage, yet must consume low power in order to prolong battery life. Solid-state storage offers low power consumption, but its capacity is an order of magnitude smaller than the hard disks needed for high-resolution photos and digital video. In order to create a device with the space of a hard drive, yet the low power consumption of solid-state storage, hardware manufacturers have proposed using flash memory as a write buffer on mobile systems. This paper evaluates the power savings of such an approach and also considers other possible flash allocation algorithms, using both hardware- and software-level flash management. Its contributions also include a set of typical multimedia-rich workloads for mobile systems and power models based upon current disk and flash technology. Based on these workloads, we demonstrate an average power savings of 267 mW (53% of disk power) using hardware-only approaches. Next, we propose another algorithm, termed Energy-efficient Virtual Storage using Application-Level Framing (EVS-ALF), which uses both hardware and software for power management. By collecting information from the applications and using this metadata to perform intelligent flash allocation and prefetching, EVS-ALF achieves an average power savings of 307 mW (61%), another 8% improvement over hardware-only techniques.

1. INTRODUCTION

Modern mobile systems, such as laptops and handhelds, require large amounts of storage—particularly for multimedia-rich applications such as high-resolution digital photos and streaming video. Unfortunately, hard disks, the only technology currently available with the storage capacity required for multimedia systems, consumes a substantial amount of power, reducing the battery life of mobile systems. Solid-state storage, such as flash memory, offers the potential for very low-power storage, but its capacity is still an order of magnitude smaller than what is needed for multimedia systems.

Recently, hardware manufacturers, including Samsung[1] and Intel,[2] have announced the development flash memory to be integrated into the hard disk or platform to act as a write buffer for the hard disk. This technology aims to offer significant energy savings, by allowing write requests to be written to flash memory whenever the disk is spun down, eliminating the need to spin up the disk on a write. Unlike buffering writes in RAM, flash memory is persistent, so in the event of a system crash or power failure, data can be recovered from flash.

This paper makes two contributions to evaluating the power savings of flash memory used in conjunction with disk storage. First, it describes the models and workloads used to gauge the power savings of flash memory on a hard disk. Most previous research on disks has been focused on the performance of server disks, so new workloads had to be created to simulate a modern, multimedia-rich laptop or embedded system. Second, it quantifies the energy savings of flash on a hard disk using current disk and flash technologies. We consider the previously proposed idea from Samsung – using flash as a write buffer – along with two additional flash allocation algorithms: using flash as a cache with LRU replacement and an algorithm we have developed called EVS-ALF (Energy-efficient Virtual Storage using Application-Level Framing). EVS-ALF is a hardware/software approach which uses application-level knowledge passed down to the operating system to use flash for both buffering writes and prefetching reads. Additional improvements to methods like ours would be gained by operating system or application changes, to change operating system schedulers and/or application behavior to further reduce the total number of disk spinups/spindowns.[3, 4]

2. RELATED WORK

Much of the previous work on power conservation of multimedia systems has focused on either network[5] or CPU[6] power consumption. Research into disks in multimedia systems has mostly focused on the performance of multimedia server systems such as Yima, a streaming video server.[7]

Disk write buffering was proposed as a performance enhancement for server systems over a decade ago by Hu and Yang with their Disk Caching Disk architecture.[8] DCD proposed the use of a small, fast disk, to buffer write requests and improve disk performance. Ideally, this buffer could be made of non-volatile RAM, but at the time, it was too small and expensive to be feasible for desktop use.

Other researchers, including,[9–11] have proposed non-volatile write buffers for network file systems. In 1992, Baker et al. quantified the amount of buffer space needed given workloads at the time, showing that a one megabyte write buffer could eliminate more than 50% of client write traffic.[9] Ruemmler and Wilkes traces showed a 65% to 90% improvement with only 200 KB, while 95% of write requests could be contained in 4 MB or less.

However, since the time of this initial work, flash memory has grown far larger and become far less expensive. In addition, laptops and other mobile systems have gained popularity, shifting much of the previous focus on performance to the current focus of energy-efficiency. Recently, Samsung announced the development of a laptop hard drive containing a 128 MB flash write buffer[1] – dozens of times larger than the capacities considered in previous research. In addition, this flash is designed to conserve energy by keeping the hard disk spun down – using the flash as temporary storage when the disk is not spinning. Meanwhile, Intel[2] has announced a similar technology, codenamed Robson, which places the flash at the platform level for disk buffering.

3. MODELS AND ASSUMPTIONS

In order to evaluate flash memory as a power-savings mechanism for hard disks, we constructed power models based upon current disk and flash technology. We selected datasheets from the Hitachi Travelstar hard disk and Samsung flash memory for our models.

Disk Model. To model disk power, we use the datasheet parameters from the Hitachi Travelstar C4K40 1.8-inch hard disk drive.[12] A summary of the relevant parameters is given in Table 1.

Symbol	Description	Value	Units
t_{SEEK}	Avg. Seek Time	15.0	ms
P_{SEEK}	Seek Power	1.70	W
P_{UP}	Spin-Up Power	2.25	W
P_{IDLE}	Idle Power	0.50	W
P_{STANDBY}	Standby Power	0.15	W

Table 1. Datasheet specifications of the Hitachi Travelstar C4K40 1.8-inch hard disk.

Using the values from Table 1, we estimate the costs of handling a disk request, spinning up, and spinning down the hard drive*:

$$E_{\text{REQUEST}} = t_{\text{SEEK}} P_{\text{SEEK}} = 25.5 \text{ mJ}$$
$$E_{\text{UP}} = t_{\text{UP}} P_{\text{UP}} = 6.75 \text{ J}$$
$$E_{\text{DOWN}} = t_{\text{UP}} P_{\text{IDLE}} = 1.50 \text{ J}$$

For each request, we assume the average-case seek time. Thus, the actual seek power may be more or less than the average, depending on the workload. However, our initial results showed that seek power was not a

*The datasheet does not provide values for spindown time or power, therefore E_{DOWN} is modeled assuming that spindown time is approximately the same as the spinup time, and the spindown power is approximately the same as the idle power.

large factor in the energy consumption of our mobile workloads—idle, standby, and spin-ups are an order of magnitude greater. Thus, we simply model seeks using the average case and in the experimental results, seek power is shown under I/O power.

Flash Memory Model. To estimate the power consumption of the flash memory, we selected parameters from the K9K4G08U0M flash memory chip from Samsung.[13] The relevant power specifications are listed in Table 2.

Symbol	Description	Value	Units
t_{PROGRAM}	Program Time	200	μs
t_{ERASE}	Erase Time	2	ms
t_{READ}	Read Time	25	μs
s_{PAGE}	Page Size	2,048	bytes
I_{IO}	Read/Program/Erase Current	15	mA
V_{CC}	Supply Voltage	3.3	V

Table 2. Datasheet specifications of Samsung K9K4G08U0M flash memory.

Using the parameters in Table 2, we use the following estimate for average power to read and write a 4 KB page from flash. Note that the flash memory chip uses a page size of 2 KB, therefore each 4 KB request requires 2 pages of flash.

$$E_{\text{READ}} = (t_{\text{READ}})(I_{\text{IO}})(V_{\text{CC}})(2 \text{ pages}) = 2.48 \ \mu J$$
$$E_{\text{WRITE}} = (t_{\text{ERASE}} + t_{\text{PROGRAM}})(I_{\text{IO}})(V_{\text{CC}})(2 \text{ pages}) = 218 \ \mu J$$

Optimal Spin-Down. Many algorithms have been proposed to efficiently spin down hard disks when idle. An accurate spin-down algorithm is crucial to energy savings using a non-volatile write buffer. However, since such a large body of previous work exists and since most existing algorithms have parameters that must be tuned for specific systems and workloads, we simply assume an optimal spindown algorithm. This algorithm spins the disk down whenever the energy consumed during an idle period exceeds the cost of spinning a disk down and back up.

$$E_{\text{IDLE}} \geq E_{\text{UP}} + E_{\text{STANDBY}} + E_{\text{DOWN}}$$
$$t_{\text{IDLE}} P_{\text{IDLE}} \geq t_{\text{UP}} P_{\text{UP}} + (t_{\text{IDLE}} - 2t_{\text{UP}}) P_{\text{STANDBY}} + t_{\text{UP}} P_{\text{IDLE}}$$
$$t_{\text{IDLE}} \geq 35 \text{ s}$$

For the Hitachi Travelstar C4K40 described in Table 1, optimal spindown occurs during any idle period of 35 seconds or greater.

4. WORKLOADS

Much of the past work related to hard drive energy and performance has utilized disk traces collected by Mummert et al.[14] at Carnegie Mellon University between February 1991 and March 1993. However, these filesystem-level traces collected from UNIX workstations are not representative of the access patterns found on modern, multimedia-rich personal or portable computers. In particular, Hsu and Smith[15] showed that the read/write ratios vary greatly between the filesystem versus device levels. That is, high-level traces, such as network filesystems show primarily reads, whereas low-level block traces show more writes than reads. This difference can be accounted for by the operating system's caching and buffering, and by journaling filesystems, which generate a large amount of write traffic. Finally, multimedia workloads tend to have more sequential accesses, and they use larger blocks of data.

Due to these limitations, the first contribution of this paper is the development of synthetic workloads that emulate two common interactive applications used on personal computers: web browsing and streaming media.

These workloads were run for extensive amounts of time, and trace files were collected. Trace files contain the following fields: request time (in microseconds), type (read/write), block number, number of blocks, file path (if available), and file offset (see `http://www.cc.gatech.edu/~lcs/disktraces/` for access to and further documentation about these traces).

Workloads were executed on a Red Hat Enterprise Linux system with a 2.6 kernel and the ext3 filesystem. The hardware consisted of a 550 MHz Intel Pentium III processor, with 1 gigabyte of RAM. The workloads described in the next two sections were run on the actual system, and trace files of the workloads were collected by augmenting the Linux kernel's block device code to capture every disk I/O request.

4.1. Web Browsing / Photo Gallery

Since a common use of laptops and mobile systems is browsing the World Wide Web, this is an important scenario to consider for optimizing energy consumption. In the case of storage, online photo galleries are a particularly I/O-intensive workload, since photographs from digital cameras are typically a few megabytes in size.

To construct a web browsing workload, 169 images were downloaded from NASA's Mars Pathfinder website. These high-resolution JPEGs consisted of a total of 535 MB of data, which was copied to a local web server. A breakdown of files by size is given in Table 3.

Size	Files	% of Total
< 128 KB	6	3.6%
128 KB - 256 KB	11	6.5%
256 KB - 512 KB	7	4.1%
512 KB - 1 MB	5	3.0%
1 MB - 2 MB	24	14.2%
2 MB - 4 MB	64	37.9%
4 MB - 8 MB	43	25.4%
8 MB - 16 MB	9	5.3%

Table 3. File distribution by size of the web browsing workload.

Files were accessed by using a normal distribution for access interarrival times. The normal distribution was generated with a mean of 60 seconds and a standard deviation of 30 seconds. The JPEGs were fetched using a JavaScript running in Mozilla Firefox. Firefox's cache size was set to 600 MB, to guarantee that all files could fit inside of the disk cache, but the cache was flushed before each experiment. Flushing Firefox's cache deletes the cache files on the filesystem, which also frees Linux's buffer cache entries for the data.

The order of the image accesses is another variable in the web browsing workload. Four usage models were considered:

- *Randomsort* – the list of images was randomly sorted. There are three random sort cases—*randomsort1*, *randomsort2*, and *randomsort3*—corresponding to the number of times each image was included the list before performing the random sort.
- *Randomwalk* – a random walk behaves like a web site with next and previous buttons for navigation. The visitor randomly moves to the next image 50% of the time and to the previous image the other 50%.
- *Paginated* – in the paginated usage model, all of the images are grouped into pages of 10 images each. The pages are randomly selected, and each image on the page is viewed before the user randomly selects another page. Each page is viewed 3 times.
- *Sequential* – the visitor views all images sequentially, repeated until each image has been viewed 3 times.

4.2. Streaming Media

Another common use of mobile systems is for portable media. From music players to personal video recorders, an increasing number of streaming media applications are available. For this usage model, we consider the recording and playback of a one-hour piece of streaming media.

To simulate realistic streaming media workloads, two different variables were considered: the bitrate and the time between recording and playback. The bitrate determines the volume of data stored – a listing of the bitrates selected is described in Table 4. However, the time between recording and playback determines how effectively this data can be cached. In some cases, such as watching streaming video over the Internet, this time is extremely short – typically a few seconds of buffering. In other applications, such as a personal video recorder, the time between recording and playback may be on the order of minutes – a user may be watching the beginning of a television show as the end of the show is still recording. And in other applications, the time between recording and playback can be days, such as a portable music player, where music is downloaded to the player once and is played back at a later time. In order to account for these differences, three use cases are considered:

- 5 minute offset – example: Internet Audio/Video.
- 30 minute offset – example: Personal Video Recorder.
- 60 minute offset – example: Music Player. Since the media is only one hour in length, this test case covers all usage scenarios where the media file is recorded at one time, and played back at a later time.

Usage Model	Bitrate	Typical Uses
sm1	128 kbps	Standard Streaming Audio
sm2	256 kbps	High-Quality Streaming Audio
sm3	1.5 Mbps	MPEG-1 Video
sm4	6 Mbps	MPEG-2 Video (DVD Quality)

Table 4. Bitrates selected for Streaming Media usage models.

4.3. Memory Pressure

While the workloads described in the previous two sections simulate common uses of mobile computer systems, a modern desktop operating system context switches between multiple processes, and these background processes consume resources that would otherwise be available to the foreground application.

Modern operating systems use much of their free RAM to cache data on disk – in Linux, this is referred to as the buffer cache. The cache optimizes disk performance, since if this data is later needed, it can be retrieved from RAM, which is much faster than fetching the data from the disk. This typically results in block level accesses being mostly write-based on a personal computer, since many reads can be handled by the operating system's disk cache.[15]

Due to this caching in the operating system, the disk access patterns are not only dependent on the workload, but also on the system's available memory. On personal computers, background tasks such as antivirus software and automatic updates often consume significant amounts of memory, taking away from the memory available to the interactive tasks and the disk cache. This typically results in more disk I/O, since less space can be devoted to the disk cache, and in the case of extreme memory pressure, applications' memory may be swapped to disk. This change results in more disk requests, and a shift towards more reads from disk.

In order to address this variation, each of the workloads were tested with three degrees of memory pressure: 0, 256, and 512 MB of RAM (out of 1 GB) were consumed by a process running in the background during trace file collection.

Memory Pressure:		0 MB		256 MB		512 MB	
Workload	Length	Req/Min	R/W Ratio	Req/Min	R/W Ratio	Req/Min	R/W Ratio
www-randomsort1	169	838	0.000	840	0.001	835	0.001
www-randomsort2	338	425	0.000	572	0.341	729	0.692
www-randomsort3	507	289	0.003	412	0.435	653	1.246
www-randomwalk	169	106	0.000	109	0.005	104	0.002
www-paginated	507	300	0.011	393	0.341	615	1.070
www-sequential	507	537	0.658	462	0.547	718	1.390
sm1-5min	65	314	0.000	313	0.000	315	0.000
sm1-30min	90	247	0.003	236	0.000	237	0.001
sm1-60min	120	191	0.001	186	0.000	187	0.001
sm2-5min	65	707	0.247	536	0.000	540	0.001
sm2-30min	90	397	0.000	397	0.000	399	0.000
sm2-60min	120	306	0.000	306	0.000	307	0.000
sm3-5min	65	2271	0.001	2288	0.008	2246	0.006
sm3-30min	90	1670	0.013	1855	0.121	2143	0.299
sm3-60min	120	1263	0.014	1310	0.050	2179	0.749
sm4-5min	65	9684	0.184	10921	0.331	14315	0.750
sm4-30min	90	11475	0.932	11738	0.985	11753	0.987
sm4-60min	120	9065	1.000	8828	0.984	8827	0.984

Table 5. Workload characteristics. Each workload is shown with the length of the trace file (in minutes), the average requests per minute, and the ratio of read/write requests for each of the three degrees of memory pressure.

4.4. Workload Characteristics

Table 5 summarizes the characteristics of the synthetic workloads. These statistics were collected from the trace files used later in the experimental results.

The data shows multiple important trends. First, the effects of memory pressure are evident in the request rate and read/write ratios. As memory pressure on the system increases, fewer blocks can be buffered in RAM, and some memory pages may be swapped to disk. In this case, the amount of disk traffic increases, and the disk traffic becomes more read-based. Second, the longer trace files generally have less disk I/O. Again, this is due to the OS's buffering of data in RAM – trace files that repeat their patterns, such as *www-randomsort3*, have much of the data they need buffered in RAM by the third repetition. Finally, the statistics show a wide variance in workloads – some traces, such as *www-randomsort1*, have a read/write ratio of zero, indicating nearly all writes, whereas other traces have ratios over 1, indicating more than 50% of disk requests were reads.

5. HARDWARE-LEVEL FLASH MANAGEMENT

The first two flash allocation algorithms we consider could be implemented entirely at the hardware-level, where firmware on the hard disk could completely manage the allocation and deallocation of blocks on the flash. To the operating system, the flash and disk then appear as a single storage device. We analyze two different algorithms for flash block allocation: (1) Write Buffer and (2) LRU Cache, which are described in the next two sections.

5.1. Write Buffer

The Write Buffer allocation algorithm uses the flash only to cache dirty blocks, before they are written to disk. This is advantageous, since the disk can remain spun down while applications perform write requests – the disk is only accessed on read requests, or when the write buffer becomes full and dirty blocks must be flushed to disk.

To some extent, the write buffer is already implemented at the operating system-level. In UNIX operating systems, writes are typically buffered in RAM, and the update daemon runs every 30 seconds to flush dirty data to disk. The 30-second flush is chosen to minimize the amount of data loss in the event of a system crash.

However, with flash, the flush time can be indefinite – in the event of a power outage or system crash, all of the data is persistent, since it was buffered in flash memory, and can be flushed to disk on the next system boot.

In the Write Buffer algorithm, we want to minimize the number of spin-ups/spin-downs for the disk, since those consume a significant amount of power. Therefore, whenever the disk spins up to handle a read request, we flush all dirty blocks in the write buffer to disk. This virtually eliminates the need to spin up the disk due to the write buffer becoming full, since this could only happen if there are 128 MB of write requests without a single read.

5.2. LRU Cache

The Write Buffer algorithm only handles write requests, and it requires the disk to spin up on every read. We wish to utilize the flash to also reduce the number of reads. Therefore, we implement a flash management algorithm that treats flash like a memory cache. The cache is a 128 MB, fully-associative read/write cache, with an LRU replacement algorithm. However, using a standard write-back or write-through replacement policy would result in unnecessary spin-ups. Therefore, at each spin-up, we flush enough dirty data to keep a reserve of free blocks. The reserve was selected to be 25% of the flash size.

6. ENERGY-EFFICIENT VIRTUAL STORAGE USING APPLICATION-LEVEL FRAMING (EVS-ALF)

The initial experiments from the hardware-level approaches (the results of which are described later in Section 7) show that in many cases, the simple hardware approaches perform non-optimally. The LRU cache does an extremely poor job of handling read requests, so that there are still substantial gains to be achieved by a better algorithm for flash allocation. In particular, power gains can be achieved by accurately prefetching data into the flash before applications need it. In this section, we describe the EVS-ALF (Energy-efficient Virtual Storage using Application-Level Framing) algorithm we have developed. This algorithm consists of two layers: the EVS layer, which is an low-level driver for managing flash blocks, and the ALF layer, a kernel component which collects Application-Level Framing metadata for power management.

6.1. Application-Level Framing Layer

Applications have a wealth of data about the contents of files and associations between files. As a result, often, they can more effectively prefetch data than the hardware or operating system. For instance, a web browser knows which pages are linked to the ones currently being browsed and can speculatively download content. Media players know the bitrates of the media files they can play, and if the media player knew the access times and speeds of the underlying I/O devices, the application could accurately predict how much data needs to be prefetched.

However, OS design abstracts physical devices from the applications, so that applications do not have to be device-specific. Rather than advocating against this level of abstraction and moving power management decisions to the applications, we propose that applications share certain metadata about file contents and associations with the operating system. Specifically, we add an Application-Level Framing (ALF) layer, which collects and stores this data, so that the OS can then use this information for power management and performance purposes.

Per-Process Metadata. The first piece of metadata provided to the OS is a per-process data structure. Passed from the application to the kernel via a system call, this structure contains two values: the expected future read (r_{READ}) and write (r_{WRITE}) rates of the application. For streaming media applications, these values are known with reasonable certainty. For interactive applications, r_{READ} and r_{WRITE} vary greatly depending on user behavior, so we instead estimate these values based on past activity.

File Relationships. The other important metadata concerns the relationships between different files. Many applications have well-defined file access patterns that are only known to the application, not the OS. For instance, a media player may have a predefined playlist. A web browser knows which images in the cache are associated with which web pages, and web pages are associated with each other via hyperlinks. Likewise, an image browser or e-mail application groups items into folders or pages, which constitutes another common file relationship.

Passing these file relationships to the kernel can greatly increase the accuracy of prefetching, which is beneficial for power management.

We store file relationships as undirected graphs – one graph (G_t) per relationship type (t). For example, for the image browser application, we create two relationship types – one for when images are viewed on a per-page screen and another for when the user shows a slideshow and views the images sequentially.

File relationships are created via an AddFileRelationship system call containing two parameters: the relationship type (t) and a set of related inodes (I). When the call is invoked, it adds undirected edges between each inode $i \in I$ to graph G_t.

Consider the aforementioned example of an image browser with two relationship types. Assume we have 5 images (with inode numbers 1 through 5) and 3 images per page in the application. And assume two relationship types: paginated access ($t = 0$) and sequential access ($t = 1$). To provide ALF relationship metadata, the application would make the following system calls to create the paginated graph:

$$\text{AddFileRelationship}(0, \{1, 2, 3\})$$

$$\text{AddFileRelationship}(0, \{4, 5\})$$

And to create the sequential graph:

$$\text{AddFileRelationship}(1, \{1, 2\})$$

$$\text{AddFileRelationship}(1, \{2, 3\})$$

$$\text{AddFileRelationship}(1, \{3, 4\})$$

$$\text{AddFileRelationship}(1, \{4, 5\})$$

Figures 1 and 2 show the resulting graphs.

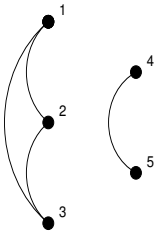

Figure 1. Graph representing paginated file access (G_0).

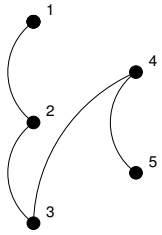

Figure 2. Graph representing sequential file access (G_1).

6.2. Energy-Efficient Virtual Storage Layer

Using the ALF metadata described in the previous section, we create a virtual block device. This layer, the Energy-efficient Virtual Storage (EVS) layer, intelligently utilizes flash as a disk cache based upon the metadata provided by the applications.

Block States. For each block of flash, the EVS algorithm assigns it to one of three states: CLEAN, DIRTY, and RESERVED. CLEAN and DIRTY are self-explanatory. RESERVED however, indicates that the block is clean *and* that the block contains speculatively prefetched data. Figure 3 shows the states and possible transitions for a flash block.

Block Replacement Algorithm. When a write request gets directed to flash or when EVS prefetches blocks, the EVS block replacement algorithm is invoked to select which block of flash to replace. The replacement policy selects blocks in the following order:

1. Check for blocks in flash which already have the mapping we need.
2. Replace CLEAN blocks. Select CLEAN blocks using a FIFO queue.

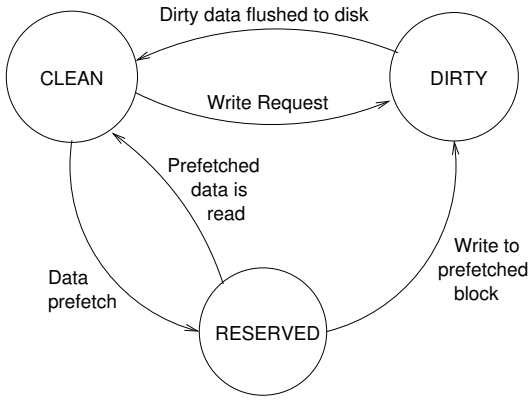

Figure 3. States and transitions for flash blocks under EVS.

3. Replace RESERVED blocks. Select RESERVED blocks using a FIFO queue.

4. We cannot replace DIRTY blocks, since they contain the most current copy of our data. If there are no CLEAN or RESERVED blocks remaining, spin up the disk, and flush all DIRTY blocks to disk, marking them CLEAN. Then select a CLEAN block.

The order of block selection is fairly intuitive. We only allow one disk block to be mapped to a single flash block, so that we always reuse existing mappings when they are available. Next, we use CLEAN before RESERVED, because by definition, RESERVED blocks contain data which our ALF layer has predicted will be read in the near future. Within the CLEAN and RESERVED groups, we select blocks using First-In-First-Out. Thus, the blocks that have gone unused for the longest period of time will be replaced first. And finally, only when the entire flash is filled with DIRTY blocks, will we spin up the disk.

Data Structures. The EVS implementation maintains the following data structures:

- **Flash-to-Disk Mapping.** Each block of flash is always mapped to one unique disk block. These mappings, along with the states of the flash blocks are kept in an array in kernel memory.

- **Disk-to-Flash Mapping.** To do a reverse lookup of the previous mapping, we keep a hash table of disk-to-flash mappings.

- **Queues by State.** For each of the three block states, a FIFO queue is kept. This is used in the implementation of the aforementioned replacement policy.

Prefetching Using ALF Metadata. Using the above algorithm, there are only two cases when the disk must be used: when the flash is completely filled with DIRTY blocks, and must be flushed to disk, or when a read request is issued for a block which is not cached in flash. Since our ultimate goal is to minimize the number of spin-ups, we take advantage of these two cases to flush dirty data to disk, and prefetch new data. The goal is to avoid future disk spin-ups.

The prefetching algorithm works as follows:

1. First, we must determine how many blocks to prefetch. Prefetching too little will result in a premature disk spin-up, increasing energy consumption. Prefetching too much will also increase energy consumption, because the prefetched data in flash may have to be overwritten with dirty data before it is used. To estimate the optimal number of blocks to prefetch (P), we use the ALF data from the current applications:

$$P = \min\left(\frac{\Sigma r_{\text{READ}}}{\Sigma r_{\text{WRITE}}}, 0.95\right) \cdot S$$

The formula uses the sum of the read and write rates of all currently-running applications. This gives us an estimate of the system's future behavior. We use the ratio of read to write rates to determine what fraction of the flash to use for prefetching. For instance, if the read and write rates are equal, we use as much of the flash as possible for prefetching, since as prefetched (RESERVED) blocks are read, they become marked as CLEAN, and can be used by write requests to hold dirty data. Since the read and write rates are only estimates, we cap the amount of prefetching to 95% of the flash, so that there will always be some CLEAN blocks available for write requests.

2. Next, we must select P blocks from disk to prefetch. We first consider the files which are currently open for reading by applications providing ALF data. If any of these files have only been partially read, we first prefetch any remaining blocks in these files.

3. If there are not P blocks remaining in the currently-open files, we use the ALF relationship graphs to predict files which are likely to be read in the near future. However, to do so, we must predict which access pattern each of the applications is following, if any. This can be a challenge, since the device level does not receive all file accesses – some accesses may be handled by the buffer cache, and never passed on to the lower levels. Thus, we probabilistically choose the closest relationship type based upon previous accesses.

   The access pattern prediction algorithm works by keeping a counter for each relationship graph. When an I/O request is received, the ALF layer compares its inode number (i_1) to the inode number of the previous I/O request (i_2). If $i_1 \neq i_2$, then we search all relationship graphs for an edge between i_1 and i_2. Graphs containing such an edge will have their counter incremented. Thus, the "correct" relationship type will have a higher counter value than the others.

4. Using the access pattern prediction to predict the relationship type t, we consider the graph G_t, and perform a breadth-first traversal beginning at the last accessed file. This gives us a sequence of files, prioritized from most- to least-likely to be accessed. We then select the blocks corresponding to these files for prefetching, until P blocks have been prefetched.

   Note that this step is very computationally intensive, and not feasible to compute during each disk spin-up. In our prototype implementation, we precompute the predictions each time an application modifies its ALF relationships—this is a fairly rare event, and takes the computation off of the critical path. In a real implementation, we speculate that these predictions could be made on a per-application basis. This would keep the size of the graphs small enough to be computationally feasible, and would allow advanced applications to override the default prediction algorithm with their own predicted sequences.

7. EVALUATION

To measure the power savings of the hardware-level and ALF approaches of flash management, the algorithms were implemented and then simulated on each of the trace files from Section 4. The resulting trace output from each of the three algorithms was run through the disk and flash power models described in Section 3. The results are shown in Figures 4 through 7. In each bar graph, the bars are clustered by input trace file and show, from left to right: the power consumed without flash (No), the flash used as a write buffer (Wr), the LRU cache (LRU), and the EVS-ALF algorithm (ALF).

The results of the low-bitrate streaming media test cases have been omitted due to space constraints. These workloads are almost entirely write based, since the media file fits well within the OS's buffer cache, and thus show negligible difference between the three flash allocation algorithms. However, all cases show a nearly 70% savings in power by using a 128 MB flash. Since the workload is mainly writes, which can be buffered in flash, the disk is spun down and remains in standby mode, achieving a large power savings.

In the higher-bitrate streaming media cases, the workload becomes more challenging as the amount of data no longer fits in RAM, offering a mix of reads and writes. Thus, the simple hardware approaches to flash management are no longer optimal. Using a flash as a write buffer works well during the write-based portions of the traces, but works poorly during periods of reads. LRU is generally ineffective with streaming media accesses, and in some cases performs worse than the original trace itself. This is due to buffer cache in RAM removing

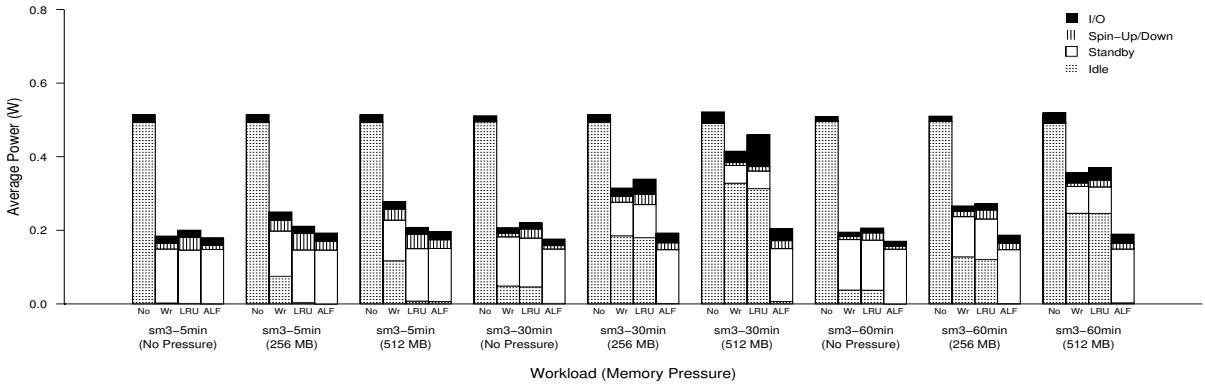

Figure 4. 1.5 Mbps Streaming Media

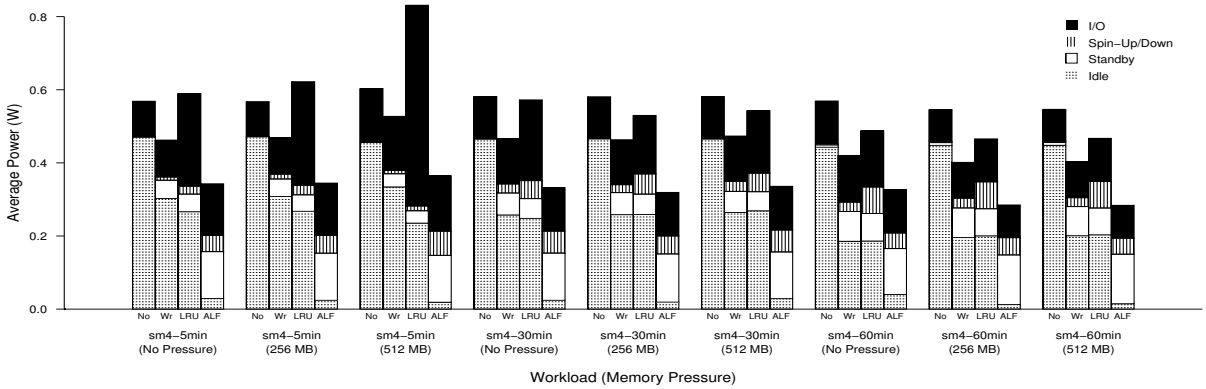

Figure 5. 6 Mbps Streaming Media

most of the temporal locality from the disk accesses. Thus, LRU results in extra I/O operations without adding any additional benefit.

In some trace files, a sort of aliasing effect with LRU occurs, where LRU performs extremely poorly. Consider the case where the offset between the writer and reader becomes close to the size of the flash. In this case, the least-recently written block in flash may be the most likely to be read next. This effect can be seen in the 1.5 Mbps case with 30 minute offset, and 6 Mbps case with 5 minute offset, where LRU performs substantially worse than a simple write buffer.

In the high-bitrate streaming media cases, the EVS-ALF algorithm excels. The prefetching algorithm effectively tracks the position of the reading process, allowing both the reading and writing process to use flash, allowing the disk to mostly remain in the spun-down (standby) state.

In the web browsing workloads, the performance of the algorithms has more variety. With random sort (Figure 6), EVS-ALF performs slightly worse than the hardware-level approaches. Since random accesses present the worst case in predicting which files to prefetch, the EVS-ALF prefetching algorithm causes some thrashing, as the accesses appear to follow a pattern for a few accesses, but then quickly prove to be wrong, resulting in unnecessary copying of disk to flash. This causes the increased I/O power consumption shown in Figure 6. However, when the web browsing workload follows a less random access pattern, such as those in Figure 7, EVS-ALF excels again, outperforming the hardware-level approaches.

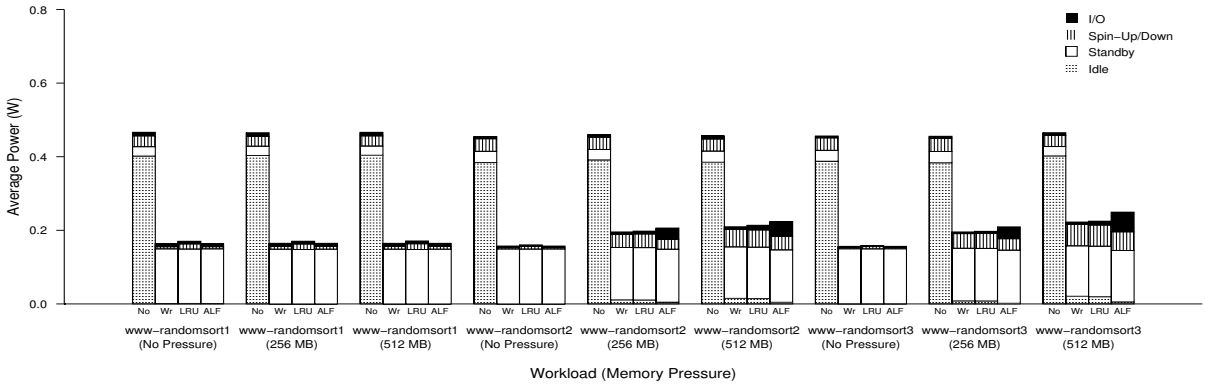

Figure 6. Web Browsing / Photo Gallery

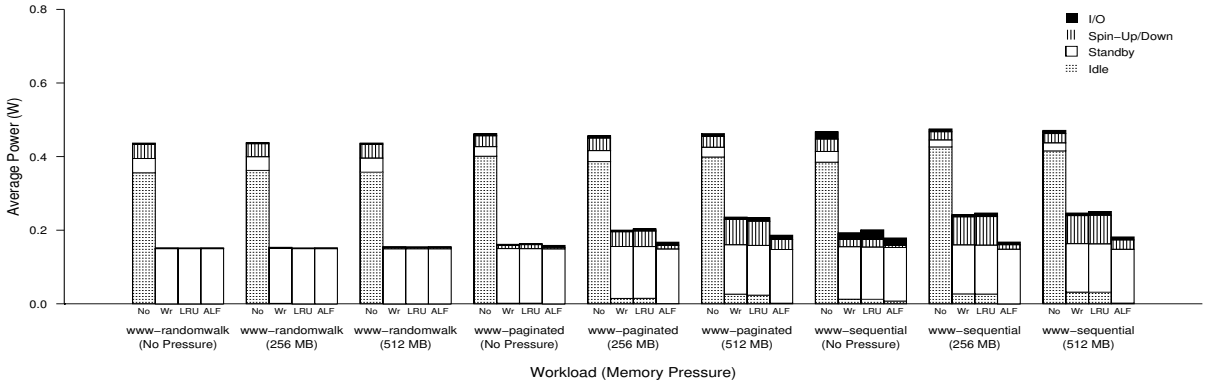

Figure 7. Web Browsing / Photo Gallery

Overall, the hardware-level approaches showed substantial power savings. The write buffer had an average power savings of 267 mW (53% of disk power), while LRU had an average power savings of 247 mW (49%). Additional gains were achieved by incorporating the OS and applications into the flash allocation decisions, with the ALF-EVS algorithm achieving an average power savings of 307 mW (61%).

8. CONCLUSION

This paper analyzes the power savings of using flash memory as a buffer for the hard disk in multimedia-rich mobile systems. We consider three possible algorithms: a hardware-level approach which uses the entire flash simply to buffer write requests, a hardware-level approach using an LRU replacement algorithm to treat flash as a disk cache, and a hardware/software-level Energy-efficient Virtual Storage (EVS) layer which relies on Application-Level Framing (ALF) metadata.

To evaluate these algorithms, a series of synthetic workloads are developed, the goal being to simulate typical multimedia-rich workloads commonly seen on modern mobile systems. These workloads consist of a web-based photo gallery, in which a user views photos using various usage models, and streaming media scenarios, where a user records and plays media at common audio and video bitrates. These scenarios were run on real systems, and traces were captured at the block-level for each scenario.

Experimental results show that the write buffer achieved an average power savings of 53% and LRU saved 49%. Thus, adding flash memory to a hard disk, and using a simple hardware approach to manage the flash blocks could significantly reduce the power consumption of a mobile system. However, by incorporating the operating system and application-level knowledge into the flash allocation process, more efficient algorithms can be developed, since applications contain a large amount of knowledge which can be used for power management. In many cases, future application disk accesses can be predicted, allowing flash to be used as a prefetching buffer to maximize the time the disk remains in a spun-down state. Using the EVS-ALF flash allocation algorithm described in this paper, an average power savings of 61% was achieved, an additional 8% improvement over hardware-only approaches.

REFERENCES

1. L. Paulson, "Hard Drive Saves Energy by Working while Resting," *IEEE Computer* **38**(8), 2005.
2. Intel Corporation, "Intel Discloses Technologies To Make The Internet More Personal And Mobile," press release. http://www.intel.com/pressroom/archive/releases/20060307corp_b.htm.
3. R. Nathuji, B. Seshasayee, and K. Schwan, "Combining compiler and operating system support for energy efficient I/O on embedded platforms," in *Proc. of the International Workshop on Software and Compilers for Embedded Systems*, 2005.
4. A. Papathanasiou and M. Scott, "Energy efficient prefetching and caching," in *Proc. of the USENIX Technical Conference*, 2004.
5. S. Chandra and A. Vahdat, "Application-specific network management for energy-aware streaming of popular multimedia formats," in *Proc. of the USENIX Technical Conference*, 2002.
6. L. Singleton, C. Poellabauer, and K. Schwan, "Monitoring of cache miss rates for accurate dynamic voltage and frequency scaling," in *Multimedia Computing and Networking*, 2005.
7. C. Shahabi, R. Zimmermann, K. Fu, and S. Yao, "Yima: A second generation continuous media server," *IEEE Computer* **35**(6), 2002.
8. Y. Hu and Q. Yang, "DCD - disk caching disk; A new approach for boosting I/O performance," in *Proc. of the 23rd International Symposium on Computer Architecture*, pp. 169–178, 1996.
9. M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," in *Proc. of the 5th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 10–22, 1992.
10. R. Katz, G. Gibson, and D. Patterson, "Disk system architectures for high performance computing," in *Proc. of the IEEE*, pp. 1842–1858, 1989.
11. C. Ruemmler and J. Wilkes, "UNIX disk access patterns," in *Proc. of Winter 1993 USENIX*, pp. 405–420, 1993.
12. Hitachi Global Storage Technologies, "Hitachi Travelstar C4K40 1.8-inch hard disk drive." http://www.hitachigst.com/tech/techlib.nsf/techdocs/4B0BCB1DB47911A386256D5500494CDC/$file/HGSTTravelstarC4K40.PDF.
13. Samsung, "Samsung K9K4G08U0M flash memory." http://www.samsung.com/Products/Semiconductor/Flash/NAND/4Gbit/K9K4G08U0M/ds_k9k4g08u0m_rev07.pdf.
14. L. Mummert and M. Satyanarayanan, "Long term distributed file reference tracing: Implementation and experience," Tech. Rep. CMU-CS-94-213, Carnegie Mellon University, 1994.
15. W. Hsu and A. Smith, "Characteristics of I/O traffic in personal computer and server workloads," *IBM Systems Journal* **42**(2), 2003.