Flushing Policies for NVCache Enabled Hard Disks

Technical Report UCSC-SSRC-07-04
May 2007

Timothy Bisson tbisson@cs.ucsc.edu
Scott A. Brandt sbrandt@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

Flushing Policies for NVCache Enabled Hard Disks

Timothy Bisson Scott A. Brandt
University of California, Santa Cruz

Abstract

One of the goals of upcoming hybrid hard disks is to reduce power consumption by adding a small amount of non-volatile flash memory (NVCache) to the drive itself. By using the NVCache to satisfy writes while the rotating media is spun-down, hard disk power consumption can be decreased by lengthening low-power periods. However, the NVCache must eventually be flushed back to the rotating media in order to cache additional data. In this paper we explore two questions: w*hen and* h*ow should NVCache content be flushed to rotating media in order to minimize the overhead of data synchronization. We show that by using traditional I/O mechanisms such as merging and reordering, combined with a "flush only when full" policy, flushing performance improves significantly.*

1 Introduction

Over the past few years, power has evolved into a first class resource [14] in computing environments. There are several reasons power consumption is relevant in desktop and laptop environments. From a monetary perspective, power is expensive and manifests itself in several ways. For example, in large organizations where each employee has a laptop or desktop the aggregate cost of several thousand computers is significant, although the individual power cost of each computer may only be dollars per year. Power is also important in the context of availability (primarily for laptops). When laptops are not running on AC they run on battery, which has a finite power supply. When that supply is exhausted, the laptop is no longer available. The implications of availability differ drastically, from minor inconvenience to mission critical. Power is considered a first class resource for other reasons as well, including environmental concerns, heat generation, and cooling.

Power management mechanisms exist in desktop and laptop computing in both the hardware and software layers. Hard disks provide different power states with varying levels of power consumption. Some power states can dynamically adjust the power consumption based on I/O activity, such as with Hitachi's ABLE technology [9]. Software solutions also exist at the operating system level, such as spin-down algorithms which determine the appropriate hard disk power state [8, 7].

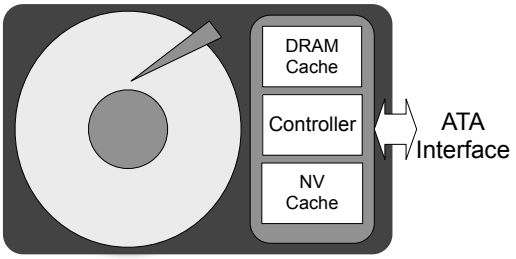


Figure 1. Hybrid Disk

Unfortunately, desktop operating system I/O behavior is often not suitable for hard disk power management because the I/O subsystem is not aware that I/O operations may reduce hard drive reliability or be blocked for several seconds while the rotating media spins up. However, work has been done to ameliorate this situation by reducing *observed* spin-up latency [3]. Additionally, applications do not consider the potential impact their I/O has on hard disks, possibly exacerbating hard drive issues associated with power management. For example, e-mail clients receive and store new e-mail messages regardless of the hard disk power state—all e-mails must be written to disk. The operating system must service such requests even if it means that the corresponding I/O will cause the hard disk platters to spin up. As a result, it is often difficult to reconcile the desires of both application I/O and power management functionality.

In the near future, hybrid disks [15] with a small amount of flash memory (NVCache) will be available, as depicted in Figure 1. The NVCache is stored logically adjacent to the rotating media. A hybrid disk maintains a single block address space, but the NVCache allows particular sectors to be stored indefinitely on the NVCache rather than on rotating media. An operating system can utilize such a device to reconcile power management and I/O performance. Continuing with the e-mail example, when the system is idle (no user activity), e-mail messages may sill arrive, but can be stored on the NVCache rather than the rotating media, allowing the heads to remain parked and the spindle spun-down.

In order for such I/O redirection to occur, hybrid disks introduce a new power mode, "NV Cache Power Mode". In this mode, the NVCache acts as a write-cache for incoming writes and the disk firmware implements its own rotating media spin-down algorithm. Read requests can be serviced solely from the NVCache if the corresponding sectors from a request are all located in the NVCache. If the NVCache becomes full or a read cannot be completely serviced from the NVCache, the rotating media must be spun up to make room for new requests or to service the read request.

The details of such I/O redirection and synchronization to and from the NVCache are left up to the manufacturer. As a result, it is unknown what the best algorithms are. For example, what is the best algorithm to flush data from the NVCache to rotating media? And, should the entire NVCache be flushed on each spin-up? These are the questions we explore in this paper with the goal of providing a performance baseline for manufacturers to compare against and an indication of the most efficient synchronization approaches.

2 Flushing Policies

Synchronizing data from the NVCache to the rotating media occurs when the NVCache is being used as a write cache (NV Cache Power Mode is on); all writes to the device are redirected to the NVCache. Eventually the NVCache will fill up and need to be flushed to the rotating media. In this paper we explore NVCache synchronization. In particular, we examine:

- When to synchronize the NVCache to the rotating media
- How to schedule synchronization

Our examination of when to synchronize the NVCache to the rotating media focuses on two alternatives: completely filling the NVCache before flushing it to the rotating media vs. flushing the NVCache on each spin-up. Our examination of how to schedule synchronization explores several scheduling algorithms used to flush NVCache data to rotating media.

2.1 When to Flush

This section discusses the trade-offs regarding the decision of when to flush NVCache data back to the rotating media. We investigate two natural policies: flush on each spin-up and flush only when the NVCache becomes full.

Flushing the NVCache after each spin-up means that if the rotating media is spun-up, regardless if it was because of a read or write, the NVCache is flushed to rotating media. As a result, the chances of the NVCache filling up decreases, meaning read requests are most likely responsible for spin-up operations. Since the NVCache is cleared of all content on each each spin-up, the coherent location of all sectors becomes rotating media—any subsequent request will only go to rotating media. As a result, the probability that the NVCache will contain read requested sectors during future spin-down periods decreases as those sectors will likely have been flushed to rotating media during a previous flushing operation.

Flushing the NVCache each spin-up also means that, while the rotating media is spun-up, subsequent I/Os will go to the rotating media. The synchronization process will also contend with user-initiated I/O, decreasing overall I/O performance. The overhead of flushing the NVCache to rotating media is relatively periodic because it occurs after every rotating media spin-up. Although somewhat subjective, users tend to find periodic stimulus more acceptable than aperiodic stimulus [4], so users may find such flushing policy acceptable.

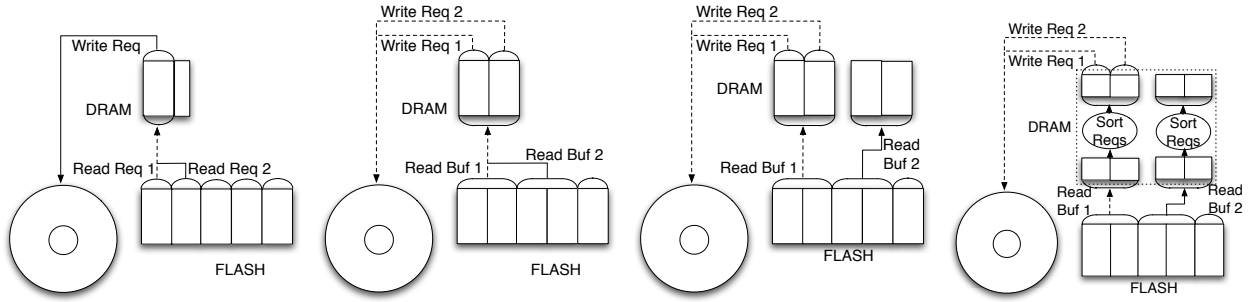
Alternatively, flushing the NVCache when it becomes full means that it will only be initiated on a redirected write request. As a result, flushing will not occur after each spin-up; flushing operations will occur less frequently, but each operation will be longer. From a user's perspective, flushing only when the rotating media is full is analogous to aperiodic stimulus. Therefore, users may be less tolerant of such performance degradation.

Although writes are more likely to cause spin-up operations when using the flush when full policy, reads are still the predominant cause of spin-up operations. However, since the NVCache will contain more valid sectors when read operations occur, the chances that read requests can be satisfied while the rotating media is spun-down increases. If valid sectors are stored on the NVCache while the rotating media is spun-up, subsequent requests (while rotating media is spun-up) may be forced to go to both rotating media and the NVCache, resulting in reduced I/O performance. However, another benefit of waiting until the NVCache is full to flush is that if sorting and merging occur, there may fewer sectors that need to be flushed back to rotating media. Such a feature is described in the next section, Section 2.2.

Considering the two approaches above, flushing on each spin-up effectively translates into flush-on-read, while flush when full translates into flush-on-write. Since read requests are often user initiated, flushing on each spin-up means that the flushing process will contend with user-initiated I/O consisting of read and probably write requests. On the other hand, flushing when full will occur in response to a write request while the rotating media is spun-down, meaning there is no user-activity. Therefore, flushing when full will likely result in data synchronization that a user never observes.

2.2 How to Flush

This section describes algorithms which aim to decrease the time to flush data from the NVCache to rotating media.



(a) Algorithm 1 (b) Algorithm 2 (c) Algorithm 3 (d) Algorithm 4

Figure 2. Flush Algorithms

An algorithm which flushes data from an NVCache to rotating media requires three resources: rotating media, DRAM, and FLASH. Hybrid disks will contain all these three resources. Fundamentally, flushing data from the NVCache to the rotating media involves reading the data from the NVCache into DRAM, and then writing it to rotating media. This in turn frees up the that data from DRAM and the NVCache. The four flushing algorithms (shown in Figure 2) build upon each other, aiming to make flushing more efficient.

The first algorithm, because of its simplicity, serves as a baseline. This algorithm is shown in Figure 2(a). It reads one request at a time from flash into the DRAM buffer. By request we mean a request redirected to the NVCache. Once the request is in DRAM it is written to the appropriate location on rotating media. This algorithm is independent of the DRAM size such that no matter how large the DRAM is only one request at a time will be read into DRAM and then written to rotating media.

The second algorithm improves upon the first by leveraging the size of the DRAM buffer to reduce the number of NVCache read operations and the time rotating media is blocked waiting for NVCache data to be read into DRAM. The main idea behind algorithm 2 is that when reading NVCache requests into DRAM, as many redirected requests (as can fit into DRAM) are read into DRAM with one large read request, as shown in Figure 2(b). We refer to reading multiple NVCache requests into DRAM at once as reading a *c*hunk. Included in the chunk is each request's data, plus metadata describing the request's location on rotating media. The size of a chunk depends on the size of the DRAM buffer allocated for synchronization. We assume redirected requests are written to the NVCache in log order, which enables performing a single NVCache read request for multiple redirected requests. Once a chunk is read into DRAM each request within the DRAM chunk is processed in the same log order.

Both of these algorithms suffer from the fact that either the rotating media or the NVCache is waiting to use the DRAM buffer. However, the NVCache is predominately blocked waiting for the rotating media to finish writing data. This happens because reading sequentially from the NVCache with a single request is faster than performing small random access writes to rotating media. Therefore, by further reducing the time rotating media is blocked on DRAM, we can increase the relative time spent writing to rotating media.

The third algorithm uses two DRAM buffers as shown in Figure 2(c). The DRAM buffer is actually split logically into two equal size DRAM buffers. By using two DRAM buffers we can ensure that the rotating media is always being written to. The main idea is that while requests from one DRAM buffer are being written to rotating media, the other DRAM buffer is being filled by the next set of redirected requests from the NVCache. The rotating media must still wait for the initial DRAM buffer to fill (starting a flush sequence) before writes can begin being written to rotating media. Therefore, relative to the question of *when to flush*, the initial wait time will occur more often with flush on each spin-up.

The fourth algorithm is shown in Figure 2(d). It extends algorithm three by adding merging and sorting to each DRAM buffer being written to rotating media. It is important to note that, as shown in the figure, sorting and merging occurs locally within the respective DRAM chunks, meaning coherency is still preserved. First, all requests are sorted in the DRAM chunk by rotating media sector address (LBA). The benefit of request sorting is that the disk arm will make a logical progression through the block address space when flushing each DRAM buffer, reducing overall seek time. Next, all overlapping requests are merged (using the request occurring last as the data source). Requests that partially overlap are also coalesced. By merging multiple requests into a single large request, disk I/O time is reduced.

2.3 Partial I/Os

It is often the case that when reading a chunk of memory into DRAM from the NVCache the last request doesn't completely fit into the DRAM chunk. For example, if the

Name	Type	Duration	Year
Eng	Linux Engineering Workstation	7 days	2005
HPLAJW	HP-UX Engineering Workstation	7 days	1992
WinPC	Windows XP Desktop	7 days	2006
Mac	Mac OS X 10.4 Powerbook	7 days	2006

Table 1. Block-Level Trace Workloads

chunk is 1MB and two redirected requests are .75MB each, one full I/O and one-third of the other will be present in the chunk. As a result, only the first redirected request in memory can be flushed to rotating media. The partial I/O request is ignored and re-read into the DRAM on the next DRAM chunk read.

Alternatively, if a small DRAM size is used to move data from flash to rotating media, the first redirected I/O may often be larger than the total DRAM chunk size. Still, the request must eventually be written to disk. In order to accomplish this, a redirected flash request is written to rotating media in the form of partial I/Os, where the partial I/O size is equal to the DRAM chunk size. When the last partial I/O for a redirected request is written out to rotating media, the next redirected request is flushed from NVCache to rotating media. Naturally, as the DRAM chunk size increases the chances of a partial I/O request occurring decreases.

3 Experimentation

To emulate a hybrid disk and the proposed algorithms we use a 2.5 in disk and compact flash card. The 2.5 in disk is a Hitachi Travelstar E7K100 and the flash device is a Sandisk Ultra II CompactFlash memory card; the flash device represents the NVCache. Traditional DDR memory simulates a hybrid disk's DRAM cache. The host system used for the experiments is a Linux 2.6 machine with a Pentium 4 3.06GHZ processor. Raw-device access is used to access the block address space of both the NVCache and rotating media.

To properly examine which sectors, written to disk, are written back to rotating media during a flushing operation, we replayed several real block-level I/O traces through a spin-down algorithm. When the spin-down algorithm spins down the rotating media, subsequent trace I/O requests are redirected to the NVCache. Writes are redirected to NVCache with a metadata sector describing the redirected requests sector number, offset, and length. We use zero-filled bytes as the actual data transferred between the different storage media. The rotating media is left spun-down while writes are redirected to the NVCache. Reads are also redirected to the NVCache if the rotating media is spun-down in the hopes that the NVCache can service the request. If the NVCache cannot service a read request, the rotating media is spun-up and services the request, possibly gathering the requested data from both rotating media and NVCache sectors. After the rotating media is spun-up, and if the NVCache should be flushed to rotating media, requests from the NVCache are read into DRAM, then written out to rotating media, according to the algorithms in Section 2.1. The traces are used to determine when to spin-down the rotating media, the size of redirected requests, and where those requests belong on rotating media.

The spin-down algorithm implemented is the multiple experts spin-down algorithm. It is an adaptive spin-down algorithm developed by Helmbold *et al*. [8]. The spin-down algorithm is based on a machine learning class of algorithms known as Multiple Experts. In the dynamic spin-down algorithm, each expert has a fixed time-out value and weight associated with it. The time-out value used is the weighted average of each expert's weight and time-out value. It is computed at the end of each idle period. After calculating the next time-out value, each expert's weight is decreased proportional to the performance of its time-out value. Multiple experts has been shown to outperform any fixed spin-down timeout.

3.1 Traces

The block-level access traces we use are from four real desktop workloads gathered from four different desktop operating systems, which are shown in Table 1. Each workload is a trace of disk requests from a single disk, and each entry contains: I/O time, sector, sector length, and read or write. The first workload, *Eng*, is a trace from the root disk of a Linux desktop used for software engineering tasks; the ReiserFS file system resides on the root disk. The trace was extracted by instrumenting the disk driver to record all accesses for the root disk to a memory buffer, and transfer it to user space (via a system call) when it became full. A corresponding user space application appended the memory buffer to a file on a separate disk. The trace, *HPLAJW*, is from a single-user HP-UX workstation [16]. The *WinPC* trace is from an Windows XP desktop used mostly for web browsing, electronic mail, and Microsoft Office applications. The trace was extracted through the use of a filter driver. The final trace, *Mac* is from a Macintosh PowerBook running OS X 10.4. The trace was recorded using the Macintosh command line tool, fs_usage, by filtering out file system operations and redirecting disk I/O operations for the root disk to a USB thumb drive.

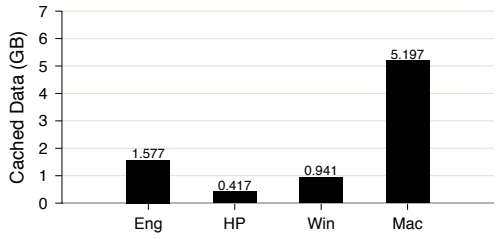
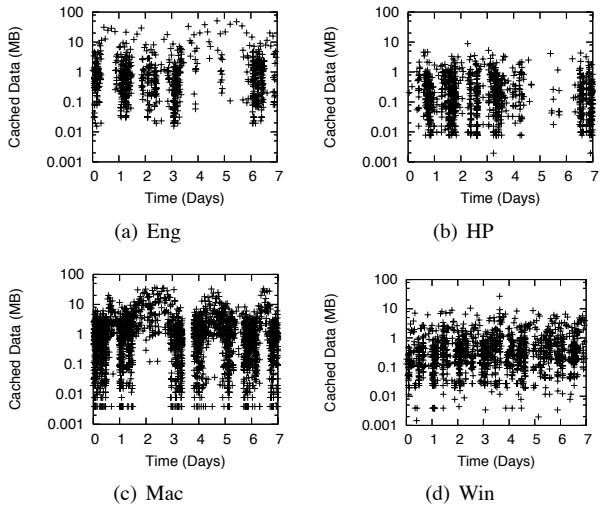


Figure 3. Optimal Cached Data



(a) Eng (b) HP

(c) Mac (d) Win

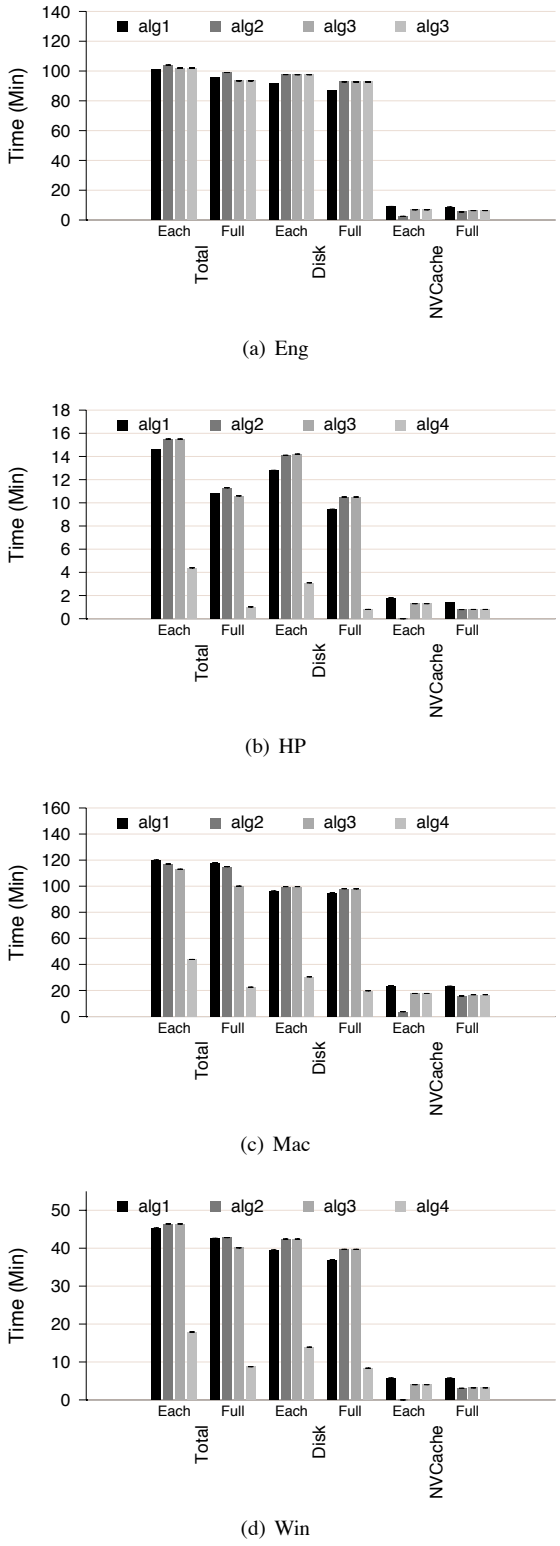
Figure 4. Optimal Cache Data distribution

4 Results

Before evaluating the performance of the proposed algorithms, we first consider write caching performance with an infinite-sized NVCache. Figure 3 shows the amount of data cached using an infinite NVCache for the 7 days of each trace. The amount of data cached with an infinite NVCache serves as a point of reference for the four proposed algorithms. For example, this figure shows that if the NVCache is flushed only when full, the NVCache need only be 5GB in order to cache a week's worth of I/O for the Mac trace. Similarly, the NVCache needs need only be .5GB to cache a week's worth of HP data. However, because the NVCache size is finite, the actual amount of data redirected will be less.

Figure 4 also shows the amount of data cached while the rotating media is spun-down with a infinite NVCache. However, this figure shows the amount of data cached per spin-down period. Even with an infinite NVCache 100MB of NVCache is never used on a single spin-down period. With the flush on each spin-up policy, at most a 100MB NVCache is needed to cache all writes per spin-down period.

The crucial metric for measuring flushing performance is time—how long does it take to flush NVCache content back



(a) Eng

(b) HP

(c) Mac

(d) Win

Figure 5. Flush Time

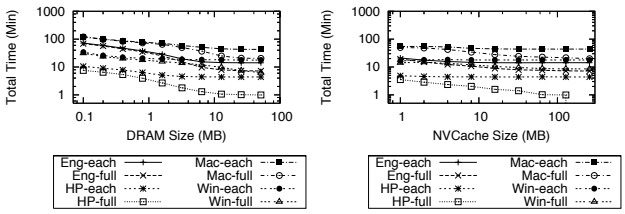
to rotating media. Figure 5 shows these results for each trace, including a break-down of time spent for each media. The amount of DRAM in these experiments is 16MB and an NVCache size of 64MB. In these figures the label *Each* represents flushing the NVCache after each spin-down period. The label *Full* represents flushing the NVCache only when the NVCache is full. And *Total* represents the total flushing time, including both time to read the flash buffers into memory and write corresponding requests back to the appropriate location on rotating media. The labels *NVCache* and *Disk* are a breakdown of the flushing time for each particular media.

This figure shows several interesting properties about flushing performance. First, we see that when comparing *Full* to *Each* for total time, *Full* is generally faster than flushing on each spin-up—it reduces the amount of time each media is blocked. Second, flushing is largely dominated by disk. This is because the NVCache can issue a single 64MB read to the NVCache, which consists of several redirected requests, each of which must be written to the rotating media.

When comparing the individual algorithms to each other the first thing to observe is that when looking at the *Total* time for *Each*, we see that algorithm alg1 performs better than alg2 many times. Looking at the respective *Disk* and *NVCache* timings, we see that the *NVCache* time typically decreases slightly from alg1 to alg2 but the disk time increases. The *NVCache* decreases because fewer I/Os are needed to read the *NVCache*. However, we suspect that the disk timing increases because with alg2 potentially hundreds of I/Os are pushed out to the disk all at once, while with alg1, the disk has a chance to catch up while the *NVCache* reads each redirected request.

The figure also shows that alg3 improves performance over alg2. This occurs even though the *NVCache* time actually increases and the *Disk* time remains the same. The reason is that because the DRAM buffer is split into two smaller portions, the *NVCache* is blocked on each for less time, resulting in a overall decrease in flush time. Alg4 improves performance because less and larger I/Os end up going to rotating media which are also ordered. As a result, *Disk* timing decreases, but the *NVCache* timing increases. Finally this figure shows that all algorithms improves when shifting to *Full*. This occurs because more data is cached per flushing period, resulting in less disk flushing time, but with roughly the same amount of *NVCache* flushing time.

Figures 6–12 show different metrics as a function of both DRAM and FLASH size, with the goal being to show how varying these parameters affect each metric. In these figure, we show results for algorithm 4 only because it is the best performing and most natural to use of the four flushing algorithms. In each set of figures the left figure, DRAM, represents scaling the DRAM size between 100KB and 64MB.



(a) DRAM (b) NVCache

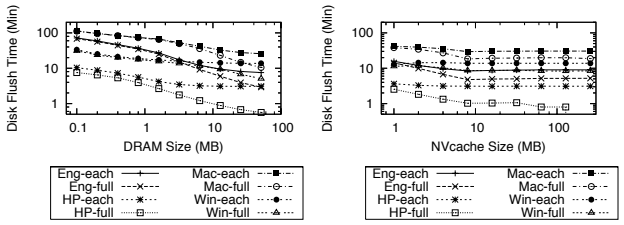
Figure 6. Total Flush Time

The right figure, NVCache, represents scaling the NVCache size from 1MB to 256MB. Each of the four traces is shown in every figure. Additionally, each trace is shown twice: once with a "-full" appended to the trace name, and once with a "-each"appended, as shown in the legend of each figure. The "-full" represents flushing when full and "-each" represents flushing on each spin-up. Note, when scaling the DRAM size, the NVCache size remains fixed at 64MB. And when scaling the NVCache size, the DRAM size is fixed at 16MB.

Figures 6, 7, and 8 show flushing performance, similar to Figure 5. However, in these results we vary the DRAM and NVCache sizes. Figure 6(a) shows that by increases DRAM size, total flushing time decreases significantly. This is because with a larger DRAM buffer more data can be read into DRAM which can then be flushed back out to rotating media. The flushing performance flattens out because the benefit of reordering and coalescing plateaus. Figure 6(b) shows that the total flushing time is stable relative to the NVCache size, which as shown in Figure 5 is because the NVCache is not the bottleneck. Both figures show that at larger NVCache and DRAM sizes, the superiority of the flush on full policy over flush on each spin-up is obvious. Note that the reason for the sharp drop off for the HP-full trace at 256MB is because the NVCache never fills up and so never flushes its data to the rotating media.

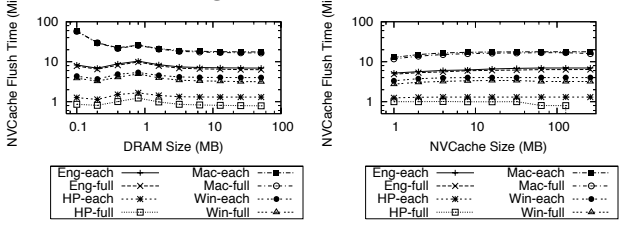
Figure 7 shows a similar pattern to that of Figure 6, except that in Figure 7(a) the disk time continues to decrease even after 10MB of DRAM is used. Figure 7(b) shows that disk time is relatively independent of the NVCache size, which is natural. Figure 8 shows the results for NVCache performance. Figure 8(a) shows that the NVCache time decreases slightly with bigger DRAM sizes, as bigger and fewer read I/Os can be issued to the NVCache. Figure 8(b) shows that as the NVCache increases, the amount of time reading from the NVCache increases only slightly. Again, the HP-full trace is an exception because between 64-128MB, fewer flush operations occur, and none with 256MB.

Figure 9 shows the number of merges (only shown for algorithm four because it is the only algorithm to provide such functionality). Figure 9(a) shows that the number of merges is directly affected by the DRAM size. However, scaling the



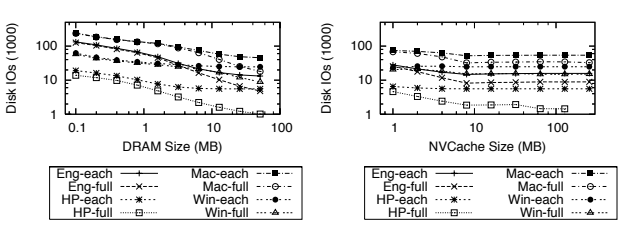
(a) DRAM (b) NVCache

Figure 7. Disk Time



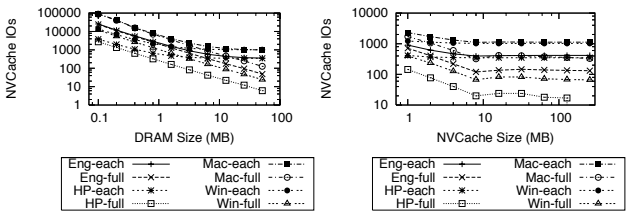
(a) DRAM (b) NVCache

Figure 8. NVCache Time



(a) DRAM (b) NVCache

Figure 10. Disk I/Os



(a) DRAM (b) NVCache

Figure 11. Flash I/Os

NVCache (Figure 9(b)) does not yield a significant increase in the number of merges. Slightly more surprising is that the number of merges for the "-full" trace is not more than for the "-each" traces, especially for the larger NVCache sizes, where the NVCache should be more utilized.

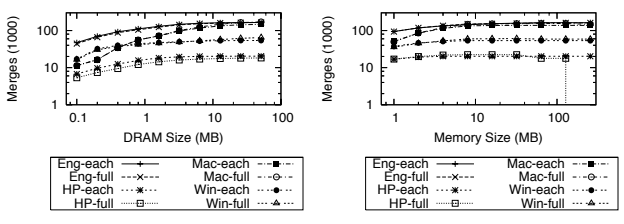
Figure 10 shows the number of disk I/Os during flushing. Figure 10(a) is an important result because it shows algorithm four is able to coalesce more requests from DRAM, up to the point when the DRAM size is larger than the NVCache size. Also, the "-full" traces do not flatten out with DRAM sizes greater than 10MB because they can leverage more of the NVCache. Figure 10(b) shows that the number of disk I/Os decreases until the DRAM size is reached. However, with the "-full" trace the number of I/Os is lower than the "-each"traces.

Figure 11 shows the number of read I/Os. The number of read I/Os decreases linearly in Figure 11(a) because the determining factor is the DRAM size. Likewise in Figure 11(b) the number of read I/Os decreases until the default DRAM size is reached. Fundamentally, the number of read I/Os is limited by both the DRAM and NVCache size.

Figure 12 shows the amount of data flushed from the NVCache to rotating media. Figure 12(a) shows that the amount of flushed data in independent of the DRAM size. This is because the DRAM just serves as temporary medium in which to flush the redirected requests stored in the NVCache. Figure 12(b) shows that the amount of flushed data increases with an increase of NVCache because as the NVCache grows the longer the disk stays spun-down. With smaller NVCache sizes, the rotating media stays spun-up more often which causes writes to be serviced by the rotating media and not the NVCache, of which does not have to flushed. Lastly, this figure shows that with a sufficiently large NVCache and DRAM, flushing performance approaches that of an infinite NVCache.

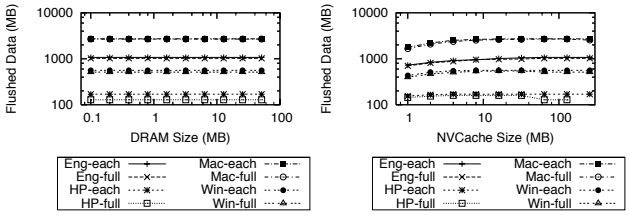
5 Related Work

There are several works which consider combining flash and rotating media to decrease hard disk power consumption. Marsh *et al.* propose that flash (FLASHCACHE) exist as a layer between DRAM and rotating media [13]. Data passes through the FLASHCACHE with an LRU policy. They show it is possible to reduce hard disk power consumption and increase performance. *NVCache* [2] and *SmartSaver* [6] are hard-disk energy saving schemes, which use flash to buffer requests during standby and prefetch disk data to increase standby periods. The significant difference



(a) DRAM (b) NVCache

Figure 9. Merges



(a) DRAM (b) NVCache

Figure 12. Flushed Data

between these approaches is the eviction policy. NVCache uses a combination of LRU and LFU, while SmartSaver uses an algorithm akin to *GreedyDual-Size* [5] originally developed for web-caching.

Alternatively, another work named FlashCache uses flash memory to reduce the power consumed by main-memory in web servers [12]. Fundamentally, FlashCache acts like a secondary buffer cache to reduce main-memory power consumption during idle-time without impacting network performance. There are several other works that utilize some form of non-volatile memory to increase I/O performance. Ruemmler and Wilkes [16], Baker *et al*. [1], Hu *et al*. [11], and WAFL [10] all buffer disk I/O in NVRAM to some extent.

Hybrid disks place a small amount of flash memory logically adjacent to the rotating media. Interfaces to leverage the NVCache are specified in the ATA8 specification [17]. However, implementation is largely left to the manufacturer. Unfortunately, this means most hybrid disk technology will not be published. Therefore, it is our goal to provide functionality and performance measurements to serve as baseline for future hybrid disk technology and research.

One work that explores ways to leverage hybrid disks to minimize power consumption is by Bisson *et al*. [3]. This work leverages hybrid disks at the OS layer to reduce power consumption, spin-up latency, and wear-leveling impact. This work presents four algorithms exploiting I/O that occurs while the rotating media is spun-down.

6 Conclusion

We have presented several algorithms which improve the efficiency of synchronizing NVCache data to rotating media for upcoming hybrid hard disks when the NVCache is used as a write-cache to reduce hard disk power consumption. We focused on two fundamental policy questions: when to flush and how to flush. We found that flushing the NVCache only when full can reduce flushing time by over 75% relative to flushing on each spin-up. We also found that ordering and merging are effective in reducing the overall number of I/O operations to rotating media, and ordering reduces disk seek time, together reducing flushing time by as much as 90% over algorithms without ordering or merging.

References

[1] 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 Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, Oct 1992.

[2] T. Bisson, S. A. Brandt, and D. D. Long. Nvcache: Increasing the effectiveness of spin-down algorithms with caching. In *Proc. of the 14th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 422–432, Sep 2006.

[3] T. Bisson, S. A. Brandt, and D. D. Long. A hybrid disk-aware spin-down algorithm with i/o subsystem support. In *Proceedings of the 26th IEEE International Performance, Computing and Communications Conference*, Apr 2007.

[4] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 297–304, Jan 2000.

[5] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 193–206, Dec 1997.

[6] 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, Oct 2006.

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

[8] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proc. of the 2nd international Conference on Mobile Computing and Networking*, pages 130–142, Nov 1996.

[9] Hitachi. Hitachi travelstar e7k100 specification. `http://www.hitachigst.com/tech/techlib.nsf/products/Travelstar_E7K100`, July 2006.

[10] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

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

[12] T. Kgil and T. Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112, Oct 2006.

[13] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, pages 451–460, Jan 1994.

[14] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pages 67–72, May 2001.

[15] R. Panabaker. Hybrid hard disk and readydrive technology: Improving performance and power for windows vista mobile pcs. http://www.microsoft.com/whdc/winhec/ pres06.mspx, 2006.

[16] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proc. of the USENIX 1993 Winter Technical Conference*, pages 405–420, Jan 1993.

[17] C. E. Stevens. At Attachment 8 - ATA/ATAPI Command Set (ATA8-ACS). http://www.t13.org, 2006.