Management Policies for Non-Volatile Write Caches

Theodore R. Haining and Darrell D. E. Long
Computer Science Department
Jack Baskin School of Engineering
University of California
Santa Cruz, CA

Abstract

Many computer hardware and software architectures buffer data in memory to improve system performance. Volatile disk or file caches are sometimes used to delay the propagation of writes to disk (called delayed writes*). While delayed writes improve system performance, volatile caches can cause the loss of vital data during sudden failure. In this study, we investigate managing non-volatile RAM (NVRAM) caches with different simple strategies to delay writes to disk. We evaluate the performance of NVRAM caches using three measures of merit: the number of stalled writes which wait while the cache is cleaned before being serviced, the mean service time for I/O requests, and the number of writes generated by cleaning the cache. Our results show that even small non-volatile write caches using simple management policies can reduce the number of writes to disk by at least 70% and as much as 80% in some cases. Our results also show that the number of stalled writes is high: 30% at best and nearly 100% at worst. Adding pro-active purging effectively decreases both stalled writes and disk write activity.*

1 Introduction

As processors and main memory become faster and cheaper, a pressing need arises to improve the write efficiency of I/O subsystems. Disks in particular are larger, cheaper, and faster than they were 10 years ago, but their performance still lags that of other subsystems. The effect of this disparity is most felt with I/O intensive tasks, especially write-dominated tasks. Read traffic is affected, but large main-memory caches are an effective technique for reducing the number of reads made to disk. This technique is suitable for writes but less effective because the volatility of main memory prevents data from being cached for more than a few seconds. In spite of this limitation, volatile write caches are part of many different kinds of storage systems, including on-line transaction processing (OLTP) systems [5], file systems [13], and disk controllers [15].

One way around this problem is to cache writes in non-volatile memory and delay writes indefinitely before being sent to disk. The longer that data is held in memory, the more likely it will be overwritten or deleted. It is also more probable that data in the cache can be grouped together to yield larger, more efficient writes to disk. Non-volatile memory can also provide guarantees of cache consistency and recovery comparable to a disk after sudden failure. There has been considerable interest in non-volatile caches for use in memory based file systems [6], monolithic and distributed disk file systems [1], [3], [4], network file systems [10], disk arrays [16], and transaction processing systems [2], [7], [12]. Advances in technology and manufacturing continue to make the cost of memory and rechargeable batteries cheaper, ensuring continued interest in the applications of non-volatile memory.

Despite a large interest in non-volatile memory, little comparative work has been done with cache management policies in file system applications [3], [15]. While this work contributed important results, it did little to compare policies and use different metrics to measure their performance. Two simulation studies focussed either on a family of caches with a single policy for cleaning blocks [3], or two cleaning policies with limited performance data [15].

Our goal is to study this problem by looking at several different ways to implement and manage a non-volatile write cache. We simulate a small cache of non-volatile RAM (NVRAM) to test different types of cache management and cleaning policies. We use the *least recently used* (LRU), the *shortest access time first* (STF), and *largest segment per track* (LST) disk scheduling algorithms to decide what data to clean from the cache. We apply these algorithms to caches managed by a simple *write behind* policy where blocks are written to the cache and previously updated blocks are written to disk only when the cache is full. We compare these results to caches using *write behind with thresholds* where purging begins when the percentage of dirty blocks in the cache rises above a high threshold and ends when it falls below a low threshold. We measure each simulation with three major metrics: the amount of time it took to service each write request, the amount of traffic used to clean the cache, and the number of cache misses in each cache.

Write behind and write behind with thresholds were both

Supported by the Office of Naval Research under grant N00014-92-J-1807.

originally applied to non-volatile caches in work by Biswas, Ramakrishnan, and Towsley [3]. This work contributed important results, but only focussed on cleaning with one disk scheduling policy. We wished to understand more about the interaction of these policies with different disk scheduling techniques.

For our experiments, we model the components of the I/O subsystem of interest: the disk head, the disk controller, the data bus, and the read and write caches. We then use file system traces collected by Ruemmler and Wilkes [15] to drive a detailed simulation of the disk subsystem. We validate the accuracy of our simulators by comparing simulation times with real trace times and measuring the error.

Section 2 discusses the disk scheduling algorithms and write policies we used in our experiments. Experimental observations and results are found in §3. The final section summarizes the major results of this work and mentions some future directions our research will take.

2 Cache management policies

To be effective, I/O requests to a write cache must either hit sectors already in the cache or empty space must be available. The hit rate for the cache is governed by two factors: the locality of reference in the request stream, and the scheduling algorithm used clean the cache. The availability of free space depends on the policy used to evict data from the cache.

We make several assumptions about the management of the cache. Data in the cache is organized by its physical track on disk. The cache is divided into track-based lists of *dirty* and *clean* sectors by the cache controller. The controller evicts sectors from the clean list when cache space is needed for incoming data. Data is then written to the cache making one or more tracks dirty. If a write request includes sectors from a track on the clean list, the clean already on the track (if any) are deleted and track becomes dirty. Dirty tracks are moved to the clean list by committing the data in that track to disk.

2.1 Cache scheduling policies

We use simple disk scheduling algorithms to order the dirty and clean lists in the cache for our experiments. We want examine the influence of three principles: temporal locality, seek distance, and write size. Three algorithms were used in our simulation study: Least Recently Used (LRU), Shortest Access Time First (STF), and Largest Segment per Track (LST).

In the LRU algorithm, the most stale data in the cache is purged first. LRU ignores the number of times data is written into the cache; it keeps track of the oldest dirty data currently in the cache. This beneficially conditions the cache when data already in the cache is modified because its lifetime in the cache is extended. This increases the opportunity for further writes of that data to be absorbed by the cache. LRU does nothing to ensure that the write is reasonably short or of significant size.

The STF algorithm attempts to minimize the amount of time between when the write is initiated and when the first bit of data is written. The cache controller models the current position of the disk head and writes the data with the lowest sum of seek time and rotational delay. This data may be overwritten in the near future and the amount written may be small. The amount of effort to model the position of the disk head is also non-trivial.

With the LST algorithm, the cache controller sorts the dirty list in the cache by the number of blocks in each dirty track. It cleans the track with the most dirty blocks first. LST requires very little state information. It has the advantage that it initiates the purge that will free the largest possible amount of space in the cache. The seek distance required to perform that purge may be large, and the data may be written to the cache again in the near future.

2.2 Cache eviction policies

The simplest policy we use to free space in the cache is write behind. Sectors are written to the cache until the cache is full, and then dirty tracks are evicted when a new write request arrives. Because blocks are evicted in cache based groups, the scheduling algorithm can amortize the cost of several write requests in one write. The disadvantage of this approach is that the write request that causes the cache to become full is *stalled* until a portion of the dirty list can be written to disk.

A simple way to improve this policy is to add thresholds to the cache to create proactive eviction policies. A single threshold cache begins to clean after the percentage of dirty blocks in the cache exceeds a *high* threshold. When the threshold is crossed, cache controller sets a "clean request" flag. Once this flag is set, the controller commits groups from the dirty list and moves them to the clean list. The clean request flag is reset when the percentage of dirty blocks falls below the high threshold. A dual threshold cache adds a second *low* threshold. In this case, the clean request flag is not reset until the percentage of dirty blocks is less than the low threshold. The combination of high and low thresholds delays the start and stop of purging from the cache, creating hysteresis.

Adding thresholds has a number of advantages. There is always some free space in the cache, abd writes stall only during bursts in write traffic. The use of thresholds means that cache cleaning does not need to be performed immediately. The controller can attempt to clean the cache when the disk would otherwise be idle, for example. The use of both high and low thresholds means that the clean request flag is set infrequently if the difference between the two is large. The task of cleaning the cache is split into smaller

parts, reducing the impact of the additional activity on other I/O requests. If the cache does fill during peak periods of load, it is still possible to immediately clean blocks in the cache until the required amount of free space is available.

3 Simulation results

We studied the impact of cache replacement policy on the utilization of the disk with a specific emphasis on overall response time. We collected several related measurements: service time, which included the seek time, settle time, and rotational delay as well as time to transfer data to or from the disk, and the queue time each request spent in a queue waiting to be serviced. From the service time and queue time, we calculated the overall response time for each I/O request. We measured the number of times that writes were made to the disk to clean the cache and the size of each write used to clean the cache. We also recorded the number of cache hits and cache misses for the non-volatile write cache.

To run our experiments, we implemented our own models of the HP2200C and the HP97560 disks. We based our models on techniques described by Ruemmler and Wilkes [14], and an implementation by Kotz, Toh, and Radhakrishnan [11]. Each disk model is implemented in C++, and designed to support multiple disks connected to one or more data buses within the same simulation. The simulation currently uses the Sim++ event simulation package [8], but can be easily ported to another environment.

We found that service time was the most useful due to abnormally long queue lengths in the simulation. Small inaccuracies of less than a millisecond in the services times of our simulators skewed the queue times of simulated read events compared to those in the traces. When the traces were collected, long groups of reads of consecutive sectors were made, presumably to back up the disks. These reads were synchronous, and each read explicitly began within a millisecond of the completion of the one before it. Our disk service times were slightly too large (< 1ms), creating long queues of events and long queuing times where none existed in the trace. This affected the mean queue times and mean response times for the disks we simulated.

We also found that the number of cache misses and the number of writes to disk also strongly affected overall response time. Each cache miss may force other events to wait in a queue for service. If cache misses occur frequently, mean queue and mean response times will increase. The number of writes needed to clean the cache compared to the number of writes in the trace is a strong metric for cache efficiency. Because the cache cleaner creates groups of writes, other events from the trace will wait in the queue while they are serviced. These bursts will also affect also the queue and response times of events in the trace.

For our experiments, we concentrated on disk 5 of the **snake** trace set and disk 0 from the **hplajw** set. Some of the important static and dynamic characteristics for these data sets are summarized in Table 1.

Table. 1
Characteristics for disks used in our analysis.

Disk	# of Read I/Os	Mean Read Size (KB)	Mean Service Time (all) (ms)	Block Size (bytes)
5	134420	5.354	11.735	512
0	41080	4.409	25.346	256

Disk	# of Write I/Os	Mean Write Size (KB)	Mean Service Time (writes) (ms)
5	129379	6.876	12.388
0	82054	6.024	27.475

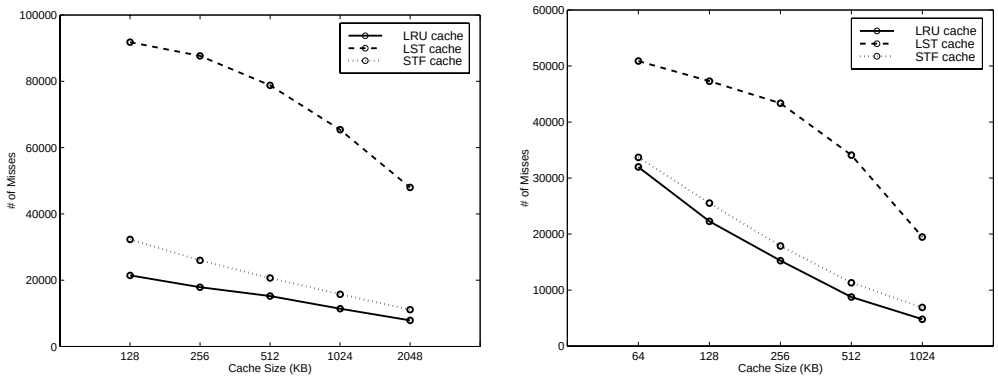
3.1 Write behind cache management

A write cache contains the modified disk blocks which must be eventually committed to disk to make room for new blocks as write events occur. A simple policy to evict blocks from the cache is to wait until the cache is completely full and then purge the cache with a write behind strategy, as explained in §2. Because the cache operates at a nearly full steady state, writes frequently stall while room is made for the new data. The idea of simple non-volatile write behind caches was first suggested elsewhere using the LST algorithm [3].

It was assumed in that work that such policies "would result in unacceptably poor performance" because of frequent stalling and were never tested.

We performed our own tests with write behind caches. We varied the cache size starting at 128KB and doubled the size on each run until we reached 2MB. For the three disk scheduling algorithms we tried, LRU performed the best in these tests, closely followed by STF, and then LST. Our service time measurements show that LRU and STF decreased the mean service time by at least 25%, scaled well as cache size increased, and reduced the number of writes to disk by at least 75% for both disks. The LST managed cache produced little reduction in the number of writes to disk especially for small caches with a correspondingly small reduction in service times.

Our results for stalled writes in some measure dispute the assumption that write behind caches are ineffective because of the high number of stalled writes (see Figure 1). While the cache miss rate is high for LST caches (nearly 100% of all writes are cache misses for small caches), cache misses for the LRU caches were much smaller (less than 20% for disk 5 and 40% for disk 0) with the perfromance of the STF cache in between. These results show that write behind



(a) Disk 5 (b) Disk 0

Fig. 1. Cache misses for disks with a write behind cache.

caches can be effective and offer significant performance improvement without having any cache parameters to tune.

3.2 Cache management with thresholds

While a simple write behind cache can improve cache performance, the best cache management algorithm we looked at only showed 30–50% improvement in mean service time. For this reason, we also studied a track based purge policy triggered by thresholds to prevent the cache from becoming completely full or (in some cases) completely empty. By preventing the cache from becoming completely full, stalled writes can be reduced because some clean space is always available to hold new data. Since the cache also does not completely empty, some data that will be overwritten in the cache in the near future will (hopefully) not be written to disk.

3.2.1 Single threshold caches. First we considered a single threshold variant of this cache eviction policy. With this type of cache, the high and low thresholds are set to the same value. The resulting cache is similar to a write behind cache, with some improvements. Because there is always some clean space in the cache, writes to the cache stall less frequently and cache writes can be made by the background cleaner. The amount of cache space cleaned in the background is small and the cache must be cleaned frequently.

First, we investigated the sensitivity of the cache to the threshold setting. We looked at single threshold caches of two different sizes for each disk; the caches were 128KB and 256KB for disk 5 and disk 0. Cache sizes were small to prevent the cache from holding the working set of written blocks. At the same time, we wanted to get some feeling for how these parameters change with cache size.

The number of writes to disk for a single threshold cache was within 10% of that of a write-behind cache of the same size for the LRU and STF algorithms until the threshold rose above 90%. The write traffic of the LST cache for disk 5 was also within 10% for both sizes, but the single threshold LST cache improved about 30% for most threshold values. The least amount of write traffic was produced for all single threshold caches when the threshold was set at 98%. This single threshold cache produced 25–50% fewer writes than the write behind cache.

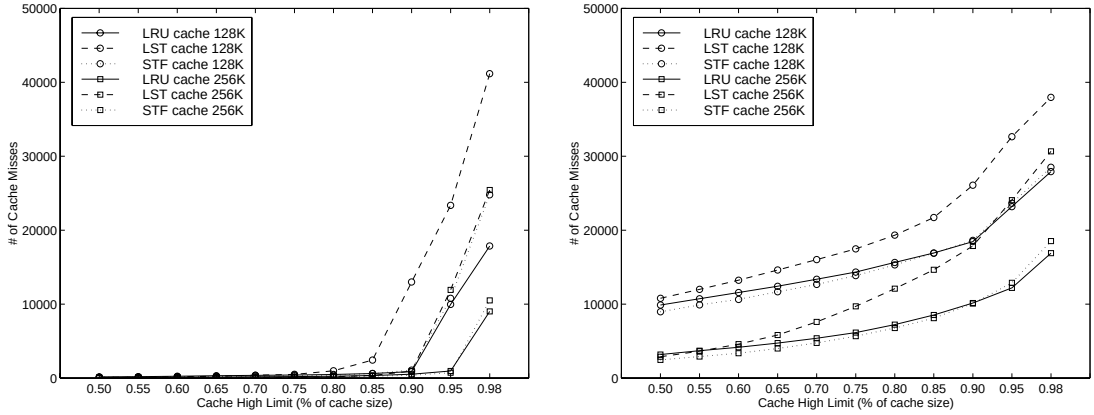
Our results showed that adding a single threshold only improved the number of stalled writes produced by the write behind cache (see Figures 1 and 2). The reduction in stalls is attributed to the clean space kept available by the threshold; data is written instead of forcing stalls. Since only the number of stalls changes, temporal locality still dominates and the LRU cache performs best.

Our results showed that choosing a good cache threshold was a trade-off between the numbers of stalled writes and writes to disk. We tried for a balance which decreased the number of writes to disk, but avoided sharp increases in the number of stalled writes. Based on these criteria, we noted that the best choice for a high limit threshold is in the range of 90–95% for both disks for all algorithms.

3.2.2 Dual threshold caches. Using a single threshold improves cache performance, but it is difficult to find a good balance between writes to disk and stalled writes. To avoid this problem, we also looked at caches using high and low thresholds to create hysteresis in the cache purging process. Because the amount of space cleaned using a dual threshold scheme is larger, the number of stalled writes will decrease.

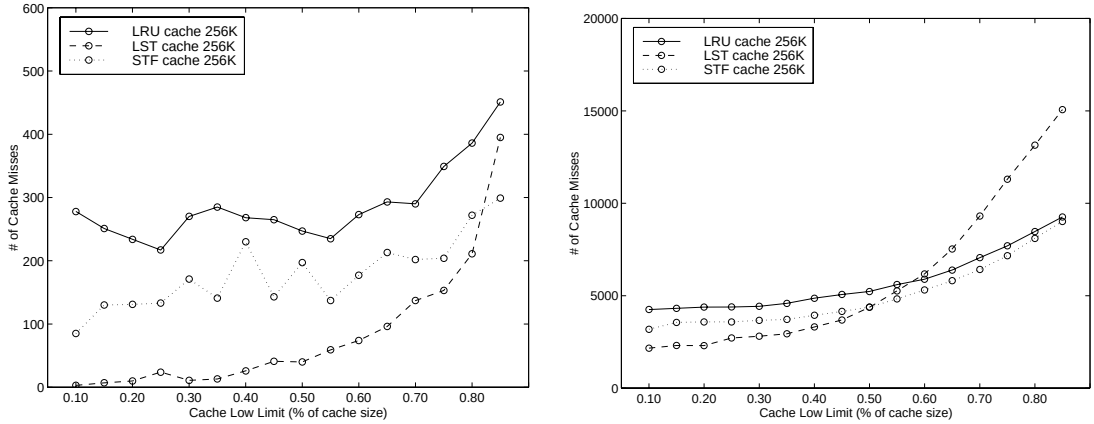
We began by testing how the interaction of the high and low threshold values affected performance. We fixed the high threshold at the single threshold values and varied the low threshold value from 10–85%. For these experiments, cache size was set at 256KB. The number of cleaning writes and cache misses for each cache are found in Figures 3 and 4.

Finding a good value for the low threshold depends on what metric is used. Based on the number of cache misses



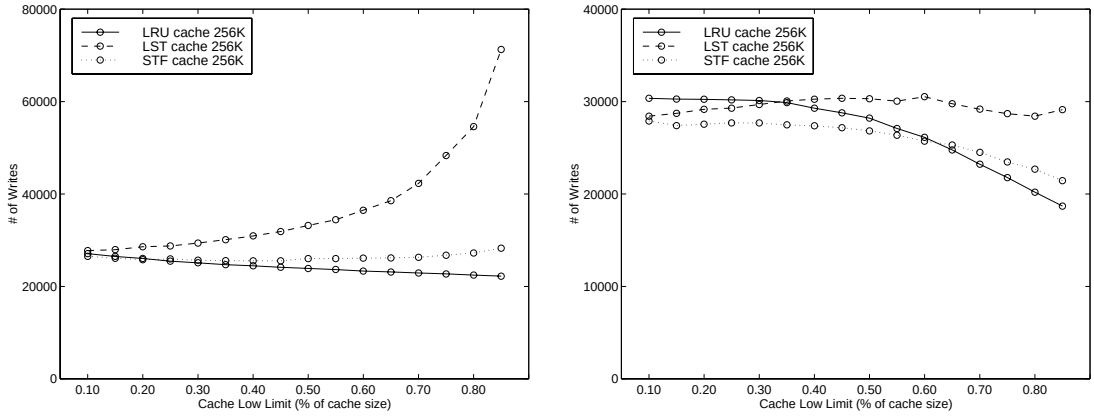
(a) Disk 5 (b) Disk 0

Fig. 2. Number of cache misses for disks with a single threshold cache.



(a) Disk 5 (b) Disk 0

Fig. 3. Number of cache misses for disks with a high and low threshold cache.



(a) Disk 5 (b) Disk 0

Fig. 4. Cache generated write traffic for disks with a high and low threshold cache.

for each cache, all three algorithms perform best when the low limit was set in the 15–20% range. It is more difficult to measure the performance of the write cache based on the number of writes generated to clean the cache. The LRU cache also shows the unusual property that number of writes decrease by almost 5% as the lower limit is raised. The LST and STF algorithms perform well for both disks when the low limit is set in the 15–20% range. The number for both algorithms is not minimal for disk 0, because of the unusual rise and fall in the number of writes.

Table. 2
Write traffic for disk 5 and disk 0 generated by a 256k cache.

	Write Behind	Single Threshold	Dual Threshold
LRU	22194	21155	26039
LST	98117	83010	28589
STF	31773	30227	25791

Disk 5

	Write Behind	Single Threshold	Dual Threshold
LRU	30187	17083	30243
LST	53569	29928	29165
STF	34633	20573	27553

Disk 0

In terms of performance, adding a second threshold causes mixed results. It reduces the number of stalled writes for the LST cache to almost zero, and reduces the number of writes for the LST cache to that of the other algorithms (see Table 2). At the same time, it degrades the performance of the LRU algorithm. Cleaning large portions of the cache benefits the LST cache because it will always use the fewest number of large writes to clean cache space. This same action reduces the number of dirty blocks in the LRU cache whenever cleaning is performed. The LRU algorithm may be able to make better choices when there are more dirty blocks in the cache. In spite of this limitation, the LRU algorithm performs well, confirming that temporal locality is still important.

The performance of the dual threshold caches leaves an open question: is it better to stall less or write to the disk less? We looked at the answer to this question while examining the impact of cache size for the LRU, STF, and LST algorithms. We used the number of cache misses as our best metric for setting high and low thresholds for each cache. High threshold values were set to the values used in our lower bound tests: 90% for the upper thresholds for all caches. Low threshold values were set to 20% for disk 5 and 15% for disk 0. For these experiments, we varied cache size from 128KB to 2MB, doubling memory size for each successive run.

The results from our cache size experiments show that all three management algorithms scale equally, though the head position aware algorithms (LST and STF) may scale slightly better than LRU. Looking at data for the number of writes generated by each cache, the LST, STF, and LRU caches produced almost identical numbers of writes for disk 0, and LRU produced approximately 10% more writes for disk 5. The rates of decrease for STF and LST were were slightly higher than that for LRU for disk 5. Perhaps temporal locality becomes less significant as the cache becomes large for that disk. At that point, algorithms which take advantage of head position may become more useful.

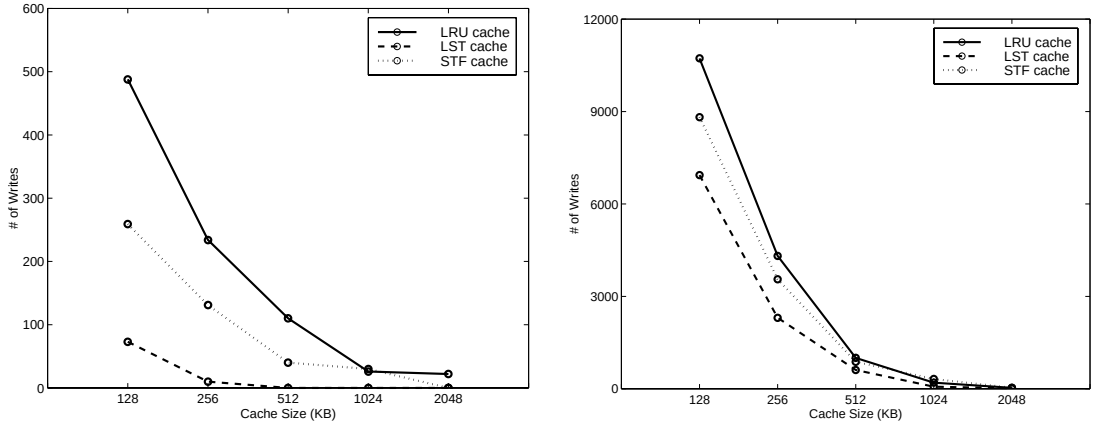
Looking at the mean service and cache miss data, fewer cache misses have a direct and beneficial effect on the service time (see Figures 5 and 6). The LST cache produces the lowest service times for both disks, and produces zero stalled writes with the smallest amount of cache space for both disks. Service times for all trace events quickly converged to an average dominated by the service time of read events for both disks.

Lower service times are only beneficial if they contribute to lower overall response times. While the LST algorithm does produce consistently fewer stalls and better service times, it also tends to write more often to disk. If these additional writes are increasing the amount of time that other events spend queueing for service, then an algorithm other than LST is a better choice.

To check to see if this was happening, we collected queue time information for two sets of events in the disk traces. The results in Figure 7 show mixed results. For disk 5, the STF cache produces mean queue times for write events that are better than either LRU and LST. The set of write events from disk 0 shows that the LST cache has a lower mean queue times than the other caches though the values are close enough not to be significant. The read queue times for disk 5 were abnormally long (for reasons mentioned at the beginning of this section), but, the read queue times for both disks did show trends similar to their write counterparts.

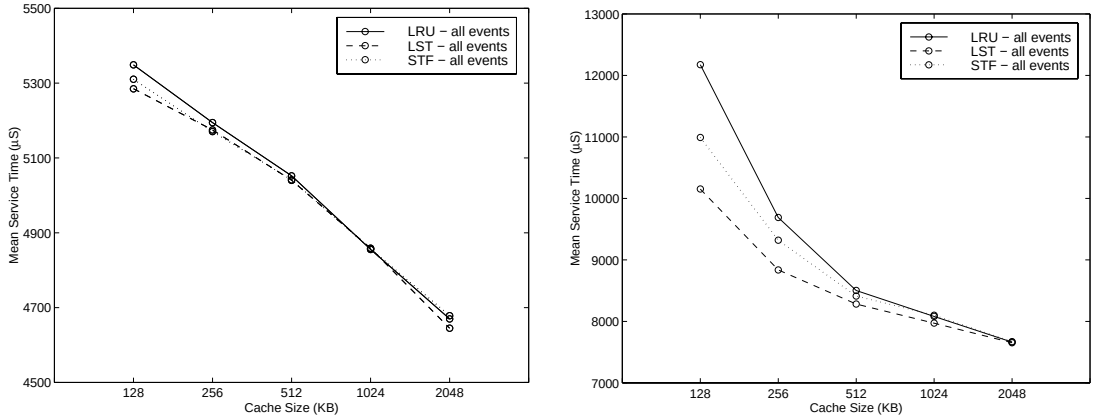
4 Conclusions

We examined several aspects of non-volatile cache management used in conjunction with delayed writes to disk. As disk subsystems continue to lag the performance of ever faster processors, such caches are an increasingly important way to remove an I/O bottleneck in write-dominated systems. Our goal was to compare different cache management techniques by looking at different ways to implement and clean disk write caches of NVRAM. We used trace-driven simulations to obtain information about the number



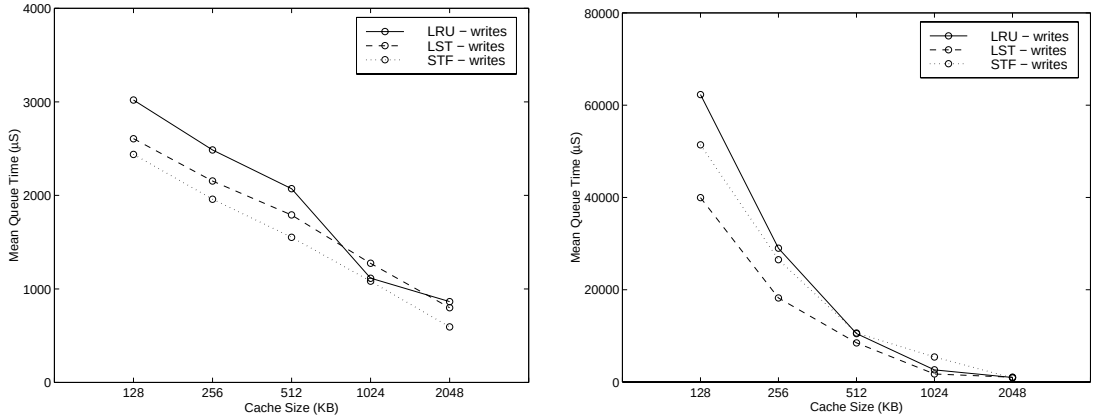
(a) Disk 5 (b) Disk 0

Fig. 5. Write cache misses for disks with different dual threshold write cache sizes.



(a) Disk 5 (b) Disk 0

Fig. 6. Mean service times for disks with different dual threshold write cache sizes.



(a) Disk 5 (b) Disk 0

Fig. 7. Mean write queue times for disks with different dual threshold write cache sizes.

of write cache misses, the mean service times, and the number of writes to disk for two disks.

Our analysis of non-volatiles caches was based on a trace driven simulation of the HP97560 and HP2200 disks. The traces used were those collected by Reummler and Wilkes from the **snake** and **hplajw** trace sets. We presented results for one disk from each set of traces to examine the effectiveness of a write cache with each disk.

Our work with NVRAM caches showed that temporal locality is a key to cache efficiency for many caches, especially small ones. We implemented and tested models of caches using a simple write-behind purging model, and using write-behind with thresholds. We used the least recently used (LRU), largest segment per track (LST) and shortest seek time first (STF) algorithms to decide what data to clean from the cache. Comparison of the data for the number of (cleaned) writes to disk and stalled writes shows that the LRU algorithm works best in many situations. Algorithms which attempt to use head position to determine what to clean next (LST and STF) can produce fewer stalled writes, but generally write to disk more often. For many kinds of caches, the cost of the additional writes outweighs the benefits of fewer stalls.

Our initial work with write-behind caches showed that the LRU managed cache was the most effective, by far. Simple write behind caches had been rejected by others under the assumption that writes to them would frequently stall. Such caches using head position algorithms, particularly LST, performed poorly, sometimes no better than the cache-less disk itself. The LRU managed cache significantly reduced the number of writes to disk and improved response time. The number of stalled writes was large compared to more complex cache management policies, but the policy is very simple and does not need tuning.

We found that one of the few cases where temporal locality is not dominant is when high and low thresholds are used. The large hysteresis of such a cache substantially reduces the number of writes that the LST algorithm must make to clean the cache, making its write performance equivalent to other algorithms. This performance improvement combined with the smaller number of stalled writes creates reduced service times. A study of how these properties scale shows that head position algorithms like LST may perform better than LRU when cache sizes are large.

The major issue we want to explore to continue this work is idle detection. The background cleaner we used in our dual threshold caches is very naive; it does not check to see if the disk is idle before it cleans. We discovered that this had some negative effects for queue size and queue time. The cleaner began to clean the cache shortly before a large burst of writes arrived at least once. None of these writes stalled once they were serviced by the cache, but they were forced to wait which the cache was cleaned. The actual response times for these writes were much larger than they could have been if an idle detector found the disk not to be idle. We hope to use existing work in idle detection and its exploitation to improve disk performance further [9].

References

[1] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Operating Systems Review*, 26(Special issue):10–22, Oct 1992.

[2] Anupam Bhide, Daniel Dias, Nagui Halim, Basil Smith, and Francis Parr. A case for fault-tolerant memory for transaction processing. In *Digest of Papers FCTS-23 23rd International Symposium on Fault-Tolerant Computing*, pages 451–60. IEEE Computer Society Press, Aug 1993.

[3] Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. Trace driven analysis of write caching policies for disks. *Performance Evaluation Review*, 21(1):12–23, Jun 1993.

[4] Prebuddha Biswas, K. K. Ramakrishnan, Don Townsley, and C. M. Krishna. Performance analysis of distributed file systems with non-volatile caches. In *Proceedings of 2nd International Symposium on High Performance Distributed Computing*, pages 252–62. IEEE Computer Society Press, 1993.

[5] Steven Bobrowski. *Oracle7 Server Concepts Manual*. Cooperative Server Technology. Oracle Corporation, 1992.

[6] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: surviving operating system crashes. *SIGPLAN Notices*, 31(9):74–83, Sep 1996.

[7] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 327–35. Morgan Kaufmann, Aug 1989.

[8] Paul A. Fishwick. Simpack: getting started with simulation programming in C and C++. Technical Report 92–022, University of Florida, Department of Computer Science and Information Science and Engineering, Gainesville, FL, Jul 1992.

[9] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *Proceedings of the Winter'95 USENIX Conference*, pages 201–22. USENIX Association, Jan 1995.

[10] D. Hitz, J. Lau, and M. Malcolm. File system design for a NFS server appliance. In *Proceedings of the Winter 1994 USENIX Conference*, pages 235–46. USENIX Association, Jan 1994.

[11] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS–TR94–220, Department of Computer Science, Dartmouth College, 1994.

[12] David E. Lowell and Peter M. Chen. Free transactions with Rio Vista. In *16th Association for Computing Machinery Symposium On Operating Systems Principles*, Oct 1997.

[13] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15. Association for Computing Machinery SIGOPS, Oct 1991.

[14] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, Mar 1994.

[15] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *USENIX Technical Conference Proceedings*, pages 405–20. USENIX, Winter 1993.

[16] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *Operating Systems Review*, 29(5):96–108, Dec 1995.