STOW: A Spatially and Temporally Optimized Write Caching Algorithm

Binny S. Gill
IBM Almaden Research Center

Michael Ko
IBM Almaden Research Center

Biplob Debnath
University of Minnesota

Wendy Belluomini
IBM Almaden Research Center

Abstract

Non-volatile write-back caches enable storage controllers to provide quick write response times by hiding the latency of the disks. Managing a write cache well is critical to the performance of storage controllers. Over two decades, various algorithms have been proposed, including the most popular, LRW, CSCAN, and WOW. While LRW leverages temporal locality in the workload, and CSCAN creates spatial locality in the destages, WOW combines the benefits of both temporal and spatial localities in a unified ordering for destages. However, there remains an equally important aspect of write caching to be considered, namely, the rate of destages. For the best performance, it is important to destage at a steady rate while making sure that the write cache is not under-utilized or over-committed. Most algorithms have not seriously considered this problem, and as a consequence, forgo a significant portion of the performance gains that can be achieved.

We propose a simple and adaptive algorithm, STOW, which not only exploits both spatial and temporal localities in a new order of destages, but also facilitates and controls the rate of destages effectively. Further, STOW partitions the write cache into a sequential queue and a random queue, and dynamically and continuously adapts their relative sizes. Treating the two kinds of writes separately provides for better destage rate control, resistance to one-time sequential requests polluting the cache, and a workload-responsive write caching policy.

STOW represents a leap ahead of all previously proposed write cache management algorithms. As anecdotal evidence, with a write cache of 32K pages, serving a 4+P RAID-5 array, using an SPC-1 Like Benchmark, STOW outperforms WOW by 70%, CSCAN by 96%, and LRW by 39%, in terms of measured throughput. STOW consistently provides much higher throughputs coupled with lower response times across a wide range of cache sizes, workloads, and experimental configurations.

1 Introduction

In spite of the recent slowdown in processor frequency scaling due to power and density issues, the advent of multi-core technology has enabled processors to continue their relentless increase in I/O rate to storage systems. In contrast, the electro-mechanical disks have not been able to keep up with a comparable improvement in access times. As this schism between disk and processor speeds widens, caching is attracting significant interest.

Enterprise storage controllers use caching as a fundamental technique to hide I/O latency. This is done by using fast but relatively expensive random access memory to hold data belonging to slow but relatively inexpensive disks.

Over a period of four decades, a large number of read cache management algorithms have been devised, including Least Recently Used (LRU), Frequency-Based Replacement (FBR) [19], Least Frequently Recently Used (LFRU) [15], Low Inter-Reference Recency Set (LIRS) [13], Multi-Queue (MQ) [24], Adaptive Replacement Cache (ARC) [17], CLOCK with Adaptive Replacement (CAR) [2], Sequential Prefetching in Adaptive Replacement Cache (SARC) [8], etc. While, the concept of a write cache has been around for over two decades, we realize that it is a more complex and less studied problem. We focus this paper on furthering our understanding of write caches and improving significantly on the state of the art.

1.1 What Makes a Good Write Caching Algorithm?

A write-back (or fast-write) cache relies on fast, non-volatile storage to hide latency of disk writes. It can contribute to performance in five ways. It can (i) harness temporal locality, thereby reducing the number of pages that have to be destaged to disks; (ii) leverage spatial locality, by reordering the destages in the most

disk-friendly order, thereby reducing the average cost of destages; (iii) absorb write bursts from applications by maintaining a steady and reasonable amount of free space, thereby guaranteeing a low response time for writes; (iv) distribute the write load evenly over time to minimize the impact to concurrent reads; and (v) serve read hits that occur within the write cache.

There are two critical decisions regarding destaging in write caching: the *destage order* and the *destage rate*. The destage order deals with leveraging temporal and spatial localities, while the destage rate deals with guaranteeing free space and destaging at a smooth rate. Write caching has so far been treated mainly as an eviction problem, with most algorithms focusing only on the destage order. The most powerful write caching algorithms will arise when we explore the class of algorithms that simultaneously optimize for both the destage order and the destage rate.

1.2 Our Contributions

Firstly, we present a detailed analysis of the problem of managing the destage rate in a write cache. While this has remained a relatively unexplored area of research, we demonstrate that it is an extremely important aspect of write caching with the potential of significant gains if done right. Further, we show that to manage the destage rate well, we actually need a new destaging order.

Secondly, we present a Spatially and Temporally Optimized Write caching algorithm (STOW), that for the first time, exploits not only temporal and spatial localities, but also manages both the destage rate and destage order effectively in a single powerful algorithm that handsomely beats popular algorithms like WOW, CSCAN, and LRW, across a wide range of experimental scenarios. Anecdotally, with a write cache of 32K pages (and high destage thresholds), serving a RAID-5 array, the measured throughput for STOW at 20 ms response time, outperform WOW by 70%, CSCAN by 96%, and LRW by 39%. STOW consistently and significantly outperforms all other algorithms across a wide range of cache sizes, workload intensities, destage threshold choices, and backend RAID configurations.

1.3 Outline of the paper

In Section 2, we briefly survey previous related research. In Section 3, we explore why the destage rate is a crucial aspect of any good write caching algorithm. In Section 4, we present the new algorithm STOW. In Section 5, we describe the experimental setup and workloads, and in Section 6, we present our main quantitative results. Finally, in Section 7 we conclude with the main findings of this paper.

2 Related Work

Although an extensive amount of work has been done in the area of read caching, not all techniques are directly applicable to write caching. While read caching is essentially a two dimensional optimization problem (maximizing hit ratio and minimizing prefetch wastage), write caching is a five dimensional optimization problem (as explained in Section 1.1).

In a write-back cache, the response time for a write is small if there is space in the write cache to store the new data. The data in the write cache is destaged to disks periodically, indirectly affecting any concurrent reads by increasing their average service response time. To reduce the number of destages from the write cache to the disks, it is important to leverage temporal locality and, just like in read caches, maximize the hit (overwrite) ratio. The primary way to maximize temporal locality is to attempt to evict the least recently written (LRW) pages from the cache. An efficient approximation of this is available in the CLOCK [5] algorithm which is widely used in operating systems and databases. These algorithms, however, do not account for the spatial locality factor in write cache performance.

Orthogonally, the order of destages can be chosen so as to minimize the average cost of each destage. This is achieved by destaging data that are physically proximate on the disks. Such spatial locality maximization has been studied mostly in the context of disk scheduling algorithms, such as shortest seek time first (SSTF) [6], SCAN [6], cyclical SCAN (CSCAN) [20], LOOK [18], VSCAN [7], and FSCAN [4]. Some of these require knowledge of the current state of the disk head [12, 21], which is either not available or too cumbersome to track in the larger context of storage controllers. Others, such as CSCAN, order the destages by logical block address (LBA) and do not rely on knowing the internal state of the disk.

In the first attempt to combine spatial and temporal locality in write caching for storage systems [10], a combination of LRW [1, 3, 11] and LST [10, 22] was used to balance spatial and temporal locality. This work had the drawback that it only dealt with one disk and it did not adapt to the workload.

In general, the order of destages is different for leveraging temporal locality versus spatial locality. One notable write caching algorithm, Wise Ordering for Writes (WOW) [9], removed this apparent contradiction, by combining the strength of CLOCK [5] in exploiting temporal locality and Cyclical SCAN (CSCAN) [20] in exploiting spatial locality. As shown in Figure 1, WOW groups the pages in the cache in terms of write groups and sorts them in their LBA order. To remember if a write group was recently used, a recency bit is main-

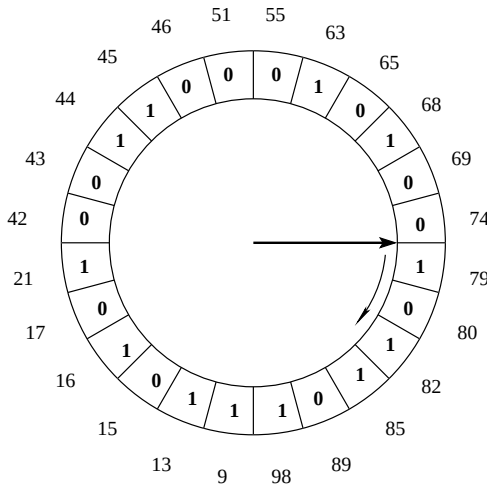


Figure 1: The data structure of the WOW algorithm

tained in each write group. When a page is written in a write group already present in the cache, the recency bit of the write group is set to 1. Destages are done in the sorted order of the write groups. However, if the recency bit of a write group is 1, the write group is bypassed after resetting the recency bit to 0.

While WOW solves the *destage order* problem, the *destage rate* problem has attracted little research. Both WOW and an earlier work on destage algorithms [23] use a linear thresholding scheme that grows the rate of destages linearly as the number of modified pages in the cache grows. While this scheme is quite robust, destage orders like WOW and CSCAN cannot achieve their full potential due to a destructive interaction between the destage rate and the destage order policies. We are unaware of any research that studies the interaction between the two vital aspects of write caching: the destage order and the destage rate.

	CSCAN	LRW	WOW	STOW
Spatial Locality	Yes	No	Yes	Yes
Temporal Locality	No	Yes	Yes	Yes
Scan Resistance	No	No	Little	Yes
Stable Destage Rate	No	Little	No	Yes
Stable Occupancy	No	Little	No	Yes

Table 1: Comparison of Various Write Cache Algorithms

Table 1 shows how the set of algorithms discussed above compare. LRW considers only recency (temporal locality), CSCAN considers only spatial locality, and WOW considers both spatial and temporal locality. Our algorithm, STOW (Spatially and Temporally Optimized Writes), tracks spatial and temporal localities, and is *scan resistant* because it shields useful random data from being pushed out due to the influx of large amounts sequential data. STOW also avoids large fluctuations in the destage rate and cache occupancy.

3 Taming the Destage Rate

Historically caching has always been treated as an eviction problem. While it might be true for demand-paging read caches, it is only partially true for write caches. The rate of eviction or the destage rate has attracted little research so far. In this section we explore why the destage rate is a crucial aspect of any good write caching algorithm.

3.1 The Goals

Any good write caching algorithm needs to manage the destage rate to achieve the following three objectives: (i) Match the destage rate (if possible) to the average incoming rate to avoid hitting 100% full cache condition (leading to synchronous writes); (ii) Avoid underutilizing the write cache space; (iii) Destage smoothly to minimally impact concurrent reads.

3.2 Tutorial: How to Get it Wrong?

Rather than simply present our approach, we explain why we reject other seemingly reasonable approaches, some of which have been used in the past.

3.2.1 Ignore Parity Groups

More often than not, a write cache in a storage controller serves a RAID array involving parity groups (e.g. RAID-5, RAID-6). In such scenarios, it is important to group together destages of separate pages within the same parity group to minimize the number of parity updates. The best case happens when all members of the parity group are present in the cache. The parity group can optionally be extended to beyond one parity stripe or even to RAID-10 (see WOW [9]).

3.2.2 Destage Quickly

One approach is to destage as soon as there are dirty pages and as fast as the system would allow. While this would guarantee that the cache stays away from the full-cache condition most effectively even for strong workloads, it wipes out any temporal and spatial locality benefits for gentler workloads. The left panel in Figure 2 shows that a quick destaging policy can lead to very low cache occupancy.

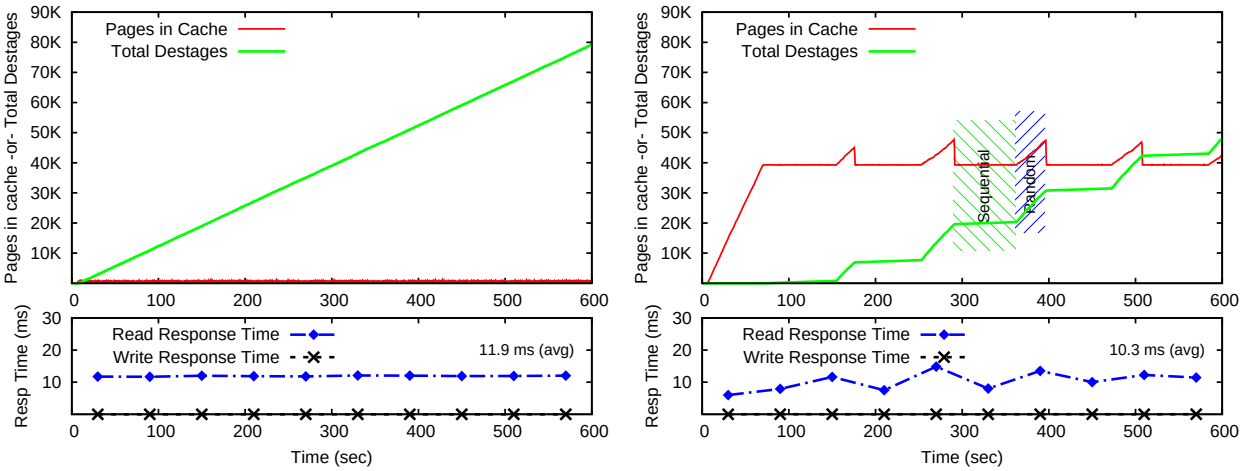


Figure 2: Left panel: We destage in the order specified by the WOW algorithm, and as fast as the disks allow. The cache remains relatively empty and the observed read response time is around 11.9 ms on average. Right panel: We destage in the WOW order, however we destage only if the cache is more than 60% full (out of 64K pages). We observe a better average read response time, but the cache occupancy exhibits spikes when random data is destaged. When sequential data is destaged, the disks are under-utilized as shown by the flat steps in the total destages curve.

3.2.3 Fixed Threshold

In the right panel of Figure 2, we examine a policy which destages quickly only if the cache is more than 60% full. This performs better on average but displays ominous "spikes" in cache occupancy which correspond to spikes in the read response times. The non-uniform destage rate is bad for concurrent reads. Furthermore, a higher fixed threshold is more likely to hit 100% cache occupancy, while a lower fixed threshold underutilizes the write cache most of the time.

The Spikes: Any write caching algorithm destages writes in a particular order. If the workload is such that the algorithm destages sequential data for some time followed by random data, such peaks are inevitable because destaging random data is far more time consuming and during such intervals, the cache occupancy can spike even for steady workloads. With WOW or CSCAN, such spikes appear when the workload has a sequential and a random component that target different portions of the LBA space. This is commonly observed in both real-life and benchmark workloads.

3.2.4 Linear Threshold

In the WOW [9] paper, a linear threshold scheme is proposed which is better than the fixed threshold scheme because it provides a gradual gradation of destage rates which is more friendly to concurrent reads and also allows the thresholds to be safely closer to the 100% mark. For example, instead of destaging at full force after reaching 60% occupancy, linear threshold would use a number of concurrent destage requests that is proportional to how close the write cache is to a high threshold (say 80% occupancy). Beyond the high threshold it will destage at the full rate. This is the best scheme so far that we are aware of, however, it cannot address the spike problem, as evidenced in Figure 3. In this paper we use an H/L notation for the thresholds: e.g. $90/80$ implies that the high threshold is at 90% of the cache and the low threshold is at 80%.

3.2.5 Adaptive Threshold

One might suggest that we should develop a scheme that adaptively determines the correct high and low thresholds for the linear threshold scheme. However, the reader will observe that the spikes are very tall, and any such effort will result in a serious underutilization of the write cache space.

3.3 Maybe Not WOW Then?

While the order of destages proposed in the WOW algorithm is disk-friendly, the same order makes the destage rate problem a tough nut to crack. We propose to change the destage order to allow us to tame the destage rate.

3.3.1 Separate Random and Sequential Data

The spikes in cache occupancy are caused by long alternating regimes of sequential and random destages. We separate the sequential data and the random data into two WOW-like data-structures in the write cache. Whenever there is a need to destage, we destage from the larger of the two queues, as a first approximation. Later, we shall

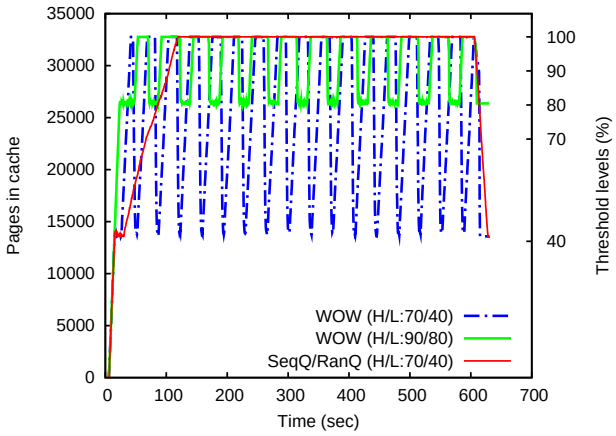


Figure 3: We examine cache occupancy for a workload with both sequential and random writes when using the linear thresholding scheme to determine destage rate. WOW exhibits spikes which go down up to the low threshold when sequential data is destaged, and rise when random data is destaged. For higher thresholds, WOW hits cache full condition more often. If we destage sequential and random data together (SeqQ/RanQ), we elimintate the spikes, but the cache is almost always full.

see that the ideal partitioning of the cache requires adapting to the workload. While the intermixing of destages from the sequential and the random queues eliminates the spikes beautifully (the SeqQ/RanQ curve in Figure 3), it pollutes the spatial locality in the destage order by sending the disk heads potentially to two separate regions on the disk causing longer seek times. The SeqQ/RanQ variant suffers from the full-cache condition almost all the time, nullifying any gains from eliminating the spikes.

3.3.2 Use Hysteresis in Destaging

We need to be able to control the spikes in cache occupancy without derailing the spatial locality of the destages. We discovered that if we destage no less than a fixed hysteresis amount from the larger queue, we reduce the negative impact of having two destage sources. This brings us to the STOW algorithm which integrates all our intuitions so far (and some more) into a powerful, practical and simple write caching algorithm.

4 STOW: The Algorithm

4.1 The STOW Principle

The STOW algorithm uses two WOW-like sorted circular queues for housing random and sequential data separately. The relative sizes of the queues is continuously adapted according to workload to maximize performance. The decisions for: which queue to destage from, when to destage, and at what rate, are carefully managed to leverage spatial and temporal locality in the workload, as well as to maintain steady cache occupancy levels and destage rates. Despite its simplicity, STOW represents the most comprehensive and powerful algorithm for write cache management.

4.2 Data Structures

4.2.1 Honoring Parity Groups

A write hit (over-write) on a page generally implies that the page and its neighbors are likely to be accessed again. In the context of a storage controller connected to a RAID controller, we would like to postpone the destage of such pages hoping to absorb further writes, thereby reducing the destage load on the disks.

In RAID, each participating disk contributes one strip (e.g. 64 KB) towards each RAID stripe. A RAID stripe consists of logically contiguous strips, one from each data disk. Destaging two distinct pages in the same RAID stripe together is easier than destaging them separately, because in the former case the parity strip needs to be updated only once. We say a **stripe hit** has occurred when a new page is written in a RAID stripe that already has a member in the write cache. In RAID 5, therefore, a stripe hit saves two disk operations (the read and write of the parity strip), while a hit on a page saves four.

While we divide the write cache in pages of 4KB each, we manage the data structures in terms of **write groups**, where a write group is defined as a collection of a fixed number (one or more) of logically consecutive RAID stripes. In this paper, we define the write group to be equal to a RAID stripe. We say a write group is present in the cache if at least one of its member pages is physically present in the cache. Managing the cache in terms of such write groups allows us to better exploit both temporal locality by saving repeated destages and spatial locality by issuing writes in the same write group together, thus minimizing parity updates.

4.2.2 Separating Sequential From Random

Sequential data, by definition, has high spatial locality, and appears in large clumps in the sorted LBA space for algorithms such as WOW or CSCAN. In between clumps of sequential data are random data. We discovered that destaging random data, even when it is sorted, is far more time consuming than destaging sequential data. Therefore, when the destage pointer is in an area full of random data, the slower destage rate causes the cache occupancy to go up. These spikes in the cache occupancy could lead to full-cache conditions even for steady workloads, severely impacting the cache performance. Fur-

ther, during a spike, a cache is especially vulnerable to reaching the 100% occupancy mark even with smaller write bursts. This discovery led us to partition the cache directory in STOW into two separate queues RanQ and SeqQ for the "random" and the "sequential" components of a workload. It is easy to determine whether a write is sequential by looking for the presence of earlier pages in the cache [8] and keeping a counter. The first few pages of a sequential stream are treated as random. If a page is deemed to belong to a sequential stream it is populated in SeqQ; otherwise, the page is stored in RanQ.

Separating sequential data from random data is a necessary first step towards alleviating the problem caused by the spikes in the cache occupancy. However, this is not sufficient by itself, as we will learn in Section 4.3.4 when we discuss destaging.

4.2.3 Two Sorted Circular Queues

The STOW cache management policy is depicted in Algorithm 1. In each queue, RanQ or SeqQ, the write groups are arranged in an ascending order of logical block addresses (LBA), forming a circular queue, much like WOW, as shown in Figure 4. A destage pointer (akin to a clock arm) in each queue, points to the next write group in the queue to be considered for destaging to disk. Upon a page miss (a write that is not an over-write), if the write group does not exist in either queue, the write group with the new page is inserted in the correct sorted order in either RanQ or SeqQ depending on whether the page is determined to be a random or a sequential page. If the write group already exists, then the new page is inserted in the existing write group.

The LBA ordering in both queues allows us to minimize the cost of a destage which depends on how far the disk head would have to seek to complete an operation. The write groups, on the other hand, allow us to exploit any spatio-temporal locality in the workload, wherein a write on one page in a write group suggests an imminent write to another page within the same write group. Not only do we save on parity updates, but we also have the opportunity to coalesce consecutive pages together into the same write operation.

When either RanQ or SeqQ is empty, because the workload lacks the random or the sequential component, then STOW converges to WOW, and by the same token, is better than CSCAN and LRU.

Algorithm 1 STOW: Cache Management Policy

Write page x in write group g:

```
1:  if g ∈ RanQ ∪ SeqQ then //a write group hit
2:     if x ∉ RanQ ∪ SeqQ then //page miss
3:        allocate x from FreePageQueue
4:        insert x in g
5:     end if
6:     if x is sequential and is last page of g then
7:        set recency-bit of g to 0
8:     else
9:        set recency-bit of g to 1
10:    end if
11: else
12:    allocate g from FreeWriteGroupQueue
13:    allocate x from FreePageQueue
14:    insert x into g
15:    if x is sequential then
16:       insert g into sorted queue in SeqQ
17:       if x is last page of g then
18:          set recency-bit of g to 0
19:       else
20:          set recency-bit of g to 1
21:       end if
22:    else
23:       insert g into sorted queue in RanQ
24:       set recency-bit of g to 0
25:    end if
26: end if
```

4.3 Operation

4.3.1 What to Destage From a Queue?

The destage pointer traverses the sorted circular queue looking for destage victims. Write groups with a recency bit of 1 are skipped after resetting the recency bit to 0. If the recency bit of the write group was found to be 0, then the pages present in the write group are destaged. Thus, write groups with a recency bit of one get an extra life equal to the time it takes for the destage pointer to go around the clock once.

Setting the recency bit in RanQ: When a new write group is created in RanQ, the recency bit is set to 0 (line 24 in Algorithm 1). On a subsequent page hit or a write group hit, the recency bit is set to 1 (line 9), giving all present members of the write group a longer life in the cache, during which they can exploit any overwrites of the present pages or accumulate new neighboring pages in the same write group. This leads to enhanced hit ratio, fewer parity updates, and coalesced writes reducing the total number of destages.

Setting the recency bit in SeqQ: Whenever a page is written to SeqQ, the recency bit in the corresponding write group is set to 1 (lines 9 and 20). This is because we anticipate that subsequent pages will soon be written to the write group by the sequential stream. Destaging to disk is more efficient if the whole write group is present since this avoids the extra read-modify-write of the parity group in a RAID-5 configuration and also coalesces consecutive pages into the same disk IO if possi-

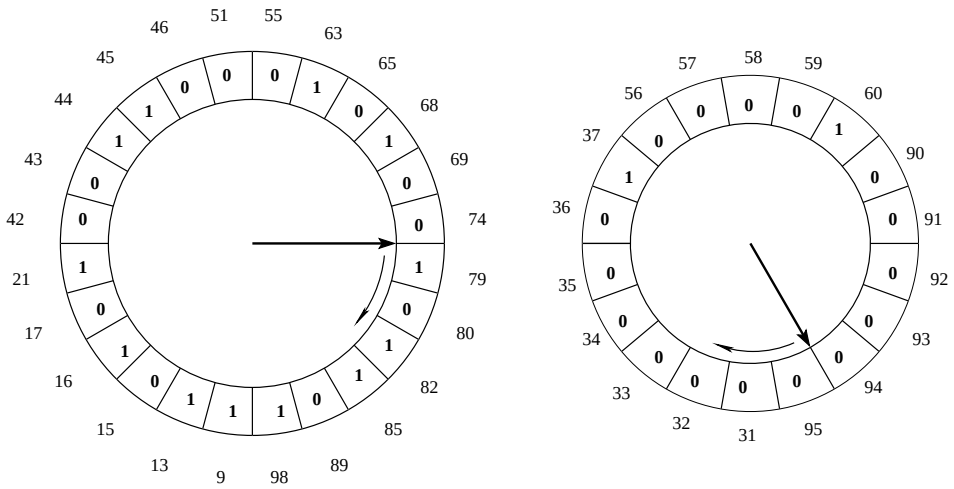


Figure 4: The data structure of the STOW algorithm

ble. However, if the page written is the last page of the write group then the recency bit is forced to 0 (lines 7 and 18). Since the last page of the write group has been written, we do not anticipate any further writes and can free up the cache space the next time the destage pointer visits this write group.

4.3.2 What Rate to Destage at?

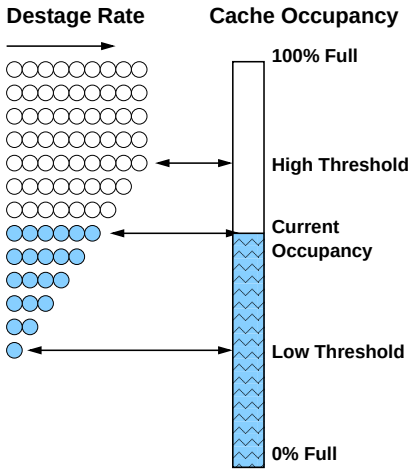


Figure 5: Linear threshold scheme for determining destage rate based on how close the cache occupancy is to the high threshold.

In STOW, we use a linear threshold scheme (see Figure 5, as described in WOW[9]) to determine when and at what rate to destage. We set a low threshold and a high threshold for the cache occupancy. When the cache occupancy is below the low threshold, we leave the write data in the cache in order to gather potential write hits. When the cache occupancy is above the high threshold, we destage data to disks at the full rate in order to avoid the full-cache condition which is detrimental to response time. When the cache occupancy is between the low and high threshold, the number of concurrent destage requests is linearly proportional to how close we are to the high threshold. Note, a higher number of concurrent destage requests to the disks results in a higher throughput for destages, but of course at the cost of making the disks busier and the concurrent reads slower. The maximum number of concurrent destage requests (queue depth) is set to a reasonable 20 [9] in our experiments.

4.3.3 Which Queue to Destage From?

Algorithm 3 shows how STOW calculates and adapts the desired size of SeqQ (DesiredSeqQSize). Algorithm 2 destages from SeqQ if it is larger than DesiredSeqQSize, else, it destages from RanQ (line 3). While strictly following this simple policy eliminates any deleterious spikes in the cache occupancy, it is not optimal because it sends the disk heads to possibly two distinct locations (the sorted order from RanQ and from SeqQ) simultaneously, resulting in an inter-mingling of two sorted orders, polluting the spatial locality in the destages.

Once we have decided to destage from a queue, we should stick with that decision for a reasonable amount of time, so as to minimize the spatial locality pollution caused by the mixing of two sorted orders. To realize this, we define a fixed number called the HysteresisCount. Once a decision has been made to destage from a particular queue, we continue destaging from the same queue until, (i) we have destaged Hystere-

Algorithm 2 STOW: Destage Policy

```
1:  while needToDestage() do
2:    if hysteresisCountDone() then
3:      if |SeqQ| > DesiredSeqQSize then
4:        set currentDestagePtr to SeqQDestagePtr
5:      else
6:        set currentDestagePtr to RanQDestagePtr
7:      end if
8:    end if
9:    g = write group pointed to by currentDestagePtr
10:   while g->recency-bit == 1 do
11:     g->recency-bit = 0
12:     g = advanceDestagePtr(currentDestagePtr)
13:   end while
14:   destage all pages in g
15:   move destaged pages to FreePageQueue
16:   move g to FreeWriteGroupQueue
17:   advanceDestagePtr(currentDestagePtr)
18: end while
```

Algorithm 3 STOW: Queue Size Management Policy

```
1:  if page x in write group g is written then
2:    if g ∈ RanQ then //(RAID-10: use x ∈ RanQ)
3:      if g->recency-bit == 0 then
4:        if (|SeqQ| - DesiredSeqQSize) <
              HysteresisCount then
5:          DesiredSeqQSize -= 1
6:        end if
7:      end if
8:    end if
9:  end if
10: if write group g is destaged then
11:   if g not contiguous with previous destage then
12:     if previous stretch <queue depth then
13:       if |RanQ| / (|RanQ| + |SeqQ|) >
              RanRq / (RanRq + SeqRq) then
14:         DesiredSeqQSize += n*|RanQ|/|SeqQ|
15:         //n = num of disks in RAID5 or RAID10
16:       end if
17:     end if
18:   end if
19: end if
```

sisCount pages from the queue; or (ii) either queue has since grown by more than HysteresisCount pages. Note that destages from RanQ are slower and the second condition avoids a large buildup in SeqQ in the meantime. Once either condition is met, we reevaluate which queue to destage from.

Normally, we fix HysteresisCount to be equal to 128 times the number of spindles in the RAID array. This ensures that a reasonable number of destage operations are performed in one queue's sorted order, before moving to the other queue's sorted order. However, we observed that fluctuations in the cache occupancy are proportional to the HysteresisCount. To maintain a smooth destage rate these fluctuations need to be small relative to the difference between the high and low thresholds. Therefore, we limit HysteresisCount to be no more than $1/8$th of the difference (in terms of pages) between the thresholds.

4.3.4 Adapting the Queue Sizes

As we stated earlier, we use the size of SeqQ relative to DesiredSeqQSize for determining which queue to destage from, every time we have destaged HysteresisCount pages. Therefore, we need to wisely and continuously adapt DesiredSeqQSize to be responsive to the workload so as to maximize the aggregate utility of the cache. The marginal utility, in terms of IOPS gained, of increasing the size of either RanQ or SeqQ, is not well understood. Therefore, we propose intuitive heuristics that are very simple to calculate and result in good performance.

Marginal utility for RanQ: We would like to estimate the extra IOs incurred if we make RanQ smaller by unit cache size (a page). We first approximate the number of misses that would be incurred if we reduce the size of RanQ. Let h be the hit rate for first time hits in RanQ (where the recency bit is previously zero). We consider only page hits for RAID-10 but any stripe hit for RAID-5, since in RAID-5, stripe hits save parity updates (two IOs) and are more common than page hits. Assuming a uniform distribution of these hits, we can compute the density of hits to be $h/|RanQ|$. Since a cache does have diminishing returns as its size grows, we add a factor of 0.5 (empirically determined). Each extra miss results in two extra IOs to the disk, yielding a marginal utility of $h/|RanQ|$.

Marginal utility for SeqQ: We would like to estimate the extra IOs incurred by making SeqQ smaller by unit cache size. We first measure the rate, s, at which there are breaks in the logical write group addresses being destaged from SeqQ. Each contiguous group of pages destaged is called a *stretch*. The smaller the size of SeqQ, the higher is the rate s. Since s is inversely proportional to the cache size, the marginal increase in s equal to $s/|SeqQ|$ (since $s*|SeqQ|$ = const, $\frac{ds}{d|SeqQ|} = -\frac{s}{|SeqQ|}$). Each extra break in SeqQ results in one extra write to all n disks. This yields a marigical utility of $n * s/|Seq|$.

We adapt the sizes of RanQ and SeqQ targeting a condition where $h/|RanQ| = n * s/|SeqQ|$, to minimize the IOs to the disk, maximizing the performance of the cache.

We implement the above in Algorithm 3 as follows:

Initialization: The initial value of DesiredSeqQSize

is the size of SeqQ when the write cache first reaches the low threshold of destaging.

Decrement: DesiredSeqQSize is reduced by one if: We have a hit (page hit for RAID-10 and write group hit for RAID-5) in RanQ (line 2), where the recency bit is zero (line 3), and the DesiredSeqQSize is not already HysteresisCount lower than the current SeqQ size (line 4).

Increment: DesiredSeqQSize is incremented whenever there is break in the logical addresses of the write groups in SeqQ being destaged (line 11). The amount incremented is $n * |RanQ|/|SeqQ|$, where n is the number of disks in the RAID array. There are two conditions when we do not increment DesiredSeqQSize: (i) When the break in the logical address occurred after a relatively long *stretch* (more than what the queue depth allows to be destaged together) (line 12); or (ii) RanQ is already below its rightful share of the cache based on the proportion of random requests in the workload (line 13).

5 Experimental Setup

A schematic diagram of the experimental system is depicted in Figure 6.

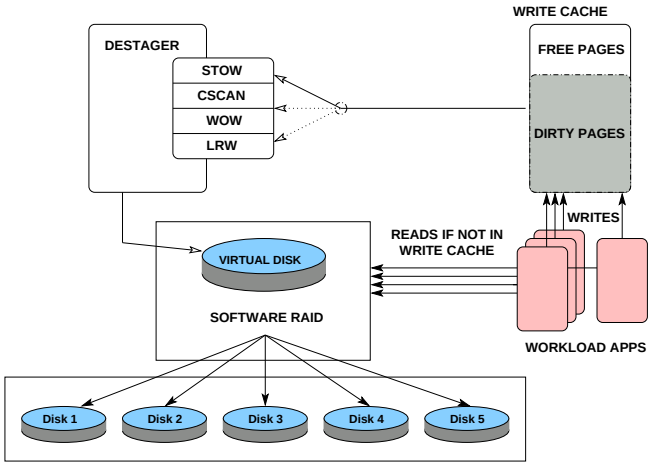


Figure 6: Overall design of the experimental system

5.1 The Basic Hardware Setup

We use an IBM xSeries 3650 machine equipped with two Intel Xeon 3 GHz processors, 4 GB DDR2 memory at 667 MHz, and eight 2.5" 10K RPM SAS disks (IBM 40K1052, 4.5 ms avg. seek time) of 73.4 GB each. A Linux kernel (version 2.6.23) runs on this machine to host all our applications and standard workload generators. We employ five SAS disks for our experiments, and one for the operating system, our software, and workloads.

5.2 Storage Configuration

We study two popular RAID configurations, viz. RAID-5 and RAID-10, using Linux Software RAID. We issue direct I/O to the virtual RAID disk device, always bypassing the kernel buffer. For RAID-5, we use 5 SAS disks to create an array consisting of 4 data disks and 1 parity disk. We choose the strip size for each disk to be 64 KB, with the resulting stripe group size being 256 KB. For RAID-10, we use 4 SAS disks to create an array in a 2 + 2 configuration. We use the same strip size of 64 KB for each disk.

We use the entire available storage in one configuration which we call the **Full Backend**. For RAID-5, with the storage capacity of four disks, Full Backend amounts to 573 million 512-byte sectors. For RAID-10, with the storage capacity of two disks, Full Backend amounts to 286 million 512-byte sectors. We also define a **Partial Backend** configuration, where we use only 1/100th of the available storage. While *Full Backend* is characterized by large disk seeks and low hit ratio, the *Partial Backend* generates only short seeks coupled with high hit ratios.

5.3 The Cache

For simplicity, we use volatile DDR2 memory as our write cache. In a real life storage controller, the write cache is necessarily non-volatile (e.g. battery-backed). In our setup, the write cache is managed outside the kernel so that its size can be easily varied allowing us to benchmark a wide range of write cache sizes.

We do not use a separate read cache in our experiments for the following reason. Read misses disrupt the sequentiality of destages determined by any write caching algorithm. A read cache reduces the read misses and amplifies the gains of the better write caching algorithm. Therefore the most adverse environment for a write caching algorithm is when there is no read cache. This maximizes the number of read misses that the disks have to service concurrent to the writes and provides the most valuable comparison of write caching algorithms.

Nevertheless, we do service read hits from the write cache for consistency.

5.4 SPC-1 Benchmark

SPC-1 [16, 14] is the most respected performance benchmark in the storage industry. The benchmark simulates real world environments in a typical server class computer system by presenting a set of I/O operations that are typical for business critical applications like OLTP systems, database systems and mail server applications. We use a prototype implementation of the SPC-1 benchmark that we refer to as SPC-1 Like.

The SPC-1 workload roughly consists of 40% read requests and 60% write requests. For each request, there is a 40% chance that the request is sequential and a 60% chance that the request is random with some temporal locality. SPC-1 scales the footprint of the workload based on the amount of storage space specified. Therefore for a given cache size, the number of read and write hits will be larger if the backend is smaller (*Partial Backend*), and smaller if the amount of storage exposed to the benchmark is larger (*Full Backend*).

SPC-1 assumes three disjoint application storage units (ASU). ASU-1 is assigned 45% of the available back-end storage and represents "Data Store". ASU-2 is assigned 45% and represents "User Store". The remaining 10% is assigned to ASU-3 and represents "Log/Sequential Write". In all configurations, we lay out ASU-3 at the outer rim of the disks followed by ASU-1 and ASU-2.

6 Results

We compare the performance of LRW, CSCAN, WOW and STOW under a wide range of cache size, workload, and backend configurations. We use linear thresholding to determine the rate of destages for all algorithms.

6.1 Stable Occupancy and Destage Rate

6.1.1 Full Backend

In Figure 7(a), we observe that the occupancy graph for WOW as well as CSCAN fluctuates wildly between the low threshold and the 100% occupancy level. For the same scenario, LRW's cache occupancy remains at 100% occupancy, which implies that most of the time it does not have space for new writes. Only STOW exhibits measured changes in the overall cache occupancy, consistently staying away from the full-cache condition. Note that with linear thresholding, the destage rate is a function of the cache occupancy, and consequently, large fluctuations are detrimental to performance.

The sequential writes in the SPC-1 benchmark are huddled in a small fraction of the address space. As the destage pointer in WOW or CSCAN moves past this sequential region and into the subsequent random region, the occupancy graph spikes upwards because the cache cannot keep up with the incoming rate while destaging in the random region. This disparity can be so large that even the maximum destage concurrency may not be sufficient to keep up with the incoming rate, leading to the dreaded full-cache condition (Figure 7(a)).

Also note the flat bottoms at the low threshold on the occupancy graphs for WOW and CSCAN in Figure 7(a). Since destaging sequential data is quick and easy, the cache occupancy quickly drops down close to the low threshold, where it uses only a small portion of the allowed destage queue depth to keep up with the incoming rate of the overall workload. The lackadaisical destage rate in the sequential region results in underutilization of disks which is ironic given that the disks cannot keep up with the incoming rate when destages move to the subsequent random regions.

6.1.2 Partial Backend

In Figure 7(b), we observe that the fluctuations in WOW are compressed together. This is because the higher hit ratio causes the destage pointer in WOW to advance much more quickly. CSCAN, on the other hand, does not skip over recently hit pages, and produces less frequent fluctuations but stays in the full-cache condition most of the time.

STOW wisely alternates between the two types of destages, ensuring that the disks are continuously utilized at a relatively constant rate. This eliminates large fluctuations in the occupancy curve for STOW in both the full and partial backend cases.

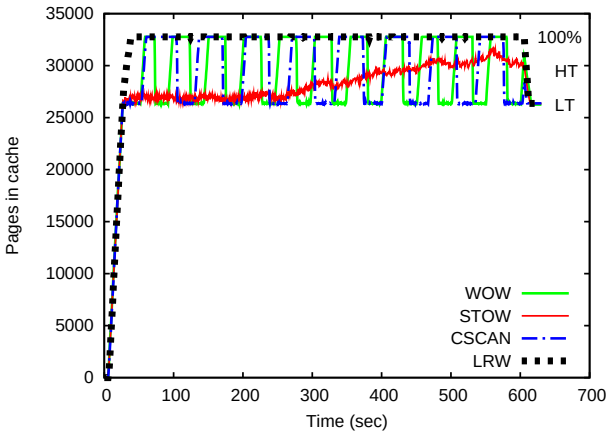
6.2 Throughput and Response Time

Presenting meaningful results: We use the best way to present results for write caching improvements: the throughput-response time curve. We present gains in terms of bandwidth improvements at the same (reasonable) response time. Another approach is to present gains in terms of response times at the same throughput. E.g., at 900 IOPS in Figure 8(a), we could report at least a 10x improvement in response time over the contenders. While it is accurate, we believe, it is not as informative because it can be cherry-picked to aggrandize even modest gains in such "hockey-stick" plots.

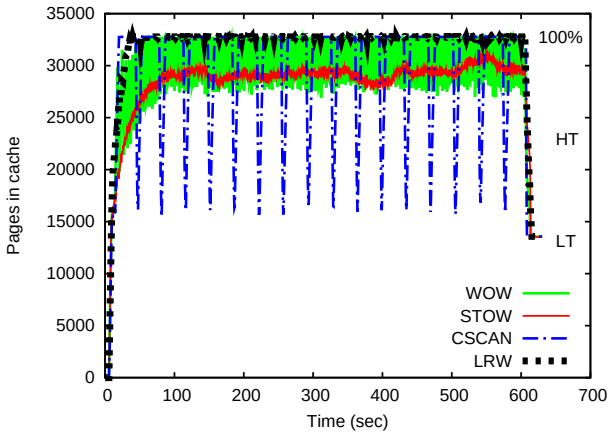
Backward bending: We also observe the *backward bending* phenomenon in some curves which happens whenever a storage controller is overdriven [9]. In this regime, congestion caused by the increasing queue lengths, lock contention, and CPU consumption, bogs down a storage controller such that the disks no longer remain the bottleneck.

6.2.1 Full Backend

In the top panel of Figure 8, we compare the average response time (aggregate read and write) as a function of the throughput achieved when the target throughput is gradually increased in the SPC1-Like workload generator. Overall, STOW outperforms all algorithms significantly across all load levels. Observe that WOW and CSCAN improve as the threshold range becomes wider since a wider range allow them to contain the fluctuations better, hitting the full-cache condition for lesser



(a) Full Backend: target 1050 IOPS, threshold:90/80



(b) Partial Backend: target 3500 IOPS, threshold:70/40

Figure 7: Cache occupancy as a function of time in a 32K page cache serving RAID-5 (RAID-10 is similar). STOW neither exhibits large fluctuations in cache occupancy, nor reaches cache full conditions for the same workload.

amount of time. Since there are no large fluctuations in STOW's cache occupancy, STOW delivers a consistent performance with any threshold, beating the best configuration for either WOW or CSCAN.

In particular, at around 20ms response time, with a threshold of 90/80, in terms of SPC-1 Like IOPS in RAID-5, STOW outperforms WOW by 70%, CSCAN by 96%, and LRW by 39%. With a threshold of 70/40, STOW beats WOW by 18%, CSCAN by 26%, and LRW by 39%. Similarly, in RAID-10 with a 90/80 threshold, STOW outperforms WOW by 40%, CSCAN by 53%, and LRW by 27%, while, with a threshold of 70/40, STOW beats WOW by 20%, and CSCAN and LRW by 27%. These gains are not trivial in the world of hard drives which sees only a meager improvement rate of 8% per year. Although we include data points at response times greater than 30ms, they are not of much practical significance as applications would become very slow at those speeds. Even the SPC-1 Benchmark disallows submissions with greater than 30ms response times.

6.2.2 Partial Backend

For the partial backend scenario, depicted in the lower panel in Figure 8, we use only the outer 1/100th of each disk in the RAID array, creating a high hit ratio scenario with short-stroking on the disks. In this setup, the fluctuations in the occupancy for WOW are closer together (Figure 7(b)), resulting in a more rapid alternation between sequential and random destages. This helps WOW somewhat, however, in terms of SPC-1 Like IOPS at 20ms, STOW still beats WOW by 12%, CSCAN by 160%, and LRW by 24% in a RAID-5 setup. In the RAID-10 setup, where writes become less important (because of no read-modify-write penalties), STOW still beats WOW by 3% (actually it is much better at lower response times), CSCAN by 120%, and LRW by 42%.

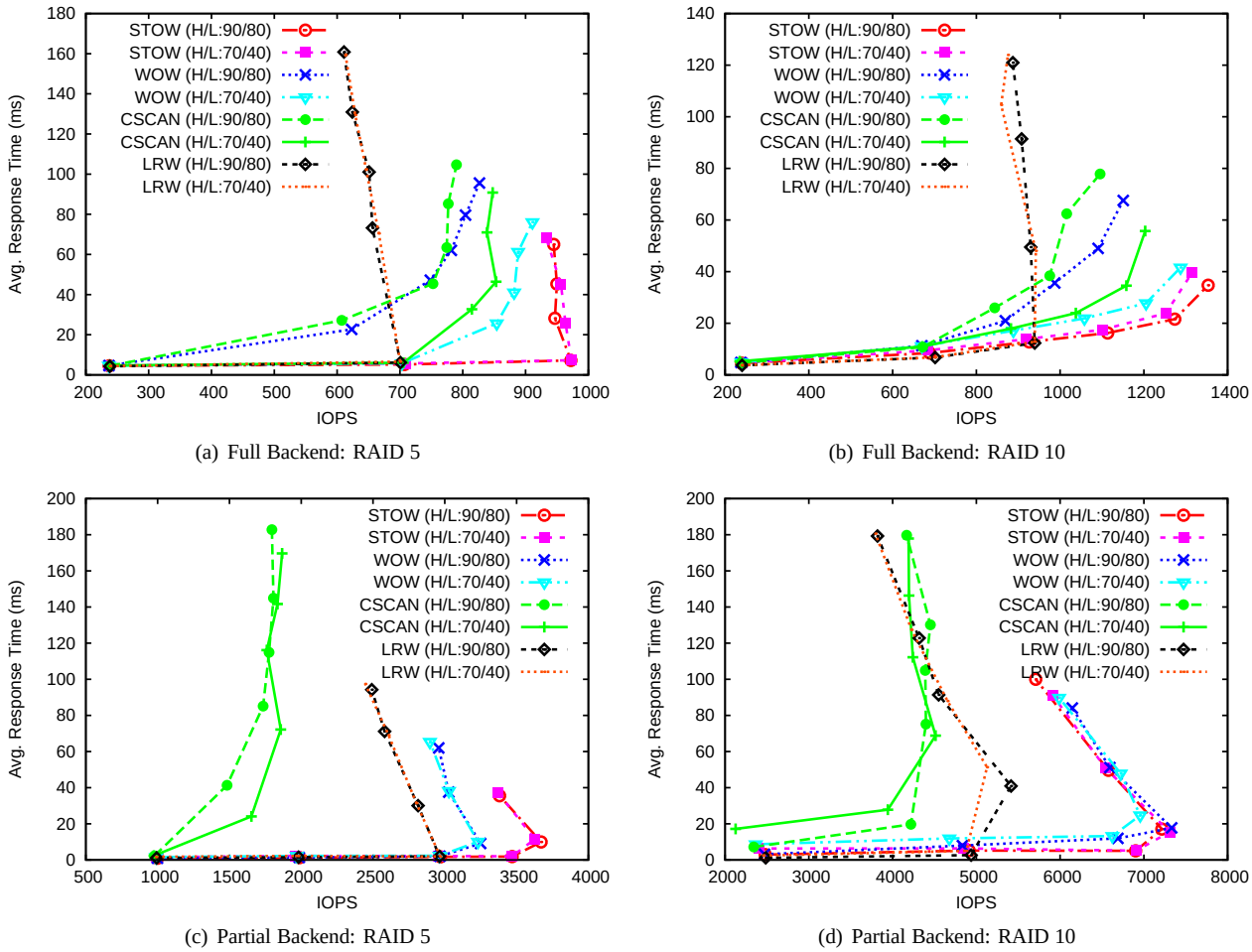
6.2.3 WOW's thresholding dilemma

	Full Backend		Partial Backend	
	H/L: 90/80	70/40	H/L: 90/80	70/40
STOW	**5.18**	5.58	**1.28**	2.23
WOW	22.69	**5.78**	**1.38**	2.26
CSCAN	27.19	**6.03**	41.32	**24.08**
LRW	**6.14**	6.58	**1.50**	2.23

Table 2: Response times (in milliseconds) at lower throughputs (better numbers in bold). For full backend, the response times correspond to a target of 750 IOPS from Figure 8(a), and for partial backend to a target of 2000 IOPS from Figure 8(c). WOW's best threshold choice, unlike STOW, depends on the backend setup.

The response time in a lightly loaded system is also an important metric [14]. We present the actual numbers corresponding to RAID-5 in Figure 8 in Table 2.

Note that in all cases, STOW beats the competition easily. However, WOW is unique in that it requires different thresholds to perform its best for different backend scenarios. While, choosing a conservative (70/40) threshold allows WOW to beat CSCAN and LRW, WOW is forced to sustain a response time of 2.26 ms in the partial backend case, even though it could have delivered 1.38 ms response time with a higher threshold which allows for higher hit ratio. Since the workload is not known *a priori*, the right choice of threshold levels remains elusive for WOW. So, in real life, STOW would cut the response time not by just 7% (1.28 ms vs 1.38 ms) when compared to WOW, but rather by 43% (1.28 ms vs



(a) Full Backend: RAID 5

(b) Full Backend: RAID 10

(c) Partial Backend: RAID 5

(d) Partial Backend: RAID 10

Figure 8: We increase the target throughput for the SPC1-Like Benchmark and present the achieved throughput as a function of the aggregate (read and write) response time for both the $90/80$ and $70/40$ destage thresholds in a 32K page cache. Each data point is the average of measurements over 5 minutes after 5 mintues of warmup time. While WOW beats LRW and CSCAN, STOW outperforms WOW consistently.

2.26 ms). An adaptive threshold determination scheme might help WOW somewhat, but in no instance would it be able to compete with STOW, which at the fixed $90/80$ threshold consistently outperforms its competition.

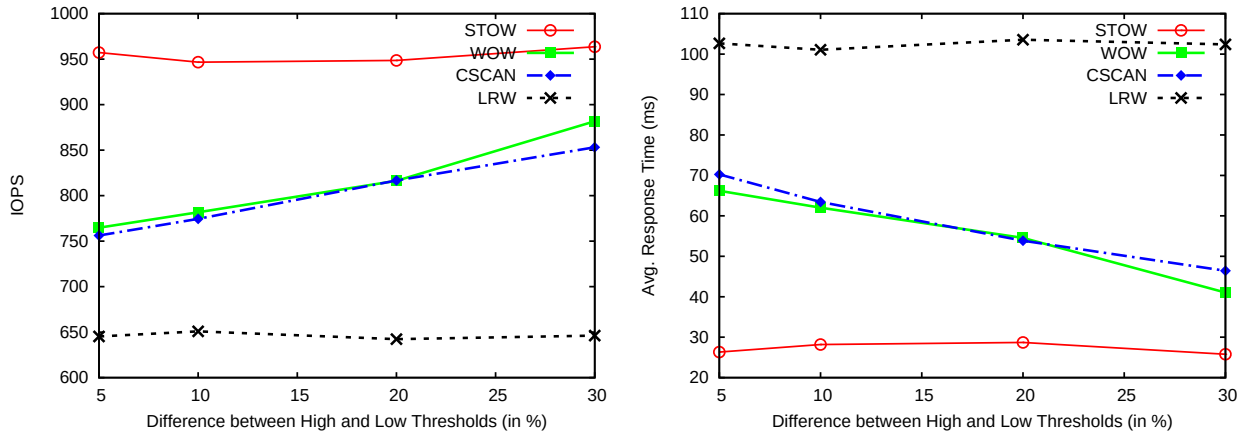
6.3 Varying Threshold Level

In Figure 9, we examine how changing the thresholds alone while keeping the workload constant changes the performance of a write cache. For WOW and CSCAN, in the full backend case, we can clearly see that as the thresholds become lower, the performance improves. While the lower thresholds help keep the occupancy fluctuations away from 100% occupancy more effectively, it cannot completely eradicate the phenomenon and, consequently, both WOW and CSCAN fare worse than STOW. STOW beats WOW and CSCAN by 19% on average, and LRW by 46% in terms of SPC-1 Like IOPS.

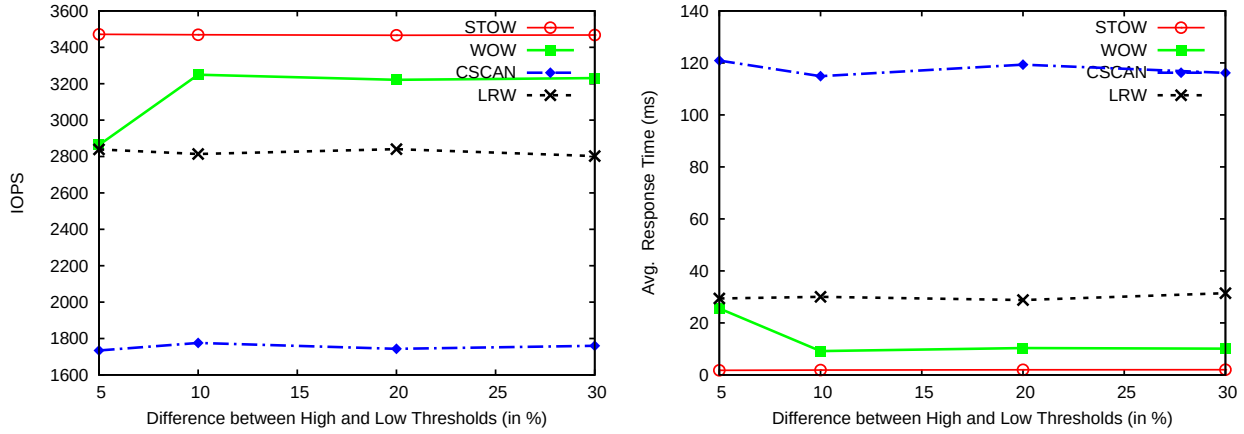
In the partial backend case, both WOW and LRW are better than CSCAN because they can leverage temporal locality more effectively. Further, the performance does not depend on the choice of the threshold in this case because what is gained by keeping lower thresholds is lost in the extra misses incurred in this high hit ratio scenario. In terms of SPC-1 Like IOPS, STOW beats WOW by about 7%, LRW by 22%, and CSCAN by about 96%.

6.4 Varying Cache Size

Any good adaptive caching algorithm should be able to perform well for all cache sizes. In Figure 10(a), we observe that for the full backend scenario, across all cache sizes, STOW outperforms WOW by 17-30%, CSCAN by 19-40%, and LRW by 35-48%. The gains are more



(a) Full Backend: Target of 1200 IOPS



(b) Partial Backend: Target of 3500 IOPS

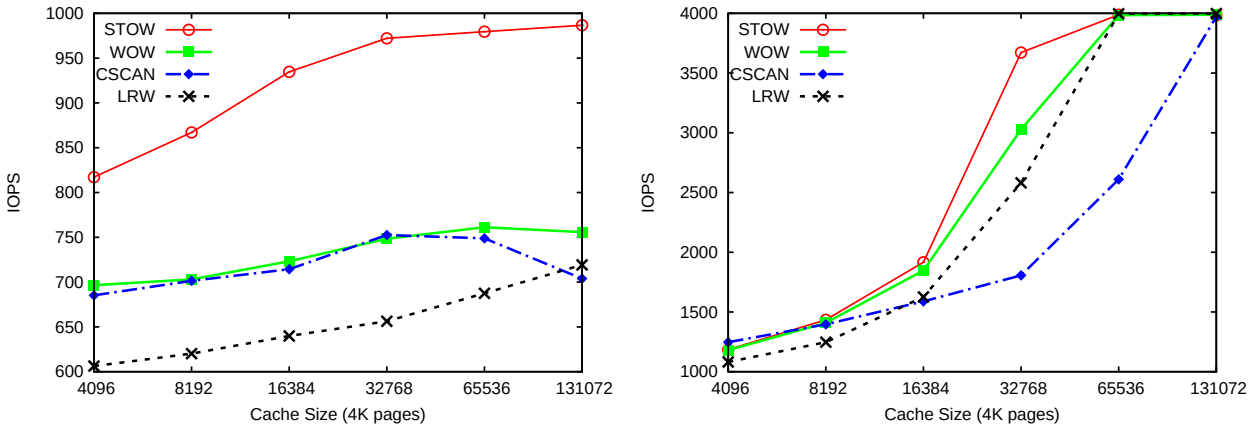
Figure 9: We vary the spread between the high and low thresholds while keeping target workload fixed (32K page cache, RAID-5). The left panel shows the measured throughput and the right panel the corresponding averge response times. RAID-10 has similar results.

significant for larger caches because of two reasons: (i) a larger cache causes the cache occupancy spikes in WOW and CSCAN to be further apart and much larger in amplitude, making it easier to hit the full-cache condition (the performance of CSCAN actually dips as the cache size increases to 131072 pages!); (ii) a larger cache in LRW, WOW, and CSCAN proportionally devotes more cache space to sequential data even though there might be nothing to gain. STOW adapts the sizes of SeqQ and RanQ, which limits the size of SeqQ in larger caches, and creates better spatial locality in the larger RanQ.

The partial backend scenario, presented in Figure 10(b), also indicates that STOW is the best algorithm overall. With smaller cache sizes, the lower hit ratio overdrives the cache for all algorithms resulting in very high response times, which are not of much practical interest. If we had scaled the workload according to what the cache could support, the benefit of STOW would be seen consistently even for lower cache sizes. At a cache size of 32K pages, in terms of SPC-1 Like IOPS, STOW outperforms WOW by 21%, CSCAN by 104%, and LRW by 43%. The performance at higher cache sizes is similar for all algorithms because the working set fits in the cache, eliminating the disk bottleneck.

7 Conclusions

STOW represents a significant improvement over the state of the art in write caching algorithms. While write caching algorithms have mainly focused on the order of destages, we have shown that it is critical to wisely control the rate of destages as well. STOW outperforms WOW by a wider margin than WOW outperforms CSCAN and LRW. The observation that the order of destages needs to change to accommodate a better control on the rate of destages is a key one. We hope that we



(a) Full Backend: Target of 1050 IOPS

(b) Partial Backend: Target of 4000 IOPS

Figure 10: Measured throughput as we vary cache size in a RAID-5 setup (RAID-10 is similar) with $90/80$ thresholds.

have furthered the appreciation of the multi-dimensional nature of the write caching problem, which will spark new efforts towards advancements in this critical field.

References

[1] BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. Non-volatile memory for fast, reliable file systems. *SIGPLAN Not. 27*, 9 (1992), 10–22.

[2] BANSAL, S., AND MODHA, D. S. CAR: Clock with Adaptive Replacement. In *Proc. Third USENIX Conf. on File and Storage Technologies (FAST 04)* (2004), pp. 187–200.

[3] BISWAS, P., RAMAKRISHNAN, K. K., TOWSLEY, D. F., AND KRISHNA, C. M. Performance analysis of distributed file systems with non-volatile caches. In *Proc. 2nd Int'l Symp. High Perf. Distributed Computing* (1993), pp. 252–262.

[4] COFFMAN, E. G., KLIMKO, L. A., AND RYAN, B. Analysis of scanning policies for reducing disk seek times. *SIAM J. Comput. 1*, 3 (1972), 269–279.

[5] CORBATO, F. J. A paging experiment with the Multics system. Tech. rep., Massachusetts Inst. of Tech. Cambridge Project MAC, 1968.

[6] DENNING, P. J. Effects of scheduling on file memory operations. In *Proc. AFIPS Spring Joint Comput. Conf.* (1967), pp. 9–21.

[7] GEIST, R., AND DANIEL, S. A continuum of disk scheduling algorithms. *ACM Trans. Comput. Syst. 5*, 1 (1987), 77–92.

[8] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in Adaptive Replacement Cache. In *Proc. USENIX 2005 Annual Technical Conf. (USENIX)* (2005), pp. 293–308.

[9] GILL, B. S., AND MODHA, D. S. WOW: Wise Ordering for Writes - combining spatial and temporal locality in non-volatile caches. In *Proc. Fourth USENIX Conf. on File and Storage Technologies (FAST 05)* (2005), pp. 129–142.

[10] HAINING, T. R. *Non-volatile cache management for improving write response time with rotating magnetic media*. PhD thesis, University of California, Santa Cruz, 2000.

[11] HSU, W. W., SMITH, A. J., AND YOUNG, H. C. I/O reference behavior of production database workloads and the TPC benchmarks—an analysis at the logical level. *ACM Trans. Database Syst. 26*, 1 (2001), 96–143.

[12] JACOBSON, D., AND WILKES, J. Disk scheduling algorithms based on rotational position. Tech. Rep. HPL-CSP-91-7rev1, HP Labs, February 1991.

[13] JIANG, S., AND ZHANG, X. LIRS: An efficient Low Interreference Recency Set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and modeling of computer systems* (2002), pp. 31–42.

[14] JOHNSON, S., MCNUTT, B., AND REICH, R. The making of a standard benchmark for open system storage. *J. Comput. Resource Management*, 101 (2001), 26–32.

[15] LEE, D., CHOI, J., KIM, J.-H., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and modeling of computer systems* (1999), pp. 134–143.

[16] MCNUTT, B., AND JOHNSON, S. A standard test of I/O cache. In *Proc. Int'l CMG Conference* (2001), pp. 327–332.

[17] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proc. Second USENIX Conf. on File and Storage Technologies (FAST 03)* (2003), pp. 115–130.

[18] MERTEN, A. G. *Some quantitative techniques for file organization*. PhD thesis, Univ. of Wisconsin, June 1970.

[19] ROBINSON, J. T., AND DEVARAKONDA, M. V. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and modeling of computer systems* (1990), pp. 134–142.

[20] SEAMAN, P. H., LIND, R. A., AND WILSON, T. L. On teleprocessing system design Part IV: An analysis of auxiliary storage activity. *IBM Systems Journal 5*, 3 (1966), 158–170.

[21] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk scheduling revisited. In *Proc. USENIX Winter 1990 Tech. Conf.* (1990), pp. 313–324.

[22] VARMA, A., AND JACOBSON, Q. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Trans. Comput. 47*, 2 (1998), 228–235.

[23] VARMA, A., AND JACOBSON, Q. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Trans. Computers 47*, 2 (1998), 228–235.

[24] ZHOU, Y., CHEN, Z., AND LI, K. Second-level buffer cache management. *IEEE Trans. Parallel and Distrib. Syst. 15*, 6 (2004), 505–519.