WOW: Wise Ordering for Writes – Combining Spatial and Temporal Locality in Non-Volatile Caches

Binny S. Gill and Dharmendra S. Modha
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
Emails: {binnyg,dmodha}@us.ibm.com

***Abstract*— Write caches using fast, non-volatile storage are now widely used in modern storage controllers since they enable hiding latency on writes. Effective algorithms for write cache management are extremely important since (i) in RAID-5, due to read-modify-write and parity updates, each write may cause up to four separate disk seeks while a read miss causes only a single disk seek; and (ii) typically, write cache size is much smaller than the read cache size – a proportion of 1 : 16 is typical.

A write caching policy must decide: what data to destage. On one hand, to exploit *temporal locality*, we would like to destage data that is least likely to be re-written soon with the goal of minimizing the total number of destages. This is normally achieved using a caching algorithm such as LRW (least recently written). However, a read cache has a very small uniform cost of replacing any data in the cache, whereas the cost of destaging depends on the state of the disk heads. Hence, on the other hand, to exploit *spatial locality*, we would like to destage writes so as to minimize the average cost of each destage. This can be achieved by using a disk scheduling algorithm such as CSCAN, that destages data in the ascending order of the logical addresses, at the higher level of the write cache in a storage controller. Observe that LRW and CSCAN focus, respectively, on exploiting either temporal or spatial locality, but not both simultaneously. We propose a new algorithm, namely, Wise Ordering for Writes (WOW), for write cache management that effectively combines and balances temporal and spatial locality.

Our experimental set-up consisted of an IBM xSeries 345 dual processor server running Linux that is driving a (software) RAID-5 or RAID-10 array using a workload akin to Storage Performance Council's widely adopted SPC-1 benchmark. In a cache-sensitive configuration on RAID-5, WOW delivers peak throughput that is 129% higher than CSCAN and 9% higher than LRW. In a cache-insensitive configuration on RAID-5, WOW and CSCAN deliver peak throughput that is 50% higher than LRW. For a random write workload with nearly 100% misses, on RAID-10, with a cache size of 64K, 4KB pages (256MB), WOW and CSCAN deliver peak throughput that is 200% higher than LRW. In summary, WOW has better or comparable peak throughput to the best of CSCAN and LRW across a wide gamut of write cache sizes and workload configurations. In addition, even at lower throughputs, WOW has lower average response times than CSCAN and LRW.

I. Introduction

Over the last three decades, processor speeds have increased at an astounding average annual rate of 60%. In contrast, disks which are electro-mechanical devices have improved their access times at a comparatively meager annual rate of about 8%. Moreover, disk capacity grows 100 times per decade, implying fewer available spindles for the same amount of storage [1]. These trends dictate that a processor must wait for increasingly larger number of cycles for a disk read/write to complete. A huge amount of performance literature has focused on hiding this I/O latency for disk bound applications.

Caching is a fundamental technique in hiding I/O latency and is widely used in storage controllers (IBM Shark, EMC Symmetrix, Hitachi Lightning), databases (IBM DB2, Oracle, SQL Server), file systems (NTFS, EXT3, NFS, CIFS), and operating systems (UNIX variants and Windows). SNIA (www.snia.org) defines a cache as "A high speed memory or storage device used to reduce the effective time required to read data from or write data to a lower speed memory or device." We shall study cache algorithms in the context of a storage controller wherein fast, but relatively expensive, random access memory is used as a cache for slow, but relatively inexpensive, disks. A modern storage controller's cache typically contains volatile memory used as a *read cache* and a non-volatile memory used as a *write cache*.

Read cache management is a well studied discipline, for a survey and for some recent work, see [2], [3], [4], [5]. There are a large number of cache replacement algorithms in this context, see, for example, LRU, CLOCK, FBR, LRU-2, 2Q, LRFU, LIRS, MQ, ARC, and CAR. In contrast, write caching is a relatively less developed subject. Here, we shall focus on algorithms for write cache management in the context of a storage controller equipped with fast, non-volatile storage (NVS).

A. Fast Writes using NVS: When to Destage

For early papers on the case for and design of write cache using NVS, see [6], [7], [8], [9], [10]. Historically, NVS was introduced to enable fast writes.

In the absence of NVS, every write must be synchronously written (destaged) to disk to ensure consistency, correctness, durability, and persistence. Non-volatility enables *fast writes* wherein writes are stored safely in the cache, and destaged later in an asynchronous fashion thus hiding the write latency of the disk. To guarantee continued low latency for writes, the

data in the NVS must be drained so as to ensure that there is always some empty space for incoming writes; otherwise, follow-on writes will become effectively synchronous, impacting adversely the response time for writes. On the other hand, if the writes are drained very aggressively, then one cannot fully exploit the benefits of write caching since average amount of NVS cache utilized will be low. Reference [11] introduced a linear threshold scheduling scheme that varies the rate of destages based on the instantaneous occupancy of the write cache. Other simpler schemes include least-cost scheduling and scheduling using high/low mark [11], [12], [13]. Another related issue is the size of a write burst that the write cache is designed to absorb while providing the low response time of fast writes. The write destage policy needs to ensure that a corresponding portion of NVS is available on an average. In Section IV-E, we describe a novel and effective approach to tackle this problem by using adaptive high/low watermarks combined with linear threshold scheduling.

B. A Fundamental Decision: What to Destage

In this paper, we shall focus on the central question of the order in which the writes are destaged from the NVS. Due to its asynchronous nature, the contents of the write cache may be destaged in any desired order without being concerned about starving any write requests. As long as NVS is drained at a sufficiently fast rate, the precise order in which contents of NVS are destaged does not affect fast write performance. However, the decision of what to destage can crucially affect (i) the peak write throughput and (ii) the performance of concurrent reads.

The capacity of disks to support sequential or nearly sequential write traffic is significantly higher than their capacity to support random writes, and, hence, destaging writes while exploiting this physical fact can significantly improve the peak write throughput of the system. This was one of the fundamental motivations for development of log-structured file systems [14], [15] in a slightly different context. Thus, any good write caching algorithm should leverage sequentiality or spatial locality to improve the write throughput and hence the aggregate throughput of the system.

In the presence of concurrent reads, both the writes being destaged and the reads compete for the attention of the disk head. From the perspective of the applications, writes represent a background load on the disks and, indirectly, make read response times higher and read throughput lower. Less obtrusive the writes, the lesser the response time for the reads. We argue that the effect of writes on reads is significant:

- The widely accepted storage benchmark SPC-1 [16], [17] contains 60% writes and 40% reads. While fast writes to NVS enable low response times for writes, these writes, when destaged, do interfere with the reads, increasing the read response times.
- When RAID is used at the back-end, writes are significantly more expensive than the reads. For example, for RAID-5, a read miss on a page may cause only one disk head to move, whereas due to read-modify-write and due to parity updates a write destage may cause up to four separate disk seeks. Similarly, for RAID-10, a write destage causes two disk heads to move.
- NVS is commonly built using battery-backed volatile RAM. The size of NVS is limited by the life of the battery which should be sufficient to dump the contents to a non-battery-backed non-volatile store like disks in case of a power failure. A write cache size of one-sixteenth of the read cache size is not unusual. Given the small relative size of the NVS, it tends to produce relatively fewer write hits, and, hence, a significantly large fraction of writes must be destaged. Further, unlike reads, writes do not benefit significantly from client side caching.

The first two arguments imply that for the SPC-1 benchmark using RAID-5 ranks, writes constitute at least six times the load of the read misses on the back-end! This underscores the tremendous influence that writes have on read performance and the peak throughput of the system. In summary, improving the order in which writes are destaged can significantly improve the overall throughput and response times of the storage controller.

C. Our Contributions

In read caches, the cost of evicting any page from the cache is the same and very small. Hence, the objective of a read cache is simple: to minimize the miss ratio. In contrast, we propose that the performance of a write destage algorithm depends upon two factors: (i) the total number of destages to disks, namely, the write miss ratio and (ii) the average cost of each destage. Roughly speaking, the objective of write destage algorithms is to minimize the product of the above two terms. Minimizing this product would result in the highest peak write throughput in absence of any reads. Even in presence of concurrent reads, this product attempts to minimize the amount of time that the disk heads are occupied in serving writes leading to minimizing the average read response time, while maximizing aggregate throughput.

To minimize the first term, the algorithm should exploit *temporal locality*, namely, should destage data that is least likely to be written to amongst all data in the write cache. To minimize the second term, the algorithm

should exploit *spatial locality* and should destage data that are closer on the disks together so as to exploit the position of the heads and the geometry of the disks in the system for higher throughput.

Classically, in read caches, LRU (least recently used) policy has been used to exploit temporal locality. The analogous policy for exploiting temporal locality in writes is known as LRW that destages the least-recently written page [18], [19], [20].

Algorithms for exploiting spatial locality have been studied in the context of disk scheduling. Many such algorithms require a detailed knowledge of the instantaneous position of the disk head, and exploit the precise knowledge of the location of each data relative to the disk head. In this paper, we are working in the context of a storage controller from which most of the disk parameters are hidden by the underlying RAID and disks. Hence, generally, speaking, spatial locality is hard to exploit at such upper memory levels, see, for example, [21]. We argue, however, that one of the algorithms, namely, CSCAN, that destages data in the ascending order of the logical addresses, can be reasonably successfully applied at even upper levels of memory hierarchy. We empirically demonstrate that as the size of NVS managed by CSCAN increases, the throughput of the system increases for a wide variety of workloads.

The destage order suggested by LRW and CSCAN are generally different, hence, it is only possible to exploit temporal locality or spatial locality, but not both. To emphasize, one reduces number of disk seeks which is the goal of caching, while the other reduces the cost of each disk seek which is the goal of scheduling. This brings us to the main focus of this paper: a novel algorithm that combines both temporal and spatial locality. As our main contribution, we combine an approximation to LRU, namely, CLOCK, with CSCAN to construct a new, simple-to-implement algorithm, namely, Wise Ordering for Writes (WOW), that effectively achieves this goal. The key new idea is to maintain a recency bit akin to CLOCK in CSCAN, and to skip destages of data that has been recently written to.

To demonstrate effectiveness of WOW, we used the following hardware, storage, and workload. The hardware was an IBM xSeries 345 dual processor server running Linux equipped with 4GB RAM that was used as NVS. The storage consisted of 5 disks. Using software RAID, we created a RAID-5 array in 4 data disks + 1 parity disk configuration. We also created a RAID-10 array in 2 + 2 configuration. As the workload, we employed an earlier implementation of the Storage Performance Council's SPC-1 benchmark that is now extremely widely used by many vendors of storage systems [16], [17]. We refer to our workload as SPC-1 Like. In a set-up with a high degree of temporal locality ("a cache-sensitive configuration"), WOW delivers peak throughput that is 129% higher than CSCAN and 9% higher than LRW. In a set-up with very little temporal locality ("a cache-insensitive configuration"), WOW and CSCAN deliver peak throughput that is 50% higher than LRW. For a random write workload with nearly 100% misses, on RAID-10, with a cache size of 64K, 4KB pages, WOW and CSCAN deliver peak throughput that is 200% higher than LRW. Similarly, for the random write workload with nearly 100% misses, on RAID-5, with a cache size of 16K, 4KB pages, WOW and CSCAN deliver peak throughput that is 147% higher than LRW. In summary, WOW has better or comparable peak throughput to the best of CSCAN and LRW across a wide gamut of write cache sizes and workload configurations. In addition, even at lower throughputs, WOW has lower average response times than CSCAN and LRW. In another experiment, using SPC-1 Like workload, as cache size is varied, we explore both cache-insensitive and cache-sensitive regimes. We clearly show that CSCAN is good for cache-insensitive regimes, LRW is good for cache-sensitive regimes, whereas WOW is evergreen and is good across the whole range of cache sizes. In summary, WOW is a practical algorithm that fundamentally enhances the capacity of a storage controller to perform more I/Os.

D. Outline of the Paper

In Section II, we briefly survey previous related research, and we argue the utility of CSCAN for exploiting spatial locality even at upper levels of cache hierarchy. In Section III, we present the new algorithm WOW. In Section IV, we describe the experimental set-up. In Section V, we describe the workloads. In Section VI, we present our main quantitative results. Finally, in Section VII, we conclude with the main findings of this paper.

II. Prior Work

A. Temporal Locality

Algorithms for exploiting temporal locality have been studied extensively in the context of read caches. Several state-of-the-art algorithms include LRU, CLOCK, FBR, LRU-2, 2Q, LRFU, LIRS, MQ, ARC, and CAR. For a detailed review of these algorithms, please see some recent papers [2], [4], [5].

These algorithms attempt to reduce the miss ratio. However, as explained in Section I-C, in write caching, it not sufficient to minimize the miss ratio alone, but we must also pay attention to the average cost of destages. The latter factor is completely ignored by the above

algorithms. We will demonstrate in the paper that a write caching algorithm has a higher hit ratio than some other algorithm and yet the second algorithm delivers a higher throughput. In other words, decreasing the miss ratio without taking into account its effect on the spatial locality is unlikely to guarantee increased performance. Thus, the problem of designing good write caching algorithms is different from that of designing algorithms for read caching.

In this paper, we focus on LRW as the prototypical temporal locality algorithm. We also exploit a simple approximation to LRU, namely, CLOCK [22], that is widely used in operating systems and databases. CLOCK is a classical algorithm, and is a very good approximation to LRU. For a recent paper comparing CLOCK and LRU, see [4].

B. Spatial Locality

The key observation is that given the geometry and design of the modern day disks, sequential bandwidth of the disks is significantly higher (for example, more than 10 times) than its performance on 100% random write workload. The goal is to exploit this differential by introducing spatial locality in the destage order.

This goal has been extensively studied in the context of disk scheduling algorithms. The key constraint in disk scheduling is to ensure fairness or avoid starvation that occurs when a disk I/O is not serviced for an unacceptably long time. In other words, the goal is to minimize the average response time and its variance. For detailed reviews on disk scheduling, see [23], [24], [25], [26]. A number of algorithms are known: First-come-first-serve (FCFS) [27], Shortest seek time first (SSTF) [28] serves the I/O that leads to shortest seek, Shortest access time first (SATF), SCAN [28] serves I/Os first in increasing order and then in decreasing order of their logical addresses, Cyclical SCAN (CSCAN)[29] serves I/Os only in the increasing order of their logical addresses. There are many other variants known as LOOK [30], VSCAN [31], FSCAN [27], Shortest Positioning Time First (SPTF) [26], GSTF and WSTF [24], and Largest Segment per Track LST [11], [32].

In the context of the present paper, we are interested in write caching at an upper level in the memory hierarchy, namely, for a storage controller. This context differs from disk scheduling in two ways. First, the goal is to maximize throughput without worrying about fairness or starvation for writes. This follows since as far as the writer is concerned, the write requests have already been completed after they are written in the NVS. Furthermore, there are no reads that are being scheduled by the algorithm. Second, in disk scheduling, detailed knowledge of the disk head and the exact position of various outstanding writes relative to this head position and disk motion is available. Such knowledge cannot be assumed in our context. For example, [21] found that applying SATF at a higher level was not possible. They concluded that "$\cdots$ we found that modern disks have too many internal control mechanisms that are too complicated to properly account for in the disk service time model. This exercise lead us to conclude that software-based SATF disk schedulers are less and less feasible as the disk technology evolves." Reference [21] further noted that "Even when a reasonably accurate software-based SATF disk scheduler can be successfully built, the performance gain over a SCAN-based disk scheduler that it can realistically achieve appears to be insignificant $\cdots$". The conclusions of [21] were in the context of single disks, however, if applied to RAID-5, their conclusions will hold with even more force.

For these reasons, in this paper, we focus on CSCAN as the fundamental spatial locality algorithm that is suitable in our context. The reason that CSCAN works reasonably well is that at anytime it issues a few outstanding writes that all fall in a thin annulus on the disk, see, Figure 1. Hence, CSCAN helps reduce seek distances. The algorithm does not attempt to optimize for further rotational latency, but rather trusts the scheduling algorithm inside the disk to order the writes and to exploit this degree of freedom. In other words, CSCAN does not attempt to outsmart or outguess the disk's scheduling algorithm, but rather complements it.

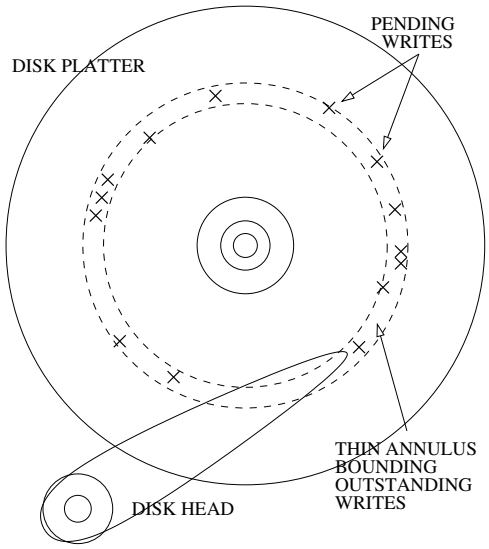

Fig. 1. A visual depiction of how CSCAN localizes the outstanding writes on a thin annulus on the disk platter.

In Figure 2, we demonstrate that as the size of NVS managed by CSCAN grows, so does the achieved throughput. We use a workload that writes 4KB pages randomly over a single disk, and that has nearly 100%

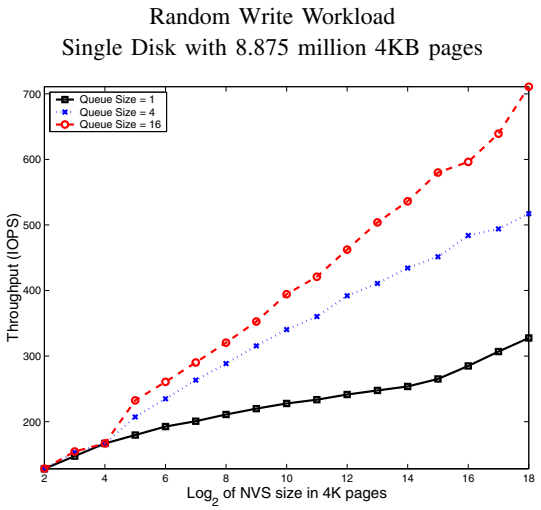

Fig. 2. A plot of the size of NVS used by CSCAN versus the throughput (in IOPS) achieved by the algorithm on a workload that writes 4KB pages randomly over a single disk. It can be seen that throughput seems to grows logarithmically as NVS size grows. Also, as the queue size, namely, the number of outstanding requests issued by CSCAN to the disk increases, the throughput also increases.

write misses. We also use several queue sizes that control the maximum number of outstanding writes that CSCAN can issue to the disk. It appears that throughput grows proportionally to the logarithm of the NVS size. As the size of NVS grows, assuming no starvation at the underlying disk, the average thickness of the annulus in Figure 1 should shrink – thus permitting a more effective utilization of the disk. Observe that the disks continuously rotate at a constant number of revolutions per second. CSCAN attempts to increase the number of writes that can be carried out per revolution.

C. Combining Temporal and Spatial Locality

The only previous work on the subject is [33], [32] that partitions the cache into a "hot" zone that is managed via LRW and a "cold" zone that is managed via LST. The authors point out several drawbacks of their approach: (i) "The work in this dissertation only deals with the interaction between the cache and one disk." [32, p. 126] and (ii) "One of the most immediate aspects of this work requiring more research is the method to determine the size of the hot zone for the stack model-based replacement algorithm. We determined the best size for the hot zone empirically in our experiments." [32, p. 125]. To continue this program further would entail developing an adaptive algorithm for tuning the size of the hot zone. However, note that the hot zone optimizes for temporal locality (say "apples") and the cold zone for spatial locality (say "oranges"). It is not currently known how to compare and trade apples versus oranges to determine the best adaptive partition. In this paper, we will present a natural combination of LRW and CSCAN that obviates this need, and yet delivers convincing performance.

III. WOW

A. Preliminaries

We are working in the context of a storage controller. Typically, a storage controller connects to a RAID controller which, in turn, connects to physical disks. We assume that there is no write cache at lower levels such as RAID and disks. In other words, there is no fast write at lower levels and I/O scheduling is limited to concurrent requests issued by the storage controller. Also, note that a typical RAID controller may choose to implement FCFS or such simpler scheduling algorithm, while an individual disk may implement a smarter disk scheduling algorithm such as SATF. We can make no assumptions about these algorithms at the level of a storage controller. Also, typically the amount of write cache in storage controllers per RAID array or per disk is much larger than the amount of cache in the RAID array or disks.

We will use 4KB pages as the smallest unit of cache management. We will divide each disk into strips, where each strip is a logically and physically contiguous set of pages. Here, we use 64KB as the strip size. In RAID, we will define a stripe as a collection of strips where each participating disk contributes one strip to a stripe. Due to mirroring in RAID-10 and parity in RAID-5, the effective storage provided by a stripe is less than its physical size.

The notion of a hit is straightforward in read caching, but is somewhat involved for write caching. In write caching, a hit can be a hit on a page, a hit on a strip, or a hit on a stripe. These different hits may have different payoffs, for example, in RAID-5, a page hit saves four seeks, whereas a stripe hit and a page miss saves two seeks because of shared parity. In RAID-5, we will manage cache in terms of stripe groups. In RAID-10, we will manage cache in terms of strip groups. This allows a better exploitation of temporal locality by saving seeks and also spatial locality by coalescing writes together. We will refer to a strip or stripe group as a *write group*.

B. WOW : The Algorithm

We now describe our main contribution which is a new algorithm that combines the strengths of CLOCK, a predominantly read cache algorithm, and CSCAN, an efficient write cache algorithm, to produce a very powerful and widely applicable write cache algorithm. See Figures 3 and 4 for a depiction of the data structures and the algorithm.

The basic idea is to proceed as in CSCAN by maintaining a sorted list of write groups. The smallest

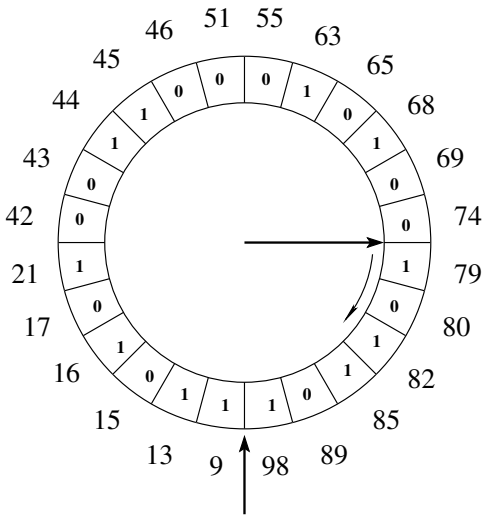

Fig. 3. A visual structure of the WOW algorithm. If the CLOCK's recency bit is ignored, the algorithm becomes CSCAN. The clock hand represents the destagePointer in Figure 4.

and the highest write groups are joined forming a circular queue. The additional new idea is to maintain a "recency" bit akin to CLOCK with each write group. The algorithm now proceeds as follows.

A write group is always inserted in its correct sorted position. Upon insertion, its recency bit is set to zero. However, on a write hit, the recency bit is set to one. The destage operation proceeds as in CSCAN, wherein a destage pointer is maintained that traverses the circular list looking for destage victims. In CSCAN every write group that is encountered is destaged. However, we only allow destage of write groups whose recency bit is zero. The write groups with a recency bit of one are skipped, however, their recency bit is turned off, and reset to zero. The basic idea is to give an extra life to those write groups that have been hit since the last time the destage pointer visited them. This allows us to incorporate recency representing temporal locality on one hand, and small average distance between consecutive destages representing spatial locality. The simplicity of the algorithm is intentional so that it succeeds in real systems. The superiority of the algorithm (demonstrated in Section VI) to the current state-of-the-art should encourage its widespread use.

C. WOW and its Parents

Since WOW is a hybrid between LRW or CLOCK, and CSCAN, we now contrast and compare these algorithms.

WOW is akin to CSCAN, since it destages in essentially the same order as CSCAN. However, WOW is

CACHE MANAGEMENT POLICY:
Page x in write group s is written:

```
1:  if (s is in NVS) // a write group hit
2:      if (the access is not sequential)
3:          set the recencyBit of s to 1
4:      endif
5:      if (x is in NVS) // a page hit
6:          set the recencyBit of s to 1
7:      else
8:          allocate x from FreePageQueue
            and insert x in s
9:      endif
10: else
11:     allocate s from
            FreeStripeGroupHeaderQueue
12:     allocate x from FreePageQueue
13:     insert x into s and s into the sorted queue
14:     initialize the recencyBit of s to 0
15:     if (s is the only write group in NVS)
16:         initialize the destagePointer to point to s
17:     endif
18: endif
```

DESTAGE POLICY:

```
19: while (needToDestage())
20:     while (the recencyBit of the write group
        pointed to by the destagePointer is 1)
21:         reset the recencyBit to 0
22:         AdvanceDestagePointer()
23:     endwhile
24:     destage all pages in the write group pointed
        to by the destagePointer and
        move them to FreePageQueue
25:     move the destaged write group to
        FreeStripeGroupHeaderQueue
26:     AdvanceDestagePointer()
27: endwhile


28: AdvanceDestagePointer()
29:     if (destagePointer is pointing to the
        highest address write group in the queue)
30:         reset the destagePointer to point to the
            lowest address write group in the queue
31:     else
32:         advance the destagePointer to the next
            higher address write group in the queue
33:     endif
```

Fig. 4. The WOW Algorithm.

different from CSCAN in that it skips destage of data that have been recently written to in the hope that that are likely to be written to again. WOW generally will have a higher hit ratio than CSCAN at the cost of an increased gap between consecutive destages.

WOW is akin to LRW in that it defers write groups that have been recently written. Similarly, WOW is akin to CLOCK in that upon a write hit to a write group a new life is granted to it until the destage pointer returns to it again. WOW is different from CLOCK in that the new write groups are not inserted immediately behind the destage pointer as CLOCK would but rather in their sorted location. Thus, initially, CLOCK would always grant one full life to each newly inserted write group, whereas WOW grants on an average half that much time. WOW generally will have a significantly smaller gap between consecutive destages than LRW at the cost of a generally lower hit ratio.

D. Is RAID just one Big Disk?

Intuitively, the reader may wish to equate the destage pointer in WOW to a disk head. It is as if WOW or CSCAN are simulating the position of the disk head, and destaging accordingly. In practice, this intuition is not strictly correct since (i) concurrent read misses may be happening which can take the disk heads to arbitrary locations on disks; and (ii) the position of the heads cannot be strictly controlled, for example, due to read-modify-write in RAID-5; and (iii) at a lower level, either the RAID controller or the individual disks may re-order concurrent write requests. In view of these limitations, the purpose of WOW or CSCAN is to spatially localize the disk heads to a relatively narrow region on the disks with the idea that the resulting disk seeks will be less expensive than random disk seeks which may move the head across a larger number of cylinders on the disks. In practice, we have seen that these observations indeed hold true.

E. WOW *Enhancements*

We anticipate that WOW will engender a class of algorithms which modify WOW along multiple dimensions. We have shown how to combine LRW and CSCAN. Another important feature of workloads that indicates temporal locality is "frequency". It is possible to incorporate frequency information into WOW by utilizing a counter instead of just a recency bit. It is extremely interesting and challenging to pursue adaptive variants of WOW that dynamically adapt the balance between temporal and spatial locality. Furthermore, it will be interesting to see if a marriage of MQ, ARC, CAR, etc. algorithms can be consummated with CSCAN to develop algorithms that separate out recency from frequency to further enhance the power of WOW.

Another aspect of temporal locality is the duration for which a new stripe of page is allowed to remain in the cache without producing a hit. For simplicity, we have chosen the initial value of the recency bit to be set to 0 (see line 14 in Figure 4). Thus, on an average, a new write group gets a life equal to half the time required by the destage pointer to go around the clock once. If during this time, it produces a hit, it is granted one more life until the destage pointer returns to it once again. If the initial value is set to 1, then–on an average– a new write group gets a life equal to 1.5 times the time required by the destage pointer to go around the clock once. More temporal locality can be discovered if the initial life is longer. However, this happens at the cost of larger average seek distances as more pages are skipped by the destage head. It may be possible to obtain the same effect without the penalty by maintaining a history of destaged pages in the spirit of MQ, ARC, and CAR algorithms.

F. WOW : Some design points

1) Sequential Writes: It is very easy to detect whether a write group is being written to by a sequential access write client or a random access one, see, for example, [3, Section II.A]. In this paper, we enhance the algorithm in Figure 4 to never set the recency bit to 1 on a sequential access. This is reflected in lines 2-4 in Figure 4. This heuristic gives the bulky sequential stripes a smaller life and frees up the cache for more number of less populated stripes that could potentially yield more hits.

2) Variable I/O sizes: In Figure 4, we have dealt with a illustrative wherein a single page x in a write group s is written to. Typical write requests may write to a variable number of pages that may lie on multiple write groups. Our implementation correctly handles these cases via simple modifications to the given algorithm.

3) Data Structures: The algorithm requires very simple data structures: a sorted queue for storing write groups, a hash-based lookup for checking whether a write group is presented in the sorted queue (that is for hit/miss determination), and a destage pointer for determining the next candidate write group for destage. The fact that insertion in a sorted queue is an $O(\log(n))$ operation does not present a practical problem due to the limited sizes of NVS and the availability of cheap computational power.

IV. Experimental Set-up

We now describe a system that we built to measure and compare the performance of the different write cache algorithms under a variety of realistic configurations.

A. The Basic Hardware Set-up

We use an IBM xSeries 345 machine equipped with two Intel Xeon 2 GHz processors, 4 GB DDR, and six 10K RPM SCSI disks (IBM, 06P5759, U160) of 36.4 GB each. A Linux kernel (version 2.6.11) runs on this machine hosting all our applications and standard workload generators. We employ five SCSI disks for the purposes of our experiments, and the remaining one for the operating system, our software, and workloads.

B. Storage: Direct Disks or Software RAID

We will study three basic types of configurations corresponding to I/Os going to either a single disk, to a RAID-5 array, or a RAID-10 array. In the first case, we issue direct (raw) I/O to one of the five disk devices (for example, /dev/sdb). In the latter two cases, we issue direct I/O to the virtual RAID disk device (for example, /dev/md0) created by using the Software RAID feature in Linux.

Software RAID in Linux implements the functionality of a RAID controller within the host to which the disks are attached. As we do not modify or depend on the particular implementation of the RAID controller, a hardware RAID controller or an external RAID array would work equally well for our experiments. We created a RAID-5 array using 5 SCSI disks in 4 data disk + 1 parity disk configurations. We chose the strip size (chunk size) for each disk to be 64KB. Resulting stripe group size was 256KB. Similarly, we created a RAID-10 array using 4 SCSI disks in 2 + 2 configuration. Once again, we chose the strip size for each disk to be 64KB.

C. NVS and Read Cache

We employ the volatile DDR memory as our write cache or NVS. The fact that this memory is not battery-backed does not impact the correctness or relevance of our results to real-life storage controllers. We shall, therefore, still refer to this memory as NVS. The write cache is implemented in shared memory and is managed by user space libraries that are linked to all the applications that refer to this shared memory cache. The size of the NVS can be set to any size up to the maximum size of the shared memory. This approach provides tremendous flexibility for our experiments by allowing us to benchmark various algorithms across a large range of NVS sizes.

For our experiments, we do not use a read cache as all disk I/Os are direct (or raw) and bypass the Linux buffer caches. This helps us eliminate an unnecessary degree of freedom for our experiments. Recall that read misses must be served concurrently, and disrupt the sequential destaging operation of WOW and CSCAN. Also, read misses compete for head time, and affect even LRW. Eliminating the read cache serves to maximize read misses, and, hence, our setup is the most adversarial for NVS destage algorithms. In real-life storage controllers equipped with a read cache, the aggregate performance will depend even more critically on the write caching algorithm and thus magnify even further the performance differences between these algorithms.

A side benefit of maintaining a write cache is the read hits that it produces. The write caching algorithms are not intended to improve the read hit ratio primarily because the read cache is larger and more effective in producing read hits. Nevertheless, in our setup we do check the write cache for these not-so-numerous read hits and return data from the write cache on a hit for consistency purposes.

D. The Overall Software System

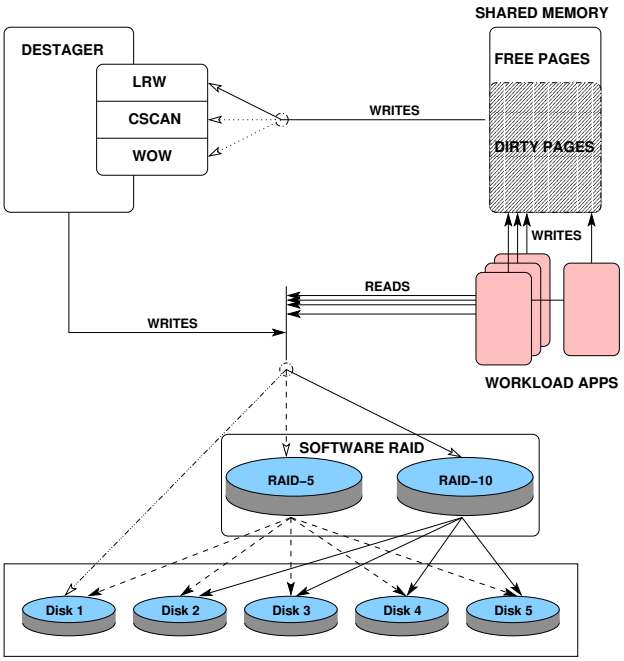

Fig. 5. The overall design of our experimental software system. The system is layered: (i) applications; (ii) NVS (shared memory) whose size can be varied from "cache-insensitive" (very small) to "cache-sensitive" (relatively large); (iii) destager which may choose to implement LRW, CSCAN, or WOW; and (iv) backend storage which may be a single disk, a RAID-5 array, or a RAID-10 array.

A schematic description of the experimental system is given in Figure 5.

The workload applications may have multiple threads that can asynchronously issue multiple simultaneous read and write requests. We implemented the write cache in user space for the convenience of debugging. As a result, we do have to modify the workload applications to replace the "read()" and "write()" system calls with our versions of these after linking our shared

memory management library. This is a very simple change that we have implemented for all the workload applications that we used in this paper.

For a single disk, NVS was managed in units of 4KB pages. For RAID-5, NVS was managed in terms of 256KB stripe write groups. Finally, in RAID-10, NVS was managed in terms of 64KB strip write groups. Arbitrary subsets of pages in a write group may be present in NVS. Write requests that access consecutively numbered write groups are termed *sequential*.

On a read request, if all requested pages are found in NVS, it is deemed a read hit, and is served immediately (synchronously). If, however, all requested pages are not found in NVS, it is deemed a read miss, and a synchronous stage (fetch) request is issued to the disk. The request must now wait until the disk completes the read. Immediately upon completion, the read request is served. As explained above, the read misses are not cached.

On a write request, if all written pages are found in NVS, it is deemed a write hit, and the write request is served immediately. If some of the written pages are not in NVS but enough free pages are available, once again, the write request is served immediately. If, however, some of the written pages are not in NVS and enough free pages are not available, the write request must wait until enough pages become available. In the first two cases, the write response time is negligible, whereas in the last case, the write response time can become significant. Thus, NVS must be drained so as to avoid this situation if possible.

We have implemented a user-space destager program that chooses dirty pages from the shared memory and destages them to the disks. This program is triggered into action when the free pages in the shared memory are running low. It can be configured to destage either to a single physical disk or to the virtual RAID disk. To decide which pages to destage from the shared memory, this program can choose between the three cache management algorithms: LRW, CSCAN, and WOW.

E. Queue Depth and When To Destage

To utilize the full throughput potential of a RAID array or even a single disk, it is crucial to issue multiple concurrent writes. This gives more choice to the scheduling algorithm inside the disks which, by design, usually tries to maximize the throughput without starving any I/Os. Furthermore, in RAID, the number of outstanding concurrent writes roughly dictates the number of disks heads that can be employed in parallel. The number of outstanding concurrent writes constitute a queue. As this queue length increases, both the throughput and the average response time increases. As the queue length increases, the reads suffer, in that, they may have to wait more on an average. We choose a value, MAXQUEUE (say 20), as the maximum of number of concurrent write requests to the disks, where a write request is a set of contiguous pages within one write group.

We now turn our attention to the important decision of "When to Destage" that is needed in line 19 of Figure 4. At any time, we dynamically vary the number of outstanding destages in accordance with how full the NVS actually is. We maintain a lowThreshold which is initially set to 80% of the NVS size, and a highThreshold which is initially set to 90% of the NVS size. If the NVS occupancy is below the lowThreshold and we were not destaging sequential write group, we stop all destages. However, if NVS occupancy is below the lowThreshold but the previous destage was marked sequential and the next candidate destage is also marked sequential, then we continue the destaging at a slow and steady rate of 4 outstanding destages at any time. This ensures that sequences are not broken and their spatial locality is exploited completely. Further, this also takes advantage of disks' sequential bandwidth. If NVS occupancy is at or above the highThreshold, then we always go full throttle, that is, destage at the maximum drain rate of MAXQUEUE outstanding write requests. We linearly vary the rate of destage from lowThreshold to highThreshold in a fashion similar to [11]. The more full within this range the NVS gets, the faster the drain rate; in other words, the larger the number of outstanding concurrent writes. Observe that the algorithm will not always use the maximum queue depth. Writing at full throttle regardless of the rate of new writes is generally bad for performance. What is desired is simply to keep up with the incoming write load without filling up NVS. Convexity of throughput versus response time curve indicates that a steady rate of destage is more effective than a lot of destages at one time and very few at another. Dynamically ramping up the number of outstanding concurrent writes to reflect how full NVS is helps to achieve this steady rate. Always using full throttle destage rate leads to abrupt "start" and "stop" situation, respectively, when the destage threshold is exceeded or reached.

We add one more new idea, namely, we dynamically adapt the highThreshold. Recall that write response times are negligible as long as NVS is empty enough to accommodate incoming requests, and can become quite large if NVS ever becomes full. We adapt the highThreshold to attempt to avoid this undesirable state while maximizing NVS occupancy. We implement a simple adaptive back-off and advance scheme. The lowThreshold is always set to be highThreshold minus 10% of NVS size. We define desiredOccupancyLevel

to be 90% of the NVS size. The highThreshold is never allowed to be higher than desiredOccupancyLevel or lower than 10% of NVS size. We maintain a variable called maxOccupancyObserved that keeps the maximum occupancy of the cache since the last time it was reset. Now, if and when the NVS occupancy drops below the current highThreshold, we decrement the highThreshold by any positive difference between maxOccupancyObserved and desiredOccupancyLevel and we reset maxOccupancyObserved to the current occupancy level. We keep a note of the amount of destages that happen between two consecutive resettings of maxOccupancyObserved in the variable resetInterval. Of course, decrementing highThreshold hurts the average occupancy levels in NVS, and reduces spatial as well as temporal locality for writes. Thus, to counteract this decrementing force, if after a sufficient number of destages (say equal to resetInterval) the maxOccupancyObserved is lower than the desiredOccupancyLevel, then we increment highThreshold by the difference between desiredOccupancyLevel and maxOccupancyObserved, and we reset maxOccupancyObserved to the current occupancy level.

V. Workloads

A. Footprint

While modern storage controllers can make available an immense amount of space, in a real-life scenario, workloads actively use only a fraction of the total available storage space known as the *footprint*. Generally speaking, for a given cache size, the larger the footprint, the smaller the hit ratio, and vice versa. We will use backend storage in two configurations: (i) *Full Backend* in which the entire available backend storage will be used and (ii) *Partial Backend* in which we will use 7.1 million 512 byte sectors. In RAID-5, we shall have effectively the storage capacity of four disks at the back-end, where Full Backend amount to 284 million 512 byte sectors. Similarly, for RAID-10, we shall have effectively the storage capacity of two disks at the back-end, where Full Backend amount to 142 million 512 byte sectors.

B. SPC-1 Benchmark

SPC-1 is a synthetic, but sophisticated and fairly realistic, performance measurement workload for storage subsystems used in business critical applications. The benchmark simulates real world environments as seen by on-line, non-volatile storage in a typical server class computer system. SPC-1 measures the performance of a storage subsystem by presenting to it a set of I/O operations that are typical for business critical applications like OLTP systems, database systems and mail server applications. For extensive details on SPC-1, please see: [16], [17], [3]. A number of vendors have submitted SPC-1 benchmark results for their storage controllers, for example, IBM, HP, Dell, SUN, LSI Logic, Fujitsu, StorageTek, 3PARdata, and DataCore. This underscores the enormous practical and commercial importance of the benchmark. We used an an earlier prototype implementation of SPC-1 benchmark that we refer to as SPC-1 Like.

SPC-1 has 40% read requests and 60% write requests. Also, with 40% chance a request is a sequential read/write and with 60% chance a request is a random read/write with some temporal locality. SPC-1 is a multi-threaded application that can issue multiple simultaneous read and writes. For a given cache/NVS size, the number of read/write hits produced by SPC-1 changes as the footprint of the backend storage changes. For example, for a given cache/NVS size, SPC-1 will produce more hits with a Partial Backend then with a Full Backend. Furthermore, it is easy to vary the target throughput in I/Os Per Second (IOPS) for the workload. Thus, it provides a very complete and versatile tool to understand the behavior of all the three write destage algorithms in a wide range of settings.

SPC-1's backend consists of three disjoint application storage units (ASU). ASU-1 represents a "Data Store", ASU-2 represents a "User Store", and ASU-3 represents a "Log/Sequential Write". Of the total amount of available back-end storage, 45% is assigned to ASU-1, 45% is assigned to ASU-2, and remaining 10% is assigned to ASU-3 as per SPC-1 specifications. In all configurations, we laid out ASU-3 at the outer rim of the disks followed by ASU-1 and ASU-2.

C. Random Write Workload

We will use a random write workload that uniformly writes 4KB pages over the Full Backend that is available. As long as the size of the cache is relatively smaller than the Full Backend size, the workload has little temporal locality, and will produce nearly 100% write misses. The think time, namely, the pause between completing one write request and issuing another one, of this workload is set to zero. In other words, this workload is capable, in principle, of driving a storage system at an infinite throughput. The throughput is limited only by the capacity of the system to serve the writes. This workload is extremely helpful in profiling behavior of the algorithms across a wide range of NVS sizes.

VI. Results

A. LRW Does Not Exploit Spatial Locality

In Figure 6, we compare LRW, CSCAN, and WOW using random write workload (Section V-C) directed to

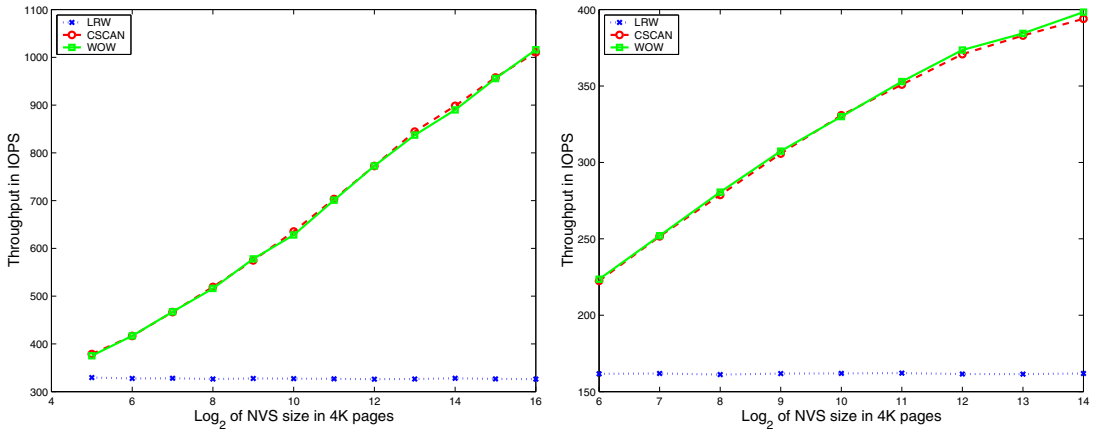

Fig. 6. A comparison of LRW, CSCAN, and WOW using random write workload using the Full Backend for both RAID-10 and RAID-5. It can be seen that the throughput of LRW does not depend upon the NVS size, whereas throughput of WOW and CSCAN exhibit a logarithmic gain as a function of the size of NVS.

Full Backend on RAID-5 and RAID-10. Since the workload has almost no temporal locality, the throughput of LRW remains constant as the NVS size increases. In contrast, WOW and CSCAN exhibit logarithmic gain in throughput as a function of the size of NVS by exploiting spatial locality (also see the related discussion in Section II-B). For RAID-10, at the lowest NVS size of 32 pages, WOW and CSCAN outperform LRW by 16%, while, quite dramatically, at the NVS size of 65,536 pages, WOW and CSCAN outperform LRW by 200%. Similarly, for RAID-5, at the lowest NVS size of 64 pages, WOW and CSCAN outperform LRW by 38%, while, quite dramatically, at the NVS size of 16,384 pages, WOW and CSCAN outperform LRW by 147%.

While, for brevity, we have shown results for a queue depth of 20. When we used a larger queue depth, performance of all three algorithms increased uniformly, producing virtually identical curves. Increasing queue depth beyond 128 in either RAID-10 or RAID-5 does not seem to help throughput significantly.

B. WOW *is Good for Cache-sensitive and -insensitive Regimes*

In Figure 7, we compare LRW, CSCAN, and WOW using SPC-1 Like workload (Section V-B) directed to Partial Backend on RAID-5 and RAID-10. We vary the size of NVS from very small (corresponding to cache-insensitive regime) to relatively large (corresponding to cache-sensitive regime). For RAID-10, a target throughput of 6000 IOPS was used for all NVS sizes. For RAID-5, a target throughput of 3000 IOPS was used for all NVS size except the largest one (100,000 pages) for which a target throughput of 6000 IOPS was used to drive the disks to full capacity.

In both graphs, it is easy to see that CSCAN dominates LRW in the cache-insensitive regime, but loses to LRW in the cache-sensitive regime. WOW, however, is evergreen, and performs well in both the regimes. This is easy to see since in a cache-insensitive regime CSCAN wins by exploiting spatial locality, whereas in cache-sensitive regime LRW puts up a nice performance by exploiting temporal locality. However, WOW which is designed to exploit both temporal and spatial locality performs well in both the regimes and across the entire range of NVS sizes.

Specifically, in RAID-10, for the smallest NVS size, in a favorable situation for CSCAN, WOW performs the same as CSCAN, but outperforms LRW by 4.6%. On the other hand, for the largest NVS size, in a favorable situation for LRW, WOW outperforms CSCAN by 38.9% and LRW by 29.4%.

Similarly, in RAID-5, for the smallest NVS size, in a favorable situation for CSCAN, WOW loses to CSCAN by 2.7%, but outperforms LRW by 9.7%. On the other hand, for the largest NVS size, in a favorable situation for LRW, WOW outperforms CSCAN by 129% and LRW by 53%. In Figure 8, we examine, in detail, the write hit ratios and average difference in logical address between consecutive destages for all three algorithms. The larger the write hit ratio the larger is the temporal locality. The larger the average distance between consecutive destages the smaller is the spatial locality. It can be seen that temporal locality is the highest for LRW, followed by WOW, and the least for CSCAN. On the contrary, spatial locality is the highest for CSCAN, followed by WOW, and the least (by 3 to 4 orders of magnitude) for LRW. As shown in the right panel of Figure 7, WOW outperforms both LRW and

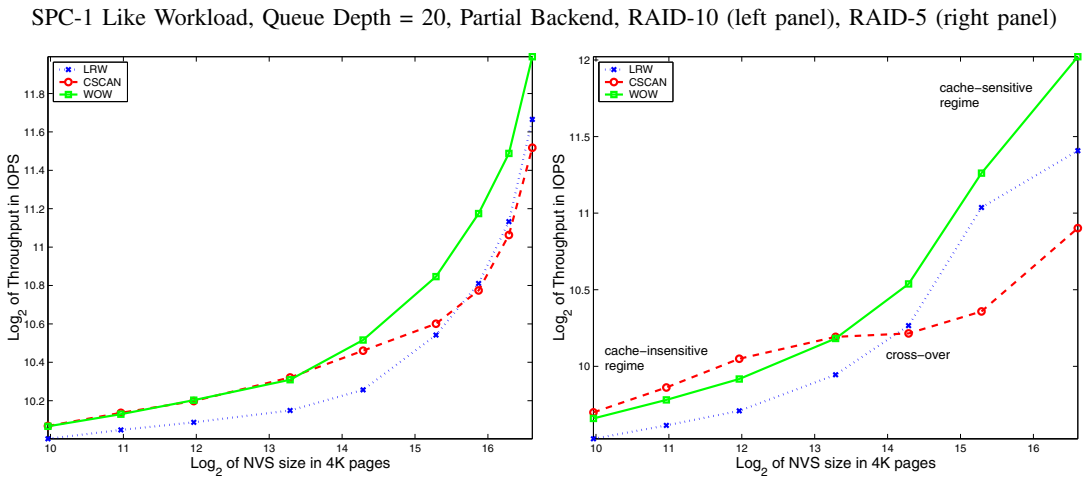

Fig. 7. A comparison of LRW, CSCAN, and WOW using SPC-1 Like workload using the Partial Backend for both RAID-10 and RAID-5. It can be seen that CSCAN is a good bet in cache-insensitive regime whereas LRW is an attractive play in cache-sensitive regime. However, WOW is attractive in both regimes.

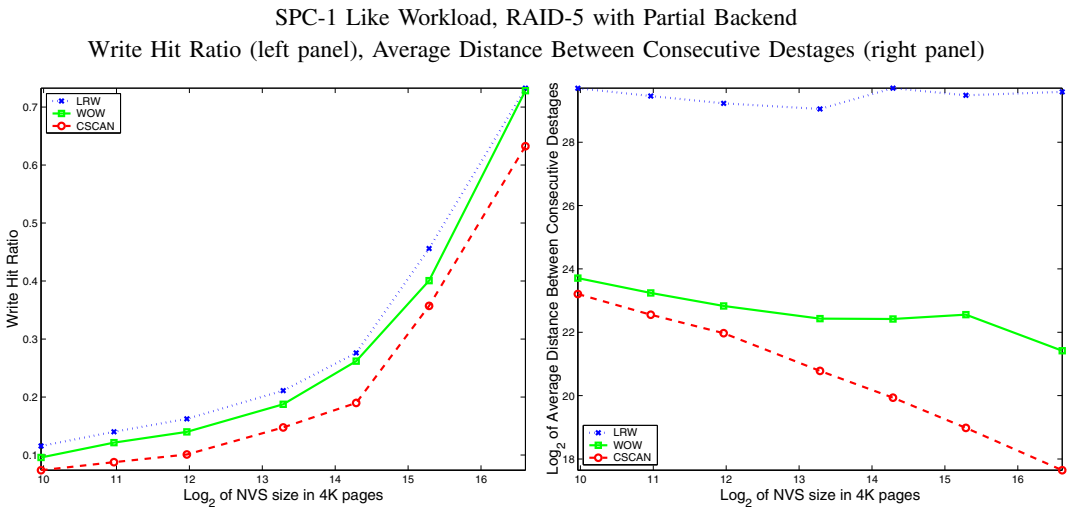

Fig. 8. The left panel shows the write hit ratios (indicating temporal locality) for LRW, CSCAN, and WOW, while using the SPC1-Like workload using partial backend for RAID-5. Temporal locality is the highest for LRW, followed by WOW, and the least for CSCAN. The right panel shows the corresponding average difference in logical address between consecutive destages for the three algorithms during the same run. The larger this average distance the smaller is the spatial locality. We can see that, contrary to the temporal locality, spatial locality is the highest for CSCAN, followed by WOW, and the least (by 3 to 4 orders of magnitude) for LRW. As shown in the right panel of Figure 7, WOW outperforms both LRW and CSCAN by effectively combining temporal as well as spatial locality.

CSCAN by effectively combining temporal as well as spatial locality.

C. Throughput versus Response Time in Cache-insensitive Scenario

In Figure 9, we compare LRW, CSCAN, and WOW using SPC-1 Like workload directed to Full Backend on RAID-5. We use an NVS size of 4K pages each of 4KB. Hence, NVS to backing store ratio is very low, namely, 0.011%, constituting a cache-insensitive scenario.

We vary the target throughput of SPC-1 Like from 100 IOPS to 1100 IOPS. At each target throughput, we allow a settling time of 10 mins, after which we record average response time over a period of 5 minutes.

It can be clearly seen that WOW and CSCAN have virtually identical performances, and both significantly outperform LRW. In particular, it can be seen that LRW finds it impossible to support throughput beyond 515 IOPS. Demanding a target throughput higher than this point does not yield any further improvements, but rather worsens the response times dramatically. In contrast, WOW and CSCAN saturate, respectively, at 774 and 765 IOPS. In other words, WOW delivers a peak throughput that is 50% higher than LRW, and

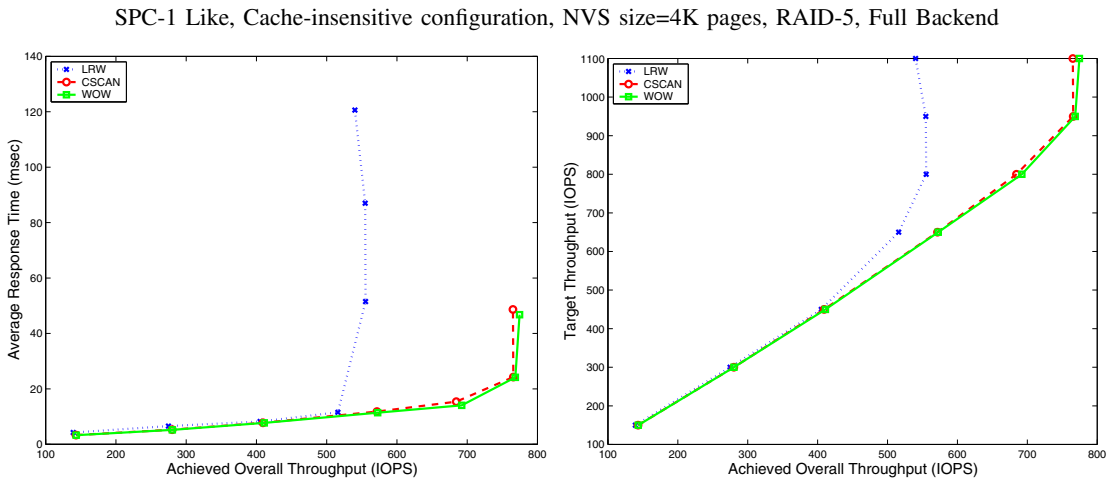

Fig. 9. A comparison of LRW, CSCAN, and WOW. The left panel displays achieved overall throughput versus achieved average response time. This set-up does not have much temporal locality. WOW and CSCAN have comparable performance, and both outperform LRW dramatically. Specifically, WOW increases the peak throughput over LRW by 50%. The right panel shows the target throughput corresponding to the data points in the left panel. It can be clearly seen that LRW hits an insurmountable stiff wall at a much lower throughput.

virtually identical to CSCAN.

D. Throughput versus Response Time in Cache-sensitive Scenario

In Figure 10, we compare LRW, CSCAN, and WOW using SPC-1 Like workload directed to Partial Backend on RAID-5. We use an NVS size of 40K pages each of 4KB. Hence, NVS to backing store ratio is relatively large, namely, 4.52%, constituting a cache-sensitive scenario.

We vary the target throughput of SPC-1 Like from 300 IOPS to 3000 IOPS. At each target throughput, we allow a settling time of 10 mins, after which we record average response time over a period of 8 minutes.

It can be clearly seen that WOW dramatically outperforms CSCAN and even outperforms LRW. In particular, it can be seen that CSCAN finds it impossible to support throughput beyond 1070 IOPS. In contrast, WOW and LRW saturate, respectively, at 2453 and 2244 IOPS. In other words, WOW delivers a peak throughput that is 129% higher than CSCAN, and 9% higher than LRW.

Remark VI.1 (backwards bending) Observe that in Figures 9 and 10 when trying to increase the target throughput beyond what the algorithms can support, the throughput actually drops due to increased lock and resource contention. This "backwards bending" phenomenon is well known in traffic control and congestion where excess traffic lowers throughput and increases average response time.

VII. Conclusions

It is known that applying sophisticated disk scheduling algorithms such as SATF at upper levels in cache hierarchy is a fruitless enterprise. However, we have demonstrated that CSCAN can be profitably applied even at upper levels of memory hierarchy for effectively improving throughput. As the size of NVS grows, for a random write workload, the throughput delivered by CSCAN seems to grow logarithmically for single disks, RAID-10, and RAID-5.

CSCAN exploits spatial locality and is extremely effective in cache-insensitive storage configurations. However, it does not perform as well in cache-sensitive storage configurations, where it loses to LRW that exploits temporal locality. Since, one cannot *a priori* dictate/assume either a cache-sensitive or a cache-insensitive scenario, there is a strong need for an algorithm that works well in both regimes. We have proposed WOW which effectively combines CLOCK (an approximation to LRW) and CSCAN to exploit both temporal locality and spatial locality.

We have demonstrated that WOW convincingly outperforms CSCAN and LRW in various realistic scenarios using a widely accepted benchmark workload. WOW is extremely simple-to-implement, and is ideally suited for storage controllers and for most operating systems. WOW fundamentally increases the capacity of a storage system to perform more writes while minimizing the impact on any concurrent reads.

References

[1] J. Gray and P. J. Shenoy, "Rules of thumb in data engineering," in *ICDE*, pp. 3–12, 2000.

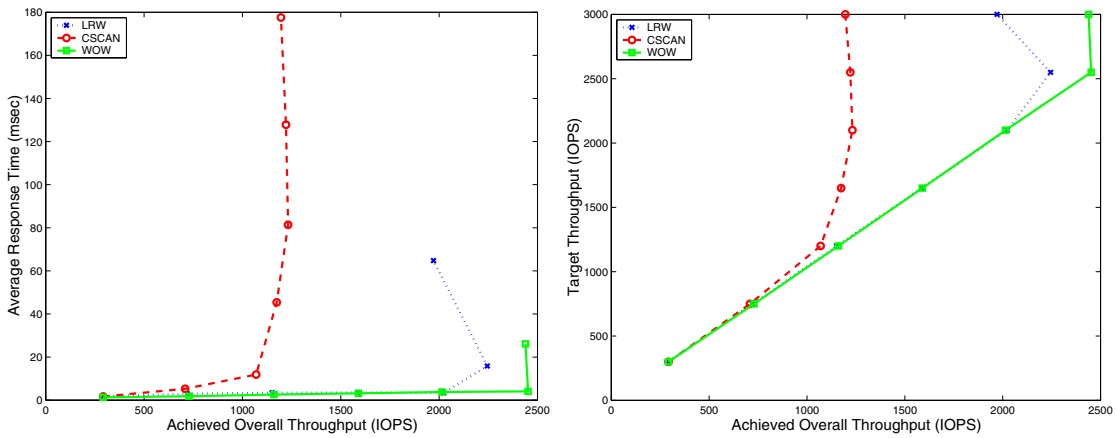

Fig. 10. A comparison of LRW, CSCAN, and WOW. The left panel displays achieved overall throughput versus achieved average response time. This set-up has significant temporal locality since the ratio of NVS size to the size of the backend is relatively high (4.52%). WOW increases the peak throughput over LRW by 9% and over CSCAN by 129%. The right panel shows the target throughput corresponding to the data points in the left panel. It can be clearly seen that CSCAN hits an insurmountable stiff wall at a much lower throughput.

[2] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," *IEEE Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[3] B. S. Gill and D. S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," in *USENIX*, 2005.

[4] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *FAST 04*, pp. 142–163, 2004.

[5] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *IEEE Trans. Parallel Distrib. Syst*, vol. 15, no. 6, pp. 505–519, 2004.

[6] J. Menon and M. Hartung, "The IBM 3990 disk cache," in *Proc. IEEE Comput. Soc. Int. COMPCON Conf.*, 1988.

[7] G. P. Copeland, T. Keller, R. Krishnamurthy, and M. Smith, "The case for safe RAM," in *VLDB*, pp. 327–335, 1989.

[8] J. Menon, "Performance of RAID5 disk arrays with read and write caching," *Distributed and Parallel Databases*, vol. 2, no. 3, pp. 261–293, 1994.

[9] J. Menon and J. Cortney, "The architecture of a fault-tolerant cached RAID controller," in *ISCA*, pp. 76–86, 1993.

[10] K. Treiber and J. Menon, "Simulation study of cached RAID5 designs," in *HPCA*, pp. 186–197, 1995.

[11] A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with nonvolatile caches," *IEEE Trans. Computers*, vol. 47, no. 2, pp. 228–235, 1998.

[12] P. Biswas, K. K. Ramakrishnan, and D. Towsley, "Trace driven analysis of write caching policies for disk," *Performance Evaluation Review*, vol. 21, no. 1, pp. 12–23, Jun 1993.

[13] Y. J. Nam and C. Park, "An adaptive high-low water mark destage algorithm for cached RAID5," in *PRDC*, pp. 177–184, 2002.

[14] J. K. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems," *Operating Systems Review*, vol. 23, no. 1, pp. 11–28, 1989.

[15] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[16] B. McNutt and S. Johnson, "A standard test of I/O cache," in *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.

[17] S. A. Johnson, B. McNutt, and R. Reich, "The making of a standard benchmark for open system storage," *J. Comput. Resource Management*, no. 101, pp. 26–32, Winter 2001.

[18] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," in *Operating Systems Review*, vol. 26, pp. 10–22, October 1992.

[19] P. Biswas, K. Ramakrishnan, D. Towsley, and C. Krishna, "Performance analysis of distributed file systems with non-volatile caches," in *Proc. 2nd Int. Symp. High Perf. Distributed Computing*, pp. 252–262, 1993.

[20] W. W. Hsu, A. J. Smith, and H. C. Young, "I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level," *ACM Trans. Database Syst.*, vol. 26, no. 1, pp. 96–143, 2001.

[21] L. Huang and T. Chiueh, "Experiences in building a software-based SATF scheduler," Tech. Rep. ECSL-TR81, SUNY at Stony Brook, July 2001.

[22] F. J. Corbató, "A paging experiment with the multics system," in *In Honor of P. M. Morse*, pp. 217–228, MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.

[23] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[24] M. Seltzer, P. Chen, and J. Ousterhout, "Disk scheduling revisited," in *Proc. USENIX Winter Tech. Conf.*, pp. 313–324, 1990.

[25] D. M. Jacobson and J. Wilkes, "Disk scheduling algorithms based on rotational position," tech. rep., HPL-CSP-91-7, HP Labs, Mar 1991.

[26] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *SIGMETRICS*, pp. 241–251, 1994.

[27] E. G. Coffman, L. A. Klimko, and B. Ryan, "Analysis of scanning policies for reducing disk seek times," *SIAM J. Comput.*, vol. 1, no. 3, pp. 269–279, 1972.

[28] P. J. Denning, "Effects of scheduling on file memory operations," in *Proc. AFIPS Spring Joint Comput. Conf.*, pp. 9–21, 1967.

[29] P. H. Seaman, R. A. Lind, and T. L. Wilson, "An analysis of auxiliary-storage activity," *IBM Systems Journal*, vol. 5, no. 3, pp. 158–170, 1966.

[30] A. G. Merten, *Some quantitative techniques for file organization*. PhD thesis, University of Wisconsin, 1970.

[31] R. Geist and S. Daniel, "A continuum of disk scheduling algorithms," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 77–92, 1987.

[32] T. R. Haining, *Non-volatile Cache Management For Improving Write Response Time with Rotating Magnetic Media*. PhD thesis, Ph.D. Dissertation, University of California, Santa Cruz, 2000.

[33] J.-F. Paris, T. R. Haining, and D. D. E. Long, "A stack model based replacement policy for a non-volatile cache," in *Proc. IEEE Sym. Mass Storage Sys.*, pp. 217–224, March 2000.