Improving throughput for small disk requests with proximal I/O

Jiri Schindler, Sandip Shete, Keith A. Smith
NetApp, Inc.

Abstract

This paper introduces proximal I/O, a new technique for improving random disk I/O performance in file systems. The key enabling technology for proximal I/O is the ability of disk drives to retire multiple I/Os, spread across dozens of tracks, in a single revolution. Compared to traditional update-in-place or write-anywhere file systems, this technique can provide a nearly seven-fold improvement in random I/O performance while maintaining (near) sequential on-disk layout. This paper quantifies proximal I/O performance and proposes a simple data layout engine that uses a flash memory-based write cache to aggregate random updates until they have sufficient density to exploit proximal I/O. The results show that with cache of just 1% of the overall disk-based storage capacity, it is possible to service 5.3 user I/O requests per revolution for random updates workload. On an aged file system, the layout can sustain serial read bandwidth within 3% of the best case. Despite using flash memory, the overall system cost is just one third of that of a system with the requisite number of spindles to achieve the equivalent number of random I/O operations.

1 Introduction

This paper focuses on an important but neglected aspect of file system performance: workloads that mix random writes with sequential reads to the same data. In particular, serial reads after random writes (SRARW) are common in many applications that are large consumers of storage in enterprise environments. For example, database systems typically acquire and update data through online transactional processing (OLTP), which is dominated by small writes, and subsequently read it in bulk for other tasks, such as analysis or backup. SRARW workloads are particularly problematic in large-scale deployments, which are often spindle-limited and too large to be moved to flash-based SSDs cost effectively.

Existing file system designs optimize either the serial read access or the random writes in a SRARW workload, but do so at the expense of the other operation type. At one end of the spectrum, *write-anywhere* file systems such as the Sprite log-structured file system (LFS) [27] and its descendants [19, 22, 3] are write optimized. They batch small or random writes into larger sequential allocations on disk, transforming updates of logically unrelated data into physically sequential I/O. However as they age, their data layout becomes fragmented, scattering related data across the disk. Thus, logically sequential access such as a database table scan leads to inefficient disk I/O. We have measured access to physically fragmented data at as little as 3% of the best-case serial read bandwidth. (See results in Section 5.3.)

At the other end of the spectrum, *update-in-place* file systems, such as FFS [21] and related designs [5, 32] are optimized for serial read and write access. These file systems attempt to allocate logically sequential data to physically sequential disk locations, providing good bandwidth for serial data access. However, this translates into inefficient non-sequential I/O, as destination blocks are predetermined by past allocation decisions, which are unlikely to be optimal in the face of random updates. Moreover, when such systems keep older versions of the data, they must perform a variant of copy-on-write [25], doubling the amount of inefficient random write I/O. Database systems often decouple this inefficient back-end I/O from foreground processing through the use of logging. The log then becomes a staging area for asynchronous bulk updates to the database tables. However, this technique alone does not mitigate the high cost of random I/Os to the back-end of a storage system that has limited I/O capacity.

To increase the back-end's effective I/O capacity without increasing disk (spindle) count, we introduce a new type of disk access pattern that we term *proximal I/O*. We demonstrate how proximal I/O leverages features of current disk drives to retire in a single revolution several I/Os scattered across dozens of tracks holding hundreds of thousands of sectors (Section 2). We propose a new data layout (Section 3), which shares the desired properties of existing copy-on-write, write-anywhere file systems that make random user writes efficient and allow for snapshots with minimal I/O overhead.

Using a prototype implementation of our data layout (detailed in Section 4), we show that with write cache sized only at 1% of the overall storage capacity, we can service 5.3 I/Os per revolution for workloads with random updates, all the while maintaining data layout on a heavily aged system that can deliver 97% of the bandwidth achieved with the best-case scenario of physically sequential layout (Section 5). The 5.3 I/Os serviced per revolution represent a $7\times$ improvement over the cost of a random disk I/O in traditional frame arrays with update-in-place data layout. With RAID5 and non-volatile write caches, they use four random disk I/Os at the back-end; with copy-on-write snapshots, this number doubles. In contrast, our approach uses about one disk I/O at the back end for every user-level random update when *both* RAID and copy-on-write are employed.

The primary contribution of this paper is the introduction of a data layout strategy that combines a small amount of flash memory with proximal I/O to efficiently service random updates to sequentially-allocated on-disk data without undermining on-disk locality. We provide efficiency both in performance and cost, significantly improving the performance of random writes at less than a third of the cost per IOPS of a pure disk solution. Finally, we present the first study to quantify the behavior of modern disk drives under proximal I/O access pattern.

2 Proximal I/O

Proximal I/O leverages the ability of modern disk drives to execute multiple I/Os per single revolution scattered across dozens of disk tracks. Given the 300-400 Gb/in^2 aerial density of the magnetic media in currently shipping disk drives, this translates to a range of hundreds of thousands of logical blocks (*LBN*s)[1]. We describe the interplay between seek-time profile and request scheduling that make proximal I/O possible.

In the material presented here, we focus primarily on one disk model (the Seagate Cheetah 15K.5, introduced to the market in late 2007). However, both the 15,000 RPM enterprise-class drives as well as the 7,200 RPM high-capacity nearline drives, colloquially referred to by their interfaces as respectively SCSI/FC and SATA drives, are capable of proximal I/O. Our measurements of over 20 different models of both drive types, representing several generations of the same family of drives and manufactured by four different vendors confirm the observations about proximal I/O described here.

2.1 Relevant technologies

Historically the seek profile, the plot of seek time as a function of radial distance, has been described by a continuous curve with two components: for small distances, one that is a square root of the cylinder distance and, for larger seek distances, one where seek time is a linear function of cylinder distance [28]. As observed by Schlosser et al. [31], the seek profile of more recent disks includes a third component: for very small seek distances of less than C cylinders, the seek time is nearly constant and is effectively equivalent to the track-switch time, or the time needed for the head to settle on a track.

The surface-serpentine disk layout adopted by more recent drives [1, 31] uses minimal seek time over a range of tracks. After mapping the last *LBN* to a given track, the disk firmware maps the next *LBN* to the adjacent track on the same surface rather than to the same track on a different surface. After mapping across C consecutive tracks, the next logical *LBN* is mapped to a sector on a track of a different surface C cylinders away. Thus, when accessing sequential run of *LBN*s, the disk heads will occasionally seek across the C cylinders to access the next logically sequential *LBN*. Using a disk extraction tool [29], we determined $C = 65$, which covers the range of 624,000 *LBN*s (1200 SPT $\times$ 65 cylinders $\times 8$ surfaces) for the 300 GB Cheetah 15K.5 disk.

The Shortest-Positioning-Time-First (SPTF) request scheduler [34] implemented in the disk firmware can effectively increase the throughput of serviced requests. It reorders requests to minimize the total positioning time (i.e., the sum of seek time and rotational latency) for each I/O request in the queue. With sufficiently large number of outstanding requests, it can lower the total positioning cost (i.e., the sum of the seek time and rotational latency) and service many more requests per unit of time [1].

2.2 Expressing request service time

Issuing only one request at a time to the disk negates the benefits of the SPTF scheduler. The service time of a small random request will then be equivalent to the sum of average seek (equivalent to 1/3 of the full-strobe seek) and rotational delay of 1/2 a revolution. For the Cheetah 15K.5 disk, this is is respectively 3.6 ms and 2 ms, resulting in service time of 5.6 ms. For a 7,200 RPM Western Digital RE3 nearline disk, the values are respectively 6.9 ms and 4.2 ms, yielding service time of 11.1 ms.

As these times for the two drives vary (by design) by a factor of $2\times$, we will use instead a relative measure of service time, here called *OP*, that lets us ignore the differences between disk types and their generations. Thus, 1 *OP* is the service time for a small random disk request or the measure of resources consumed when servicing a random disk I/O.

Our enterprise-class disk has the capacity to service about 180 I/Os per second, while the nearline disk only 90. These drives demonstrate a useful rule of thumb;

[1]The number of sectors per track (SPT) for recent 3.5” disk drives ranges between 800 and 2800. edge. The 15,000 RPM enterprise-class disks employing 65 mm platters have fewer SPT at their outer-most track compared to the 95 mm platters in the 7,200 RPM disk drives [2].

the average seek time of a disk is roughly equal to the time for a full rotation. Thus, the seek component of an OP is roughly 2/3 *OP* while the remaining portion is attributed to the rotational latency of half a revolution. This rule predicts that a disks can typically service 0.66 random requests per revolution. Our two drives do slightly better—0.71 for the enterprise-class disk and 0.75 for the nearline disk. This trend has held across many disk generations with different rotational speeds and seek times.

2.3 Measurements

To quantify the benefits of proximal I/O, we measured the per-request service time in a batch of requests, here called a strand, under a variety of conditions. We chose at random a location on the disk and controlled the span of *LBN*s covered by the requests as well as the number of requests in the strand issued simultaneously (i.e., the disk queue depth). The measured response time for the entire strand, listed in Table 1, is the sum of the service times of the individual requests in the strand.

To service a strand of requests, the disk must first seek to the general vicinity of the requests. Servicing the first request in the strand thus incurs the cost of an average seek in addition to some rotational latency. However, if the requests are near each other, servicing the remaining requests, incurs only some additional rotational latency and potentially minimal seek/track switch, since all requests in the strand are within *C* cylinders of each other. As we batch all requests, the disk is free to reorder them.

Figure 1 shows the effective per-request service time as the number of requests in the strand (and hence the queue depth) increases, expressed both absolutely and in relative units of *OP*s. The graph compares three different access modes. The track-approximate access limits the *LBN* span to 1024, the approximate size of the disk's track, rounded down to the power of two. The proximal access uses a span of 100,000 *LBN*s. The semi-sequential access represents the best possible disk access after sequential streaming [30] — the requests are carefully chosen such that each request is positioned at a different track and at an offset equivalent to the minimal seek/track switch time. For semi-sequential access, we need to know detailed disk drive parameters. On the other hand, proximal I/O does not require the knowledge of track switch time or precise track size (SPT).

We remark on the following trends. First, as the number of requests in the strand increases, the effective per-request service time decreases from 1 *OP* to 0.39 *OP* for a strand of 8 requests — a 2.5× improvement over the case with one request per strand. Second, both track-approximate and proximal mode are very similar, despite the ten-fold difference in the *LBN* span. And third, the

	Strand response time (ms)			
Requests per strand	2	4	6	8
Semi-sequential	5.9	7.3	8.6	9.9
Track-approximate	7.4	10.8	13.9	17.0
Proximal READ	7.4	11.2	14.2	17.1
Proximal WRITE	8.6	12.9	16.8	20.4

Table 1: Comparison of strand response times for Seagate Cheetah 15K.5. The mean service time of single READ request is 5.6 ms. For WRITE, it is 5.8 ms due to extra write-settle time.

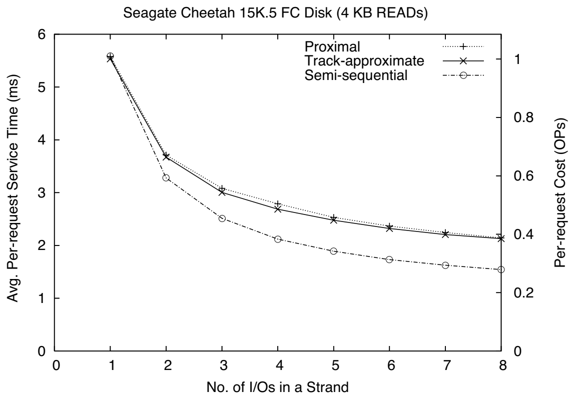


Figure 1: Per-request cost of small requests in a strand.

semi-sequential mode yields 0.23 *OP* for 8 requests per strand compared to 0.39 *OP* of proximal mode — an additional 1.7 times improvement. The results for WRITEs (not shown here) exhibit a similar trend; the slightly higher strand response time (see Table 1) is due to additional write-settle time for track-to-track seeks.

2.4 Detailed model comparisons

Two hypotheses might explain why we do not see values for proximal I/O that are closer to the semi-sequential mode. First, with randomly chosen blocks, some of them may land on the same track and the disk firmware opts to prefetch the remainder of the track before servicing other requests. Second, even without triggering prefetching, the random placement of the requests can cause extra (missed) revolutions as we describe below.

The semi-sequential access carefully chooses the placement of blocks so as to eliminate any rotational delays between requests after a track switch. With randomly chosen requests in proximal access, requests on different tracks can have rotational offset that is smaller than the time needed to switch tracks. The following paragraphs help illustrate how the disk scheduler minimizes overall rotational latency. They also show that it is the stochastic nature of the request placement rather than an artifact of the disk firmware causing the extra revolutions.

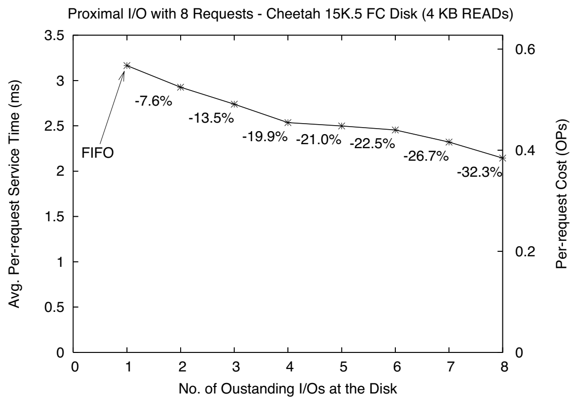


Figure 2: Per-request service time for strand of 8 requests with a varying number of READ requests outstanding at the drive. The first point is equivalent to a FIFO scheduler. The percentages next to the data points represent improvement relative to the FIFO data point. The second graph compares the modeled and the observed (measured) behavior.

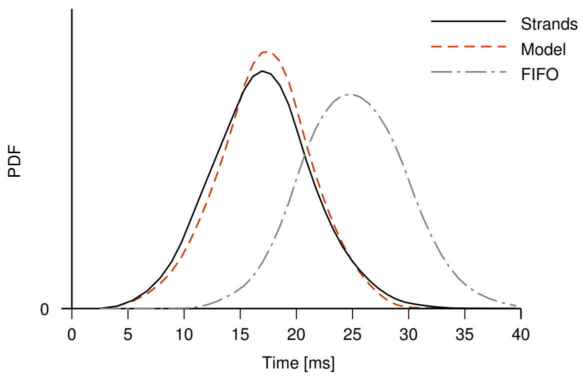


Figure 3: Model-predicted vs. measured (observed) values.

To verify our hypothesis, we set up an experiment, whose results are shown in Figure 2, where we fixed the number of requests per strand to 8 and varied the number of requests queued at the drive. With one request outstanding, the requests are serviced in FIFO order. As the number of outstanding requests increases, the disk scheduler can choose a request with smaller rotational latency, yielding a 32% reduction in per-request service time for a queue depth of 8 requests. This result confirms that proximal access effectively leverages the SPTF scheduler.

To obtain the expected number of revolutions needed to service a strand of requests with proximal I/O, we developed an analytical model for computing the expected strand response time and the probability distribution of missing revolution(s) for proximal I/O access. The model is based on the birthday paradox principle [33] and works at a high level as follows. It divides the disk into equally sized wedges or bins. When two requests (on different tracks) fall into the same bin, the disk heads cannot move fast enough to reach the second request in time and will have to service it during the next revolution. Because of the high track switch time relative to the revolution time (0.4–0.8 ms), there are only a few bins (days in a month) available and several requests are likely to fall into the same bin (i.e., having birthday on the same day). See Appendix A for model details.

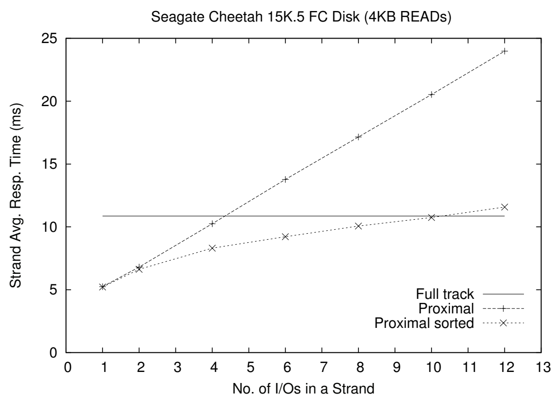
Figure 3 demonstrates the high accuracy of our model, comparing the measured and modeled distributions of the strand response times. The two curves labeled Strands and Model are very similar with nearly identical distributions. The curve labeled FIFO corresponds to measurements with one request outstanding at the disk drive, which is the scenario described in Figure 2.

2.5 Practical considerations

There are a few practical considerations for proximal I/O. First, we discovered that the manner in which we issue the requests in the strand is important. Issuing requests in a random order and relying solely on the scheduler's ability to reorder them does not work. However, when we issue the requests in ascending *LBN*, the scheduler works as expected — it picks from the strand a request with the lowest positioning cost, services it first, and reorders the remaining ones as necessary. Figure 4 shows the effect of issuing requests in ascending *LBN* for two different disk drives. The previously reported results include this workaround.

We attribute this limitation to two factors: the lack of the embedded CPU power (especially for nearline drives) and a firmware bug. We consulted disk manufactures who acknowledged both factors. In one case, our inquiry led to a fix in a subsequent firmware release. In practice, even with current limitations, pre-sorting requests is not an issue. Second, to engage the request scheduler properly, the strand must be issued to a non-empty disk queue. Again, in practice this limitation is not a problem. In many deployed systems, disks are seldom idle; they are busy servicing either client-generated workload or a variety of background scanning and grooming tasks. Third, we explored strands with at most 8 requests outstanding, although deeper queues would likely result in better results. This is again driven by a practical consideration. Many commercial storage systems [20, 8, 9] limit the number of pending requests to 4 or 8 to put a bound on the response time of a time-critical request.

As a final remark note that our experiments assumed a purely random workload, which we simulated by a uniform distribution of requests in the given range of *LBN*s. We believe that workloads that have more locality (but which are not sequential by nature) will benefit at least as much (if not more) from using proximal I/O.



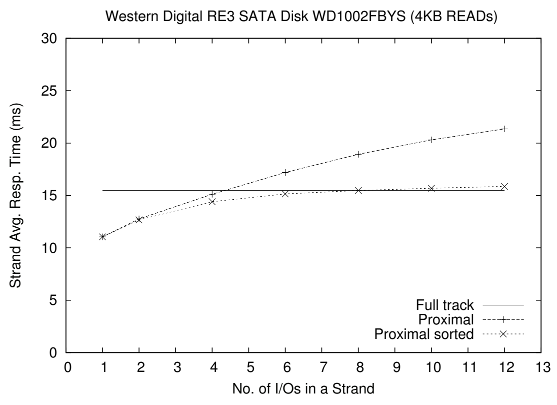


Figure 4: The effects of sorting requests in ascending *LBN* order before issuing them to the disk. The full track data serves as a reference of request scheduling efficiency and corresponds to reading approximately one full track. Same trends hold for WRITEs.

3 Data layout with proximal I/O

Previous section explained how proximal I/O can retire several I/Os per revolution within the span of approximately 100 tracks or half a million *LBN*s. In this section we describe the design principles for data layouts that leverage the proximal I/O construct. For our discussion, we will use the term block to designate the basic data allocation unit in file systems (typically 4 KB) to distinguish them from sectors, or *LBN*s, which are typically 512 bytes and serve as the basic unit of disk I/O.

3.1 Increasing I/O density

We start by considering how to increase the density of writes in the SRARW workload. The goal is to take a stream of random requests and produce sequences of I/O that will benefit from proximal I/O. As long as a storage system can produce a batch of several, say eight, write requests, and the data layout engine can place them within the span of $\approx 100,000$ blocks, each request will be serviced in time equivalent to much less than one revolution and consume only 3.2 *OP* of resources (i.e., 0.4 *OP* per request as shown in Figure 1) regardless of the previous position of the disk heads. In contrast, servicing eight blocks randomly scattered across the entire media will require 8 *OP*. Put differently, we need to find a way to increase the effective I/O density instead of spreading out a given batch of I/Os across the entire disk.

We use two complementary approaches to achieving the necessary I/O density. First, we leverage indirection when assigning data to their physical locations akin to inodes in file systems that map file offset to a physical address at the underlying storage. Write-anywhere systems with no-overwrite semantics [19, 27] already take advantage of this approach; random writes at the storage system interface are mapped to the same segment (allocation area) at the physical layer. Our technique expands on this notion by allocating data to free space in the vicinity of the previously written logically related data. Second, when the I/O density of 6–8 requests within the zone of effectiveness of proximal I/O is not enough, we complement the new type of write allocation described above with the use of a staging area. With a large-enough staging area, one can selectively pick appropriate requests and write-allocate them to achieve the required I/O density as determined by the disk technology.

Our approach contrasts with existing ones in several ways. Traditional frame arrays that export logical volumes composed of disk drives organized into a RAID group typically do not have much flexibility in mapping their blocks to the underlying devices. They stripe data in a round-robin fashion across the constituent disks. Such systems do not require any additional metadata; they can compute the disk number and disk offset with simple modulo and divide arithmetic. However, a given write operation at the storage interface will land at a specific location on the disk, negating the desire for decoupling the front-end workload from the back-end. As a result, they need much larger write caches to achieve the required write density compared to our approach.

A back-end system with hundreds of large-capacity near-line disk drives, will require hundreds of GB of staging area. Using that much NV-RAM (i.e., some form of battery-backed DRAM) would make the overall system cost prohibitively high (although there are commercial systems that offer such configurations [10, 9]). A more cost-effective solution is to use Flash memory an append-only log [23], at approximately 1/10th of the cost per GB. Another possibility is to use a dedicated disk as commercial relational database systems do for their log. However, with disk-based staging area, we would require some additional DRAM to hold the data during the destage operation to perform random reads from DRAM rather than disk during destage operation.

3.2 Overwrites and snapshots

In contrast to purpose-built storage systems [4, 7, 14, 18]) that function as fixed-content repositories or that handle specialized scientific workloads that write lots of intermediate results, most writes in commercial systems are (logical) overwrites or updates rather than new writes. For example, commercial databases and email servers [11] update individual records within database pages. New, or append-only, writes occur infrequently as these systems typically pre-allocate their table space by writing out empty (but non-zero) pages in bulk. Similarly, writes to objects containing virtual machine disk images create clones of a baseline golden image with relatively few unique blocks.

Proximal I/O can also reduce the overhead of preserving multiple versions of the same block, be it snapshots for fast recovery after a crash or keeping diverging replicas of original files. A storage system will turn an update into a copy-on-write operation that will write data to a new location. A write-anywhere file layout, for example, lends itself to keeping snapshots with very little overhead, as in WAFL [19]. Other systems, such as frame arrays, with direct mapping of logical blocks to disk locations must issue an extra I/O to preserve old block versions.

Both types of systems exhibit a similar shortcoming. A version (the new one in case of WAFL and the old one in case of frame arrays) of the data is put in a location that is convenient for the system *without* considering the semantic relationship to the original data. This can adversely affect the efficiency of subsequent reads. Logically related data may end up too far away from each other, incurring high positioning cost when they are both first written and then later on retrieved. Therefore, when a data layout engine maintains physical proximity of logically related data (be it a live version or a snapshot), it can leverage proximal I/O for copy-on-write of data that are updates in place from the client's perspective.

Most storage systems use RAID to protect their data against disk failures and grown media defects. The RAID read-modify-write (RMW) operation is not proximal I/O per se. However, we can combine copy-out and RMW operations and leverage proximal I/O; we can pipeline them such that we write out the just-read old version of the data within the effective span of proximal I/O in time before the disk spins around to write the new version of the block. With enough flexibility in the data layout, we can accomplish two RMW operations, that is four media accesses, in time equivalent to slightly more than one and a half revolution plus the initial seek.

3.3 Efficiency of reads

So far we have discussed proximal I/O in the context of writes. However, it can also improve the efficiency of subsequent reads. The careful placement of related versions of the data during writes allows the disk to collect physically non-contiguous blocks with minimal positioning overhead for logically sequential access. Proximal I/O can access both the current data as well as the snapshots with similar efficiency.

Systems that do not place logically related data near each other are likely to perform differently depending on which version of the data they access. For example, a sequential table scan on an aged system may be less efficient than one performed against a snapshot made earlier. In contrast, when systems can place related data within the span of blocks that can be serviced with proximal I/O, they will likely exhibit much smaller variations in performance regardless of the version/snapshot they are reading. This is because both the old and new versions of data blocks, as well as logically related unmodified blocks will be in close proximity of each other, allowing proximal I/O to read either old or new versions of the data with high efficiency.

3.4 Summary of key design points

We summarize the key design points of a data layout engine suited for proximal I/O:

1. Flexible mapping of object data to the physical on-disk location is an effective mechanism for increasing I/O density. Put differently, given a certain level of "randomness" in the front-end workload, systems with flexible per-block location pointers will need smaller staging area compared to systems that use rigid mapping of front-end blocks to on-disk physical locations.

2. The system needs to employ large-enough write staging area to achieve the required I/O density for the given front-end workload. Naturally, completely random workloads will require the largest size. In practice, workloads are rarely purely random — there are typically hot spots where relatively small portion of the data is updated frequently. These hotspots reduce the amount of staging area required for effective proximal I/O.

3. A data layout engine with built-in efficient copy-on-write mechanism is well suited for proximal I/O; only some adjustments will be necessary to marry the constraints of proximal I/O with their already existing mechanisms.

We conclude by examining the various access patterns encountered by enterprise storage systems. In addition to serial reads after random updates (SRARW) that we target with proximal I/O, we must consider random reads, sequential writes, and sequential reads not coupled with frequent (small) updates. It is our belief that storage systems already employ effective techniques that can handle these other access patterns. In our view, proximal I/O is the missing link that fixes the inefficiencies of disk accesses in today's deployments.

Increasing the size of read caches, for example by employing devices based on flash memory, effectively copes with random reads. A modicum of NV-RAM can turn sequential writes into efficient disk accesses. In contrast, increasing the efficiency of random updates requires large buffers. Finally, sequential reads (in the absence of small updates) are easy to achieve. For example, workloads typical for fixed-content repositories often write out and read entire objects. When individual objects would be too small for efficient disk I/O, they are grouped into larger allocation and access units [4, 35].

3.5 Target workloads

Our work targets workloads in large-scale storage systems and is motivated by the emergence of virtualized data center environments. The computing infrastructure includes a storage manager that allocates data from the underlying storage systems in large chunks or extents. For example, both the Oracle ASM [12] and VMWare VMFS [13] allocate data in 1 MB chunks. Storage systems in these environment in turn provide data management features such as fine-grain snapshots, writable clones, etc. [20].

In these environments SRARW workloads are typical. The storage manager reads and prefetches data in full allocation units (chunks). However updates are typically at a finer granularity—for Oracle ASM, the update size is equivalent to the DBMS page size (typically 4–8 KB). For VM hypervisors, the update size is governed by the block size of the file system in the VM guest operating system. The writes from the storage managers to the underlying storage systems may turn into copy-on-write (rather than update-in-place) operations in order to preserve older versions data for disaster recovery. Our work focuses on these logical update-in-place operations with serial reads for prefetching or OLAP data scans.

4 Prototype data layout engine

The goal of our work is not to build an entire new file system. Instead, we have built a *data layout engine* (DLE) that uses our staging and allocation algorithms to demonstrate the feasibility of using proximal I/O to greatly improve random write performance while maintaining (near) optimal serial performance for SRARW workloads. We believe these algorithms are readily adaptable to both update-in-place and write-anywhere file systems.

Our DLE is, in effect, a stripped-down object storage system. We store logical extents of data in a flat namespace, where each extent is named only by a unique ID. Extents can be created, read, written (and overwritten) and deleted. For simplicity, we only support reads and writes that are properly-aligned on block boundaries. Our DLE includes all of the necessary file systems structures to support this functionality, inode-like structures for each extent, allocation maps to track free space, and additional metadata to facilitate layouts friendly to proximal I/O.

Because we are primarily interested in addressing the SRARW workload, our DLE is designed to efficiently support moderately large extents (1 MB or larger)—large enough for the serial read portion of the workload to benefit substantially from sequential layout. Our DLE works correctly for smaller extents, but we have not tested or optimized its performance in those cases. We believe that for those workloads, file systems would benefit more from using allocation algorithms that are different from those implemented in our prototype DLE. We describe here only the major pieces of our prototype necessary to understand the experiments presented in Section 5.

4.1 Extent interface

The DLE operates on extents. An extent is a contiguous logical range of bytes. The DLE decides how to best allocate extent data into fixed-size blocks (4 KB in our prototype) of the underlying storage subsystem—a logical volume created from raw disks in a RAID group. Internally, each extent is represented by an inode, which is the root of a constant-height tree of indirect blocks. The leaf nodes of this tree contain the extent data.

4.2 Staging area

Our DLE uses a separate flash device as a staging area to accommodate random writes. When the DLE writes data into the staging area, it also updates the corresponding metadata including inode and indirect blocks for the just-written extent. Thus the staging area is the full-fledged home (albeit temporary) for new data, rather than a write cache with a copy of the data. When the system achieves the required I/O density (or the staging area runs out of capacity) we use proximal I/O to move data from the flash-based staging area to the on-disk location. More importantly, during destage, we make just-in-time allocation decisions for the best on-disk placement in relation to the data already-allocated to the disk. Because of our desire to keep previous versions of data for snapshots, we don't overwrite in place and instead write to a new location. When the destage operation finishes, we update the extent and DLE metadata accordingly to reflect its new on-disk location.

As part of its metadata, the staging area maintains a table mapping each block in the staging area to the extent and offset that the block belongs to. This allows us to efficiently locate items in the staging area for destage. Our DLE also uses the flash device to store its internal metadata, so metadata access does not interfere with the SRARW access patterns we wish to study.

4.3 Allocation policies

The more interesting feature of our prototype is the set of write allocation policies we implemented. When new data is written to an extent, we use the size of the write to determine whether to write the data to the staging area or directly to disk. In the current implementation this threshold is 168 KB—a number chosen to be approximately the break-even point between the response time of a random I/O of that size and a full-track read for our current disks. Although, we have not examined alternate settings, we believe that the precise value has little qualitative effect on our system; it only serves to distinguish between small and large writes.

We have three different I/O allocation scenarios in our system: small writes allocated to the staging area, large writes allocated on disk, and collections of small writes allocated on disk when destaging. We manage the staging area as an append-only log. Other more involved schemes are possible, but we have not explored them. When the staging area fills, we destage its full contents and start refilling it again from the beginning. As described earlier, when we write a block to the staging area, we update the metadata that points to it, freeing any on-disk block containing older data at that offset if that block is not used for a snapshot.

When we receive a large write request, we write it directly to disk, allocating new space if necessary, as when first writing an extent. Since we assume that extents will be large, and we want to provide good serial performance, we map large sequential ranges of an extent to similarly sized physical extents on disk that we call *allocation ranges*. By allocating at first fewer physical blocks than the size of the allocation range, we can provide extra space for future updates and write-anywhere-style snapshots [19] at the cost of a corresponding fraction of serial bandwidth. We have not yet explored this capability in our prototype.

We follow the recommendations of Chen et al. [6] for stripe unit size to approximate the disk track size. Given the current disk parameters, we chose 1 MB as the size for the RAID stripe unit and allocation range for near-line drives and 512 KB for enterprise-class drives. Note however, that we need not know the precise disk parameters. The allocation unit size is a configurable parameter in our allocation algorithm and can loosely follow technology trends over time as track size increases.

Small writes (or updates) are first written to a staging area and held there until sufficient number of random updates is accumulated to achieve the required proximal I/O density. At that point we collect the relevant data (using our metadata info) and destage them to their final place. That is where alternative storage technologies work to our advantage; we can perform random reads directly from the staging area backed by e.g., flash memory. If using disks instead, we need to perform (possibly multiple) sorting pass(es) and use additional DRAM.

Destage is a two phase process. First, by scanning the staging area tables, we identify sets of blocks that can be allocated together. We do this by sorting the blocks first by extent and then by logical offset within each extent. Second, from the extent metadata, we determine the allocation range(s) that contain related data i.e., data at the logical offsets immediately preceding and following the data being destaged. If there is enough space in the corresponding allocation range we simply write-allocate data there. When no additional space exists, we look for another allocation range that has enough free space to absorb the blocks and is in the vicinity of proximal I/O.

In the worst case we inspect up to approximately 100 allocation ranges (given current disk characteristics) for each group of blocks i.e., all blocks in the staging area belonging to a single extent and that are logically offset by the range of proximal I/O. In practice, this number is much smaller; when we wrote blocks to the staging area, we typically deallocate the older version of the block on disk, unless they are kept for snapshots. If we are destaging to an allocation range that had no underlying physical storage (i.e., we are writing to a sparse extent), we first allocate a physical extent for the allocation range, and then allocate the destage blocks within it. Figure 5 illustrates the destaging process, showing the layout of data in both flash memory and disk.

Our allocation algorithm uses two parameters dependent on disk technology trends: (a) the SPT governs the efficient allocation and serial I/O size and (b) minimal seek time governs the effective range of proximal I/O. The first parameter dictates the size of our allocation range, the second one, expressed in the number of allocation ranges, provides the flexibility in our allocation decisions during destage operation. The parameters need not closely follow the technology trends. One adjustment for every few disk generations is sufficient. The trends are evolving to our advantage (see discussion in Section 5.4).

4.4 RAID Layer

Our user-level prototype also includes a stand-alone RAID subsystem, which presents a logical volume abstraction to our DLE. This has several benefits compared to using an off-the-shelf one such as hardware RAID controller or a software implementation such as the `md` block device driver in Linux (our prototype platform).

Our RAID implementation offers fine control over scheduling requests to individual disks. We use Linux SCSI generic device (`/dev/sg`) interface that bypasses the kernel block device's buffer cache and the block device schedulers. Linux can issue SCSI commands directly to both SCSI/FC and SATA drives thanks to the libsata layer. Most importantly, our own implementation more closely emulates the operation of an enterprise-class RAID layer and includes features that are missing from the aforementioned RAID implementations.

First, we perform updates either by addition or subtraction so as to minimize the number of disks engaged in I/O operations. Second, just like many other RAID subsystems [20, 8, 9], we maintain additional information for every data block, including, a write generation number for lost write protection and additional data checksum. Since SATA disks support only 512-byte sectors, we must use a separate sector for the additional per-block information. We use 64 bytes of additional information per single 4 KB block grouped into one checksum block for every 63 data blocks, emulating the features of Data ONTAP [20] running on systems with commodity SATA disks. Thus, accessing one block above the DLE interface results in two distinct block accesses.

5 Results

We evaluate the effectiveness of proximal I/O using our DLE prototype. We first study random updates to large extents comprising a logical volume (LUN) exported by a storage system. and then analyze serial reads after our volume has been aged with many small random updates.

5.1 Experimental setup

Our prototype runs as a user-level process on a host with one dual core 3 GHz Intel CPU under Linux 2.6.24 (from stock Ubuntu Server 9.04 distribution). We use a 4+1 RAID4 of 1 TB Western Digital RE3 (WD1002FBYS) SATA drives. We chose these 7200 RPM drives despite their lower performance compared to their enterprise-class counterparts because they are more cost effective.

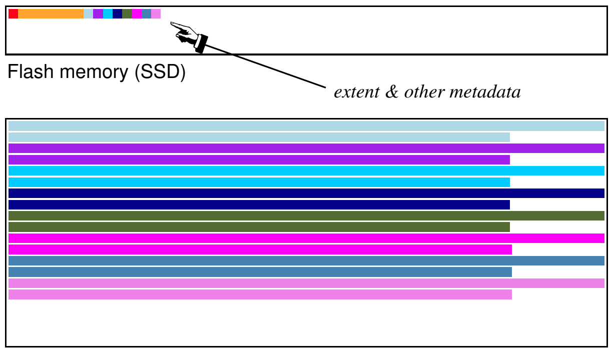
We fill our DLE with 16 MB extents to 89% of its capacity, writing them serially directly to the disks. We then issue 2000 small (4 KB) random updates per extent, thus re-allocating half of all blocks we initially wrote. Our DLE accumulates these updates to an SSD-based staging area, destaging them to back-end disk each time the staging area fills. For measuring serial reads after writes, we read every single extent from our aged DLE (in random order). These requests for 974 logically serial blocks at a time (governed by the fan-out of our indirect blocks) result in several scatter-gather disk I/Os. Figure 5 shows the layout during the execution of updates.

Before the DLE issues a set of requests to the RAID layer, we execute random I/Os to each of the constituent disks so as to avoid "short-stroking" (i.e., generating artificially short seeks due to using only a subset of the disk capacity). We wait for the disks complete the random I/Os and exclude these unrelated I/Os from our analysis. Executing a set of small updates results in many more individual disk I/Os than there are requests in the batch — the RAID layer needs to access the additional checksum blocks and to perform read-modify-write operations.

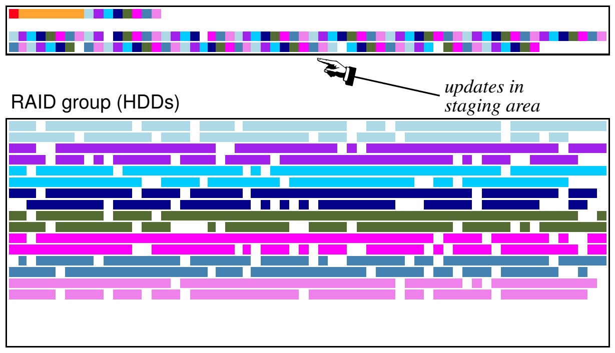
5.2 Random updates

The results for the random updates are summarized in Table 2. Each table row represents measurements with a different size of the staging area relative to the RAID group size. We collect statistics for each *batch* of I/O, where one batch is the disk I/O generated in destaging the accumulated changes to a single extent. Thus, a entire destage operation will generate one batch of I/O for each extent with at least one block in the staging area. We measure the mean response time and the number of user updates for each batch (columns 2 and 3). These times reflect the disk activity (i.e., the operations on the data). The DLE and extent metadata are updated on the SSDs, on average, with fewer than three I/Os for each batch. Since the SSD I/O service time is much smaller, the disk I/O dominates the batch response time.

We collect the service time for each disk I/O (computed as the difference between the completions of the last two I/Os). We list the mean number of disk I/Os (reads and writes across all five drives) in column 4. Column 5 shows the I/O amplification, the mean number of disk I/Os needed for each user-initiated update. Column 6 shows the equivalent number of disk I/Os serviced per revolution. Finally, we show for the data and parity disks the mean number of I/Os, per-I/O service time, and the resulting disk utilization.



(a) Initial setup with sequentially written extents.



(b) Aged layout with updates still in the Flash memory staging area.

Figure 5: An example of block allocation in the prototype DLE. Initially, extents are written out directly to the RAID group. The Flash memory (SSD) holds extent and DLE metadata. Random updates are first put into the Flash staging area. As the layout ages, the extents are no longer contiguously laid out. However, the DLE maintains the proximity of the related blocks of the same extent by "pluging" holes in the layout created by overwrites two destage operations.

Stage	Resp.	User	Disk	I/O	I/Os	Data disks			Parity disk		
Area	time	writes	I/Os	ampl.	p. rev.	I/Os	ST	Util.	I/Os	ST	Util.
Baseline	16.8 ms	1	8	8x	2.0	4	4.2	97%	4	4.2	97%
1%	129.5	47.5	295.3	6.2x	5.3	43.3	2.0	65%	122.0	1.0	91%
2%	155.0	85.2	465.3	5.5x	7.2	66.1	1.4	61%	201.1	0.7	93%
3%	182.9	119.7	614.8	5.1x	8.1	86.2	1.2	57%	269.9	0.6	94%
4%	227.6	149.3	723.7	4.8x	7.7	99.8	1.1	47%	324.3	0.6	86%
5%	235.9	179.1	847.6	4.7x	8.7	117.0	1.0	48%	379.8	0.6	95%
6%	278.4	226.1	1014.7	4.5x	9.0	136.3	0.9	44%	469.3	0.6	97%
7%	315.4	259.5	1151.5	4.4x	9.0	155.6	0.9	42%	529.2	0.6	97%
8%	320.9	254.8	1166.7	4.6x	8.7	163.5	0.8	43%	512.6	0.6	96%

Table 2: Random updates for various sizes of the staging area. Resp. time is the response time of the batch of user I/Os being destaged and ST is the the per disk I/O service time, both reported in milliseconds. I/O ampl. is the ratio of disk I/Os to user writes. The I/Os per revolution represents the number of I/Os serviced by a drive averages across both data and parity disks. The base data represents a system without staging area, whereby every user write results in RMW at the RAID back-end.

In our baseline data, we show the performance when a batch contains exactly one block. This has a latency of 16.8 ms, and results in 8 separate disk I/Os (an I/O amplification of 8×). Writing a single block to RAID4 results in 4 individual disk I/Os—a read and write of both the data disk and the parity disk. Updating the checksum incurs an additional 4 I/Os.

Next, we explore how our write allocation, coupled with 1% of staging area, leverages proximal I/O to improve the efficiency of disk accesses. We observe that, on average, 47.5 user updates result in 295.3 disk I/Os for a 6.2× amplification that are serviced in 139.5 ms. The per-I/O service time for the data and parity disk is thus 2.0 ms and 1.0 ms respectively. Even though the RAID4 parity disk is more efficient, it has to service many more I/Os and thus is the bottleneck.

Since the batch of I/Os is serviced by proximal I/O, we can retire on average 5.3 I/Os per revolution. Yet, as shown in Figure 1, we were only able to retire 1.9 I/Os per revolution in a strand of 8 requests (17.1 ms to retire 8 read requests with 4 ms rotational time). We achieve this improvement because of the greater I/O density; we are writing strands that contain many more blocks, typically within a single track or two. Also, the RAID layer must also update the checksum block for data blocks that are being written out. This further increases the number of disk I/Os, but also the I/O density — for most data blocks the checksum block is on the same track. For the same reason, we also see only a 6.2× write amplification (instead of 8×); we need to access the same checksum block only once for several data block updates.

As the size of the staging area increases, the batch size increases (from 47.5 to 369.7 updates for staging area of 1% and 10% respectively) and the destage operation for each batch becomes more efficient. The I/O amplification decreases from 6.2× to 4.5× and the number of disk I/Os serviced per revolution grows from 5.3 to 8.6.

5.3 Serial reads after random updates

Table 3 shows the results for sequential reads from an aged layout as depicted in Figure 5(b). Our system reads up to 974 logically consecutive blocks. Given the extent sizes, our DLE requests on average 819.2 blocks that are returned with a mean response time of 9.5 ms. This translates to effective bandwidth of 86.2 MB/s per disk (345 MB / 4 data disks in RAID group). Because of fragmented layout, the request for up to 974 logical blocks results in a batch of 47.5 logically sequential runs of blocks issued to the RAID group that are further broken into individual per-disk I/Os. Because of striping and the need to access the checksum block in addition to the data blocks, each disk services on average 20.7 individual I/O requests. Given the 1 MB stripe unit size on our RAID setup, the original request for 974 logically sequential blocks is typically serviced by three disks.

We repeated the same experiment on the non-aged data layout depicted in Figure 5(a) where extents are laid out on physically contiguous disk sectors. We measured mean response time of 9.2 ms, which translated to per-disk bandwidth of 89 MB/s. Thus, sequential reads after random updates on our system are within 3% of the the best-case scenario of physically contiguous layout.

Finally, we evaluated the performance of serial reads after random updates with write-anywhere-style allocation. In our system, we induced this behavior by eliminating the staging area and writing out data by greedily plugging the holes created by deletes of earlier versions of the data (we did not implement an LFS-style segment cleaner). In this setup, logically serial data increasingly dispersed over the disk over time, resulting in dramatically lower bandwidth compared to the baseline case.

	Per disk statistics			
	Read BW	**Diff**	**I/Os**	**Util.**
Baseline	89.0 MB/s		11.7	85%
Aged layout	86.2 MB/s	-3%	20.7	82%
Write-anywhere	2.6 MB/s	-97%	210.2	85%

Aged layout reads – detailed statistics			
	mean	**min**	**max**
Request response time (ms)	9.5	6.5	32.9
Request size (4 KB blocks)	819.2	200	974
Requests per batch	43.9	28	114
Span of blocks	1002.8	914	1008
Number of I/Os per disk	20.7	2	58
Per-disk resp. time (ms)	8.8	0.9	32.8

Aged layout – read response time quantiles

10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
7.4	7.5	7.6	7.7	7.9	8.1	8.4	22.1	27.8	31.9

Table 3: Serial reads after random updates.

5.4 System cost and technology trends

Lowering overall cost is one of the driving forces behind changing the internal architecture and design of commercial enterprise-scale storage systems. The adjustments to the write allocation policies presented here coupled with deployment of some additional device(s) for the staging area is but one example of such force. Making the prevalent access pattern (e.g., the serial read after random write described here) more efficient allows the system to run workloads with larger I/O demand for the same dollar cost. We now explore the trade off between the cost of additional hardware for the staging area and the resulting improvement in the back-end disk I/O capacity.

Consider the WD1002FBYS disk drive we used in our experiments. It has a measured average seek time of 7.5 ms and rotational speed of 7,200 RPM. With the time of 8.4 ms for a single rotation, the mean time to service a random I/O is 11.7 ms. Thus our drive can perform 86 random IOPS. With a street price of \$130, this means we are paying \$1.52 per IOPS. Now consider the effects of adding 1% of capacity as a flash staging area. Table 2 shows that in this configuration we can write 5.3 blocks in a single revolution. Adding an average seek means that our system performs 5.3 writes in 15.9 ms, or 3 ms per write. This is equivalent to 333 random IOPS, an improvement of 289% over the basic disk solution.

Adding the flash staging area increases the cost of the system. With cost of flash at \$3.13/GB, based on a 160 GB Intel X25 SSD with a street price of \$500, a 1% staging area for our 1 TB drive requires 10 GB of flash, increasing our cost by \$31.30, to a total of \$161.30 for a configuration capable of 333 random IOPS. Thus, in our system, the cost is \$0.48 per IOPS, less then a third of the per IOPS cost of the raw disk drive.

With our system, we pay an extra 25% to add a flash staging area and in return we get nearly a 3$\times$ performance increase on random writes, while preserving near sequential on-disk layout.[2] Note that these numbers are pessimistic. They assume that the staging area is scaled to the entire disk-based storage capacity. In reality, the staging area need only be 1% of the write working set, further reducing the flash costs.

We conclude by considering the impact of technology trends on the effectiveness of proximal I/O. The disk trends are in our favor. Growing areal media densities

[2]These numbers are shown here only to illustrate our point. our simplified model considers only the cost of individual devices. Also, we ignore many practical system issues such as RAID group size, etc.

(i.e., the increase in both SPT and track density), increase the span of *LBN*s over which proximal I/O will be effective. Larger span gives more options to our system to lay out its data. Similarly, as the flash memory cost decreases, the relative size of the deployed staging area may likely increase relative to the disk storage capacity. This will also increase the effectiveness of our destage process. In the end however, the ratio of flash memory to disk capacity will be driven by customer needs and their ability to get the right performance for the least cost.

6 Related Work

Our work explores the design principles for a data layout suitable for the SRARW workload while leveraging a more efficient disk access pattern. We review some additional related work not mentioned earlier in the paper.

Our data layout is similar in principle to the data journalling mode employed by some journalling file systems [5, 26, 32]. As in those systems, we write data initially to a designated staging area (journal, separate device etc.) and later on destage them to their final location. The difference in our approach, which utilizes proximal I/O, is that the efficiency of our destage operation is much much higher; journalling file systems typically write to a specific location on the disk constrained by their "overwrite-in-place" policy. In contrast, we destage data with fewer constraints offered by the span of blocks in proximal I/O. Additionally, we can consider the best location with respect to the related data and thus make the write operations more efficient.

The Disk Caching Disk (DCD) [24] explored a different technique for using write caching to improve storage system performance. DCD aggregates small writes on a separate caching disk, achieving serial performance when flushing dirty data from the buffer cache. During idle periods it destages data from the cache disk to its home on the primary disk. This design improves the latency of small writes, but does not leverage proximal I/O to achieve better I/O efficiency. A similar technique has also been used in database systems [16].

The idea of proximal I/O combines and expands on the observations about (1) efficient disk access across adjacent tracks with minimal positioning cost [30] and (2) minimal positioning cost when seeking across an ever-increasing range of cylinders [31]. Unlike semi-sequential access, however, proximal I/O does not require detailed knowledge of disk geometry or specialized device interface that provides the position of the next semi-sequential block relative to the current position; it works on systems with standardized interfaces (SATA or SCSI) and off-the-shelf commodity disk drives.

Our DLE design relies on write-anywhere allocation, similar to LFS [27], WAFL [19] and related designs such as ZFS [22] and btrfs [3]. Like these systems, it never overwrites old data in place, making it possible to preserve older versions of data, or snapshots, with minimal I/O overhead. Traditional write-anywhere file systems batch temporally related dirty data for efficient disk writes. Thus logical data locality is lost, requiring segment cleaning [27] or other defragmentation techniques [15] to re-establish sequential layout. In contrast, our DLE allocates data close to logically related on-disk data, preserving logical locality with proximalI/O plus the staging area to achieve efficient write performance.

The Loge [17] disk controller represents another variation of write-anywhere; it virtualizes block addresses so that it can write incoming data at the free locations nearest to the current disk head location. However, the work does not target SRARW workloads; it explicitly assumed that randomly written data would also be randomly read. In principle, many aspects of our DLE design could be implemented in a Loge-like disk controller rather than in a file system, although it would loose the semantic information about which blocks of data are logically related and are likely to be read together. Moreover, our design does not require detailed knowledge of disk head position and thus is time-invariant.

Appendix A: Proximal I/O model

Our objective is to find the expected number of revolutions needed to serve D requests in a strand. Recall that a strand is a collection of proximal I/Os that are sent together to a disk and that are close enough such that servicing any one of the requests incurs a minimal seek equivalent to head/track switch time.

Assume there are SPT sectors per track and D requests, each of size S sectors, and a seek between each request in the strand equivalent to head switch time, H. We express H in terms of the number of sectors that pass by the disk head during track switch. We can formulate the problem of finding the expected number of revolutions in terms of binning requests into B bins. Each request of size S is then randomly placed into any one of the K slots along a circular track. This is analogous to a roulette wheel with K slots and D balls spun simultaneously.

With such a formulation, if two balls (i.e., requests on different tracks) fall into the same bin, that is, if they are within $K \times S/H$ slots, the disk arm cannot service those requests in a single revolution and we get

$$B = \frac{SPT}{H} = \frac{K \times S}{H}.$$

Let's express the probability, Q_1, that no two requests are in the same bin. This is analogous to the probability that no extra revolutions are required when servicing a schedule of D requests in a strand. This can be solved by the birthday paradox problem, where we look for the probability that no two people out of a group of n people in a room have a birthday on the same day out of b possible, and equally likely, birthdays.

$$Q_1 = \frac{b!}{(b-n)!b^n}$$

Using our analogy, we have B bins, which is equivalent to the b possible birthdays, and D, the number of requests in a strand, is the number of people, n. This is equivalent to not using any extra revolutions (since each request is in a separate bin) when servicing the D requests.

We can now calculate the probability that at least one extra revolution will be required as

$$P_1 = 1 - Q_1.$$

More generally, the probability that a birthday is shared by exactly k (and no more) people is expressed as [33]

$$Q_k(n,b) = \sum_{i=1}^{\lfloor n/k \rfloor} \left(\frac{n!b!}{b^{ik} i!(k!)^i (n-ik)!(b-i)!} \sum_{j=1}^{k-1} Q_{k-1} \frac{(b-i)^{n-ik}}{b^{n-ik}} \right)$$

This is equivalent to the probability of servicing a given strand in k revolutions or using exactly $k-1$ extra revolutions. This assumes that each request landed on a separate track and requires a track switch when servicing it.

The probability that we will require at least k extra revolutions in servicing a request (or that $k+1$ or more people share a birthday in our analogy), we have

$$P_k = 1 - \sum_{i=1}^{k} Q_i$$

Now let's express the probability that we will not use any extra revolutions when servicing a strand as a function of number of sectors, H, that pass by during track switch time. With values for the Seagate Cheetah 15K.5 disk's first zone we have $SPT = 1200$, track switch time 0.475 ms, $H = SPT \times \lceil 0.475 \text{ ms}/4 \text{ ms} \rceil = 142$, and the number of bins $B = 8.45$. Therefore, we set $\lfloor B \rfloor = 8$, meaning that this disk can at best schedule 8 proximal I/Os in a revolution when the requests are properly offset from each other. With strand where $D = 8$, the probability of not using any extra revolutions is close to zero.

We express the expected number of revolutions for servicing a strand of D requests as

$$E[\text{Revs}] = \frac{1}{2} + \sum_{i=1}^{D} iQ_i(D, SPT/H)$$

For $D = 8$, we get $E[\text{Revs}] = 3.4$, assuming that each request lands on a separate track. Normalized (or per-request) number of revolutions is then 0.43.

Next, we assume eight requests in a strand even though this disk can service at best six in a single revolution. We choose the value of eight because it gives, on average, 12% lower per-request service time compared to a strand with $D = 6$. Adding an initial average seek of 3.5 ms for each strand, the per-request service time is 2.16 ms or 17.26 ms for the entire strand of $D = 8$ with variance $\sigma^2 = 9$ ms. This comes to within 1% of the measured mean service time of 17.14 ms with $\sigma^2 = 9.6$ ms.

Finally, we examine the probability of using exactly one, two, three, and so on, revolutions when servicing a strand of $D = 8$ requests. From our model, the most prevalent value is two extra revolutions (three in total). When $D = 6$ (with $H \approx 7$ for our disk), the probability of not using any additional revolutions is still only 0.02.

Acknowledgments

We would like to thank the program committee and the anonymous reviewers for their helpful comments. In particular, we thank our shepherd, Arif Merchant, for his insight and suggestions for improving this paper. Thanks also to Steve Kleiman for asking the questions that started us down this path and for many fruitful conversations along the way. Finally, we thank Jeffrey Heller, Art Harkin, and the all of NetApp's Advanced Technology Group (ATG) for their advice, support, and suggestions—and for making it fun to come to the office everyday!

References

[1] Dave Anderson. You don't know jack about disks. *Queue*, 1(4):20–30, 2003.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface—SCSI vs. ATA. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 245–257, 2003.

[3] Valerie Aurora. A short history of btrfs. `http://lwn.net/Articles/342892`, Jul 2009.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, Oct 2010.

[5] Mingming Cao, Theodore Y. Tso, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of Ottawa Linux Symposium (OLS)*, pages 69–96, Jul 2005.

[6] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *Computer Surveys*, 26(2):145–185, 1994.

[7] EMC Corporation. EMC Centera: Content Addressed Storage. `http://www.emc.com/products/systems/centera.jsp`, 2007.

[8] EMC Corporation. CLARiiON CX4 Series. `http://www.emc.com/products/series/cx4-series.htm`, 2010.

[9] EMC Corporation. Symmetrix DMX-4: Enterprise networked storage system. `http://www.emc.com/products/detail/hardware/symmetrix-dmx-4.htm`, 2010.

[10] IBM Corporation. IBM System Storage DS8000 Turbo. `http://www-03.ibm.com/systems/storage/disk/ds8000/index.html`, 2010.

[11] Microsoft Corporation. Microsoft Exchange Server. `http://www.microsoft.com/exchange/2010/en/us/default.aspx`, 2010.

[12] Oracle Corporation. Oracle database storage administrator's guide 11g release 1 (11.1). `http://download.oracle.com/docs/cd/B28359_01/server.111/b31107/toc.htm`, 2008.

[13] VMware Corporation. VMware vSphere. `http://www.vmware.com/products/vmfs/index.html`, 2010.

[14] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a scalable secondary storage. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 197–210, 2009.

[15] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, Jun 2008.

[16] Klaus Elhardt and Rudolf Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems*, 9:503–525, 1984.

[17] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing disk controller. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 237–251, Jan 1992.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[19] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

[20] NetApp Inc. Data ONTAP 8. `http://www.netapp.com/us/products/platform-os/data-ontap-8/`, 2010.

[21] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *Transactions on Computer Systems*, 2(3):181–197, 1984.

[22] Sun Microsystems. ZFS at OpenSolaris community. `http://opensolaris.org/os/community/zfs/`.

[23] Sun Microsystems. Logzilla: Hybrid storage pools in the 7410. `http://dtrace.org/blogs/ahl/2008/11/10/hybrid-storage-pools-in-the-7410/`, 2008.

[24] Tycho Nightingale, Yiming Hu, and Qing Yang. The design and implementation of a DCD device driver for unix. In *Proceedings of USENIX Annual Technical Conference*, pages 295–308, Jun 1999.

[25] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Transactions on Storage*, 1(2):190–212, 2005.

[26] Hans Reiser. Reiserfs. `http://www.namesys.com/`.

[27] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

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

[29] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, 1999.

[30] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: A disk array volume manager for orchestrated use of disks. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 159–172, Mar 2004.

[31] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On multidimensional data and modern disks. In *Pro-*

ceedings of Conference on File and Storage Technologies (FAST), pages 225–238, Dec 2005.

[32] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of USENIX Annual Technical Conference*, pages 1–14, Jan 1996.

[33] Eric Weisstein. Birthday problem. `http://mathworld.wolfram.com/BirthdayProblem.html`, 2007.

[34] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *SIGMETRICS Perform. Eval. Rev.*, 22(1):241–251, 1994.

[35] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 269–282, Feb 2008.