Deconstructing Storage Arrays

Timothy E. Denehy, John Bent, Florentina I. Popovici,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Department of Computer Sciences, University of Wisconsin, Madison

ABSTRACT

We introduce Shear, a user-level software tool that characterizes RAID storage arrays. Shear employs a set of controlled algorithms combined with statistical techniques to automatically determine the important properties of a RAID system, including the number of disks, chunk size, level of redundancy, and layout scheme. We illustrate the correctness of Shear by running it upon numerous simulated configurations, and then verify its real-world applicability by running Shear on both software-based and hardware-based RAID systems. Finally, we demonstrate the utility of Shear through three case studies. First, we show how Shear can be used in a storage management environment to verify RAID construction and detect failures. Second, we demonstrate how Shear can be used to extract detailed characteristics about the individual disks within an array. Third, we show how an operating system can use Shear to automatically tune its storage subsystems to specific RAID configurations.

Categories and Subject Descriptors: D.4.2 [Storage Management]: Storage hierarchies

General Terms: Measurement, Performance.

Keywords: Storage, RAID.

1. INTRODUCTION

Modern storage systems are complex. For example, a high-end storage array can contain tens of processors and hundreds of disks [8] and a given array can be configured many different ways, most commonly using RAID-0, RAID-1, or RAID-5. However, regardless of their internal complexity, RAID arrays expose a simple interface consisting of a linear array of blocks. All of the internal complexity is hidden; a large array exports exactly the same interface as a single disk.

This encapsulation has many advantages, the most important of which is *transparent* operation of unmodified file systems on top of any storage device. But this transparency has a cost: users and applications cannot easily obtain more information about the storage system. For example, most storage systems do not reveal how data blocks are mapped to each of the underlying disks and it is well known that RAID configuration has a large impact on performance and reliability [4, 17, 22, 29]. Furthermore, despite the fact that configuring a modern array is difficult and error-prone, administrators are given little help in verifying the correctness of their setup.

In this paper, we describe Shear, a user-level software tool that automatically identifies important properties of a RAID. Using this tool to characterize a RAID allows developers of higher-level software, including file systems and database management systems, to tailor their implementations to the specifics of the array upon which they run. Further, administrators can use Shear to understand details of their arrays, verifying that they have configured the RAID as expected or even observing that a disk failure has occurred.

As is common in microbenchmarking, the general approach used by Shear is to generate controlled I/O request patterns to the disk and to measure the time the requests take to complete. Indeed, others have applied generally similar techniques to single-disk storage systems [23, 27, 30]. By carefully constructing these I/O patterns, Shear can derive a broad range of RAID array characteristics, including details about block layout strategy and redundancy scheme.

In building Shear, we applied a number of general techniques that were critical to its successful realization. Most important was the application of *randomness*; by generating random I/O requests to disk, Shear is better able to control its experimental environment, thus avoiding a multitude of optimizations that are common in storage systems. Also crucial to Shear is the inclusion of a variety of *statistical clustering techniques*; through these techniques, Shear can automatically come to the necessary conclusions and thus avoid the need for human interpretation.

We demonstrate the effectiveness of Shear by running it upon both simulated and real RAID configurations. With simulation, we demonstrate the breadth of Shear, by running it upon a variety of configurations and verifying its correct behavior. We then show how Shear can be used to discover interesting properties of real systems. By running Shear upon the Linux software RAID driver, we uncover a poor method of parity updates in its RAID-5 mode. By running Shear upon an Adaptec 2200S RAID controller, we find that the card uses the unusual left-asymmetric parity scheme [13].

Finally, we demonstrate the utility of the Shear tool through three case studies. In the first, we show how administrators can use Shear to verify the correctness of their configuration and to determine whether a disk failure has occurred within the RAID array. Second, we demonstrate how Shear enables existing tools [23, 27, 30] to extract detailed information about individual disks in an array. Third, we show how a file system can use knowledge of the underlying RAID to improve performance. Specifically, we show that a modified Linux ext2 file system that performs *stripe-aware writes* improves sequential I/O performance on a hardware RAID by over a factor of two.

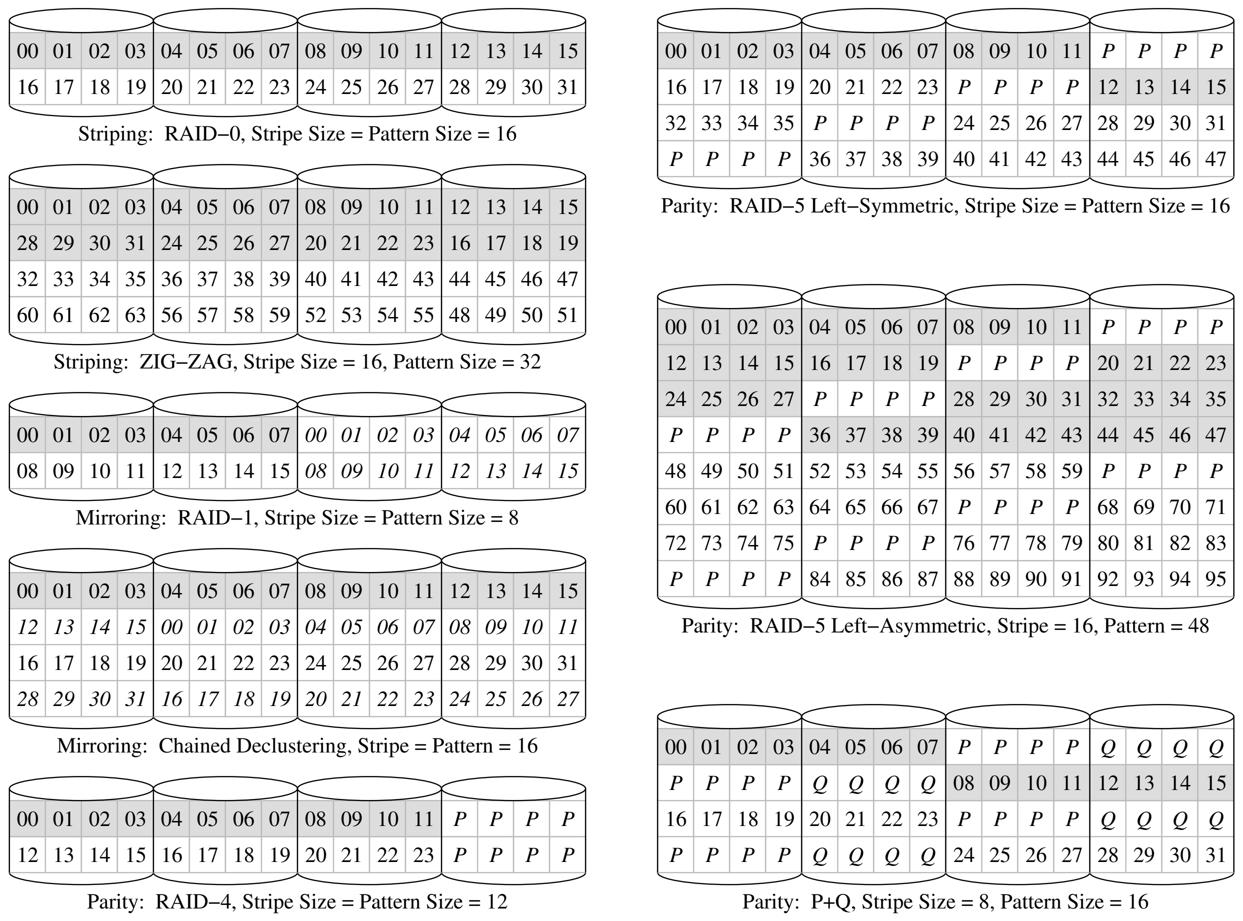

00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Striping: RAID−0, Stripe Size = Pattern Size = 16
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
28 29 30 31 24 25 26 27 20 21 22 23 16 17 18 19
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
60 61 62 63 56 57 58 59 52 53 54 55 48 49 50 51
Striping: ZIG−ZAG, Stripe Size = 16, Pattern Size = 32
00 01 02 03 04 05 06 07 00 01 02 03 04 05 06 07
08 09 10 11 12 13 14 15 08 09 10 11 12 13 14 15
Mirroring: RAID−1, Stripe Size = Pattern Size = 8
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
12 13 14 15 00 01 02 03 04 05 06 07 08 09 10 11
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
28 29 30 31 16 17 18 19 20 21 22 23 24 25 26 27
Mirroring: Chained Declustering, Stripe = Pattern = 16
00 01 02 03 04 05 06 07 08 09 10 11 P P P P
12 13 14 15 16 17 18 19 20 21 22 23 P P P P
Parity: RAID−4, Stripe Size = Pattern Size = 12
00 01 02 03 04 05 06 07 08 09 10 11 P P P P
16 17 18 19 20 21 22 23 P P P P 12 13 14 15
32 33 34 35 P P P P 24 25 26 27 28 29 30 31
P P P P 36 37 38 39 40 41 42 43 44 45 46 47
Parity: RAID−5 Left−Symmetric, Stripe Size = Pattern Size = 16
00 01 02 03 04 05 06 07 08 09 10 11 P P P P
12 13 14 15 16 17 18 19 P P P P 20 21 22 23
24 25 26 27 P P P P 28 29 30 31 32 33 34 35
P P P P 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 P P P P
60 61 62 63 64 65 66 67 P P P P 68 69 70 71
72 73 74 75 P P P P 76 77 78 79 80 81 82 83
P P P P 84 85 86 87 88 89 90 91 92 93 94 95
Parity: RAID−5 Left−Asymmetric, Stripe = 16, Pattern = 48
00 01 02 03 04 05 06 07 P P P P Q Q Q Q
P P P P Q Q Q Q 08 09 10 11 12 13 14 15
16 17 18 19 20 21 22 23 P P P P Q Q Q Q
P P P P Q Q Q Q 24 25 26 27 28 29 30 31
Parity: P+Q, Stripe Size = 8, Pattern Size = 16

Figure 1: Examples and Terminology. ***This picture displays a number of four disk arrays using several of the layout patterns discussed in the paper. The numbers represent blocks, P and Q indicate parity blocks, and redundant data is denoted with italics. In each case, the chunk size is four blocks and the stripe size and pattern size in blocks are listed. Each array depicts at least two full patterns for the given layout scheme, the first of which is shaded in gray.***

The rest of this paper is organized as follows. In Section 2 we describe Shear, illustrating its output on a variety of simulated configurations and redundancy schemes. Then, in Section 3, we show the results of running Shear on software and hardware RAID systems, and in Section 4, we show how Shear can be used to improve storage administration and file system performance through three case studies. Finally, we discuss related work in Section 5 and conclude in Section 6.

2. SHEAR

We now describe Shear, our software for identifying the characteristics of a storage system containing multiple disks. We begin by describing our assumptions about the underlying storage system. We then present details about the RAID simulator that we use to both verify Shear and to give intuition about its behavior. Finally, we describe the algorithms that compose Shear.

2.1 Assumptions

In this paper, we focus on characterizing block-based storage systems that are composed of multiple disks. Specifically, given certain assumptions, Shear is able to determine the mapping of logical block numbers to individual disks as well as the disks for mirrored copies and parity blocks. Our model of the storage system captures the common RAID levels 0, 1, 4, and 5, and variants such as P+Q [4] and chained declustering [11].

We assume a storage system with the following properties. Data is allocated to disks at the block level, where a *block* is the minimal unit of data that the file system reads or writes from the storage system. A *chunk* is a set of blocks that is allocated contiguously within a disk; we assume a constant chunk size. A *stripe* is a set of chunks across each of D data disks.

Shear assumes that the mapping of logical blocks to individual disks follows some repeatable, but unknown, pattern. The *pattern* is the minimum sequence of data blocks such that block offset i within the pattern is always located on disk j; likewise, the pattern's associated mirror and parity blocks, i_m and i_p, are always on disks k_m and k_p, respectively. Note that in some configurations, the pattern size is identical to the stripe size (*e.g.*, RAID-0 and RAID-5 left-symmetric), whereas in others the pattern size is larger (*e.g.*, RAID-5 left-asymmetric). Based on this assumption, Shear cannot detect more complex schemes, such as AutoRAID [29], that migrate logical blocks among different physical locations and redundancy levels.

Figure 1 illustrates a number of the layout configurations that we analyze in this paper. Each configuration contains four disks and uses a chunk size of four blocks, but we vary the layout algorithm and the level of redundancy.

RAID systems typically contain significant amounts of memory for caching. Shear currently does not attempt to identify the amount of storage memory or the policy used for replacement; however, techniques developed elsewhere may be applicable [2, 23, 28, 30]. Due to its use of random accesses and steady-state behavior, Shear operates correctly in the presence of a cache, as long as the cache is small relative to the storage array. With this assumption, Shear is able to initiate new read requests that are not cached and perform writes that overwhelm the capacity of the cache.

Our framework makes a few additional assumptions. First, we assume that all of the disks are relatively homogeneous in both performance and capacity. However, the use of random accesses again makes Shear more robust to heterogeneity, as described in more detail below. Second, we assume that Shear is able to access the raw device; that is, it can access blocks directly from the storage system, bypassing the file system and any associated buffer cache. Finally, we assume that there is little traffic from other processes in the system; however, we have found that Shear is robust to small perturbations.

2.2 Techniques

The basic idea of Shear is that by accessing sets of disk blocks and timing those accesses, one is able to detect which blocks are located on the same disks and thus infer basic properties of block layout. Intuitively, sets of reads that are "slow" are assumed to be located on the same disk; sets of reads that are "fast" are assumed to be located on different disks. Beyond this basic approach, Shear employs a number of techniques that are key to its operation.

Randomness: The key insight employed within Shear is to use random accesses to the storage device. Random accesses are important for a number of reasons. First, random accesses increase the likelihood that each request will actually be sent to a disk (*i.e.*, is not cached or prefetched by the RAID). Second, the performance of random access is dominated by the number of disk heads that are servicing the requests; thus Shear is able to more easily identify the number of disks involved. Third, random accesses are less likely to saturate interconnects and hide performance differences. Finally, random accesses tend to homogenize the performance of slightly heterogeneous disks: historical data indicates that disk bandwidth improves by nearly 40% per year, whereas seek time and rotational latency improve by less than 10% per year [10]; as a result, disks from different generations are more similar in terms of random performance than sequential performance.

Steady-state: Shear measures the steady-state performance of the storage system by issuing a large number of random reads or writes (*e.g.*, approximately 500 outstanding requests). Examining steady-state performance ensures that the storage system is not able to prefetch or cache all of the requests. This is especially important for write operations that could be temporarily buffered in a write-back RAID cache.

Statistical inferences: Shear automatically identifies the parameters of the storage system with statistical techniques. Although Shear provides graphical presentations of the results for verification, a human user is not required to interpret the results. This automatic identification is performed by clustering the observed access times with K-means and X-means [18]; this clustering allows Shear to determine which access times are similar and thus which blocks are correlated.

Safe operations: All of the operations that Shear performs on the storage system are safe; most of the accesses are read operations and those that are writes are performed by first reading the existing data into memory and then writing out the same data. As a result, Shear can be run on storage systems containing live data and this allows Shear to inspect RAIDs that appear to have disk failures or other performance anomalies over time.

2.3 Simulation Framework

To demonstrate the correct operation of Shear, we have developed a storage system simulator. We are able to simulate storage arrays with a variety of striping, mirroring, and parity configurations; for example, we simulate RAID-0, RAID-1, RAID-4, RAID-5 with left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric layouts [13], P+Q redundancy [4], and chained declustering [11]. We can configure the number of disks and the chunk size per disk. The storage array can also include a cache.

The disks within the storage array are configured to perform similarly to an IBM 9LZX disk. The simulation of each disk within the storage array is fairly detailed, accurately modeling seek time, rotation latency, track and cylinder skewing, and a simple segmented cache. We have configured our disk simulator through a combination of three methods [23]: issuing SCSI commands and measuring the elapsed time, by directly querying the disk, and by using the values provided by the manufacturer. Specifically, we simulate a rotation time of 6 ms, head switch time of 0.8 ms, a cylinder switch time of 1.8 ms, a track skew of 36 sectors, a cylinder skew of 84 sectors, 272 sectors per track, and 10 disk heads. The seek time curve is modeled using the two-function equation proposed by Ruemmler and Wilkes [20]; for short seek distances (less than 400 cylinders) the seek time is proportional to the square root of the cylinder distance (with endpoints at 0.8 and 6.0 ms), and for longer distances the seek time is proportional to the cylinder distance (with endpoints of 6.0 and 8.0 ms).

2.4 Algorithm

Shear has four steps; in each step, a different parameter of the storage system is identified. First, Shear determines the pattern size. Second, Shear identifies the boundaries between disks as well as the chunk size. Third, Shear extracts more detailed information about the actual layout of blocks to disks. Finally, Shear identifies the level of redundancy.

Although Shear behaves correctly with striping, mirroring, and parity, the examples in this section begin by assuming a storage system without redundancy. We show how Shear operates with redundancy with additional simulations in Section 2.5. We now describe the four algorithmic steps in more detail.

2.4.1 Pattern Size

In the first step, Shear identifies the pattern size. This *pattern size*, P, is defined as the minimum distance such that, for all B, blocks B and $B + P$ are located on the same disk.

Shear operates by testing for an assumed pattern size, varying the assumed size p from a single block up to a predefined maximum (a slight but unimplemented refinement would simply continue until the desired output results). For each p, Shear divides the storage device into a series of non-overlapping, consecutive segments of size p. Then Shear selects a random segment offset, o_r, along with N random segments, and issues parallel reads to the same offset o_r within each segment. This workload of random requests is repeated R times and the completion times are averaged. Increasing N has the effect of concurrently examining more segments on the disk; increasing R conducts more trials with different random offsets.

The intuition behind this algorithm is as follows. By definition, if p does not match the actual pattern size, P, then the requests will be sent to different disks; if p is equal to P, then all of the requests will be sent to the same disk. When requests are serviced in parallel by different disks, the response time of the storage system is expected to be less than that when all requests are serviced by the same disk.

To illustrate this behavior, we consider a four disk RAID-0 array with a block size of 4 KB and a chunk size of 4 blocks (16 KB);

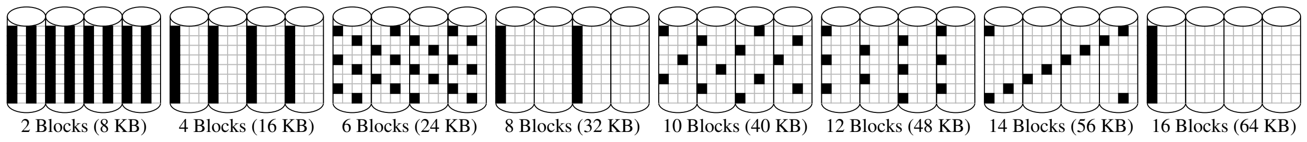


Figure 2: Pattern Size Detection: Sample Execution. ***Given 4 disks and a chunk size of 4 blocks, the shaded blocks are read as Shear increments the assumed pattern size. For compactness, the figure starts with an assumed pattern size of 2 blocks and increases each time by 2 blocks. The figure highlights all blocks at the given stride; in reality, only N random blocks are read.***

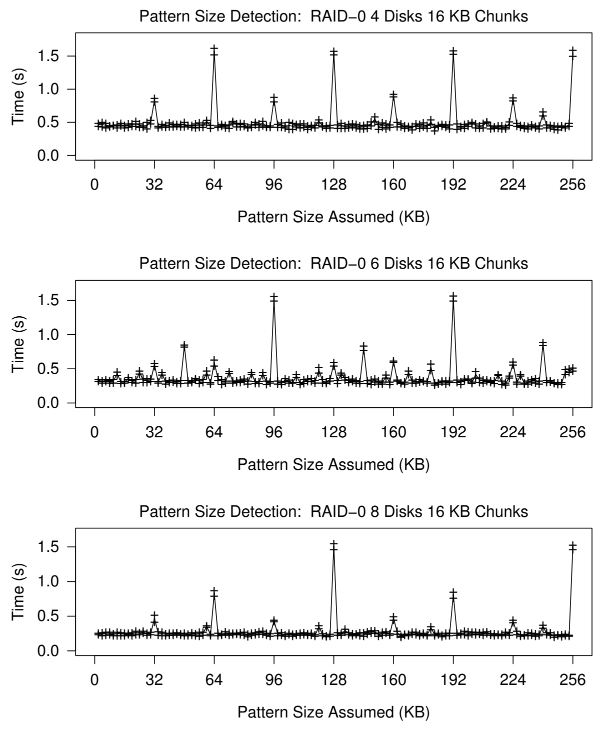


Figure 3: Pattern Size Detection: Simulations. ***The graphs show the results of running the pattern size detection algorithm on RAID-0 with 16 KB chunks and 4, 6, and 8 disks.***

thus, the actual pattern size is 16 blocks (64 KB). Figure 2 shows the location of the reads as the assumed pattern size is increased for a sample execution. The top graph of Figure 3 shows the corresponding timings when this workload is run on the simulator.

The sample execution shows that when the assumed pattern is 2, 4, or 6 blocks, the requests are sent to all disks; as a result, the timings with a stride of 8, 16, and 24 KB are at a minimum. The sample execution next shows that when the assumed pattern is 8 blocks, the requests are sent to only two disks; as a result, the timing at 32 KB is slightly higher. Finally, when the assumed pattern size is 16 blocks, all requests are sent to the same disk and a 64 KB stride results in the highest time.

To detect pattern size automatically, Shear clusters the observed completion times using a variant of the X-means cluster algorithm [18]; this clustering algorithm does not require that the number of clusters be known *a priori*. Shear then selects that cluster with the greatest mean completion time. The correct pattern size, P, is calculated as the greatest common divisor of the pattern size assumptions in this cluster.

To demonstrate that Shear is able to detect different pattern sizes, we configure the simulator with six and eight disks in the remaining two graphs of Figure 3. As desired, blocks with a stride of 96 KB (*i.e.*, 6 disks $\times$ 16 KB) and 128 KB (*i.e.*, 8 disks $\times$ 16 KB) are located on the same disk and mark the length of the pattern.

2.4.2 Boundaries and Chunk Size

In the second step, Shear identifies the data boundaries between disks and the chunk size. A data boundary occurs between blocks a and b when block a is allocated to one disk and block b to another. The chunk size is defined as the amount of data that is allocated contiguously within a single disk.

Shear operates by assuming that a data boundary occurs at an offset, c, within the pattern. Shear then varies c from 0 to the pattern size determined in the previous step. For each c, Shear selects N patterns at random and creates a read request for offset c within the pattern; Shear then selects another N random patterns and creates a read request at offset $(c-1)\ mod\ P$. All $2N$ requests for a given c are issued in parallel and the completion times are recorded. This workload is repeated for R trials and the times are averaged.

The intuition is that if c does not correspond to a disk boundary, then all of the requests are sent to the same disk and the workload completes slowly; when c does correspond to a disk boundary, then the requests are split between two disks and complete quickly (due to parallelism).

To illustrate, we consider the same four disk RAID-0 array as above. Figure 4 shows a portion of a sample execution of the chunk size detection algorithm and the top graph of Figure 5 shows the timings. The sample execution shows that when c is equal to 0 and 4, the requests are sent to different disks; for all other values of c, the requests are sent to the same disk. The timing data validates this result in that requests with an offset of 0 KB and 16 KB are faster than the others.

Shear automatically determines the chunk size C by dividing the observed completion times into two clusters using the K-Means algorithm and selecting the cluster with the smallest mean completion time. The data points in this cluster correspond to the disk boundaries; the RAID chunk size is calculated as the difference between these boundaries.

To show that Shear can detect different chunk sizes, we consider a few striping variants. We begin with RAID-0 and a constant pattern size (*i.e.*, 64 KB); we examine both 8 disks with 8 KB chunks and 16 disks with 4 KB chunks in the next two graphs in Figure 5. As desired, the accesses are slow at 8 KB and 4 KB intervals, respectively. To further stress boundary detection, we consider ZIG-ZAG striping in which alternating stripes are allocated in the reverse direction; this scheme is shown in Figure 1. The last graph shows that the first and last chunks in each stripe appear twice as large, as expected.

2.4.3 Layout

The previous two steps allow Shear to determine the pattern size and the chunk size. In the third step, Shear infers which chunks within the repeating pattern fall onto the same disk.

To determine which chunks are allocated to the same disk, Shear

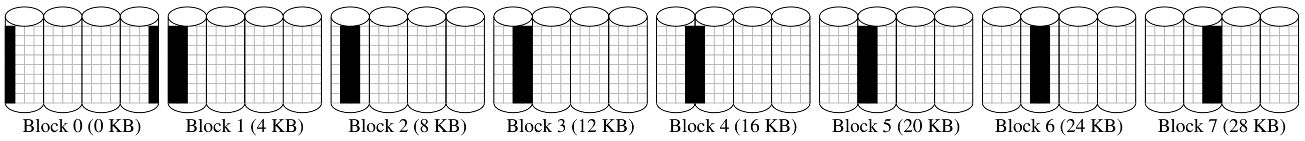


Figure 4: Chunk Size Detection: Sample Execution. ***Given 4 disks and 4 block chunks, the shaded blocks are read as Shear increments the offset within the pattern. Although requests are shown accessing every pattern, only N are selected at random.***

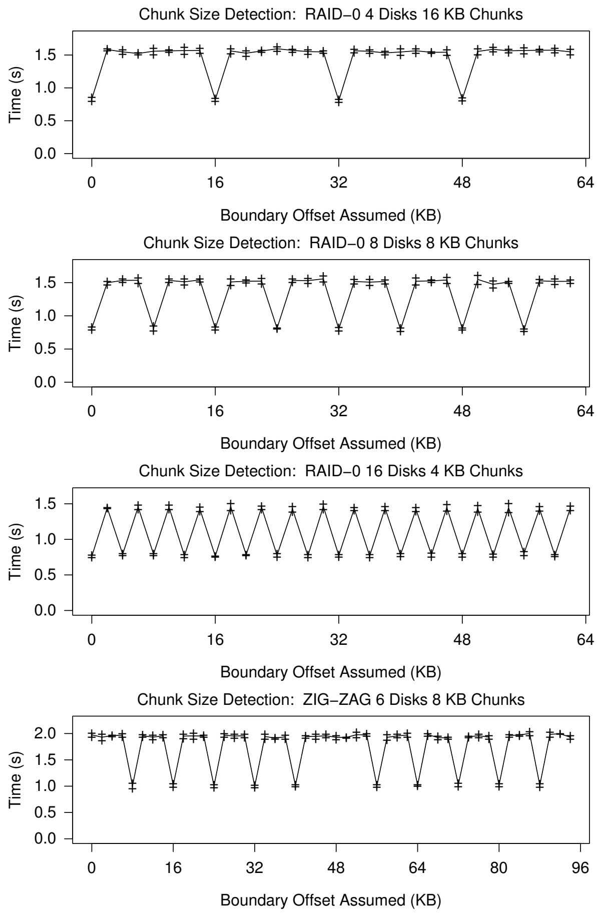


Figure 5: Chunk Size Detection: Simulations. ***The first three graphs use RAID-0 configurations: 4 disks with 16 KB chunks, 8 disks with 8 KB chunks, and 16 disks with 4 KB chunks. The last graph uses the ZIG-ZAG striping configuration in which alternating stripes are allocated in the reverse direction; this has 6 disks and 8 KB chunks.***

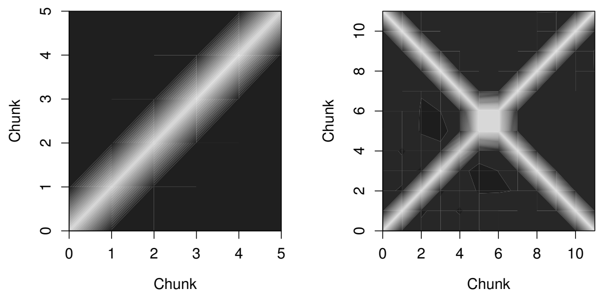


Figure 6: Read Layout Detection: Simulations. ***The first graph uses RAID-0; the second graph uses ZIG-ZAG. Both configurations use 6 disks and 8 KB chunks. The points in the graph correspond to pairs of chunks within a pattern that are accessed simultaneously. Lighter points indicate the workload finished more slowly and therefore those chunks reside on the same disk.***

examines in turn each pair of chunks, c_1 and c_2, in a pattern. First, Shear randomly selects N patterns and creates read requests for chunk c_1 within each pattern; then Shear selects another N patterns and creates read requests for c_2 within each pattern. All of the requests for a given pair are issued in parallel and the completion times are recorded. This workload is repeated over R trials and the results are averaged. Shear then examines the next pair.

Figure 6 shows that these results can be visualized in an interesting way. For these experiments, we configure our simulator to model both RAID-0 and ZIG-ZAG with 6 disks and 8 KB chunks. Each point in the graph corresponds to a (c_1, c_2) pair; light points indicate slow access times and thus fall on the same disk. The diagonal line in each graph corresponds to pairs where $c_1 = c_2$ and thus always fall on the same disk. In RAID-0, no chunks within a pattern are allocated to the same disk; thus, no pairs are shown in conflict. However, in ZIG-ZAG, the second half of each pattern conflicts with the blocks in the first half, shown as the second (upper-left to lower-right) diagonal line.

To automatically determine which chunks are on the same disk, Shear divides the completion times into two clusters using K-means and selects the cluster with the largest mean completion time. Shear infers that the chunk pairs from this cluster are on the same physical disk. By dividing the chunks into associative sets, Shear can infer the number of primary data disks in the system.

The above algorithm elicits read dependencies between pairs of chunks. Running the same algorithm with writes instead of reads allows Shear to identify write dependencies, which may occur due to rotating mirrors as in chained declustering or a shared parity block in RAID-4 or RAID-5. For example, consider the RAID-5 left-asymmetric array in Figure 1. Writing to blocks 0 and 16 at the same time will result in a short response time because the operations are spread across all four disks. Writing to blocks 0 and 52, however, will result in a longer response time because they share a parity disk. Similarly, writing to blocks 0 and 20 will take longer because the parity block for block 0 resides on the same disk as block 20.

The write layout results can reinforce conclusions from the read layout results, and they will be used to distinguish between RAID-4, RAID-5, and P+Q, as well as between RAID-1 and chained declustering. We discuss write layouts further and provide example results in Section 2.5.

2.4.4 Redundancy

In the fourth step, Shear identifies how redundancy is managed within the array. Generally, the ratio between random read bandwidth and random write bandwidth is determined by how the disk array manages redundancy.

Therefore, to detect how redundancy is managed, Shear compares the bandwidth for random reads and writes. Shear creates N block-sized random reads, issues them in parallel, and times their

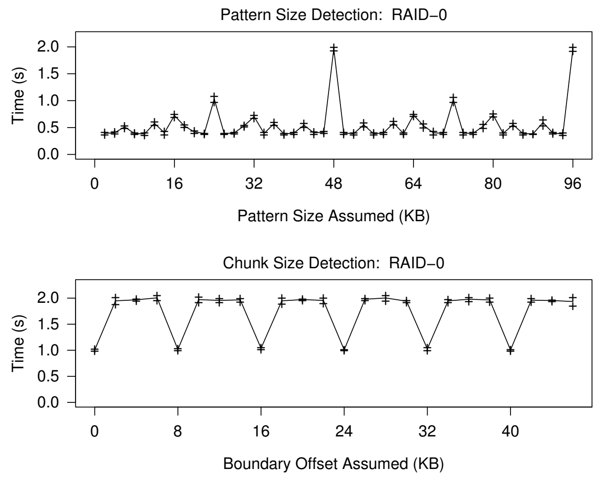


Figure 7: Pattern Size and Chunk Size Detection: RAID-0. ***We simulate RAID-0 with 6 disks and 8 KB chunks. The first graph confirms that the pattern size is 48 KB; the second graph confirms that the chunk size is 8 KB.***

completion. Shear then times N random writes issued in parallel; these writes can be performed safely if needed, by first reading that data from the storage system and then writing out the same values (with extra intervening traffic to flush any caches). The ratio between the read and write bandwidth is then compared to our expectations to determine the amount and type of redundancy.

For storage arrays with no redundancy (*e.g.*, RAID-0), the read and write bandwidths are expected to be approximately equal. For storage systems with a single mirror (*e.g.*, RAID-1), the read bandwidth is expected to be twice that of the write bandwidth, since reads can be balanced across mirrored disks but writes must propagate to two disks. More generally, the ratio of read bandwidth to write bandwidth exposes the number of mirrors. For systems with RAID-5 parity, write bandwidth is roughly one fourth of read bandwidth, since a small write requires reading the existing disk contents and the associated parity, and then writing the new values back to disk. In RAID-4 arrays, however, the bandwidth ratio varies with the number of disks because the single parity disk is a bottleneck. This makes RAID-4 more difficult to identify, and we discuss this further in Section 3.

One problem that arises in our redundancy detection algorithm is that instead of solely using reads, Shear also uses writes. Using writes in conjunction with reads is essential to Shear as it allows us to observe the difference between the case when a block is being read and the case when a block (and any parity or mirrors) is being committed to disk.

Unfortunately, depending on the specifics of the storage system under test, writes may be buffered for some time before being written to stable storage. Some systems do this at the risk of data loss (*e.g.*, a desktop drive that has immediate reporting enabled), whereas higher-end arrays may have some amount of non-volatile RAM that can be used to safely delay writes that have been acknowledged. In either case, Shear needs to avoid the effects of buffering and move to the steady-state domain of inducing disk I/O when writes are issued.

The manner in which Shear achieves this is through a simple, adaptive technique. The basic idea is that during the redundancy detection algorithm, Shear monitors write bandwidth during the write phase. If write performance is more than twice as fast as the previously observed read performance, Shear concludes that some

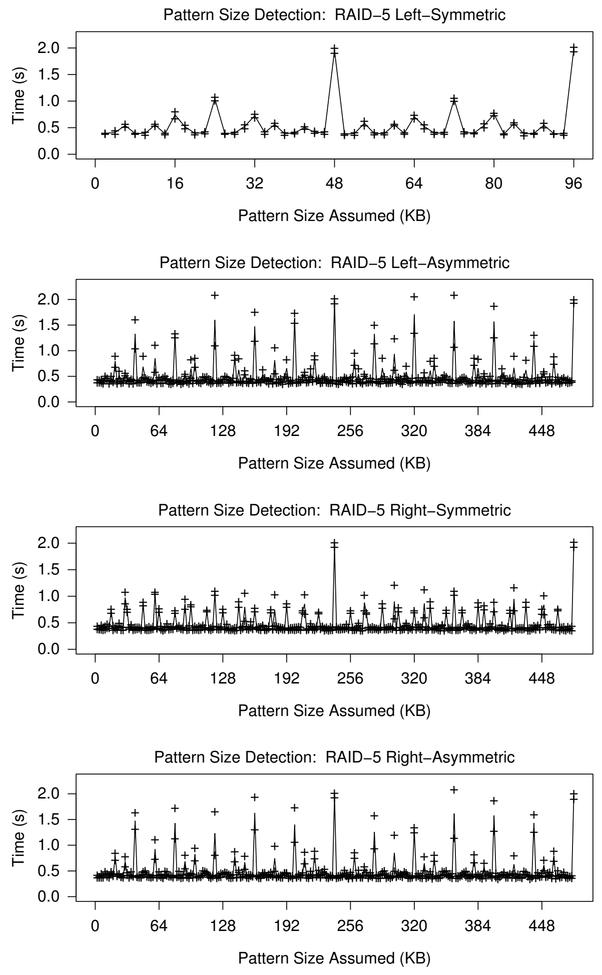


Figure 8: Pattern Size Detection: RAID-5. ***We simulate RAID-5 with left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric layouts. Each configuration uses 6 disks and a chunk size of 8 KB. The pattern size is 48 KB for RAID-5 left-symmetric and 240 KB for the rest.***

or all of the writes were buffered and not written to disk, so another round of writes is initiated. Eventually, the writes will flood the write cache and induce the storage system into the desired steady-state behavior of writing most of the data to disk; Shear detects this transition by observing that writes are no longer much faster than reads (indeed, they are often slower). We explore this issue more thoroughly via experimentation in Section 3.

2.4.5 Identifying Known Layouts

Finally, Shear uses the pattern size, chunk size, read layout, write layout, and redundancy information in an attempt to match its observations to one of its known schemes (e.g. RAID-5 left-asymmetric). If a match is found, Shear first re-evaluates the number of disks in the system. For instance, the number of disks will be doubled for RAID-1 and incremented for RAID-4. Shear completes by reporting the total number of disks in the array, the chunk size, and the layout observed.

If a match is not found, Shear reports the discovered chunk size and number of disks, but reports that the specific algorithm is unknown. By assuming that chunks are allocated sequentially to disks, Shear can produce a suspected layout based on its observations.

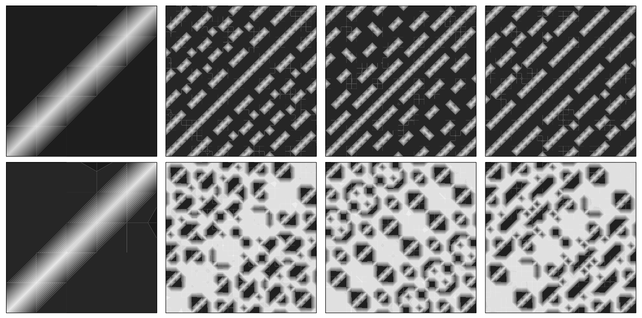


Figure 9: Read and Write Layout Detection: RAID-5. ***We simulate (from left to right) RAID-5 left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric, with 6 disks. The first row displays the read layouts and the second row shows the write layout graphs.***

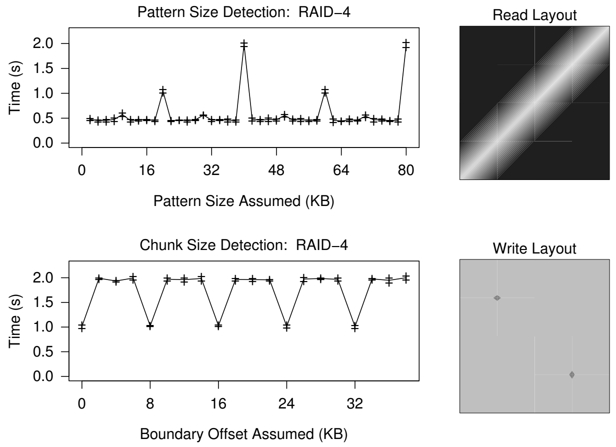


Figure 10: Pattern Size, Chunk Size, and Layout Detection: RAID-4. ***We simulate RAID-4 with 6 disks and 8 KB chunks. The first graph confirms that the pattern size of 40 KB is detected; the second graph shows the chunk size of 8 KB is detected. The read layout graph on the right resembles that for RAID-0, but the write layout graph uniquely distinguishes RAID-4 from other parity-based schemes.***

2.5 Redundancy Simulations

In this section, we describe how Shear handles storage systems with redundancy. We begin by showing results for systems with parity, specifically RAID-4, RAID-5, and P+Q. We then consider mirroring variants: RAID-1 and chained declustering. In all simulations, we consider a storage array with six disks and an 8 KB chunk size. For the purpose of comparison, we present a base case of RAID-0 in Figure 7.

2.5.1 Parity

Shear handles storage systems that use parity blocks as a form of redundancy. To demonstrate this, we consider four variants of RAID-5, RAID-4, and P+Q redundancy [4].

RAID-5: RAID-5 calculates a parity block for each stripe of data, and the location of the parity block is rotated between disks. RAID-5 can have a number of different layouts of data and parity blocks to disks, such as left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric [13]. Left-symmetric is known to deliver the best bandwidth [13], and is the only layout in which the pattern size is equal to the stripe size (*i.e.*, the same as for RAID-0); in the other RAID-5 layouts, the pattern size is $D - 1$ times the stripe size.

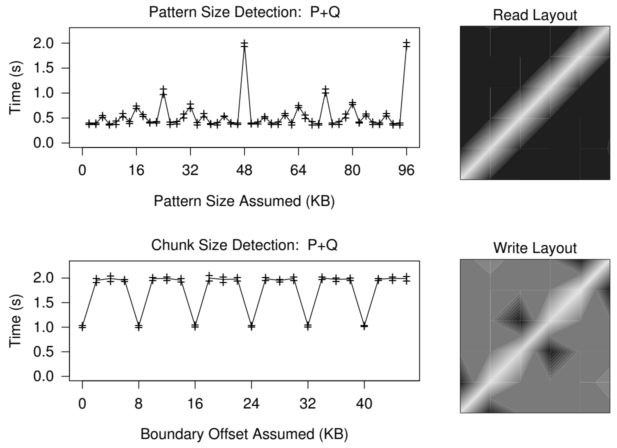


Figure 11: Pattern Size, Chunk Size, and Layout Detection: P+Q. ***We present simulated results for P+Q redundancy with 6 disks and a chunk size of 8 KB. The first graph confirms that the pattern size of 48 KB is detected; the second graph shows the chunk size of 8 KB is detected. The read layout graph on the right resembles RAID-0, but the write layout graph distinguishes P+Q from other schemes.***

The pattern size results for the four RAID-5 systems are shown in Figure 8. The first graph shows that the pattern size for left-symmetric is 48 KB, which is identical to that of RAID-0; the other three graphs show that left-asymmetric, right-symmetric, and right-asymmetric have pattern sizes of 240 KB (*i.e.*, 30 chunks), as expected. Note that despite the apparent noise in the graphs, the X-means clustering algorithm is able to correctly identify the pattern sizes. The chunk size algorithm does not behave differently for RAID-5 versus RAID-0; therefore we omit those results.

Figure 9 shows the read layout and write layout graphs for RAID-5. Note that each of the four RAID-5 variants leads to a very distinct visual image. As before, light points correspond to dependent chunk pairs that are slow; points that are dark correspond to independent chunk pairs that offer fast concurrent access. A read dependence occurs when the two chunks are located on the same disk. Write dependencies occur when the two chunks reside on the same disk, share a parity disk, or cause interference with a parity disk. These instances result in an overburdened disk and a longer response time.

Each graph depicts a pattern-sized grid that accounts for all possible pairs of chunks. For example, the RAID-5 left-asymmetric read layout graph is a 30 chunk by 30 chunk grid. The points that pair chunk 0 with chunks 5, 10, 15, 20, and 25 are all light in color because those chunks are located on the same disk. With this knowledge, Shear is able to identify if the storage system is using one of these standard RAID-5 variants and it can calculate the number of disks.

RAID-4: RAID-4 also calculates a single parity block for each stripe of data, but all of the parity blocks reside on a single disk. The pattern size, chunk size, read layout, and write layout results for RAID-4 are shown in Figure 10. The pattern size is 40 KB because the parity disk is invisible to the read-based workload. The read layout graph resembles the RAID-0 result because the pattern size is equal to the stripe size, and therefore each disk occurs only once in the pattern.

On the other hand, the write layout graph for RAID-4 is quite unique. Because the parity disk is a bottleneck for writes, all pairs of chunks are limited by a single disk and therefore exhibit similar

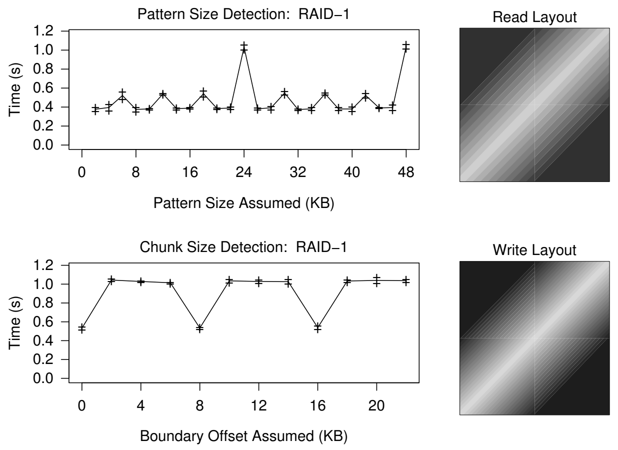


Figure 12: Pattern Size, Chunk Size, and Layout Detection: RAID-1. ***We present simulated results for RAID-1 with 6 disks and a chunk size of 8 KB. The first graph confirms that the pattern size of 24 KB is detected; the second graph shows the chunk size of 8 KB is detected. The read layout and write layout graphs on the right resemble those for RAID-0.***

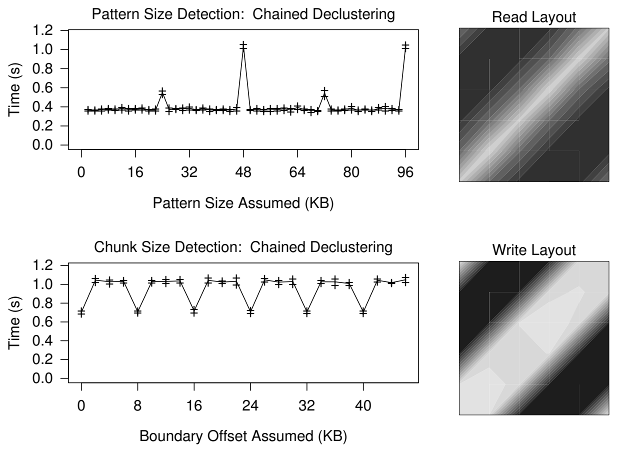


Figure 13: Pattern Size, Chunk Size, and Layout Detection: Chained Declustering. ***We present simulated results for chained declustering with 6 disks and a chunk size of 8 KB. The first graph confirms the pattern size of 48 KB; the second graph shows the chunk size of 8 KB is detected. The wider bands in the read layout and write layout graphs show that two neighboring chunks are mirrored across a total of three disks; this uniquely identifies chained declustering.***

completion times. This bottleneck produces a relatively flat RAID-4 write layout graph, allowing us to distinguish RAID-4 from other parity schemes.

P+Q: To demonstrate that Shear handles other parity schemes, we show the results of detecting pattern size and chunk size for P+Q redundancy (RAID-6). In this parity scheme, each stripe has two parity blocks calculated with Reed-Solomon codes; otherwise, the layout looks like left-symmetric RAID-5. In Figure 11, the first graph shows that the pattern size of 48 KB is detected; the second graph shows an 8 KB chunk size.

Figure 11 also shows the read layout and write layout graphs for P+Q. The read layout graph resembles that for RAID-0. The write layout graph, however, exhibits three distinct performance regions. The slowest time occurs when all requests are sent to the same chunk (and disk) in the repeating pattern. The fastest time occurs when the requests and parity updates are spread evenly across four disks, for instance when pairing chunks 0 and 1. A middle performance region occurs when parity blocks for one chunk conflict with data blocks for the other chunk. For example, when testing chunks 0 and 2, about half of the parity updates for chunk 2 will fall on the disk containing chunk 0. Again, this unique write layout allows us to distinguish P+Q from the other parity-based schemes.

2.5.2 Mirroring

Using the same algorithms, Shear can also handle storage systems that contain mirrors. However, the impact of mirrors is much greater than that of parity blocks, since read traffic can be directed to mirrors. A key assumption we make is that reads are balanced across mirrors; if reads are sent to only a primary copy, then Shear will not be able to detect the presence of mirrored copies. To demonstrate that Shear handles mirroring, we consider both simple RAID-1 and chained declustering.

RAID-1: The results of running Shear on a six disk RAID-1 system are shown in Figure 12. Note that the pattern size in RAID-1 is exactly half of that in RAID-0, given the same chunk size and number of disks. The first graph shows how the RAID-1 pattern size of 24 KB is inferred. As Shear reads from different offsets throughout the pattern, the requests are sent to both mirrors. As desired, the worst performance occurs when the request offset is equal to the real pattern size, but in this case, the requests are serviced by two disks instead of one. This is illustrated by the fact that the worst-case time for the workload on RAID-1 is exactly half of that when on RAID-0 (*i.e.*, 1.0 instead of 2.0 seconds).

The second graph in Figure 12 shows how the chunk size of 8 KB is inferred. Again, as Shear tries to find the boundary between disks, requests are sent to both mirrors; Shear now automatically detects the disk boundary because the workload time increases when requests are sent to two disks instead of four disks. Since the mapping of chunks to disks within a single pattern does not contain any conflicts, the read layout and write layout graphs in Figure 12 resemble RAID-0.

Chained Declustering: Chained declustering [11] is a redundancy scheme in which disks are not exact mirrors of one another; instead, each disk contains a primary instance of a block as well as a copy of a block from its neighbor. The results of running Shear on a six disk system with chained declustering are shown in Figure 13.

The first graph shows that a pattern size of 48 KB is detected, as desired. As with RAID-1, each read request can be serviced by two disks, and the pattern size is identified when all of the requests are sent to only two disks in the system. Note that the chained declustering pattern size is twice that of RAID-1 since each disk contains a unique set of data blocks.

The second graph in Figure 13 shows that four block chunks are again detected. However, the ratio between best and worst-case performance differs in this case from RAID-0 and RAID-1; in chained declustering the ratio is 2:3, whereas in RAID-0 and RAID-1, the ratio is 1:2. With chained declustering, when adjacent requests are located across a disk boundary, those requests are serviced by three disks (instead of four with RAID-1); when requests are located within a chunk, those requests are serviced by two disks.

The mapping conflicts with chained declustering are also interesting, as shown in the remaining graphs in Figure 13. With chained declustering, a pair of chunks can be located on two, three, or four disks; this results in three distinct performance regimes. This new case of three shared disks occurs for chunks that are cyclically adjacent (*e.g.*, chunks 0 and 1), resulting in the wider bands in the read and write layout graphs.

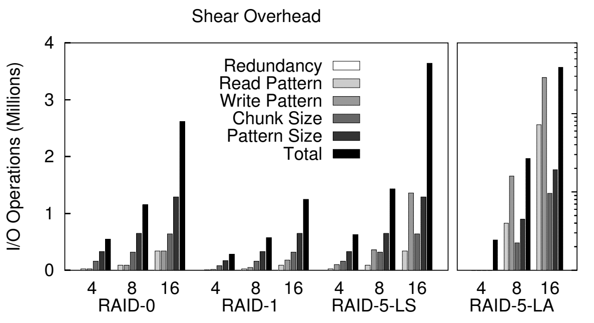


Figure 14: Shear Overhead. ***The graph shows the number of I/Os generated by each phase of Shear. Four simulated redundancy schemes are shown (RAID-0, RAID-1, RAID-5 left-symmetric, and RAID-5 left-asymmetric), each with three numbers of disks (4, 8, and 16) and 32 KB chunks.. Each bar plots the number of I/Os taken for a phase of Shear except the last (right-most) bar which shows the total. The RAID-5 left-asymmetric results are plotted with a log scale on the y-axis.***

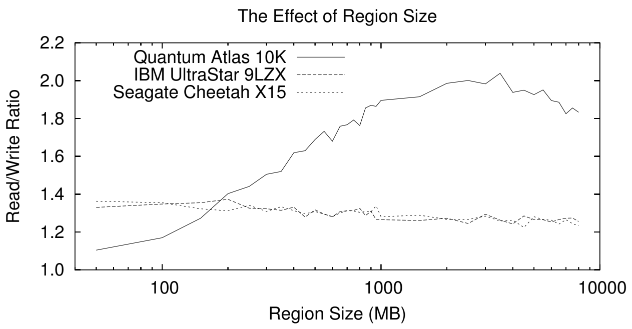


Figure 15: Sensitivity to Region Size. ***The figure plots the bandwidth ratio of a series of random read requests as compared to a series of random write requests. The x-axis varies the size of the region over which the experiment was run. In each run, 500 sector-sized read or write requests are issued. Lines are plotted for three different disks: a Quantum Atlas 10K 18WLS, an IBM 9LZX, and a Seagate Cheetah X15.***

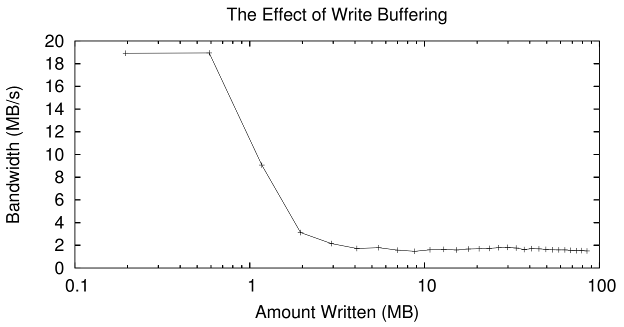


Figure 16: Avoiding the Write Buffer. ***The figure plots the performance of writes on top of the RAID-5 hardware with write-buffering enabled. The x-axis varies the number of writes issued, and the y-axis plots the achieved bandwidth.***

2.6 Overhead

We now examine the overhead of Shear, by showing how it scales as more disks are added to the system. Figure 14 plots the total number of I/Os that Shear generates during simulation of a variety of disk configurations. On the x-axis, we vary the configuration, and on the y-axis we plot the number of I/Os generated by the tool. Note that the RAID-5 left-asymmetric results are shown with a log scale on the y-axis.

From the graphs, we can make a few observations. First, we can see that the total number of I/Os issued for simple schemes such as RAID-0, RAID-1, and RAID-5 left-symmetric is low (in the few millions), and scales quite slowly as disks are added to the system. Thus, for these RAID schemes (and indeed, almost all others), Shear scales well to much larger arrays.

Second, we can see that when run upon RAID-5 with the left-asymmetric layout, Shear generates many more I/Os than with other redundancy schemes, and the total number of I/Os does not scale as well. The reason for this poor scaling behavior can be seen from the read layout and write layout detection bars, which account for most of the I/O traffic. As illustrated in Figure 1, the RAID-5 left-asymmetric pattern size grows with the square of the number of disks. Because the layout algorithms issue requests for all pairs of chunks in a pattern, large patterns lead to large numbers of requests (although many of these can be serviced in parallel); thus, RAID-5 left-asymmetric represents an extreme case for Shear. Indeed, in its current form, Shear would take roughly a few days to complete the read layout and write layout detection for RAID-5 left-asymmetric with 16 disks. However, we believe we could reduce this by a factor of ten by issuing fewer disk I/Os per pairwise trial, thus reducing run time but decreasing confidence in the layout results.

3. REAL PLATFORMS

In this section, we present results of applying Shear to two different real platforms. The first is the Linux software RAID device driver, and the second is an Adaptec 2200S hardware RAID controller. To understand the behavior of Shear on real systems, we ran it across a large variety of both software and hardware configurations, varying the number of disks, chunk size, and redundancy scheme. Most results were as expected; others revealed slightly surprising properties of the systems under test (*e.g.*, the RAID-5 mode of the hardware controller employs left-asymmetric parity placement). Due to space constraints, we concentrate here on the most challenging aspect of Shear: redundancy detection.

While experimenting with redundancy detection, we uncovered two issues that had to be addressed to produce a robust algorithm. The first of these was the size of the region over which the test was run. Figure 15 plots the read/write ratio of a single disk as the size of the region is varied.

As we can see from the figure, the size of the region over which the test is conducted can strongly influence the outcome of our tests. For example, with the Quantum disk, the desired ratio of roughly 1 is achieved only for very small region sizes, and the ratio grows to almost 2 when a few GB of the disk are used. We believe the reason for this undesirable inflation is the large settling time of the Quantum disk. Thus, we conclude that the redundancy detection algorithm should be run over as small of a portion of the disk as possible.

Unfortunately, at odds with the desire to run over a small portion of the disk is our second issue: the possible presence of a write-back cache within the RAID. The Adaptec 2200S card can be configured to perform write buffering; thus, to the host, these writes complete quickly, and are sent to the disk at some later time. Note that the presence of such a buffer can affect data integrity (*i.e.* if the buffer is non-volatile).

Because the redundancy detection algorithm needs to issue write requests to disk to compare with read request timings, Shear must

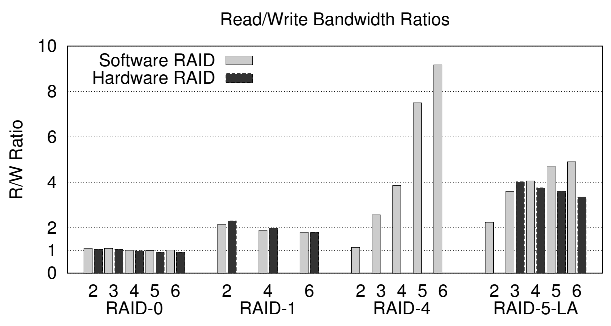


Figure 17: Redundancy Detection. ***The figure plots the ratio of read to write bandwidth over a variety of disk configurations. The x-axis varies the number of disks and the configuration: RAID-0, RAID-1, RAID-4, or RAID-5 left-asymmetric, with either software or hardware RAID.***

circumvent caching effects. Recall that Shear uses a simple adaptive scheme to detect and bypass buffering by issuing successive rounds of write requests and monitoring their performance. At some point, the write bandwidth decreases, indicating the RAID system has moved into the steady-state of writing data to disk instead of to memory, and thus a more reliable result can be generated. Figure 16 demonstrates this technique on the Adaptec hardware RAID adapter with write caching enabled.

With these enhancements in place, we study redundancy detection across both the software and hardware RAID systems. Figure 17 plots the read bandwidth to write bandwidth ratio across a number of different configurations. Recall that the read/write ratio is the key to differentiating the redundancy scheme that is used; for example, a ratio of 1 indicates that there is no redundancy, a ratio of 2 indicates a mirrored scheme, and a ratio of 4 indicates a RAID-5 style parity encoding. Note that our hardware RAID card does not support RAID-4 and will not configure RAID-5 on two disks.

The figure shows that Shear's redundancy detection does a good job of identifying which scheme is being used. As expected, we see read/write ratios of approximately 1 for RAID-0, near 2 for RAID-1, and 4 for RAID-5. There are a few other points to make. First, the bandwidth ratios for RAID-4 scale with the number of disks due to the parity disk bottleneck. This makes it more difficult to identify RAID-4 arrays. To do so, we rely on the write layout test described previously that exhibits this same bottleneck in write performance. The unique results from the write layout test allow us to distinguish RAID-4 from the other parity-based schemes.

Second, note the performance of software RAID-5 on 5 and 6 disks; instead of the expected read/write ratio of 4, we instead measure a ratio near 5. Tracing the disk activity and inspecting the source code revealed the cause: the Linux software RAID controller does not utilize the usual RAID-5 small write optimization of reading the old block and parity, and then writing the new block and parity. Instead, it will read in the entire stripe of old blocks and then write out the new block and parity. Finally, the graph shows how RAID-5 with 2 disks and a 2-disk mirrored system are not distinguishable; at two disks RAID-5 and mirroring converge.

4. SHEAR APPLICATIONS

In this section, we illustrate a few of the benefits of using Shear. We begin by showing how Shear can be used to detect RAID configuration errors and disk failures. We then show how Shear can be used to discover information about individual disks in an array. Finally, we present an example of how the storage system parameters uncovered by Shear can be used to better tune the file system; specifically, we show how the file system can improve sequential bandwidth by writing data in full stripes.

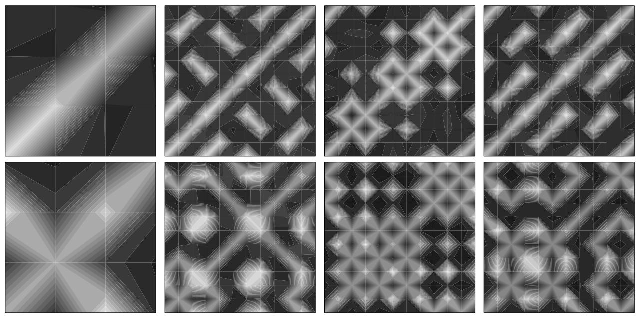


Figure 18: Detecting Misconfigured Layouts. ***For RAID-5 left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric, the upper graph shows the read layout graph when the RAID of IBM disks is correctly configured. The lower graphs show the read layout when two logical partitions are misconfigured such that they are placed on the same physical device.***

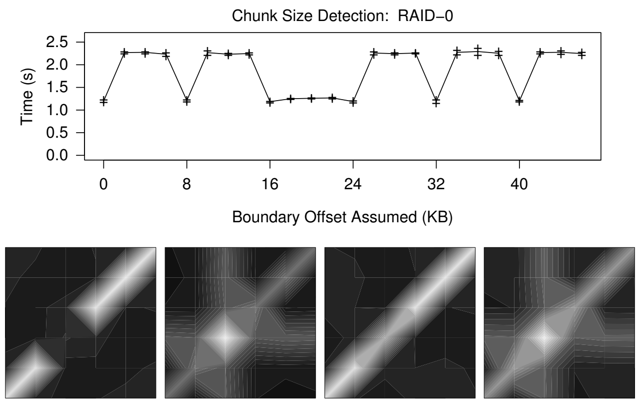
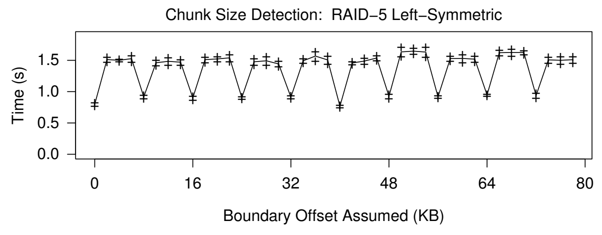


Figure 19: Detecting Heterogeneity. ***The first graph shows the output of the chunk size detection algorithm run upon an array with a single heterogeneous fast rotating disk. The second row of figures shows the results of the read layout algorithm on four different simulated disk configurations. In each configuration, a single disk has different capability than the others. A fast rotating, slow rotating, fast seeking, and slow seeking disk is depicted in each of the figures.***

4.1 Shear Management

One of our intended uses of Shear is as an administrative utility to discover configuration, performance, and safety problems. Figure 18 shows how a failure to identify a known scheme may suggest a storage misconfiguration. The upper set of graphs are the expected read layout graphs for the four common RAID-5 levels. The lower are the resulting read layout graphs when the disk array is misconfigured such that two logical partitions actually reside on the same physical disk. These graphs were generated using disk arrays comprised of four logical disks built using Linux software RAID and the IBM disks. Although the visualization makes it obvious, manual inspection is not necessary; Shear automatically determines that these results do not match any existing known schemes.

Shear can also be used to detect unexpected performance heterogeneity among disks. In this next experiment, we run Shear across a range of simulated heterogeneous disk configurations; in all ex-



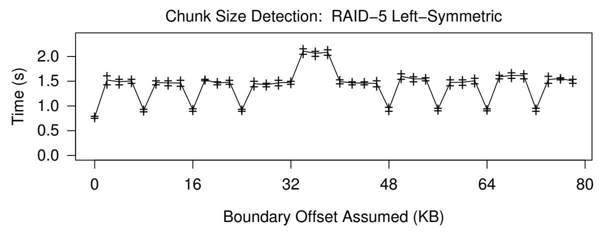


Figure 20: Detecting Failure. ***Using the chunk size detection algorithm, Shear can discover failed devices within a RAID system. The upper graph shows the initial chunk size detection results collected after building a 10 disk software RAID system using the IBM disks. The lower graph is for the same system after the fifth disk was removed.***

periments, one disk is either slower or faster than the rest. Figure 19 shows results when run upon these heterogeneous configurations.

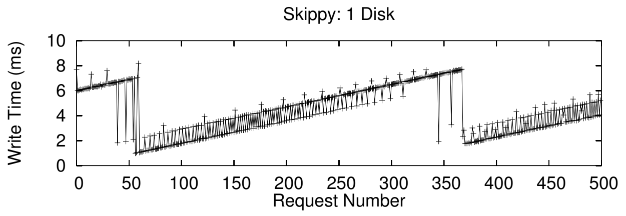
As one can see from the figure, a faster or slower disk makes its presence known in obvious ways in both the read layout graphs as well as in the chunk size detection output (the pattern size detection is relatively unaffected). Thus, an administrator could view these outputs and clearly observe that there is a serious and perhaps unexpected performance differential among the disks and take action to correct the problem.

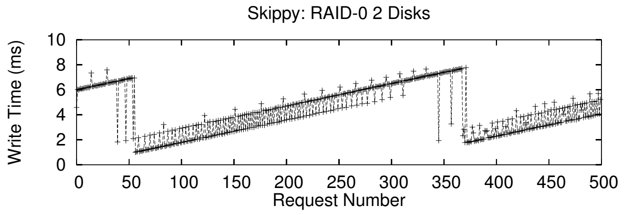
Finally, the chunk size detection algorithm in Shear can be used to identify safety hazards by determining when a redundant array is operating in degraded mode. Figure 20 shows the chunk size detection results for a ten disk software RAID system using the IBM disks. The upper graph shows the chunk size detection correctly working after the array was first built. The lower graph shows how chunk size detection is changed after we physically remove the fifth disk from the array. Recall that chunk size detection works by guessing possible boundaries and timing sets of requests on both sides of the boundary. Vertical downward spikes should be half the height of the plateaus and indicate that the guessed boundary is correct because the requests are serviced in parallel on two disks. The plateaus are false boundaries in which all the requests on both sides of the guessed boundary actually are incurred on just one disk. The lower graph identifies that the array is operating in degraded mode because the boundary points for the missing disk disappear, and its plateau is higher due to the extra overhead of performing on-the-fly reconstruction.

4.2 Shear Disk Characterization

Related projects have concentrated on extracting specific properties of individual disk drives [23, 27, 30]. Several techniques have been built on top of this characteristic knowledge, such as aligning files to track boundaries [24] and free-block scheduling [14]. Shear enables such optimizations in the context of storage arrays. Shear can expose the boundaries between disks, and then existing tools can be used to determine specific properties of those individual disks.

We demonstrate this ability using the Skippy disk characterization tool [27]. Skippy uses a sequence of write operations at in-





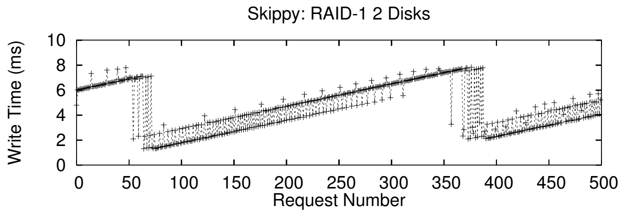


Figure 21: Skippy. ***The figures plot the results of running the Skippy disk characterization tool on a single Quantum disk, a two disk RAID-0 array, and a two disk RAID-1 array.***

creasing strides to determine the disk sector to track ratio, rotation time, head positioning time, head switch time, cylinder switch time, and the number of recording surfaces. The first graph in Figure 21 shows the pattern generated by Skippy on a single Quantum disk.

The second graph in Figure 21 shows the results of running a modified version of Skippy on a RAID-0 array with two disks. This version of Skippy uses the array information provided by Shear to map its block reference stream to the corresponding logical blocks residing on the first disk in the array. This results in a pattern that is nearly identical to that running on a single disk, allowing us to extract the individual disk parameters. The final graph in Figure 21 shows the results of the same technique applied to a two disk RAID-1 array. Again, the results are nearly identical to the single disk pattern except for some small perturbations that do not affect our analysis.

There are some limitations to this approach, however. For example, in the case of RAID-1, the Skippy write workload performs as expected, but a read workload produces spurious results due to the fact that reads are balanced across disks. Conversely, reads work well under RAID-5 whereas writes do not due to the need to update parity information. Additionally, because the parity blocks under RAID-5 cannot be directly accessed, characterization tools may obtain an incomplete set of data. Despite these limitations, we have tested a read-based version of Skippy on RAID-5 and successfully extracted all parameters from the individual disks.

4.3 Shear Performance

The stripe size within a disk array can have a large impact on performance [3, 5]. This effect is especially important for RAID-5 storage, since writes of less than a complete stripe require additional I/O. Previous work has focused on selecting the optimal stripe size for a given workload. We instead show how the file system can adapt the size and alignment of its writes as a function of a given stripe size.

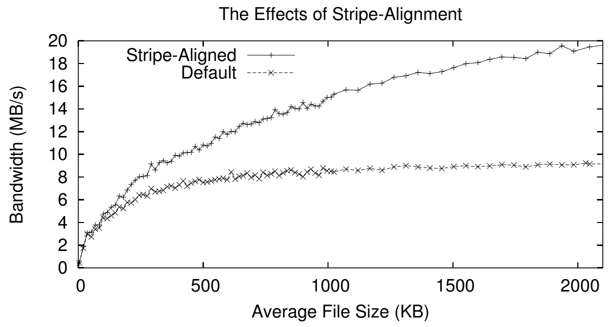


Figure 22: Benefits of Stripe Alignment. ***The figure plots the bandwidth of a series of file creations of an average size, as varied along the x-axis. Two variants are shown: one in which the file system generates stripe-sized writes and the default Linux. The workload consists of creating 100 files. The x-axis indicates the mean size of the files, which are uniformly distributed between 0.5 $\times$ mean and 1.5 $\times$ mean.***

The basic idea is that the file system should adjust its writes to be stripe aligned as much as possible. This optimization can occur in multiple places; we have modified the Linux 2.4 device scheduler so that it properly coalesces and/or divides individual requests such that they are sent to the RAID in stripe-sized units. This modification is straight-forward: only about 20 lines of code were added to the kernel.

This simple change to make the file system stripe-aware leads to tremendous performance improvements. The experiments shown in Figure 22 are run on a hardware RAID-5 configuration with 4 Quantum disks and a 16 KB chunk size. These results show that a stripe-aware file system noticeably improves bandwidth for moderately-sized files and improves bandwidth for larger files by over a factor of two.

5. RELATED WORK

The idea of providing software to automatically uncover the behavior of underlying software and hardware layers has been explored in a number of different domains. Some of the earliest work in this area targeted the memory subsystem; for example, by measuring the time for reads of different amounts and with different strides, Saavedra and Smith reveal many interesting aspects of the memory hierarchy, including details about both caches and TLBs [21]. Similar techniques have been applied to identify aspects of a TCP protocol stack [9, 16], to determine processor cycle time [26], and CPU scheduling policies [19].

The work most related to ours is that which has targeted characterizing a single disk within the storage system. For example, in [30], Worthington *et al.* identify various characteristics of disks, such as the mapping of logical block numbers to physical locations, the costs of low-level operations, the size of the prefetch window, the prefetching algorithm, and the caching policy. Later, Schindler *et al.* and Talagala *et al.* build similar but more portable tools to achieve similar ends [23, 27]. We have shown how Shear can be used in conjunction with such low-level tools to discover properties of single disks inside arrays.

Evaluations of storage systems have usually focused on measuring performance for a given workload and not on uncovering underlying properties [1, 12, 15]. One interesting synthetic benchmark adapts its behavior to the underlying storage system [6]; this benchmark examines sensitivity to parameters such as the size of requests, the read to write ratio, and the amount of concurrency.

Finally, the idea of using detailed storage-systems knowledge within a file system or storage client has been investigated. For example, Schindler *et al.* investigate the concept of track-aligned file placement [24] in single disk systems; in this work, a modified file system allocates medium-sized files within track boundaries to avoid head switches and thus deliver low-latency access to files. Other systems, like I·LFS [7] and Atropos [25], augment the array interface to provide information about individual disks. I·LFS uses knowledge of disk boundaries to dynamically allocate writes based on performance and to control redundancy on a per-file basis. The Atropos volume manager extends the storage interface to expose disk boundary and track information and provide efficient semi-sequential access to two-dimensional data structures. Shear enables the use of such information in multiple disk systems without the need for an enhanced interface.

6. CONCLUSIONS

We have presented Shear, a tool that automatically detects important characteristics of modern storage arrays, including the number of disks, chunk size, level of redundancy, and layout scheme. The keys to Shear are its use of randomness to extract steady-state performance and its use of statistical techniques to deliver automated and reliable detection. We have verified that Shear works as desired through a series of simulations over a variety of layout and redundancy schemes. We have subsequently applied Shear to both software and hardware RAID systems, revealing properties of both. Specifically, we found that Linux software RAID exhibits poor performance for RAID-5 parity updates, and the Adaptec 2200S RAID adapter implements RAID-5 left-asymmetric layout.

We have also shown how Shear could be used through three case studies. Storage administrators can use Shear to verify properties of their storage arrays, monitor their performance, and detect disk failures. Shear can help extract individual parameters from disks within an array, enabling performance enhancements previously limited to single disk systems. Finally, we have shown a factor of two improvement in performance from a file system tuning its writes to the stripe size of its RAID storage.

Storage systems, and computer systems in general, are becoming more complex, yet their layers of interacting components remain concealed by a veil of simplicity. We hope the techniques developed within Shear can help reveal the true power of future systems and subsequently make them more manageable, composable, and efficient.

7. ACKNOWLEDGEMENTS

We would like to thank Bradford Beckmann, Nathan Burnett, Vijayan Prabhakaran, Muthian Sivathanu, and the anonymous reviewers for their excellent feedback. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM, EMC, and the Wisconsin Alumni Research Foundation.

8. REFERENCES

[1] T. Bray. The Bonnie File System Benchmark. http://www.textuality.com/bonnie/.

[2] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, California, June 2002.

[3] P. Chen and E. K. Lee. Striping in a RAID Level 5 Disk Array. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 136–145, Ottawa, Canada, May 1995.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[5] P. M. Chen and D. A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, pages 322–331, Seattle, Washington, May 1992.

[6] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 1–12, Santa Clara, California, May 1993.

[7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, California, June 2002.

[8] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

[9] T. Glaser. TCP/IP Stack Fingerprinting Principles. http://www.sans.org/newlook/resources/IDFAQ/ TCP_fingerprinting.htm, October 2000.

[10] E. Grochowski. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. *Datatech*, September 1999.

[11] H.-I. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proceedings of 6th International Conference on Data Engineering (ICDE '90)*, pages 456–465, Los Angeles, California, February 1990.

[12] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[13] E. K. Lee and R. H. Katz. Performance Consequences of Parity Placement in Disk Arrays. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 190–199, Santa Clara, California, April 1991.

[14] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 87–102, San Diego, California, October 2000.

[15] W. Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[16] J. Padhye and S. Floyd. Identifying the TCP Behavior of Web Servers. In *Proceedings of SIGCOMM '01*, San Diego, California, August 2001.

[17] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[18] D. Pelleg and A. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the 17th International Conference on Machine Learning*, June 2000.

[19] J. Regehr. Inferring Scheduling Behavior with Hourglass. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[20] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[21] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

[22] S. Savage and J. Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, pages 27–39, San Diego, California, January 1996.

[23] J. Schindler and G. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.

[24] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[25] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. In *Proceedings of the 3nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2004.

[26] C. Staelin and L. McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 155–166, New Orleans, Louisiana, June 1998.

[27] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[28] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):559–574, June 2004.

[29] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[30] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 146–156, Ottawa, Canada, May 1995.