# Deconstructing Storage Arrays

Timothy Denehy
John Bent
Florentina Popovici
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report #1506

May 2004

# 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*
*{tedenehy, johnbent, popovici, dusseau, remzi}@cs.wisc.edu*

## ABSTRACT

*We present 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. We illustrate the correctness of Shear by running it upon numerous simulated configurations, and then verify its real-world applicability by applying Shear to both software-based and hardware-based RAID systems. Finally, we demonstrate the utility of Shear through two case studies. First, we show how Shear can be used in a storage management environment to verify RAID construction and detect failures. Second, we show how an operating system can use Shear to automatically tune its storage subsystems to specific RAID configurations.*

## 1. INTRODUCTION

Modern storage systems are complex. For example, a high-end storage array can contain tens of processors and hundreds of disks [6] and a given array can be configured many different ways. However, regardless of their internal complexity, RAID arrays expose a simple interface to the file system: 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 *transparency*: file systems can operate without modification on top of any storage device. But this transparency has a cost: users and applications are not easily able to obtain more information about the storage system. For example, most storage systems do not reveal how data blocks have been mapped to each of the underlying disks and it is well known that RAID configuration has a large impact on performance and reliability [3, 14, 19, 23]. Furthermore, despite the fact that configuring a modern RAID 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 the important properties of a RAID. Using this tool to discover the properties of 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 RAIDs, 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 [20, 24]. 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* where possible; 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 simulated 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 uncovered a poor implementation 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 encoding scheme [11].

Finally, we demonstrate the utility of the Shear tool through two 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 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 system by over a factor of two.

The rest of this paper is organized as follows. In Section 2 we describe Shear, illustrating its output on simpler simulated configurations, and present more detailed results across many different redundancy schemes in Section 3. Then, in Section 4, we show the results of running Shear on software and hardware RAID systems, and in Section 5, we show how Shear can be used to improve storage administration as well as file system performance. Finally, we discuss related work in Section 6 and conclude in Section 7.

## 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 Shear's algorithms.

### 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 map-

| Disk 0 | | | | Disk 1 | | | | Disk 2 | | | | Disk 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *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* |

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 size = Pattern size = 16

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *00* | *01* | *02* | *03* | *04* | *05* | *06* | *07* | *08* | *09* | *10* | *11* | *PP* | *PP* | *PP* | *PP* |
| 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* | *PP* | *PP* | *PP* | *PP* |
| *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* |

Parity: RAID-5 Left-Asymmetric. Stripe = 16; Pattern size = 48

Figure 1: **Examples and Terminology.** *The picture depicts a variety of 4-disk systems; the chunk size is set to 4 blocks. A full pattern is indicated by italics for each redundancy scheme.*

ping 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, and 5, as well as variants such as chained declustering [9].

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 the unit of data that is allocated contiguously within a disk; we assume that the chunk size is constant. A *stripe* is a set of chunks across each of $D$ disks; a stripe may include mirrored copies and/or parity blocks.

Shear assumes that the mapping of logical blocks to individual disks follows some repeatable, but unknown, pattern. The *pattern* is the minimum sequence of blocks such that block offset $i$ within the pattern is always located on disk $j$; likewise, the mirror and parity blocks within the pattern, $i_m$ 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 -5 left-symmetric), whereas in others the pattern size is larger (*e.g.*, RAID-5 left-asymmetric).

Figure 1 illustrates a number of the layout configurations that we analyze in this paper. Each configuration is performed on four disks with 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 [20, 24]. Due to its use of random accesses and steady-state behavior, Shear does operate 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 are robust to small perturbations.

## 2.2 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-5 with left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric layouts [11], P+Q redundancy [3], and chained declustering [9]. 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, a fixed rotation latency, track and cylinder skewing, and a simple segmented cache. We have configured our disk simulator through a combination of three methods [20]: 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 [17]; 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).

## 2.3 Algorithm

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 correct 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 band-

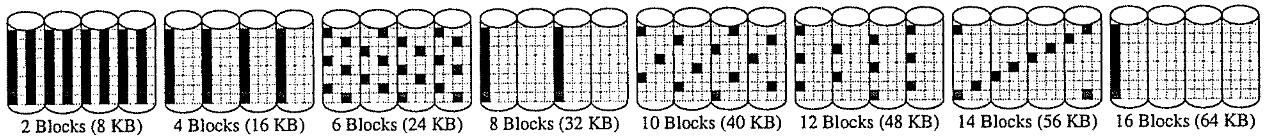


Figure 2: **Pattern Size Detection: Sample Execution.** *Given four disks and a chunk size of 4 blocks, the shaded blocks are read as Shear increments the assumed pattern size. For compactness, the figure starts with an assumed pattern size of 2 blocks and increases each time by 2 blocks. The figure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

width improves by nearly 40% per year, whereas seek time and rotational latency improve by less than 10% per year [8]; 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 [15]; this clustering helps Shear 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 safely, 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 allows Shear to inspect RAIDs that appear to have disk failures or other performance anomalies over time.

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 and the chunk size. Third, Shear extracts more detailed information about the actual layout of blocks to disks. Finally, Shear identifies the standard RAID levels.

Although Shear behaves correctly with striping, mirroring, and parity, the examples in this section begin by assuming a storage system without redundancy (*i.e.*, mirroring or parity). We show how Shear operates with redundancy with additional simulations in Section 4. We now describe the four algorithmic steps in more detail.

### *2.3.1 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 blocks $B$, $B$ 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 randomly selects $N$ segments and issues a read to the same offset, $o_r$, within each segment in parallel. This workload of random requests

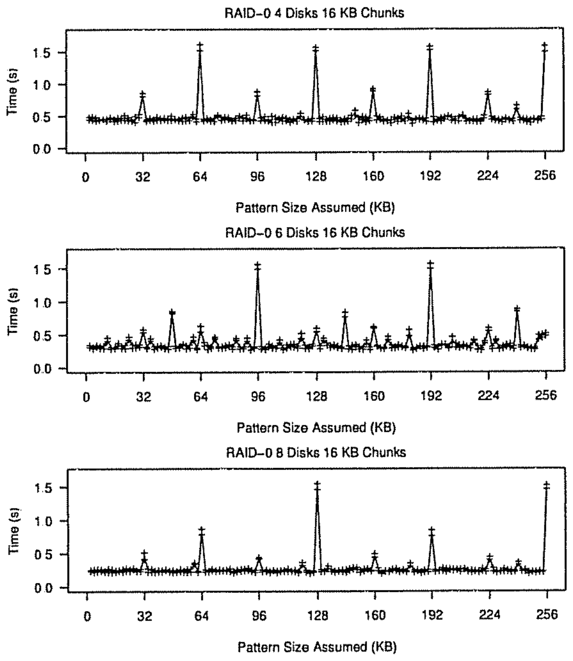


Figure 3: **Pattern Size Detection: Simulations.** *Pattern size detection on RAID-0 with 16 KB chunks and 4, 6, and 8 disks.*

is repeated $R$ times and the completion times are averaged. Increasing $N$ has the effect of concurrently examining more segments on the disk; increasing $R$ repeats this workload more times 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 sent to different disks, the bandwidth delivered by the storage system is expected to be greater 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); 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.

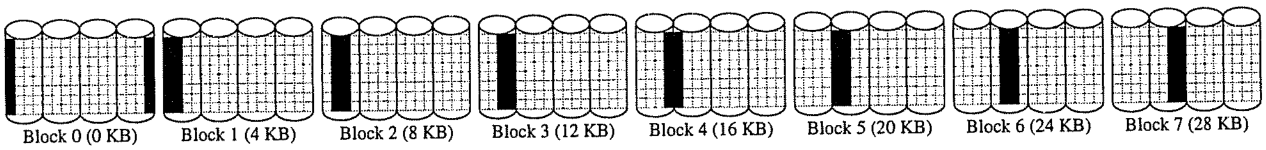


Figure 4: **Boundary Detection: Sample Execution** *Given four disks and 4-block chunks, the shaded blocks are read as Shear increments the offset within the pattern. Although requests are shown accessing every pattern, only N are selected at random.*

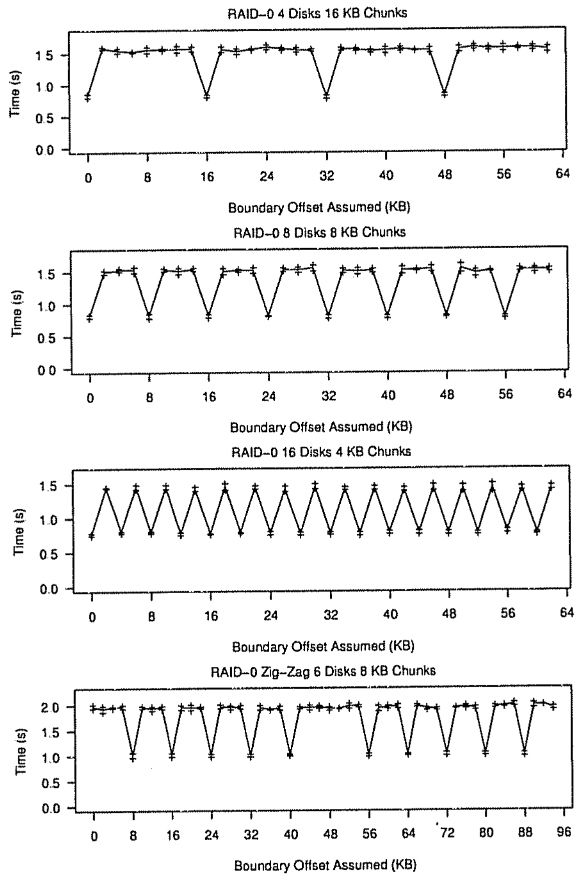


Figure 5: **Boundary Detection: Simulations.** *The first three graphs use RAID-0 configurations: 4 disks with 16 KB chunks, 8 disks with 8 KB chunks, and 16 disks with 4 KB chunks. The last graph uses the ZIGZAG striping configuration in which alternating stripes are allocated in the reverse direction;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

To detect pattern size automatically, Shear clusters the observed completion times using a variant of the x-mean cluster algorithm [15]; 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 * 16 KB) and 128 KB (*i.e.*, 8 disks * 16 KB) are located on the same disk and mark the length of the layout pattern.

### 2.3.2 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 $(c-1)\%P$. All of the requests for a given $c$ are issued in parallel and the completion times are recorded. This workload is repeated for $R$ trials and the completion times are averaged.

The intuition is that if $c$ does not correspond to a disk boundary, then all of 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 boundary 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 $C$ 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 ZIGZAG striping in which alternating stripes are allocated in the reverse direction; this layout is shown in Figure 1. The last graph shows that the first and last chunks in each stripe are twice as large, as expected.

### 2.3.3 Pattern 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 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 chunk

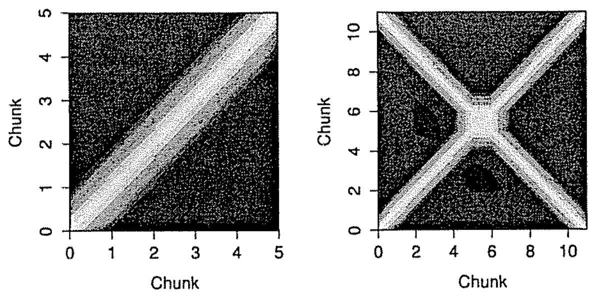


Figure 6: **Pattern Layout Detection: Simulations** *The first graph uses a RAID-0 layout; the second graph uses ZIGZAG. The points in the graph correspond to pairs of offsets within a pattern that are accessed simultaneously. Lighter points indicate the workload finished more slowly and therefore those offsets reside on the same disk.*

$c_2$ 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 ZIGZAG. 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 ZIGZAG, 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. This step also allows Shear to infer the number of disks in the system (assuming they are not simple mirrors of each other).

### *2.3.4 Redundancy*

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

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 "significantly" faster than the previously observed read performance, Shear concludes that some or all of the writes were buffered and not written to disk. The number of writes is then doubled and the write test reinitiated. 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 the writes are no longer "significantly" faster than reads (indeed, they are often slower).

The major weakness of our current approach is that setting this threshold value is difficult. We explore this issue more thoroughly via experimentation in Section 4.

## 2.4 Shear Overhead

We now examine the overhead of Shear, by showing how it scales as more disks are added to the system. Figure 7 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. Two plots of the same data are shown, one with a log scale on the y-axis and one without.

From the graphs, we can make a few observations. First, we can see that the total number of I/Os issued for simple patterns such as RAID 0 and RAID 1 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 (LA) 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 pattern layout detection bar, which accounts for most of the I/O traffic. As discussed before, RAID-5 LA has quite a large pattern size; because the pattern layout algorithm issues 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 LA represents a worst case behavior for Shear. In-

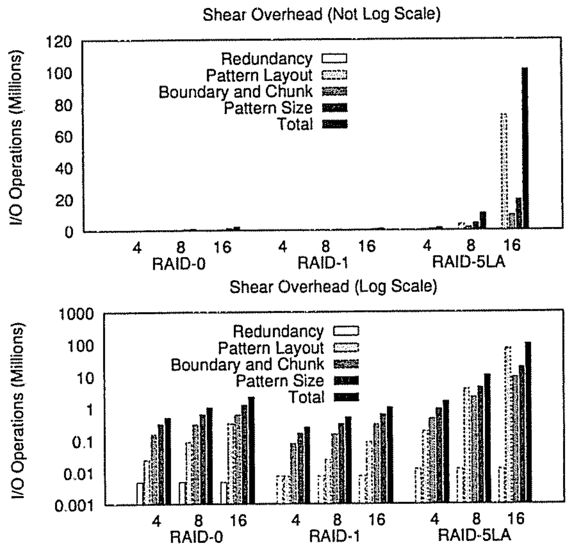


Figure 7: **Shear Overhead.** *The plots show the number of I/Os generated by each phase of Shear. Three simulated redundancy schemes are shown (RAID-0, RAID-1, and RAID-5 left-asymmetric), and for each scheme, three different number of disks (4, 8, and 16). Each bar save the last in each group plots the number of I/Os taken for a phase of Shear; the last (rightmost) bar shows the total. In all experiments, the chunk size is set to 32 KB.*

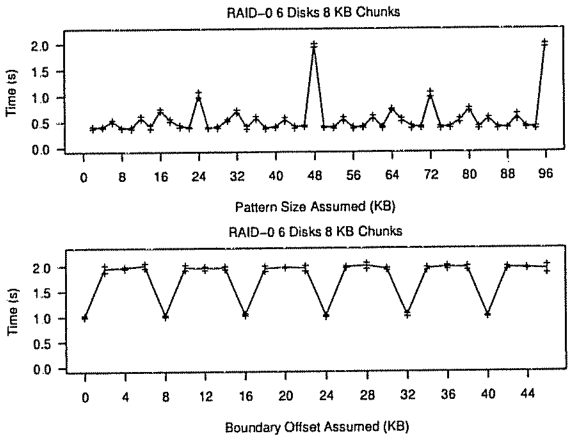


Figure 8: **Pattern Size and Boundary Detection: RAID-0.** *We simulate RAID-0 with 6 disks and 8 KB chunks. The first graph confirms that the pattern size is 48 KB; the second graph confirms that the chunk size is 8 KB.*

deed, in its current form, Shear would take roughly a few days to complete the pattern layout detection for the 16 disk RAID-5 LA. 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 pattern layout result.

## 3. 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-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 8.

### 3.1 Parity

Shear handles storage systems that use parity blocks as a form of redundancy. To demonstrate this, we consider four variants of RAID-5 as well as P+Q redundancy [3].

**RAID-5:** RAID-5 calculates a single parity block for each stripe of data; across stripes, 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 and asymmetric, and right symmetric and asymmetric [11]. Left-symmetric is known to deliver the best bandwidth [11], 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 equal to $D-1$ times the stripe size.

The pattern size results for the four RAID-5 systems are shown in Figure 9. The first graph shows that the left-symmetric pattern size 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 longer 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 10 shows the layout of data chunks within a pattern to disks for RAID-5. Note that each of the four RAID-5 variants leads to a very distinct visual pattern. As before, points that are light correspond to block pairs that are slow, and are assumed to be located on the same disk; points that are dark are located on different disks. For example, with left-asymmetric, chunks 0, 5, 10, 15, 20, and 25 all fall 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 can calculate the number of disks.

**P+Q:** To demonstrate that Shear can handle other parity schemes, we show the results of detecting pattern size and chunk size for P+Q redundancy (RAID Level 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 that the chunk size of 8 KB. We omit the graph showing the layout of blocks within a pattern since it is identical to RAID-0.

### 3.2 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

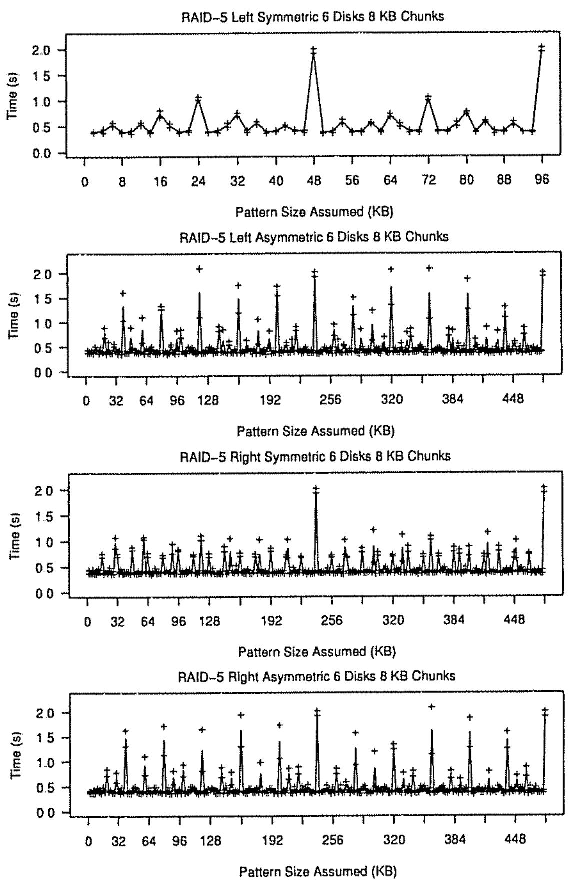


Figure 9: **Pattern Size Detection: RAID-5** *We simulate RAID-5 with Left-Symmetric, Left-Asymmetric, Right-Symmetric, and Right-Asymmetric layouts, respectively. Each configuration uses 6 disks and a chunk size of 8 KB. The pattern size is 48 KB for Left-Symmetric and otherwise 240 KB.*

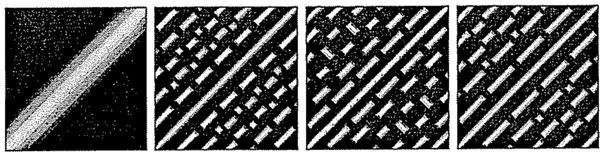


Figure 10: **Pattern Layout Detection: RAID-5.** *We simulate Left-Symmetric, Left-Asymmetric, Right-Symmetric, and Right-Asymmetric, respectively, with 6 disks.*

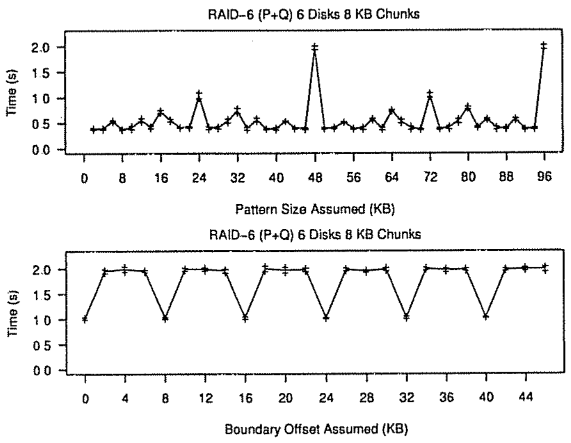


Figure 11: **Pattern Size and Boundary Detection: P+Q** *Simulated results for P+Q redundancy (RAID-6) with 6 disks and a chunk size of 8 KB. The first graph confirms that the pattern size of 48 KB is detected; the second graph shows the chunk size of 8 KB is detected.*

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, we omit the pattern layout graph (*i.e.*, it is identical to that for RAID-0).

**Chained Declustering:** Chained declustering 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 each 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 to two disks, and the pattern size is identified when all of the requests are sent to only two disks in the system. Note that chained declustering pattern size is twice that of RAID-1 since each disk does contain 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 third graph in Figure 13. With chained declustering, a pair of chunks can be located on two, three, or four disks; as a result, there are three distinct performance regimes. As shown in the figure, this new case of three shared disks occurs for chunks that are cyclically adjacent (*e.g.*, chunks 0 and 1).

## 4. RESULTS

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;

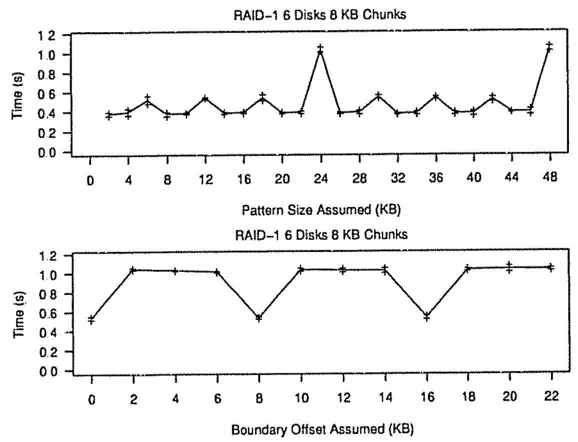


Figure 12: **Pattern Size and Boundary Detection: RAID-1** *Simulated results for RAID-1 with 6 disks and a chunk size of 8 KB. The first graph confirms that the pattern size of 24 KB is detected; the second graph shows the chunk size of 8 KB is detected.*

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.

We now study redundancy detection across both the software and hardware RAID systems. Figure 14 plots the read/write 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 parity encoding.

As we can see from the figure, Shear's redundancy detection does a good job of identifying which scheme is being used. There are two other points to make. First, note the performance of software RAID-5 on 5 disks; instead of the expected read/write ratio of 4, we instead measure a ratio of 5. Further inspection of the source code revealed the cause: the Linux software RAID controller does not implement 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 set of old blocks and then write out the new block and parity. Second, 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.

However, we found two aspects of the redundancy detection that had to be fine-tuned to result in a robust detection 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 concluded 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

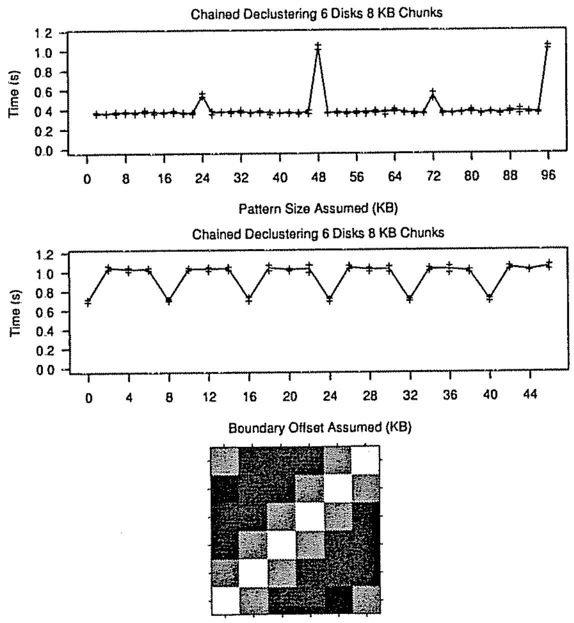


Figure 13: **Pattern Size, Boundary, and Pattern Layout Detection: Chained Declustering** *Simulated results for chained declustering with 6 disks and a chunk size of 8 KB. The first graph confirms that the pattern size of 48 KB is detected; the second graph shows the chunk size of 8 KB is detected. The third graph shows that two neighboring chunks are mirrored across a total of three disks; this uniquely identifies the pattern layout of chained declustering.*

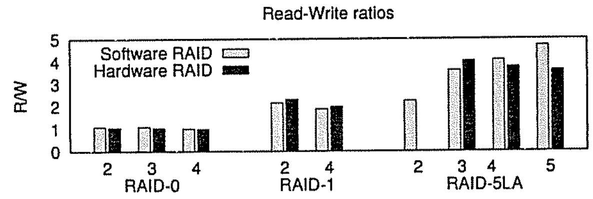


Figure 14: **Redundancy Detection.** *The figure plots the ratio of reads to writes over a variety of disk configurations. The x-axis varies the configuration: RAID-0, RAID-1, or RAID-5 (LA), with either software or hardware RAID.*

portion of the disk is the possible presence of a write-back cache within the RAID. The Adaptec card that we have 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 (depending on if the buffer is non-volatile).

Because the redundancy detection algorithm needs to be able to issue write requests to disk to compare with read request timings, Shear needs to circumvent caching effects. Shear does so with a simple adaptive scheme that issues a fixed number $W$ write requests, times them, and then compares them to the performance of read requests. If the write requests are significantly faster, then Shear increases $W$ by a fixed amount, and repeats the process. At some point, the write bandwidth drops, indicating that we have successfully moved the RAID system into the steady-state of writing data to disk instead of to memory, and thus a more reliable result can be generated.

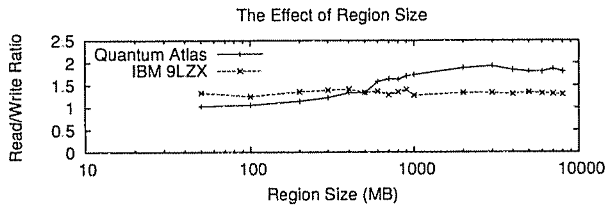


Figure 15: **Sensitivity to Region Size.** *The figure plots the ratio of a series of random read requests as compared to a series of random write requests. The x-axis varies the size of the region over which the experiment was run. In each run, 500 sector-sized read or write requests are issued. Lines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

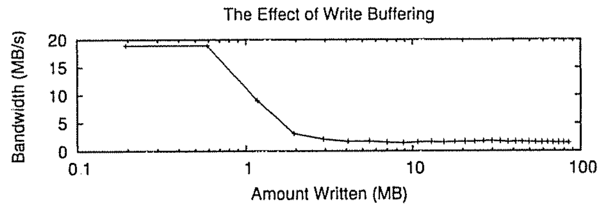


Figure 16: **Avoiding the Write Buffer.** *The figure plots the performance of writes on top of the RAID-5 hardware with write-buffering enabled. The x-axis varies the number of writes that are issued in the test, and the y-axis plots the achieved bandwidth.*

## 5. CASE STUDIES

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 give 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.

### 5.1 Shear Management

One of our intended uses of Shear is as an administrative utility to discover configuration, performance, and safety problems. Figure 17 shows how a failure to identify a known pattern may suggest a storage misconfiguration. The upper set of graphs are the expected patterns for RAID-0 and the four common RAID-5 levels. The lower are the resulting patterns 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 detection is not necessary; Shear automatically determines that these patterns do not match any existing known patterns.

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 experiments, one disk is either slower or faster than the rest. Figure 18 shows results when run upon a variety of heterogeneous disk configurations.

As one can see from the figure, a faster or slower disk makes it presence known in obvious ways in both the pattern layout

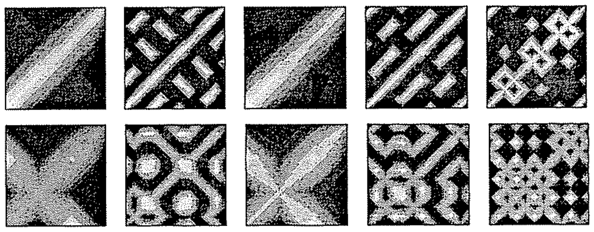
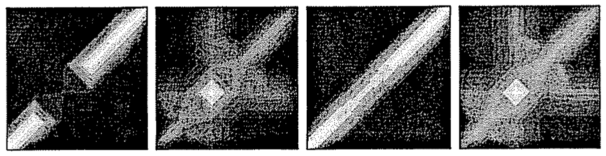


Figure 17: **Misconfigured Patterns.** *For levels 0, 5-LA, 5-LS, 5-RA, 5-LS, the upper graph shows the pattern when the RAID of IBM disks is correctly configured and the lower shows the pattern when two logical partitions are misconfigured such that they are placed on the same physical device.*



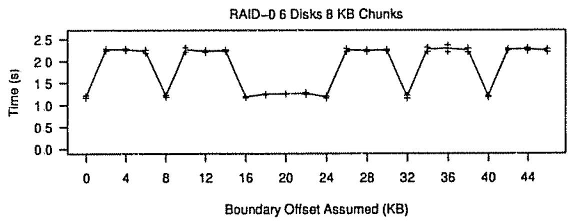


Figure 18: **Detecting Heterogeneity.** *The row of figures shows the results of the pattern layout algorithm on four different simulated disk configurations. In each configuration, a single disk has different capability than the others. A fast rotating, slow rotating, fast seeking, and slow seeking disk is depicted in each of the figures. The bottom figure shows the output of the boundary detection algorithm run upon the configuration with the fast rotating disk.*

graphs as well as in the boundary 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 boundary detection algorithm in Shear can be used to identify safety hazards by determining when a redundant array is operating in degraded mode. Figure 19 shows the boundary detection results for a ten disk software RAID system using the IBM disks. The upper graph shows the boundary detection correctly working after the array was first built. The lower graph shows how boundary detection is changed after we physically removed the fifth disk from the array. Recall that boundary 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.

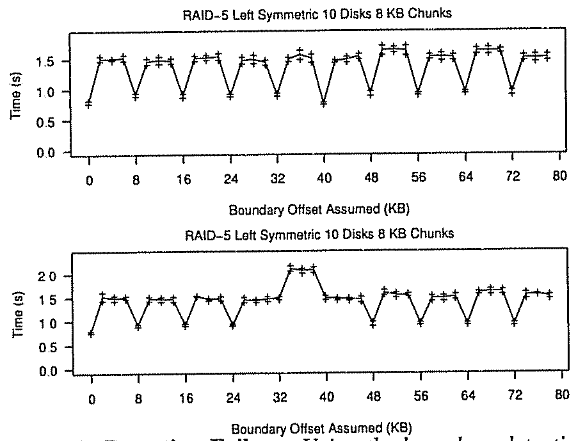


Figure 19: **Detecting Failure.** *Using the boundary detection algorithm, Shear can discover failed devices within a RAID system. The upper graph shows the initial boundary detection results collected after building a 10 disk software RAID system using the IBM disks. The lower graph is for the same system after a fault was induced on one of the disks.*

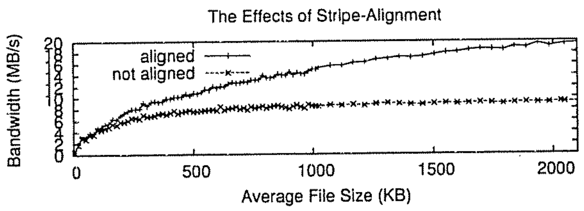


Figure 20: **Benefits of Stripe Alignment.** *The figure plots the bandwidth of a series of file creations of an average size, as varied along the x-axis. Two variants are shown: one in which the file system generates stripe-sized writes versus default Linux. The workload consists of creating 100 files. The x-axis indicates the mean size of the files, which are uniformly distributed between* $0.5 \times$ *mean and* $1.5 \times$ *mean.*

## 5.2 Shear Performance

The striping unit within a disk array can have a large impact on performance [4, 2]. 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 striping unit 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 striping unit.

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 very straight-forward: only about 20 lines of code were added to the file system.

This very simple change to make the file system stripe-aware leads to tremendous performance improvements. The experiments shown in Figure 20 are run on the 4-disk hardware RAID-5 described previously, with caching disabled. 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.

## 6. 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 [18]. Similar techniques have been applied to identify aspects of a TCP protocol stack [7, 13], to determine processor cycle time [22], and CPU scheduling policies [16].

The work most related to ours is that which has targeted characterizing a single disk within the storage system. For example, in [24], 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.* build a similar but more portable tool to achieve similar ends [20]. We plan to explore the use of such low-level single-disk tools in conjunction with Shear. In this scenario, Shear first exposes the boundaries between disks; a lower-level tool could then be used to understand more specific aspects of each disk.

Evaluations of storage systems have usually focused on measuring performance for a given workload and not on uncovering underlying properties [1, 10, 12]. One interesting synthetic benchmark adapts its behavior to the underlying storage system [5]; this benchmark examines sensitivity to parameters such as the size of requests, the ratio of reads versus writes, and the amount of concurrency.

Finally, the idea of using detailed storage-systems knowledge within the file system has been investigated, usually for systems composed of a single disk. For example, Schindler *et al.* investigate the concept of track-aligned file placement [21]; 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.

## 7. CONCLUSIONS

We have presented Shear, a tool that automatically detects characteristics of modern storage arrays. The key to Shear is its use of randomness combined with statistical techniques to deliver reliable detection. We have verified that Shear works as desired through a series of simulations, and have subsequently applied Shear to both software and hardware RAID systems, revealing properties of both. Finally, we have shown how Shear could be used by both administrators, to better understand their storage arrays, and the file system itself, to improve performance by tuning writes to the characteristics of the RAID underneath.

## 8. REFERENCES

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

[2] P. Chen and E. K. Lee. Striping in a RAID Level 5 Disk Array. In *SIGMETRICS '95*, May 1995.

[3] 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.

[4] P. M. Chen and D. A. Patterson. Maximizing Performance in a Striped Disk Array. In *ISCA '90*, pages 322–331, June 1990.

[5] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks. In *SIGMETRICS '93*, 1993.

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

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

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

[9] H.-I. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *ICDE '90*, 1990.

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

[11] 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.

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

[13] J. Padhye and S. Floyd. Identifying the TCP Behavior of Web Servers. In *SIGCOMM01*, June 2001.

[14] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, June 1988.

[15] D. Pelleg and A. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML '00*, 2000.

[16] J. Regehr. Inferring Scheduling Behavior with Hourglass. In *FREENIX '02*, June 2002.

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

[18] 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.

[19] S. Savage and J. Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *USENIX '96*, pages 27–39.

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

[21] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *FAST '02*, Monterey, California, January 2002.

[22] 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.

[23] 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.

[24] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *SIGMETRICS '95*.