AFRAID--A Frequently Redundant Array of Independent Disks

Stefan Savage, University of Washington
John Wilkes, Hewlett-Packard Laboratories

AFRAID — A Frequently Redundant Array of Independent Disks

Stefan Savage
University of Washington, Seattle, WA

John Wilkes
Hewlett-Packard Laboratories, Palo Alto, CA

Abstract

Disk arrays are commonly designed to ensure that stored data will *always* be able to withstand a disk failure, but meeting this goal comes at a significant cost in performance. We show that this is unnecessary. By trading away a fraction of the enormous reliability provided by disk arrays, it is possible to achieve performance that is almost as good as a non-parity-protected set of disks.

In particular, our AFRAID design eliminates the small-update penalty that plagues traditional RAID 5 disk arrays. It does this by applying the data update immediately, but delaying the parity update to the next quiet period between bursts of client activity. That is, AFRAID makes sure that the array is *frequently* redundant, even if it isn't always so. By regulating the parity update policy, AFRAID allows a smooth trade-off between performance and availability.

Under real-life workloads, the AFRAID design can provide close to the full performance of an array of unprotected disks, and data availability comparable to a traditional RAID 5. Our results show that AFRAID offers 42% better performance for only 10% less availability, 97% better for 23% less, and as much as a factor of 4.1 times better performance for giving up less than half RAID 5's availability.

We explore here the detailed availability and performance implications of the AFRAID approach.

1. Introduction

In a RAID 5 disk array, small writes take a long time to complete [Patterson88]. This is known as the "small update problem". In such an array, redundancy for a stripe of data is provided by a parity block, computed as the XOR of the data blocks in the stripe, in order to allow recovery if any disk fails. If a portion of a stripe is updated, the parity data must also be updated to preserve the recoverability property (Figure 1). To do this, it is necessary to (1) read the old value of the data to be overwritten, unless it is already cached in the array controller; (2) read the old parity; (3) XOR the new data with the old, and XOR the result with the old parity to generate the new parity data; (4) write the new data and (5) write the new parity.

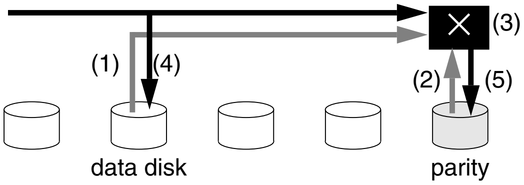

Figure 1: doing a small update in a traditional RAID 5.

Thus, three or four disk I/Os are needed to achieve one small write — all of which are in the critical path. In contemplating this problem we made the following observations:

- modern disks are extremely reliable—so much so that disk array reliability is limited more by its support components than its disks;
- many real workloads have slack periods between bursts of client activity;
- people are already well-used to the notion of time-limited exposure to risk.

These eventually led us to the idea of AFRAID (*A Frequently Redundant Array of Independent Disks*).[1] AFRAID is a RAID 5 disk array that relaxes the coherency between data and parity for short periods of time; parity is made consistent again in the idle periods between bursts of client writes. Thus the stored data is *frequently* held redundantly, rather than always guaranteed to be so.

In this approach, small updates are not required to wait for the parity to be updated, thereby reducing the four I/Os in the critical path of the traditional small-update protocol to just one: write the new data. The benefit is that performance approaches that of an unprotected array. The disadvantage is a slightly increased risk of data loss from a disk failure, but we will show that this increase is small in practice, and also that it can be bounded at the cost of some performance. That is, AFRAID allows a smooth trade-off between increased reliability and increased performance.

[1] Like so many good ideas, ours was of course developed by back-determination from the acronym.

1.1. The AFRAID design

A write in AFRAID does two things: it updates the on-disk copy of the original data, and it causes the target stripes to be marked *unredundant*—i.e., that their parity needs rebuilding. This is indicated by setting a bit per stripe in a non-volatile memory in the array controller; attempting to re-mark an already-marked stripe does nothing. Once one or more stripes have been marked, the AFRAID controller waits until the array is idle and then starts to process the pending parity updates where they can be achieved at effectively zero performance cost to the clients of the array.

Each parity update requires reading all the data blocks in the stripe and XORing them together to generate a new parity block. The new parity block is then written over the top of the old one, after which the unredundant mark for the stripe is removed. The rebuilding of adjacent unprotected stripes can be coalesced to increase the efficiency of disk accesses. Since the overhead of the parity update is linear with the number of disks in a stripe group, AFRAID is best suited to arrays with smaller numbers of disks.

The additional cost to build an AFRAID is just the cost of the marking memory: one bit per stripe. With an array that is 5 disks wide and has a stripe unit size, or stripe depth, of 8KB, this is ~3 bits per 100KB, or 3 KB of memory per 1GB of stored data—a trivial cost compared to the cost of the disk storage itself. The recovery technique for a failed marking memory is simply to rebuild parity for the whole array. This rebuilding can proceed in parallel with continued use.

Any write to a stripe unprotects it all—not just the data being written to. Somewhat counterintuitively, this loss may include data that has not been written to recently. This failure mode is a natural consequence of RAIDs protecting whole stripes, rather than individual blocks. In practice, the exposure is quite small, because the likelihood of data being lost is minimal.

Rather than simply waiting for an idle period before starting to reconstruct parity, it is possible to configure AFRAID to be more aggressive about availability, at the possible cost of greater interference with foreground I/Os. Sample policies include:

- allowing parity rebuilding to start even when there is a non-zero client load on the array;
- giving parity rebuilding priority over foreground client I/Os;
- reverting to normal RAID 5 behavior.

These techniques can be enabled dynamically and adaptively to achieve specified long-term availability and performance goals. We explore the performance and availability effects of some of these policies below.

1.2. AFRAID design assumptions

In this section we provide some additional information about the suppositions on which we based our work.

Disk reliability. Modern disks have published mean time to failure (MTTF) times of 0.5–1 million hours, and this number is increasing every year. As a result, small disk arrays have vanishingly small chances of experiencing a dual-disk failure, which would cause a data loss. The expected disk-related mean time to data loss (MTTDL) in a small RAID 5 with a half-dozen disks is measured in hundreds of millions of hours—several tens of thousands of years. This is far larger than the limits imposed by other "support" components such as the array's power supply, controller, and cabling. Small disk arrays with less than a dozen disks are the most common in practice, and their overall data availability may not be reduced much if full on-disk redundancy is not provided for short periods.

As we will show, any data-redundancy scheme that produces a disk-related MTTDL of a few million hours or better will be dominated in practice by the array support components. An aggregate MTTDL of a million hours (114 years) translates into only a 2.6% likelihood of any data loss at all during a typical 3-year array lifetime. This is much lower than the rate of problems due to software failures, operator errors, and other environmental difficulties [Gray90, Gray91a]—that is, a small-to-medium sized array that achieves an overall MTTDL of 1M hours or better will probably be entirely adequate for the majority of its applications.

In addition to reduced failure rates, modern disks also provide feedback mechanisms for predicting when such failures will occur. These can warn of impending disk failures hours or days in advance by looking at soft-error logs (e.g., the number of retries required on reads), or the variation in head flying-height. [Lin90c] discusses one such experiment, which was able to predict 93.7% of system failures in a distributed file system, typically many hours before they happened. More recently, IBM disks drives have incorporated a scheme that has been shown to offer a mean of 10 days warning of disk failures [IBMpfa95], with an anticipated success rate of 50–60%. Other manufacturers are following suit. With such techniques available, the likelihood of experiencing an unexpected disk failure can be made very small.

Bursty access patterns. Many (if not most) real uses of disk arrays have bursty access patterns, with periods of relative inactivity between groups of client accesses. [Ruemmler93] offers one quantification of this, in some detail. Indeed, we believe that it is usually only in benchmarks or very large, carefully-tuned systems that arrays are driven to saturation for long periods of time. Given this, there will be spare I/O time available in the idle periods between bursts. If expensive operations such as parity updates can be delayed to these less busy periods, they can be achieved at little or no apparent performance cost to the client.

Time-limited exposure to data loss. This principle is already well understood and frequently exploited. For example, data in UNIX[2] file systems is held unprotected in volatile memory buffers before it is written out to disk [Ritchie84a]. Because data is typically only held in volatile memory for short periods of time before being written to disk, this exposure is tolerated in exchange for the increased performance that results. This idea has been extended still further by special file systems that deliberately store data in volatile memory [McKusick90, Ohta90].

Most systems that use non-volatile memory (NVRAM) assume that a single copy of the stored data is sufficient: providing true single-failure-tolerant NVRAM systems is difficult [Menon93], and so is rarely done. Examples of common systems that make this tradeoff include the popular PrestoServe card [Moran90] for NFS servers, as well as recent file system designs, such as LFS [Rosenblum92] and Zebra [Hartman95]. All these rely on assembling large amounts of data in NVRAM to obtain both good performance and acceptable reliability. Other studies have suggested ways to extend this use of NVRAM still further [Baker92b].

Together, these thoughts led us to the AFRAID notion: consciously sacrificing a small amount of data redundancy in order to achieve considerably better performance.

The remainder of the paper explores the AFRAID idea in some detail. We begin with a short discussion of closely-related prior work. This is followed by a description of analytic availability models for AFRAID, and then a quantitative evaluation of the availability and performance effects of the AFRAID design as compared to a traditional RAID 5 and a set of unprotected disks. We conclude with some observations on what this study taught us.

2. Related work

The most obviously related prior solution to the small update problem is parity-logging [Stodolsky93]. A parity-logging array defers the parity-update cost to a later time, at which point it can be performed more efficiently. It does this by performing the traditional RAID 5 read-modify-write operation on the data block being updated, but then, instead of doing the same for the parity block, it writes the XOR of the old and new data to a log—thereby preserving full redundancy all the time. At a later date, the log file is replayed against the disk array, and the parity updated in situ.

By comparison, AFRAID avoids a pre-read of the old data in the critical path for writes, and thus saves a complete disk revolution on most small writes. It also avoids potentially long delays during parity rebuilds. For efficiency, the parity logging scheme applies a batch of parity updates at a time, which can interfere with foreground I/O requests. Although some of the policies used in AFRAID to control availability can also generate interference with foreground I/Os, they are less intrusive because parity updates may be preempted between stripes.

The parity logging scheme could be extended to apply its parity updates in idle periods, like AFRAID. This would improve its performance, except under workloads in which the parity log fills up, when either the pending parity updates must be applied immediately, interrupting foreground processing to do so, or the array must revert to a regular RAID 5 model of operation until it becomes idle enough to apply updates. Efficiency will drop for either approach. There is no parity log to fill up in AFRAID—all that happens is that the data becomes less well protected.

Finally, parity logging is quite a complicated scheme; AFRAID is much simpler.

Another approach to the same problem is the floating-parity scheme [Menon89]. This reduces the cost of parity updates by writing the new parity data to an empty, rotationally-nearby slot in the array, rather than waiting for a full revolution to go by to update it in place. Such an array still needs to do the old-data and old-parity reads. The floating-data scheme extends this placement optimization to data blocks, too, but this requires considerably larger amounts of non-volatile state information: a word or two per stored block.

In contrast to these two schemes, AFRAID has a less efficient parity update scheme (reading all the data portions of the stripe and recalculating the parity from scratch), but it uses it during a time when the array is less utilized, so that the resulting client-visible cost is small or zero. The result is better performance when the array is active, at the cost of a small exposure to data loss if a (rare) disk failure happens before the new parity has been calculated and written. We quantify the degree of this exposure more precisely in Section 3.

The idea of allowing a file to become unredundant while it was being created was proposed in [Cormen93], in the context of parallel file systems for scientific computing. This paper also suggested the notion of *paritypoints*, by analogy to checkpoints, at which an application could ask for the parity to be computed for the file. Our AFRAID design takes these ideas several steps further: it automates the process of recomputing parity; and does so in idle periods rather than only on demand; it isolates the parity rebuilding

[2] UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

inside the disk array where it need not be visible to application programs; and it is not limited to stripe-aligned files.

Determining when the array is going to be idle enough to do the rebuild without impacting incoming work is a problem that has been studied already. [Golding95] discusses this problem and a variety of solutions to it.

All the well-known techniques that have been developed for performing stripe rebuilds in a recently repaired disk array can be applied to the problem of rebuilding the parity in AFRAID (e.g., [Muntz90, Holland92]). These include opportunistically piggybacking the parity updates on other "nearby" activity done in the foreground; batching together updates that are physically close together; or simply doing a single, linear sweep through the disks.

Similarly, existing schemes for balancing disk traffic under failure conditions can be applied to AFRAID (e.g., [Gray90c, Muntz90, Blaum94, Reddy91]). For ease of exposition, however, we concentrate here on a straightforward left-symmetric RAID 5 data layout.

3. Availability model of AFRAID

In this section we develop analytic models of data-loss mechanisms for AFRAID, basing them on similar models for traditional RAIDs. In the next section we apply these models to the data from our simulation experiments to provide a quantitative evaluation of data availability in AFRAID.

Following [Gibson93], we do not separate the cases of inaccessible data from data that has been lost irrevocably. We use the term *availability* in this paper to refer to the amount of time that data is accessible and/or not lost. To make our discussion concrete, we apply a set of assumptions about typical failure rates for modern array components; these are summarized in Table 1.

Because manufacturers do not yet publish $\text{MTTF}_{\text{unexpected}}$ separately from overall MTTF, we have been fairly conservative and set the coverage factor C for disk failure predictions to 0.5 in our calculations. That is, we assume that half of the disk failures will not be predicted ahead of time. In what follows, we include the coverage factor in the $\text{MTTF}_{\text{disk}}$ calculations:

$$\text{MTTF}_{\text{disk}} = \text{MTTF}_{\text{disk-raw}} / (1 - C)$$

3.1. Mean time to first data loss

Most RAID data-loss calculations look only at the time to the first catastrophic data loss due to disk failure, which occurs if two disk failures occur so close together that the first failure has not yet been recovered from. If this happens, two disks worth of data is lost. A convenient measure for this kind of catastrophe is the *mean time to data loss* (MTTDL). The equation for a RAID 5 disk array with N+1 disks, assuming rare, independent, exponentially-distributed disk failures is:[3]

Table 1: values assumed for calculations in this paper.

Parameter	*Value*
disk mean time to failure $\text{MTTF}_{\text{disk-raw}}$	1 M hours
support hardware mean time to data loss $\text{MTTDL}_{\text{support}}$	2 M hours
disk failure-prediction coverage (C)	0.5
mean time to repair (MTTR)	48 hours
stripe unit size (S)	8 KB
size of disk (V_{disk})	2 GB

$$\text{MTTDL}_{\text{RAID-catastrophic}} = (\text{MTTF}_{\text{disk}})^2 \,/\, (\, N\,(N{+}1) \times \text{MTTR}_{\text{disk}}\,) \qquad (1)$$

With a 5-disk array, and the parameters of Table 1, this gives a theoretical MTTDL of $\sim 4.10^9$ hours, or about 475,000 years.

In addition to the regular RAID failure mode, AFRAID exhibits data loss if a single disk fails unexpectedly while there is some unprotected data. To determine the combined effect of these two modes, we look at the likelihood of data loss occurring when there is unprotected data (T_{unprot}) and when there is not ($T_{\text{total}} - T_{\text{unprot}}$). A conservative measure of the contribution to MTTDL for the period in which there is unprotected data is:

$$\text{MTTDL}_{\text{AFRAID-unprotected}} = (T_{\text{total}}/T_{\text{unprot}}) \times \text{MTTF}_{\text{disk}}/(N{+}1) \qquad (2a)$$

This measure is conservative because we do not take account of the fact that in some cases only parity data will be lost: we just simplify and assume that there will always be some data loss. The rest of the time, when there is no unprotected data, AFRAID behaves just like a RAID for the MTTDL measure:

$$\text{MTTDL}_{\text{AFRAID-RAID-catastrophic}} = T_{\text{total}}/(T_{\text{total}} - T_{\text{unprot}}) \times \text{MTTDL}_{\text{RAID-catastrophic}} \qquad (2b)$$

Summing these two contributions, which are best through of as inverses of rates, gives:

$$\text{MTTDL}_{\text{AFRAID}} = 1 \,/\, (\, 1/\text{MTTDL}_{\text{AFRAID-unprotected}} + 1/\text{MTTDL}_{\text{AFRAID-RAID-catastrophic}}\,) \qquad (2c)$$

[3] [Gibson93] includes several rather fancier formulae (e.g., equations 12 and 14) that give additional accuracy for large arrays with many tens to hundreds of disks. In addition to the fact that it would need another page or so to explain them, they don't help characterize the much smaller arrays that are the common case, and the target of AFRAID.

Section 4.3 presents the results of experimental determinations of this value over several workloads.

One more kind of multiple failure can afflict an AFRAID: if its NVRAM marking memory fails, the array will start reconstructing parity across all the stripes. This will take a little while (about ten minutes for an array using 2GB disks that can read at a sustained rate of 5MB/s). If a disk failure occurs before the parity has been completely rebuilt, the array has no way of knowing which stripes were still unprotected, if any, although it will be bounded by the knowledge of how far the reconstruction has progressed. The likelihood of this failure is exceedingly small, however, because of the small window of vulnerability (MTTDL > 10^{11} hours), so we can safely ignore it here.[4]

3.2. Mean data loss rate

The MTTDL measure indicates the expected rate of failures leading to *any* data loss. In any RAID 5-based system this occurs on a dual-disk failure, at which point a catastrophe occurs: two whole disks worth of data vanishes. In addition, unprotected data in AFRAID is vulnerable to loss from a single-disk failure. However, the amount of data lost in this case is bounded by how much is unprotected—and we will show later that it is often quite small.

There is an important qualitative difference between losing a block or two on a disk and losing the whole disk. For example, all disks have a few defective sectors or tracks, and new ones are occasionally added to this list over its lifetime—but the occurrence of a bad block on a disk doesn't mean that the entire disk has been lost, or even that it should be discarded. Similarly, the effect of accidentally deleting a single small file is usually much less severe than that of losing an entire file system. The former may be merely tedious, while the latter can be a catastrophe.

There are several reasons for this qualitative difference: not all data is equally valuable; some data can easily be reconstructed or recomputed; much data "dies young"—that is, it will be deleted or overwritten soon after it is created [Ousterhout85a]; recovering a single file is often simpler than rebuilding an entire disk set. Others have taken advantage of this difference before us. For example, the BSD fast file system [McKusick84] and its journalling file system successors take considerable care to maintain consistency of file system metadata, but are much more cavalier with user information.

As a result, we feel that it is important to measure the *amount* of data subject to loss, as well the time to lose the first byte. A good metric for this is the *mean data loss rate* (MDLR): the product of the amount of data loss and the rate at which it is likely to occur. In addition to quantifying the effects of such losses, it has the advantage of being a reminder that mean time to failure values should be used only to define failure *rates*, not expectations of *lifetimes*.

The catastrophic data loss rate for a regular array due to a two-disk failure can be cast in these terms as:

$$\text{MDLR}_{\text{RAID-catastrophic}} = 2V_{\text{disk}} \times N/(N+1) \times 1/\text{MTTDL}_{\text{RAID-catastrophic}} \qquad (3)$$

where V_{disk} is the capacity of a single disk, which is reduced by the second term to reflect that some of the lost disk space held parity rather than data blocks. The RAID 5 array we considered earlier would have a MDLR of ~0.8 bytes/hour from this failure mode.

Analyzing the impact of single disk failures in AFRAID requires additional information. To provide a basis for this availability analysis, we first introduce the notion of *parity lag*, which is the amount of unredundant non-parity data present in the array at any time, measured in bytes. The *mean parity lag* is the average parity lag over some test period, such as the duration of a test workload. Note that parity lag is workload dependent.

Data loss only occurs from a single disk failure if the parity lag is non-zero at the time of the failure. When this happens, one stripe unit (block) from each unredundant stripe is lost (the one on the failed disk), unless the lost block is a parity block, in which case no actual data loss occurs.

The mean data loss rate for a single-disk failure on AFRAID with unprotected data is:

$$\text{MDLR}_{\text{unprotected}} = (\textit{mean-parity-lag}/N) \times (N+1)/\text{MTTF}_{\text{disk}} \qquad (4)$$

where the first term reflects the average amount of unprotected non-parity data vulnerable to a single disk failure, and the second term gives the total failure rate of all the disks in the array. We will present experimental determinations of these values in section 4.3.

Summing the different component contributions gives us the final mean data loss rate for the disk-related components of AFRAID:

$$\text{MDLR}_{\text{AFRAID}} = \text{MDLR}_{\text{RAID-catastrophic}} + \text{MDLR}_{\text{unprotected}} \qquad (5)$$

3.3. The effect of support components

We have concentrated so far only on disk failures. This emphasis made sense when disks were much less reliable than support components such as cooling fans, power supplies, cabling, and other passive components. But one of our contentions is that disks are no longer the primary cause of problems in a disk array. The reliability of the support hardware and the

[4] The formula, for the curious, is:

$$\text{MTTDL}_{\text{NVRAM+disk}} = \text{MTTF}_{\text{NVRAM}} \times \text{MTTF}_{\text{disk}} / ((N+1) \times \textit{rebuild-time})$$

array controller is little or no better than that of the disks.

The data in [Schulze89] suggests that these support components would together lead to a mean time to failure of a small array of about 46k hours; [Gibson93] simply increases this to "a more reasonable value of 150k hours" without further discussion. Fortunately, more recent designs and pressure by manufacturers to boost reliability seem to have increased the quality of these components, and although many array manufacturers disconcertingly consider the data either irrelevant or proprietary, a few do not, and we were quoted MTTF numbers of 20–35k hours, and MTTDL values of 270k to 5M hours. Some typical component MTTF examples are: 150k or 0.5–1M hours for the controller; 300–500k hours for a host bus adapter; 50–350k hours for a power supply module; and 1–3M hours for cabling and packaging.[5]

It takes considerable engineering effort and use of redundant components to increase the overall MTTDL above 1M hours. For example, the HP AutoRAID array [Wilkes95] uses two redundant power supplies, three fans (any two of which can keep the system cool enough for continued operation), and can support a dual controller configuration; each controller has a separate NVRAM that uses dual rechargeable batteries that are periodically discharge-tested. The result: with a fully-populated system (12 disks and 2 controllers), the array's overall MTTDL is specified (probably conservatively) as 1.97M hours, together with an overall MTTF of 31k hours. Few designers of small arrays go to all this trouble: for example, the Network Appliance's FAServer350 product has a predicted MTTF of around 20–30k hours with four disks, and disks are its only redundant components.[6]

Together, these figures suggest that the current "more reasonable value" for the aggregated non-disk components of a *conservatively-engineered* array is probably about 2M hours. This is a far cry from the 4.10^9 hours calculated from the independent-disk failure model considered earlier. With a 2M hour MTTDL, our 5-disk array would suffer a MDLR of 4.0KB/hour; using the 150k hour figure from [Gibson93] would increase this to 53KB/hour.

The lesson here is that it is the support components that determine the availability of a modern disk array, not its disks.

3.4. Non-volatile memory

Despite the extensive use of NVRAM in high-availability systems, remarkably little data has been published on its reliability.

Integral lithium-cell-backed static RAM is probably one of the most reliable kinds of NVRAM: it offers retention lifetimes of 25–87k hours and extremely low failure rates [Dallas94, Dallas95], but it is quite expensive: ~$350/MB for a representative state-of-the-art part from Dallas Semiconductor.

To avoid this expense, many systems use dynamic RAM backed by rechargeable batteries based on NiCd cells. The battery technology often limits the resulting availability: achieving MTTF values above a few tens of thousands of hours requires the use of redundant batteries whose status is periodically tested by controlled discharging, and careful attention to charging circuitry and battery conditioning. The complexity and cost of this design means that it is not often used, so most battery-backed NVRAM has a much lower MTTF than the Li-cell backed RAM. For example, the popular PrestoServe card has a predicted MTTF of 15k hours [Neary91]; with 1MB of vulnerable data, this corresponds to an MDLR of 67 bytes/hour.

We will show that this means that single-copy NVRAM applications are already accepting significantly higher risk of data loss than results from the temporary lack of parity protection in AFRAID.

3.5. Power failure

One additional support component that is particularly important is external power. Up to this point, our discussion has assumed that external power failures simply don't happen. This matters because a power failure that happens while a RAID 5 is writing can lead to data loss unless a separate, non-volatile intentions log is kept.

[Gibson93] reports a MTTF of 4300 hours for mains power (i.e. a power failure about every 6 months). This is probably reasonable for parts of North America and Europe, but would be optimistic in some other parts of the world. In our traces, we saw outstanding writes up to 59% of the time, with a mean of 20%. Even using a more conservative value of a 10% write duty cycle on a 5-disk RAID 5 gives a MTTDL of only 43k hours due to external power failures. The effect on MDLR is roughly to double it (0.7 bytes/hour), but the change in MTTDL represents losing about 98% of the availability that the array offers.

It might be thought that providing an uninterruptible power supply, or UPS, would be overkill for a small array, but it may be the single largest contributor to preventing data loss. Using a high-grade UPS with an MTTF of 200k hours [Best95] and a 10% write duty cycle returns the MTTDL for the array's external power components to 2M hours.

The large variability in power and UPS reliability can obscure the other support contributions to MTTDL, so we have chosen not to include external power failure in the calculations in this paper.

[5] Storage Dimensions technical support line, personal communication, October 1995.

[6] Rich Boburg, Network Appliance, personal communication, October 1995.

3.6. How much availability is enough?

When RAIDs were first being discussed as a replacement for large disks, the MTTF for small disks was 20–30,000 hours, and the target was to match the reliability of a single, large disk with a MTTF of 30–100,000 hours [Patterson88]. Things have improved since then: modern small-form-factor disks typically have a published MTTF of 0.5–1.0M hours. Given that the expected useful lifetime of a disk or disk array is probably no more than 3 years, or about 26k hours, this is equivalent to a lifetime expected failure likelihood of 3–5%. If it held 2GB, its mean data loss rate would be 2–4KB/hour. This means that the *best* of the traditional 5-disk RAIDs, limited to a MTTDL of about 1–2M hours by their support components, are achieving a MDLR for the whole array roughly equivalent to that of a single disk.

We contend that the combination of modern, highly reliable disks with traditional RAID technology provides more than enough protection against disk failures, and that further efforts to increase data availability are attacking a solved problem for the vast majority of customers. Instead, we suggest that it may be worth exchanging some of the "excess" disk availability for better performance—which is precisely what AFRAID does.

4. Evaluation

To provide a quantitative evaluation of the AFRAID concept, we used a detailed event-driven simulator to compare the performance and availability of an AFRAID array with a non-AFRAID system under a variety of workloads. We report here on three aspects of this evaluation:

- the relative performance of AFRAID, RAID 5, and RAID 0 (an unprotected array);
- quantitative availability measures;
- the relationship between performance and availability under AFRAID.

We begin with a description of our experimental setup.

4.1. Experimental methodology

In order to evaluate whether real-life workloads are bursty enough for an AFRAID array to rebuild parity quickly, it was necessary to look at some real-life workloads. So we did. Here are the ones we used:

- *hplajw* — a single user HP-UX [Clegg86] system used mainly for email and document editing.
- *snake* — an HP-UX file server for a cluster of workstations at UC Berkeley.
- *cello* — an HP-UX timesharing system for about 20 people doing text editing and program development. We used two subsets of the full cello trace: *cello-usr* is the set of three disks holding the root file system, /usr, and /users; *cello-news* is a single disk holding the Usenet news database: it received half of all the disk I/Os in the system.
- *netware* — an intensive database-loading benchmark measured on a Novell Netware server.
- ATT — a production telephone-company database system. On the real system, the entire dataset was mirrored; for our tests, we just used one copy of the data.
- IBM AS400 — four production AS400 systems. These traces were supplied to us by Bruce McNutt of IBM San Jose; we called them AS400-1 through AS400-4.

The workloads for the first three of these systems are described in great detail in [Ruemmler93]. We used one-day subsets of them for this work.

To evaluate our claims we constructed a detailed event-driven performance simulation of AFRAID using the Pantheon[7] simulator, which includes the calibrated disk models discussed in [Ruemmler94]. To simplify the discussions and save space, we just consider spin-synchronized arrays here.

To add AFRAID to Pantheon, we started with a detailed RAID 5 model and adapted it to support AFRAID. The changes were small: they consisted of adding the marking memory and updating it on writes, and not doing the read-modify-write cycle for the parity in AFRAID mode. We added a background idle-task to do the parity rebuilds, triggered by an idle-detection network [Golding95] or explicit foreground policies. By default, we used a timer-based idleness detector with a 100ms delay: that is, AFRAID started processing parity updates once the array had been completely idle for 100ms; the output from the idle-period predictor was ignored.

To make sure that we were seeing the effects of the AFRAID policies themselves rather than just the disk array's cache policies [Ruemmler94], we chose a small (256KB) write staging area with a write-through policy together with a small (256KB) read cache with no array-level readahead. Since our workloads came from systems with much larger file buffer caches, read hits in the array's cache were rare. We limited the number of concurrently active client requests inside the array to the number of physical disks it had; the host device driver used the CLOOK policy [Worthington94a], the back-end device drivers inside the array used FCFS. We modelled HP C3325 2GB 3.5" 5400 RPM disks in the array [HPC3324].

Multiple writes to the same stripe were allowed to proceed in parallel, but would block if a parity-rebuild

[7] The simulator used to be known as TickerTAIP: we changed its name to avoid confusion with the parallel RAID array of the same name [Cao94b].

on that stripe was in progress. Requests were never preempted: once started, they ran to completion.

The I/O times we report in this paper start when a request is given to the device driver, and stop when the request is completed by the array. They include both the time spent in the array itself and any time spent queued in the device driver. Given that we are using an open-queueing, trace-driven workload, this provides the fairest assessment of the performance that would be seen by a user or file system.

We took no special action for synchronous writes: ones for which the file system waits until the data being written has been put onto non-volatile media. Such writes are designed to ensure resilience against power-failure, not against disk failure; for example, they are used to disable immediate-reporting in disks that allow this [Ruemmler93, Ruemmler94]. Even if we had chosen to force a parity-update on a stripe updated by a synchronous write, the redundancy would go away again on the next update to any block in the stripe—not just the one that had been written to synchronously—because parity protection is at the stripe level, not the block level.

Because almost all of the code was the same between the various array models, direct performance comparisons between them are possible. Indeed, to make sure that the exact same disk and cache algorithms was executed in all cases, we modelled RAID 0 as an AFRAID that simply never did parity updates.

About the only things that we did not model were a few performance improvements for AFRAID, of which the most important were probably aggregation of adjacent stripes needing parity rebuilds and piggybacking parity updates on disk accesses to nearby blocks.

In addition to the baseline AFRAID design, which updated parity only in idle periods, we implemented a policy which directly traded off performance for improved availability. This MTTDL-X policy is designed to keep the disk-based MTTDL above a particular target value (X). To do this, it continuously calculates the MTTDL that has been achieved so far, and reverts to RAID 5 mode if the goal is not being met. (It also starts the parity update for any unprotected stripes at this time.) The policy attempts to limit MDLR by automatically starting a parity update when more than 20 stripes are unprotected, even if the array is not idle; we had found earlier that this was fairly effective and caused little performance degradation.

4.2. Performance evaluation of AFRAID

Figure 2 and Table 2 present the results of exploring the relative performance of AFRAID, RAID 5 and RAID 0 across a range of workloads and parity-update policies. The figure and table show that, as predicted, pure AFRAID performance is very close to that of RAID 0, with a smooth degradation in performance towards that of RAID 5 as AFRAID is configured to increase data availability.

The performance of the baseline AFRAID was a geometric mean of 4.1 times that of RAID 5 across our test workloads. By comparison, RAID 0 performance was 4.2 times that of RAID 5. Thus, AFRAID is living up to the first part of its promise: performance comparable to non-protected arrays.

4.3. Availability measures for AFRAID

Our next experiments determined the availability delivered by the different parity update policies under real workloads. The results are shown in Table 3 and Table 4. The AFRAID contribution to MDLR from unprotected data is extremely low: with the exception of the heavy load from the ATT trace, $\text{MDLR}_{\text{unprotected}}$ contributes less than one byte per hour to the overall MDLR. This is tiny by comparison to the overall MDLR,

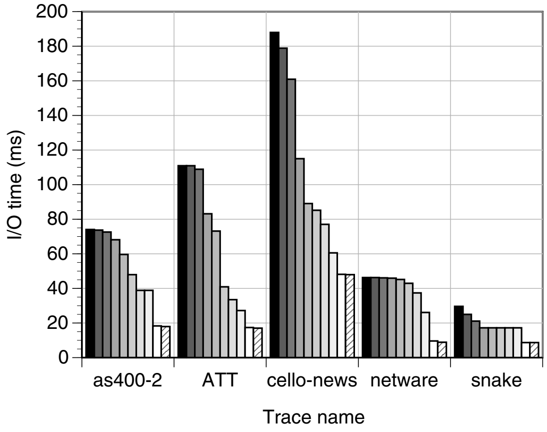

Figure 2: absolute performance of RAID 5 (leftmost bars), AFRAID-baseline and RAID 0 (rightmost bars) with a range of AFRAID-MTTDL-X policies in between.

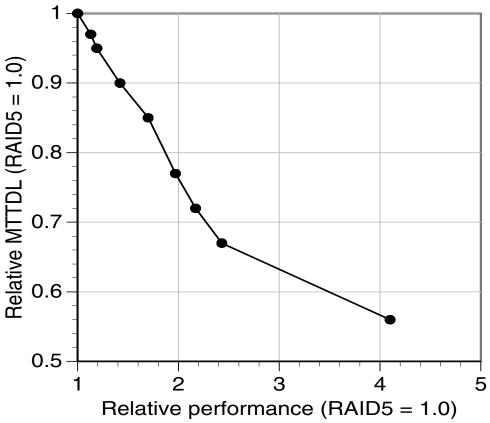

Figure 3: relative performance and MTTDL for RAID 5 (top left) to AFRAID-baseline (bottom right). These are geometric means across all the workloads we studied.

Table 2: performance data for the traces we studied.
AFRAID-MTTDL-X is an AFRAID that reverts to RAID 5 when the availability drops below a target threshold.

Workload	*AS400-1*	*AS400-2*	*AS400-3*	*AS400-4*	*ATT*	*cello-news*	*cello-usr*	*hplajw*	*netware*	*snake*
Number of disks (N+1)	4	4	4	4	5	4	4	4	8	4
Trace duration (hours)	0.6	1.5	1.7	1.0	1.0	24	24	24	1.1	24
Mean I/O time (milliseconds)										
RAID0	58.0	18.0	12.8	22.6	17.1	48.1	13.0	20.9	8.9	8.7
AFRAID-baseline	58.5	18.4	12.9	22.9	17.5	48.2	13.2	21.2	9.7	8.8
AFRAID-MTTDL-2M	125	38.9	23.8	39.3	33.6	77.1	18.5	27.2	37.4	17.2
AFRAID-MTTDL-16M	179	68.2	33.5	73.6	83.2	115	43.3	27.2	45.9	17.2
AFRAID-MTTDL-64M	183	73.8	37.2	79.9	111	179	96.8	27.2	46.4	25.1
RAID5	183	74.2	37.8	80.4	111	188	104	70.2	46.4	29.7

Table 3: mean data loss rate (MDLR) for the traces we studied.
MDLR-nosupport excludes data losses due to the support hardware, while MDLR-total includes them.

Workload	*AS400-1*	*AS400-2*	*AS400-3*	*AS400-4*	*ATT*	*cello-news*	*cello-usr*	*hplajw*	*netware*	*snake*
MDLR-unprotected (bytes/hour)										
AFRAID-baseline	0.08	0.02	0.01	0.02	5.93	0.47	0.06	<0.01	0.71	0.02
MDLR-nosupport (bytes/hour)										
RAID0	18K	18K	18K	18K	32K	18K	18K	18K	98K	18K
AFRAID-baseline	0.51	0.45	0.44	0.46	6.70	0.90	0.50	0.43	3.06	0.45
RAID5	0.43	0.43	0.43	0.43	0.77	0.43	0.43	0.43	2.35	0.43
MDLR-total (bytes/hour)										
RAID0	21K	21K	21K	21K	36K	21K	21K	21K	105K	21K
RAID5, AFRAID	3K	3K	3K	3K	4K	3K	3K	3K	7K	3K

Table 4: mean time to data loss (MTTDL) data for the traces we studied.
MDLR-nosupport excludes data losses due to the support hardware, while MDLR-total includes them.

Workload	*AS400-1*	*AS400-2*	*AS400-3*	*AS400-4*	*ATT*	*cello-news*	*cello-usr*	*hplajw*	*netware*	*snake*
Percentage of time with unprotected data										
AFRAID-baseline	51.1%	18.3%	13.4%	22.4%	22.5%	7.9%	8.8%	<0.1%	51.3%	4.2%
AFRAID-MTTDL2M	25.6%	12.7%	7.4%	13.7%	10.7%	3.9%	3.2%	<0.1%	12.3%	2.1%
AFRAID-MTTDL64M	0.8%	0.8%	0.8%	0.8%	0.6%	0.7%	0.7%	<0.1%	0.3%	0.7%
MTTDL-nosupport (hours)										
RAID0	0.33M	0.33M	0.33M	0.33M	0.25M	0.33M	0.33M	0.33M	0.14M	0.33M
AFRAID-baseline	0.98M	2.74M	3.74M	2.23M	1.77M	6.29M	5.63M	300M	0.49M	11.8M
AFRAID-MTTDL2M	1.95M	3.92M	6.74M	3.66M	3.74M	12.8M	15.5M	556M	2.02M	24.0M
AFRAID-MTTDL64M	61.2M	63.5M	63.6M	62.9M	62.9M	66.4M	66.4M	556M	64.6M	75.3M
RAID5	6.94G	6.94G	6.94G	6.94G	4.17G	6.94G	6.94G	6.94G	1.49G	4.17G
MTTDL-total (hours										
RAID0	0.29M	0.29M	0.29M	0.29M	0.22M	0.29M	0.29M	0.29M	0.13M	0.29M
AFRAID-baseline	0.66M	1.16M	1.30M	1.05M	0.94M	1.52M	1.48M	1.99M	0.39M	1.71M
AFRAID-MTTDL2M	0.99M	1.32M	1.54M	1.29M	1.30M	1.73M	1.77M	1.99M	1.01M	1.85M
AFRAID-MTTDL64M	1.94M	1.94M	1.94M	1.94M	1.94M	1.94M	1.94M	1.99M	1.94M	1.95M
RAID5	2.00M	2.00M	2.00M	2.00M	2.00M	2.00M	2.0M	2.00M	2.00M	2.00M

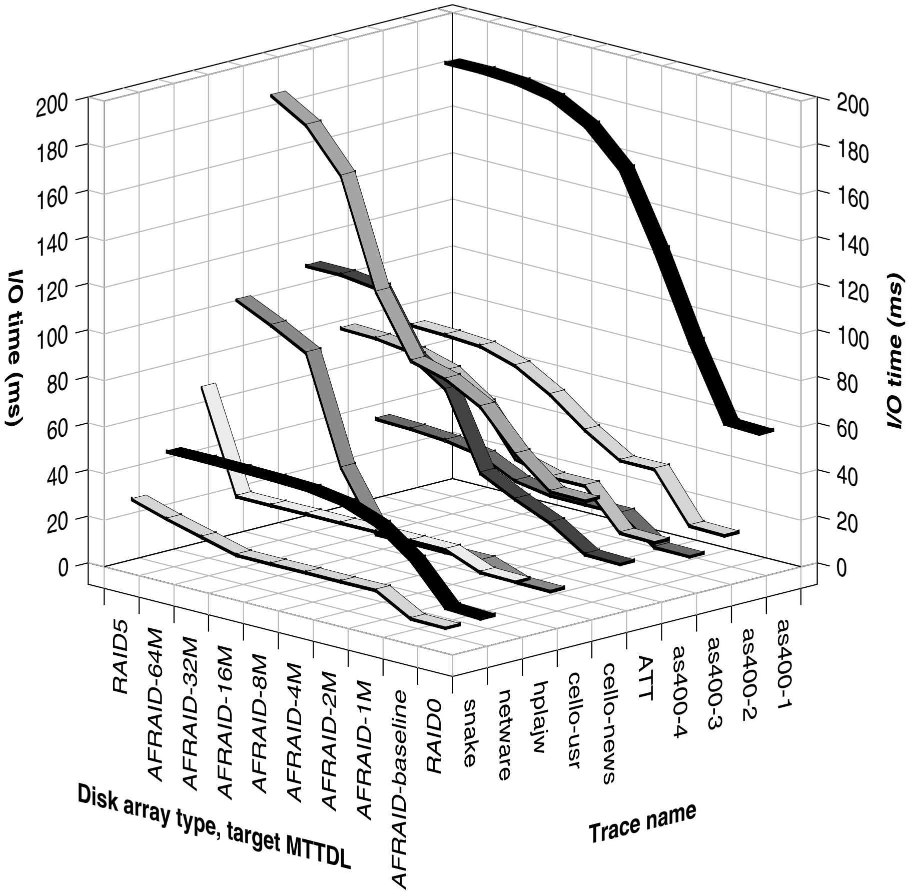

Figure 4: performance of RAID 0, RAID 5 and AFRAID under different workloads and policies.

which is dominated by support-component effects. Consequently, AFRAID and RAID 5 have essentially identical MDLRs. The MDLR$_{\text{unprotected}}$ drops to less than 0.1 bytes/hour if any of the MTTDL-X policies are used.

Because the MDLR values for AFRAID are so close to those for RAID 5, the more interesting comparison is between the MTTDL values. The first thing to note is that even the baseline AFRAID design is uniformly better than an unprotected disk array. It delivers a geometric mean MTTDL 4.3 times better than RAID 0, and is only a factor of 1.8 worse than pure RAID 5.

The MTTDL-X policy can bring the overall AFRAID MTTDL as close to RAID 5 as desired. Even the simple implementation of this policy that we used proved highly effective: the disk-related MTTDL was never more than 5% below its target, and usually far exceeded it.

As with MDLR, the dominant factor in overall MTTDL comes from the support components, which limit overall MTTDL to 2 million hours for all but the baseline AFRAID with the busiest workloads.

Thus, AFRAID is living up to the second part of its promise: availability comparable to RAID 5.

4.4. How changing availability affects performance

Figure 3 indicates just how little of the availability of a RAID 5 is relinquished by AFRAID in order to obtain better performance. The graph indicates relative performance and availability (MTTDL) by comparison to RAID 5 (the top left data point); it uses the geometric mean of the results obtained from all of our workloads. As the target MTTDL-X value is reduced (points further to the right), performance increases rapidly, while availability drops off much more slowly. For example, AFRAID offers 42% better performance for only 10% less availability, and 97% better for 23% less. By the time pure AFRAID is reached at the bottom right of the graph, performance is 4.1 times better than RAID 5, at a cost of less than half its availability.

Thus, a great deal of performance improvement can be had for a small reduction in data availability.

Figure 4 shows how performance varies with the parity-update policy for each of the traces that we studied. This figure highlights what AFRAID is all about: providing a choice between more performance or more availability.

The tradeoff between performance and availability is directly related to the characteristics of the workload. For instance, the highly bursty workloads such as snake, hplajw, and cello-usr show relatively little change in mean I/O time as availability is increased by choice of more conservative MTTDL-X policies. This is because the workloads have enough idle time to update unredundant stripes and therefore the amount of unprotected time usually stays low; in turn, this means that there is little need to revert to RAID 5 mode. In workloads with fewer idle periods and more write traffic, such as AS400-1 and ATT, there is a smooth decline in mean I/O time as MTTDL is increased across the entire range between RAID 5 and pure AFRAID.

This adaptability is one of the key features of AFRAID. Once a desired level of availability has been specified, an AFRAID array will translate any unneeded redundancy into performance. A typical bursty workload will show performance close to that of an unprotected RAID 0 disk array, while even the most highly utilized workload will deliver performance no worse than a RAID 5.

The net result is that AFRAID lives up to the last part of its promise: it offers a smooth trade-off between performance and protection that a regular RAID cannot.

5. Refinements of the AFRAID ideas

This section suggests some further applications and refinements of the AFRAID idea.

An array could begin in a "conservative" RAID 5 mode, and automatically switch into AFRAID behavior once it had determined that the I/O patterns included sufficient idle time to keep the redundancy deficit below some bound. This would be a more conservative scheme than the one we used in the MTTDL-X policy, which took the opposite approach, switching into RAID 5 when it felt that its target could not be achieved.

Stripe-aligned subsets of an AFRAID's storage space could be permanently flagged with different redundancy properties, from full RAID 5 redundancy-preservation to zero-redundancy RAID 0-style storage. Data could then be mapped to portions of the array that provided different redundancy guarantees, allowing fine-tuning of the array's availability properties according to user-specified goals [Wilkes91]. The host could then actively request that a set of stripes be made redundant, analogous to the traditional database COMMIT operation.

The units of parity-reconstruction can have a smaller "height" than the stripes used for data layout if more marker memory can be provided. For example, if M memory bits can be afforded per stripe, then parity computations will still be efficient for small writes that update only $1/M$ of a stripe unit.

A RAID 6 array keeps two parity blocks for each stripe, and thus pays an even higher penalty for doing small updates than does RAID 5. The AFRAID technique could be combined with the RAID 6 parity scheme to delay either or both parity-block updates: if only one was deferred, partial redundancy protection would be available immediately, and full redundancy once the parity-rebuild happened for the other parity block.

6. Conclusions

The main AFRAID idea is the notion of allowing deliberate, controlled, temporary non-redundancy in a disk array in order to get significantly better performance. Because real-life workloads are very often bursty, these performance gains can be achieved with a minimally increased chance of data loss—and indeed, there may be less exposure to data loss than existing single-point-of-failure solutions such as single-copy volatile or NVRAM caches. AFRAID also offers a choice that has not been possible before: that of selecting just how much availability is wanted in a particular situation.

The AFRAID design appears to be highly appropriate for workloads that have even moderate amounts of idle time between bursts of activity. Like other RAID designs, there are some workloads and applications for which it is not particularly well suited. For example, we would not advocate AFRAID for the cases where data must be protected at all costs, but it does offer a very good solution for the majority of people who want something between completely unprotected data and a fully-redundant, high-end disk array with its performance, purchase, and configuration costs.

In particular, we believe AFRAID is an appropriate design for low-load environments where latency is important, such as systems with a small number of interactive users. We hypothesize that these applications are also the ones least likely to benefit from the full availability improvements of RAID 5.

What did we learn as a result of this study? In addition to the performance and availability results we have described already, a few lessons stand out:

- Throughout this paper we have been attempting to reinforce a larger point that deserves more attention in system design: there is little value in bolstering the fault-tolerance of a single component to heroic levels if the rest of the system is less reliable. We call this the *end-to-end availability* argument, by analogy with [Saltzer81]. Making simplifying assumptions about end-to-end availability (for example, that complete data redundancy in the disk layer of an array is sacrosanct, or that NVRAM storage never fails) prevents taking advantage of performance opportunities like AFRAID.
- Real-life workloads really are bursty (we've been saying this for a while, but it bears repeating).

- Although the amount of unprotected data in the array is a function of the workload, there are several algorithms for bounding it, at the cost of some of the performance gains from pure AFRAID. Unbounded AFRAID and pure RAID 5 are simply different points on a continuum of allowed parity lag—and our design allows a user to choose where on this scale they would like their array to be.
- Thinking of different availability solutions in terms of data-loss-rate proved a useful way to unify a number of effects.

Finally, just because an idea has a strange acronym doesn't mean you should be worried by it:

> **"Always do what you are afraid to do."**
> *- Ralph Waldo Emerson*

Acknowledgments

Richard Golding, Chris Ruemmler, Carl Staelin, Tim Sullivan, and Bruce Worthington built much of the experimental infrastructure on which this work was based. Denny Georg funded our pursuit of an oddball idea with commendable(?) alacrity. Terri Watson provided the transportation to Orcas Island's Mount Constitution in which the idea took shape, and gave us invaluable feedback on drafts of this paper. Bruce McNutt provided us with the IBM AS400 traces. Our USENIX paper shepherd, Miche Baker-Harvey, also provided many useful comments.

References

[Baker92b] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20**(special issue):10–22, October 1992.

[Best95] Best Power Technology Incorporated, Necedah, WI. *Product catalog*, September 1995.

[Blaum94] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: an optimal scheme for tolerating double disk failures in RAID Architectures. *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL). Published as *Computer Architecture News*, **22**(2):245–54, 18–21 April 1994.

[Clegg86] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX operating system on HP Precision Architecture computers. *Hewlett-Packard Journal*, **37**(12):4–22, December 1986.

[Cormen93] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. *Proceedings of the 1993 DAGS/PC Symposium*, (Hanover, NH), pages 64–74, Dartmouth Institute for Advanced Graduate Studies, June 1993.

[Dallas94] Dallas Semiconductor, Dallas, TX. *Using nonvolatile static RAMs*, Application note 63, September 1994.

[Dallas95] Dallas Semiconductor, Dallas, TX. *DS1250Y/AB 4096K nonvolatile SRAM: data sheet*, October 1995.

[Gibson93] Garth A. Gibson and David A. Patterson. Designing disk arrays for high data reliability. *Journal Parallel and Distributed Computing*, **17**(1–2):4–27. Academic Press, Incorporated, January/February 1993.

[Golding95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Proceedings of Winter USENIX 1995 Technical Conference* (New Orleans, LA), pages 201–12. Usenix Association, Berkeley, CA, 16–20 January 1995.

[Gray90] Jim Gray. *A census of Tandem system availability between 1985 and 1990*. Technical Report 90.1. Tandem Computers Incorporated, September 1990.

[Gray90c] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disc arrays: low-cost reliable storage with acceptable throughput. *Proceedings of 16th International Conference on Very Large Data Bases* (Brisbane, Australia), pages 148–59, 13–16 August 1990.

[Gray91a] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, **24**(9):39–48, September 1991.

[Hartman95] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, **13**(3):274–310, August 1995.

[Holland92] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA). Published as *Computer Architecture News*, **20**(special issue):23–35, 12–15 October 1992.

[HPC3324] Hewlett-Packard Company, Boise, Idaho. *HP C3324/C3724, HP C3325/C3725 3.5-inch SCSI-2 disk drives: technical reference manual*, Part number 5963-0277, edition 3, February 1995.

[IBMpfa95] Predictive Failure Analysis: advanced condition monitoring. International Business Machines Corporation, World-wide web page http://www.almaden.ibm.com/storage/oem/tech/predfail.htm, 21 August 1995.

[Lin90c] T.-T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, **39**(4):419–32, October 1990.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[McKusick90] Marshall Kirk McKusick, Michael J. Karels, and Keith Bostic. A pageable memory based filesystem. *UKUUG Summer 1990* (London), pages 109–15. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.

[Menon89] Jai Menon and Jim Kasson. *Methods for improved update performance of disk arrays*. Technical report, rJ 6928 (66034). IBM Almaden Rescarch Center, San Jose, CA, 13 July 1989. Declassified 21 Nov. 1990.

[Menon93] Jai Menon and Jim Courtney. The architecture of a fault-tolerant cached RAID controller. *Proceedings of 20th International Symposium on Computer Architecture* (San Diego, CA), pages 76–86, 16–19 May 1993.

[Moran90] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon. Breaking through the NFS performance barrier. *Proceedings of EUUG Spring 1990* (Munich, Germany), pages 199–206, 23–27 April 1990.

[Muntz90] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. *Proceedings of 16th International Conference on Very Large Data Bases* (Brisbane, Australia), pages 162–73, 13–16 August 1990.

[Neary91] Annette M. Neary. MM-1350 reliability prediction. Micro Memory Incorporated, Chatsworth, CA, 15th March 1991. Personal communication from Dennis Doe, 23rd Oct. 1995.

[Ohta90] Masataka Ohta and Hiroshi Tezuka. A fast /tmp file system by delay mount option. *1990 Summer USENIX Technical Conference* (Anaheim, California, June 1990), pages 145–50. USENIX, June 1990.

[Ousterhout85a] John K. Ousterhout, HervéDa Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of SIGMOD.* (Chicago, Illinois), 1–3 June 1988.

[Reddy91] A. L. Narasimha Reddy and P. Banerjee. Gracefully degradable disk arrays. *Proceedings of FTCS-21*, pages 401–8. Institute of Electrical and Electronics Engineers, June 1991.

[Ritchie84a] D. M. Ritchie. The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal*, **63**(8, part 2):1577–93, October 1984.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[Ruemmler93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 405–20, 25–29 January 1993.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

[Saltzer81] J. H. Saltzer. End-to-end arguments in system design. *Proceedings of 2nd International Symposium on Operating Systems* (Paris), April 1981.

[Schulze89] Martin Schulze, Garth Gibson, Randy Katz, and David Patterson. How reliable is a RAID? *Spring COMPCON'89* (San Francisco), pages 118–23. IEEE, March 1989.

[Stodolsky93] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging: overcoming the small write problem in redundant disk arrays. *Proceedings of 20th International Symposium on Computer Architecture* (San Diego, CA), pages 64–75, 16–19 May 1993.

[Wilkes91] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review*, **25**(1):56–9, January 1991.

[Wilkes95] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system technology. *Proc 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, **29**(4), 3–6 December 1995.

[Worthington94a] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN), 16–20 May 1994.

Author information

John Wilkes graduated with degrees in Physics (BA 1978, MA 1980), and a Diploma (1979) and PhD (1984) in Computer Science from the University of Cambridge. He has worked since 1982 as a researcher and project manager at Hewlett-Packard Laboratories. His main research interest is in fast, highly available, distributed storage systems, although he also dabbles in network architectures (the Hamlyn sender-based message model), OS design (most recently in the Brevix project), harassing PhD students at random universities whenever he gets the chance, and learning about early Renaissance art and architecture.

Stefan Savage graduated with a degree in Applied History (BS 1991) from Carnegie Mellon University. He worked there for three years developing real-time operating systems and then left for Seattle, where he learned to like coffee and became a graduate student at the University of Washington. His current research interests are in safe extensible operating systems (such as SPIN, which has been occupying most of his time for the last two years) although he also finds time to be harassed by visiting researchers from random industrial research labs. He prefers art with an explanation.

The authors can be reached by electronic mail at savage@cs.washington.edu and wilkes@hpl.hp.com.