The HP AutoRAID hierarchical storage system

John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan
Hewlett-Packard Laboratories, Palo Alto, CA
{wilkes,golding,staelin,sullivan}@hpl.hp.com

Abstract

Configuring redundant disk arrays is a black art. To properly configure an array, a system administrator must understand the details of both the array and the workload it will support; incorrect understanding of either, or changes in the workload over time, can lead to poor performance.

We present a solution to this problem: a two-level storage hierarchy implemented inside a single disk-array controller. In the upper level of this hierarchy, two copies of active data are stored to provide full redundancy and excellent performance. In the lower level, RAID 5 *parity protection is used to provide excellent storage cost for inactive data, at somewhat lower performance.*

The technology we describe in this paper, known as HP AutoRAID*, automatically and transparently manages migration of data blocks between these two levels as access patterns change. The result is a fully-redundant storage system that is extremely easy to use, suitable for a wide variety of workloads, largely insensitive to dynamic workload changes, and that performs much better than disk arrays with comparable numbers of spindles and much larger amounts of front-end* RAM *cache. Because the implementation of the* HP AutoRAID *technology is almost entirely in embedded software, the additional hardware cost for these benefits is very small.*

We describe the HP AutoRAID *technology in detail, and provide performance data for an embodiment of it in a prototype storage array, together with the results of simulation studies used to choose algorithms used in the array.*

1 Introduction

Modern businesses and an increasing number of individuals depend on the information stored in the computer systems they use. Even though modern disk drives have mean-time-to-failure (MTTF) values measured in hundreds of years, storage needs have increased at an enormous rate, and a sufficiently-large collection of such devices can still experience inconveniently frequent failures. Worse, such failures can be extremely costly to repair: it may take hours, or even days, to completely reload a large storage system from backup tapes, and this can result in very costly downtime for a business that relies on its computer systems being continuously on-line.

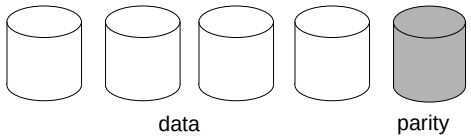
For small numbers of disks, the preferred method to provide fault protection is to duplicate (*mirror*) data on two disks with independent failure modes. This solution is simple, and it performs well.

However, once the total number of disks gets large, it becomes more cost-effective to employ an array controller that uses some form of partial redundancy (such as parity) to protect the data it stores. Such RAIDs (for Redundant Arrays of Independent Disks) were first described in the early 1980s [Lawlor81, Park86], and popularized by the work of a group at UC Berkeley [Patterson88, Patterson89]. By storing only partial redundancy for the data, the incremental cost of the desired high availability is reduced to as little as 1/N of the total storage-capacity cost (where N is the number of disks in the array), plus the cost of the array controller itself.

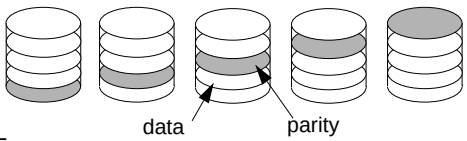
The UC Berkeley RAID terminology has a number of different *RAID levels*, each one representing a different amount of redundancy and a placement rule for the redundant data. Most disk array products implement RAID level 3 or 5. In RAID level 3, host data blocks are bit- or byte-interleaved across a set of data disks, and parity is stored on a dedicated data disk (see Figure 1). In RAID level 5, host data blocks are block-interleaved across the disks, and the disk on which the parity block is stored rotates in round-robin fashion for different stripes. Both hardware and software RAID products are available from many vendors.

Unfortunately, current *RAID arrays* are often difficult to use [Chen93]: the different RAID levels have different performance characteristics, and perform well only for a relatively narrow range of workloads. To accommodate this, RAID systems typically offer a great many configuration parameters: data- and parity-layout choice, stripe depth, stripe width, cache sizes and write-back policies, etc. Setting these correctly is difficult, and requires knowledge of workload characteristics that most people are unable (and unwilling) to acquire. As a result, setting up a RAID array is often a daunting task, that requires skilled, expensive people and—in too many cases—a painful process of trial and error.

Making the wrong choice has two costs: the resulting system may perform poorly; and changing from one layout to another almost inevitably requires copying data off to a second device,



a. RAID 3.



b. RAID 5.

Figure 1. Data and parity layout for two different RAID levels.

reformatting the array, and then reloading it. Each step of this process can take hours; it is also an opportunity for inadvertent data loss through operator error—one of the commonest source of problems in modern computer systems [Gray90].

Adding capacity to an existing array is essentially the same problem: taking full advantage of a new disk usually requires a reformat and data reload.

Since RAID 5 arrays suffer reduced performance in "degraded mode"—when one of the drives has failed—many include a provision for one or more spare disks that can be pressed into service as soon as an active disk fails. This allows redundancy reconstruction to commence immediately, thereby reducing the window of vulnerability to data loss from a second device failure and also minimizing the duration of the performance degradation. In the normal case, however, these spare disks are not used, and contribute nothing to the performance of the system. (There's also the secondary problem of being convinced that a spare disk is in fact still working: because it is idle, the array controller may not find out that it has failed until it is needed—by which time it is too late.)

1.1 The solution: a managed storage hierarchy

Fortunately, there's a solution to these problems for a great many applications of disk arrays: a redundancy-level storage hierarchy. The basic idea is to combine the performance advantages of mirroring with the cost-capacity benefits of RAID 5 by mirroring active data and storing relatively inactive data (or data that are just read, not written) in RAID 5.

To make this work, only part of the data must be active (else the cost-performance would reduce to that of mirrored data), and the active subset must change relatively slowly over time (to allow the array to do useful work, rather than just move data between the two levels). Fortunately, studies on I/O access patterns, disk shuffling and file-system restructuring have shown that these conditions are often met in practice [Akyurek93, Deshpande88, Floyd89, Geist94, Majumdar84, McDonald89, McNutt94, Ruemmler91, Ruemmler93, Smith81].

Such a storage hierarchy could be implemented in a number of different ways:

- *Manually*, by the system administrator. (This is how large mainframes have been run for decades. [Gelb89] discusses a slightly refined version of this basic idea.) The advantage of this approach is that human intelligence can be brought to bear on the problem—and perhaps globally-better solutions can be developed, using knowledge that is simply not available to the lower levels of the I/O and operating systems. However, it is obviously error-prone (the wrong choices can be made, and mistakes can be made in moving data from one level to another); it cannot adapt to rapidly-changing access patterns; it requires highly skilled people; and it does not allow new resources (such as disk drives) to be added to the system easily.
- *In the file system*, perhaps on a per-file basis. This might well be the best possible place in terms of there being a good balance of knowledge (the file system can track access patterns on a per-file basis) and implementation freedom. Unfortunately, there are many file system implementations in customers' hands, so deployment is a major problem.
- *In a smart array controller*, behind a block-level device interface such as the Small Systems Computer Interface (SCSI) standard [SCSI91]. Although this level has the disadvantage that knowledge about files has been lost, it has the enormous compensating advantage of being easily deployable—strict adherence to the standard means that an array using this approach can look just like a regular disk array, or even just a set of plain disk drives. As we will show, the performance that can be attained by operating at this level is outstanding in almost all cases.

Not surprisingly, we are describing an array-controller-based solution here. We use the name "HP AutoRAID" to refer both to the collection of technology developed to make this possible, and its embodiment in an array controller.

1.2 Summary of the features of HP AutoRAID

We can summarize the features of HP AutoRAID as follows:

Mapping. Metadata are maintained to map host block addresses to physical locations, allowing transparent migration of individual blocks.

Mirroring. Write-active data are mirrored for best absolute performance and to provide single-disk failure redundancy.

RAID 5. Write-inactive data are stored in RAID 5 for good cost-capacity while retaining single-disk failure redundancy.

Adaptation to changes in the amount of data stored. Space is allocated to mirrored storage until there is more data than can be stored in the array this way. When this happens, storage space is automatically allocated to the RAID 5 storage class, and data migrated down into it. Since this is a more compact data representation, more data can now be stored in the array. This re-apportionment is allowed to proceed until the capacity of the mirrored storage has shrunk to about 10% of the total usable space. (The exact number is a policy choice made by the implementors of the HP AutoRAID firmware to maintain good performance.) Space is apportioned in coarse-granularity units (1MB in the prototype).

Adaptation to workload changes. As the active set of data changes, newly-active data are promoted to mirrored storage, and relatively inactive data are demoted to RAID 5 in order to keep the amount of mirrored data roughly constant. With care, this can be done in the background, and need not impact the performance of the array. This movement occurs completely automatically, in relatively fine granularity units (64KB in the prototype).

Hot-pluggable disks, fans, power supplies, and controllers. These allow a failed component to be removed and a new one inserted while the system continues to operate. Although these are relatively commonplace features in higher-end disk arrays, they are important in enabling the next three features.

On-line storage capacity expansion. A disk can be added to the array at any time, up to the maximum allowed by the physical packaging (currently 12 disks in the prototype). The system automatically takes advantage of the additional space by allocating mirrored storage. As time and the workload permits, the active data will be rebalanced across the available drives to even out the workload between the newcomer and the previous set—thereby getting maximum performance from the system.

Easy upgrade to new disks. Unlike conventional arrays, the disks do not all need to have the same capacity. This has two advantages: when a new drive is added, it can be purchased at the optimal capacity/cost/performance point, without regard to prior purchases. In addition, the automatic data reconstruction and rebalancing facilities can be used to completely upgrade an array to a new disk-capacity point by simply removing each old disk, inserting a replacement disk, and then waiting for the reconstruction to complete. (To eliminate the reconstruction, data could first be "drained" from the disk being replaced if there is sufficient spare capacity in the system.)

Controller fail-over. A single array can have two controllers, each capable of running the entire subsystem. On failure of the primary,

the operations are rolled over to the other. (A future implementation could allow concurrently active controllers.) A failed controller can be replaced while the system is active.

Active hot spare. Thanks to the way in which data are allocated to the disks (more on this below), the spare space needed to perform a reconstruction can be spread across all of the disks, and used for storing mirrored data. This means that the disk spindle that would have been idle in a regular RAID array can contribute to the normal operation of an HP AutoRAID array, thereby improving its performance.

If a disk fails, mirrored data are demoted to RAID 5 to provide the space to reconstruct the desired redundancy. Once this has happened, a second disk failure can be tolerated—and so on, until the physical capacity is entirely filled with data in the RAID 5 storage class.

Simple administration and setup. The array presents one or more logical units (LUNs in SCSI terminology) to the host. Creating a new LUN is a trivial matter from the front panel: it takes about 10 seconds to go through the menus, select a size, and confirm the request. Since the array does not need to be formatted in the traditional sense, the creation of the LUN doesn't require a pass over all the newly-allocated space to zero it and initialize its parity. (This operation can take hours in a regular array.) Instead, all that is needed is for the controller's data structures to be updated.

Log-structured RAID 5 writes. A well-known problem of RAID 5 disk arrays is the so-called small-write problem. Doing an update-in-place of part of a stripe takes 4 I/Os: old data and parity have to be read, new parity calculated, and then new data and new parity written back. HP AutoRAID avoids this overhead (in most cases; see section 2.3.2) by writing to its RAID 5 storage in a log-structured fashion: that is, only empty areas of disk are written. (The indirection mechanism used to find whether data are mirrored or in RAID 5 provides the empty/full information for free.)

1.3 Related work

Many papers have been published on RAID reliability, performance, and on design variations for parity placement and recovery schemes (see [Chen94] for an annotated bibliography). The HP AutoRAID work builds on many of these studies: we concentrate here on the architectural issues of using multiple RAID levels (specifically 1 and 5) in a single array controller.

Storage Technology Corporation's Iceberg [Ewing93, STK95] uses a similar indirection scheme to map logical IBM mainframe disks (count-key-data format) onto an array of 5.25" SCSI disk drives [Art Rudeseal, private communication, Nov. 1994]. IceBerg has to handle variable-sized records; HP AutoRAID has a SCSI interface, and can handle the indirection using fixed-size blocks. The emphasis in the IceBerg project seems to have been on achieving extraordinarily high levels of availability; the emphasis in HP AutoRAID has been on performance once the single-component failure model of regular RAID arrays had been achieved. IceBerg does not include multiple RAID storage levels: it simply uses a single level modified RAID 6 storage class [Dunphy91, Ewing93].

A team at IBM Almaden has done extensive work in improving RAID array controller performance and reliability, and several of their ideas have seen application in IBM mainframe storage controllers. Their floating parity scheme [Menon89, Menon92] uses an indirection table to allow parity data to be written in a nearby slot, not necessarily its original location. This can help to reduce the small-write penalty of RAID 5 arrays. Their distributed sparing concept [Menon92a] spreads the spare space across all the disks in the array, allowing all the spindles to be used to hold data. HP AutoRAID goes further than either of these: it allows both data and parity to be relocated, and it uses the distributed spare capacity to increase the fraction of data held in mirrored form, thereby improving performance still further. Some of the schemes described in [Menon93] are also used in the dual-controller version of the HP AutoRAID array to handle controller failures.

The Loge disk drive controller [English92], and its follow-ons Mime [Chao92] and Logical Disk [deJonge93], all used a scheme of keeping an indirection table to fixed-sized blocks held on secondary storage. None of these supported multiple storage levels, and none were targeted at RAID arrays. Work on an Extended Function Controller at HP's disk divisions in the 1980s looked at several of these issues, but awaited development of suitable controller technologies to make the approach adopted in HP AutoRAID cost effective.

The log-structured writing scheme used in HP AutoRAID owes an intellectual debt to the body of work on log-structured file systems (LFS) [Carson92, Ousterhout89, Rosenblum92, Seltzer93, Seltzer95], and cleaning (garbage-collection) policies for them [McNutt94, Blackwell95].

There is a large literature on hierarchical storage systems and the many commercial products in this domain (for example [Chen73, Cohen89, DEC93, Deshpande88, Epoch88, Gelb89, Henderson89, Katz91, Miller91, Misra81, Sienknecht94, Smith81], together with much of the proceedings of the IEEE Symposia on Mass Storage Systems). Most of this work has been concerned with wider performance disparities between the levels than exist in HP AutoRAID. For example, they often use disk and robotic tertiary storage (tape or magneto-optical disk) as the two levels.

Several hierarchical storage systems have used front-end disks to act as a cache for data on tertiary storage. In HP AutoRAID, however, the mirrored storage is not a cache: instead data move between the storage classes, residing in precisely one of them at a time.

The Highlight system [Kohl93] extended LFS to two-level storage hierarchies (disk and tape) and also used fixed-size segments. Highlight's segments were around 1MB in size, however, and therefore were much better suited for tertiary-storage mappings than for two secondary-storage levels.

Schemes in which inactive data are compressed [Burrows92, Cate90, Taunton91] exhibit some similarities to the storage-hierarchy component of HP AutoRAID, but operate at the file system level rather than at the block-based device interface.

Finally, like most modern array controllers, HP AutoRAID takes advantage of the kind of optimizations noted in [Baker91, Ruemmler93] that become possible with non-volatile memory.

1.4 Roadmap to remainder of paper

The remainder of the paper is organized as follows. We begin with an overview of the technology: how an HP AutoRAID array controller works. Next come two sets of performance studies. The first is a set of measurements of a laboratory prototype; the second a set of simulation studies used to evaluate algorithm choices for HP AutoRAID. Finally, we summarize what we have learned from this project and identify a few areas for possible future work.

2 The technology

This section of the paper introduces the basic technologies used in HP AutoRAID. It starts with an overview of the hardware, then discusses the layout of data on the disks of the array, including the structures used for mapping data to their locations on disk. This is followed by brief descriptions of normal read and write operations to illustrate the flow of data through the system, and then by a series

of operations that are (usually) performed in the background, to ensure that the performance of the system remains high over long periods of time.

2.1 The HP AutoRAID array controller hardware

As far as its hardware goes, an HP AutoRAID array is fundamentally similar to a regular RAID array. That is, it has a set of disks, managed by an intelligent controller that incorporates a microprocessor, mechanisms for calculating parity, caches for staging data (some of which are non-volatile), a connection to one or more host computers, and appropriate speed-matching buffers. Figure 2 is an overview of this hardware.

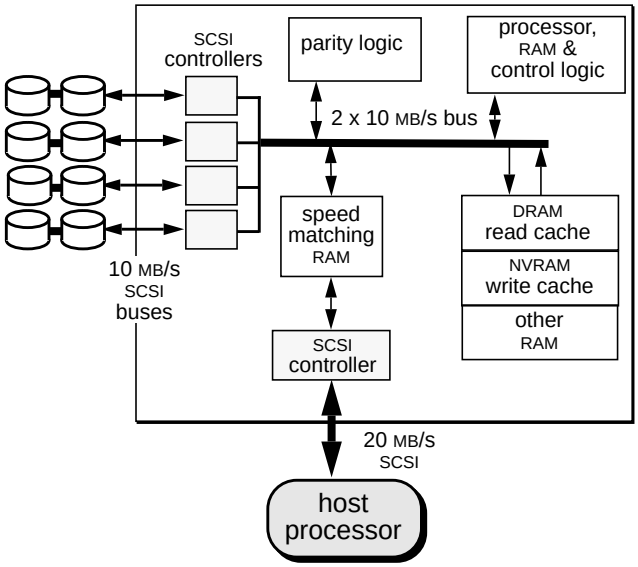


Figure 2. Overview of HP AutoRAID hardware.

The hardware prototype for which we provide performance data uses four back-end SCSI buses to connect to its disks, and a fast-wide SCSI bus for its front-end host connection. Many other alternatives exist for packaging this technology, but are outside the scope of this paper.

The array presents one or more SCSI logical units (LUNs) to its hosts. Each of these is treated as a virtual device inside the array controller: their storage is freely intermingled. A LUN's size may be increased at any time (subject to capacity constraints). Not every block in a LUN must contain valid data—if nothing has been stored at an address, the array controller need not allocate any physical space to it.

2.2 Data layout

Much of the intelligence in an HP AutoRAID controller is devoted to managing data placement on the disks. A two-level allocation scheme is used.

2.2.1 Physical data layout: PEGs, PEXes, and segments

First, the data space on the disks is broken up into large-granularity objects called Physical EXtents (PEXes), as shown in Figure 3. PEXes are typically 1MB in size. Several PEXes can be combined to make a Physical Extent Group (PEG). In order to provide enough redundancy to make it usable by either the mirrored or the RAID 5 storage class, a PEG includes at least three PEXes on different disks. At any given time, a PEG may be assigned to the mirrored storage class or the RAID 5 storage class, or may be unassigned—thus, we speak of mirrored, RAID 5, and free PEGs.

PEXes are allocated to PEGs in a manner that balances the amount of data on the disks (and thereby, hopefully, the load on the disks), while retaining the redundancy guarantees (no two PEXes from one disk can be used in the same stripe, for example). Because the disks in an HP AutoRAID array can be of different sizes, this allocation process may leave uneven amounts of free space on different disks.

Segments are the units of contiguous space on a disk that are included in a stripe or mirrored pair; each PEX is divided into a set of segments. In the prototype, segments are 128KB in size. As Figure 4 shows, mirrored and RAID 5 pegs are divided into segments in exactly the same way, but the segments are logically grouped and used by the storage classes in different ways.

2.2.2 Logical data layout: RBs

The logical space provided by the array—that visible to its clients—is divided into relatively small units called *Relocation Blocks* (RBs). These are the basic units of migration in the system. When a LUN is created or is increased in size, its address space is mapped onto a set of RBs. An RB is not assigned space in a particular PEG until the host issues a write to a LUN address that maps to the RB.

In the prototype, RBs are 64KB in size. This size is a compromise over the following pressures. Decreasing the size of an RB requires more mapping information to record where the RBs have been put. It also means that a larger fraction of the time spent moving whole-RB units is spent on disk-arm seek and rotational delays. On the other hand, a larger RB may increase migration costs if only small amounts of data are being updated in each RB. We describe the relationship between RB size and performance in section 4.1.2.

Each PEG holds a predetermined number of RBs, as a function of its size and its storage class; unused RB slots are marked as "free" until they have an RB (data) allocated to them.

2.2.3 Mapping structures

A subset of the overall mapping structures are shown in Figure 5. These data structures are optimized for looking up the physical disk address of an RB, given its logical (LUN-relative) address, since that is the most common operation. In addition, data are held about access times and history; the amount of free space in each PEG (for

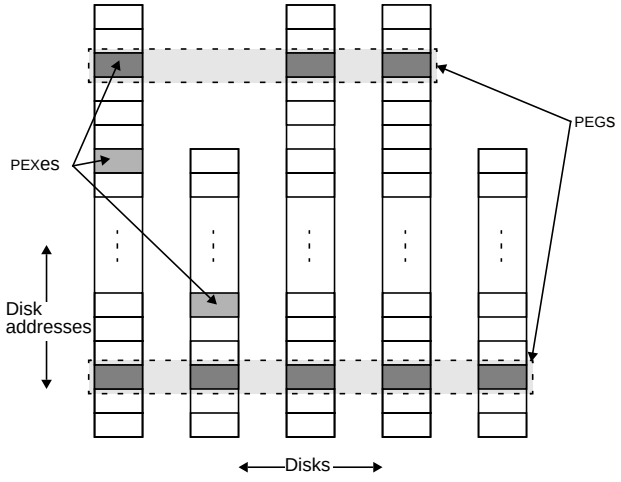


Figure 3. Mapping of PEGs and PEXes onto disks. Figure is taken from [Burkes95].

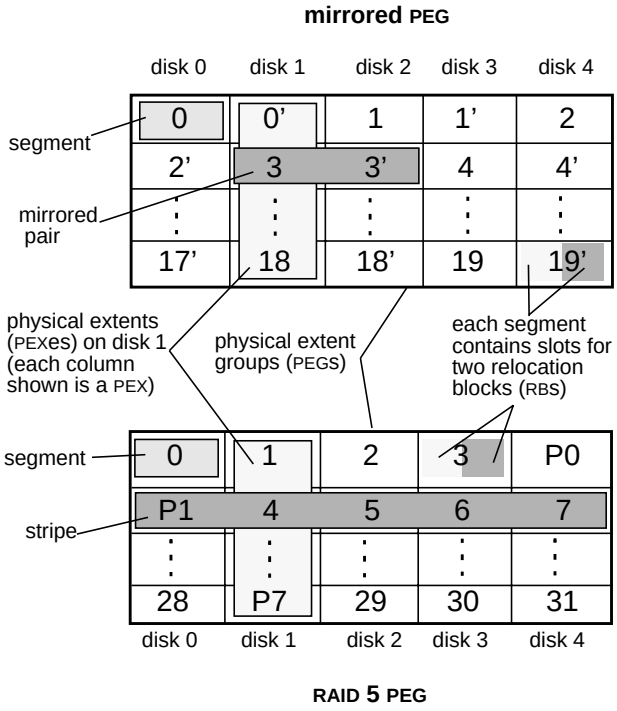


Figure 4. Layout of mirrored and RAID 5 PEGS spread out across five disks. The PEGS are physically the same (both contain a PEX from each disk), but the mirrored and RAID 5 storage classes logically group the segments differently. The RAID 5 PEG uses segments from all five disks to make its stripes; the mirrored PEG uses segments from two disks to form mirrored pairs.

cleaning and garbage-collection purposes), and various other statistics. Not shown are various back pointers that allow additional scans.

2.3 Normal operations

This section describes what happens to a host-initiated read or write operation.

To start a request, the host sends a SCSI Command Descriptor Block (CDB) to the HP AutoRAID array, where it is parsed by the controller. Up to 32 CDBs may be active at a time. An additional 2048 CDBs may be held in a FIFO queue waiting to be serviced; above this limit, requests are queued in the host. (Delays can be caused by controller resource limits, or if a request's addresses overlap with any of those already active.) Long requests are broken up into 64KB segments, which are handled sequentially: this limits the amount of controller resources a single I/O can consume, at minimal performance cost.

If the request is a Read, a test is made to see if the data being read are already in the controller's cache: either in the read cache, or (completely) in the non-volatile write cache. If the data are completely in memory, they are transferred to the host via the speed-matching buffer, and the command then completes, once various statistics have been updated. Otherwise, space is allocated in the front-end buffer cache, and one or more requests are dispatched to the back-end storage classes.

Writes are handled slightly differently, because the non-volatile front-end write buffer (*NVRAM*) allows the host to consider the request complete as soon as a copy of the data has been made in this memory. First a check is made to see if any cached data need

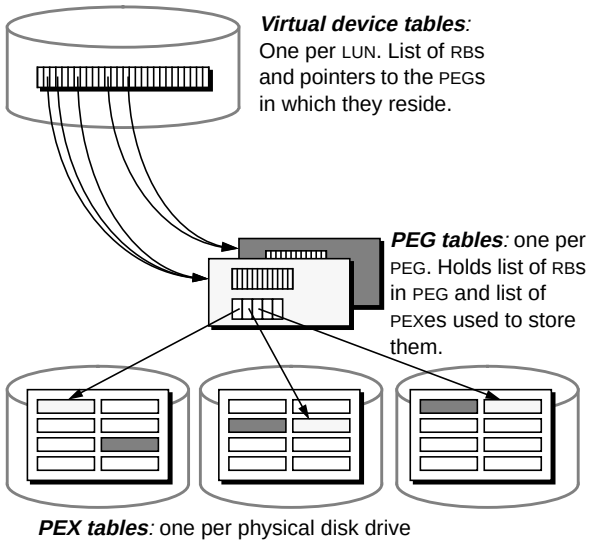


Figure 5. Structure of the tables that map from addresses in virtual volumes to PEGS, PEXes, and physical disk addresses (simplified).

invalidating, and then space is allocated in the NVRAM. (This may have to wait until space is available; in doing so, it will usually trigger a flush of existing dirty data to a back-end storage class.) The data are copied into the NVRAM, and the host told that the request is complete. Depending on the NVRAM cache-flushing policy, a back-end write may be initiated at this point. More often, nothing is done, in the hope that another subsequent write can be coalesced with this one to increase efficiency.

Flushing data to a back-end storage class simply causes a back-end write of the data if they are already in the mirrored storage class. Otherwise, it will usually trigger a promotion of the RB from RAID 5 to mirrored. (There are a few exceptions, which we will discuss later.)

This promotion is done by calling the migration code, which allocates space in the mirrored storage class and copies the RB from RAID 5. If there is no space in the mirrored storage class (because the background daemons have not had a chance to run, for example), this may in turn provoke a demotion of some mirrored data down to RAID 5. There are some tricky details involved in ensuring that this cannot in turn fail—in brief, the free-space management policies must anticipate the worst-case sequence of such events that can arise in practice.

2.3.1 Mirrored reads and writes

Reads and writes to the mirrored storage class are straightforward: a read call picks one of the copies, and issues a request to the associated disk. (More on this below.) A write call causes writes to two disks; it returns only when both copies have been updated. Note that this is a back-end write call that is issued to flush data from the NVRAM, and is not synchronous with the host write.

2.3.2 RAID 5 reads and writes

Back-end reads to the RAID 5 storage class are as simple as for the mirrored storage class: in the normal case, a read is issued to the disk that holds the data. In the recovery case, the data may have to be reconstructed from the other blocks in the same stripe. (The usual RAID 5 recovery algorithms are followed in this case, so we will not discuss the failure case more in this paper.)

Back-end RAID 5 writes are rather more complicated, however. RAID 5 storage is laid out as a log: that is, freshly-demoted RBs are appended to the end of a "current RAID 5 write PEG", overwriting virgin storage there. Such writes can be done in one of two ways: per-RB writes or batched writes. The former are simpler; the latter more efficient.

- For *per-RB writes*, as soon as an RB is ready to be written, it is flushed to disk. Doing so causes a copy of its contents to flow past the parity-calculation logic, which XORs it with its previous contents—the parity for this stripe. (To protect against power failure during this process, the prior contents of the parity block are first copied into a piece of non-volatile memory.) Once the data have been written, the parity can also be written. With this scheme, each data-RB write causes two disk writes: one for the data, one for the parity RB. This scheme has the advantage of simplicity, at the cost of slightly worse performance.
- For *batched writes*, the parity is only written once all the data-RBs in a stripe have been written, or at the end of a batch. If, at the beginning of a batched write, there is already valid data in the PEG being written, the prior contents of the parity block are copied to non-volatile memory along with the index of the PEG's highest-numbered RB that contains valid data. (The parity was calculated by XORing only RBs with indices less than or equal to this value.) RBs are then written to the data portion of the stripe until the end of the stripe is reached or the batch completes; at that point, the parity is written. (The parity has been computed by then because as each data RB was being written, the parity calculation logic incorporated it into the parity.) If the batched write fails to complete for any reason, the old parity and valid RB index that were stored in non-volatile memory are restored, returning the system to its pre-batch state, and the write is retried using the per-RB method.

  Batched writes require a bit more coordination than per-RB writes, but require only one additional parity write for each full stripe of data that is written. Most RAID 5 writes are arranged to be batched writes.

In addition to these logging write methods, the method typically used in non-logging RAID 5 implementations (*read-modify-write*) is also used in some cases. This method, which reads old data and parity, modifies them, and rewrites them to disk, is used to allow forward progress in rare cases when no PEG is available for use by the logging write processes. It is also used when it is better to update data (or *holes*; see section 2.4.1) in place in RAID 5 than to migrate an RB into mirrored storage, such as in background migrations when the array is idle.

2.4 Background operations

In addition to the foreground activities described above, the HP AutoRAID array controller executes many background activities like garbage collection and layout balancing. These background algorithms attempt to provide "slack" in the resources needed by foreground operations so that the foreground never has to trigger a synchronous version of these background tasks; such synchronous invocations can dramatically reduce performance.

The background operations are triggered when the array has been "idle" for a period of time. When an idle period is detected (using an algorithm based on current and past device activity—the array does not have to be completely devoid of activity to be declared "idle"), the array performs one set of background operations. Each subsequent (or continuation of the current) idle period triggers another set of operations.

After a long period of activity, it may take a moderate amount of time to detect that the array is idle. We hope to apply some of the results from [Golding95] to improve the idle period detection and prediction accuracy, which will in turn allow us to be more aggressive about executing the background algorithms.

2.4.1 Compaction: cleaning and hole-plugging

The mirrored storage class acquires *holes*, empty RB slots, when RBs are demoted to the RAID 5 storage class. (Updates to mirrored RBs are written in place, so they generate no holes.) These holes are added to a free list in the mirrored storage class, and may subsequently be used to contain promoted or newly-created RBs. If a new PEG is needed for the RAID 5 storage class, and no free PEXes are available, a mirrored PEG may be chosen for *cleaning*: all the data are migrated out to fill holes in other mirrored PEGs, after which the PEG can be reclaimed and reallocated to the RAID 5 storage class.

Similarly, the RAID 5 storage class acquires holes when RBs are promoted to the mirrored storage class, usually because the RBs have been updated. Because the normal RAID 5 write process uses logging, the holes cannot be reused directly; we call them *garbage*.

If the RAID 5 PEG containing the holes is almost full, the array performs *hole-plugging* garbage collection. RBs are copied from a PEG with a small number of RBs, and used to fill in the holes of an almost-full PEG. This minimizes data movement if there is a spread of fullness across the PEGs, which is often the case.

If the PEG containing the holes is almost empty and there are no other holes to be plugged, the array does PEG-*cleaning*: that is, it appends the remaining valid RBs to the current end of the RAID 5 write log, and reclaims the complete PEG as a unit.

2.4.2 Migration: moving RBs between levels

A background migration policy is run to move RBs from mirrored storage to RAID 5. This is done primarily to provide enough empty RB slots in the mirrored storage class to handle a future write burst. As [Ruemmler93] showed, such bursts are the common manner in which current UNIX[1] systems emit updates.

RBs are selected for migration by an approximate Least-Recently-Written algorithm. Migrations are performed in the background until the number of free RB slots in the mirrored storage class or free PEGs exceeds a high water mark that is chosen to allow the system to handle a burst of incoming data. This threshold can be set to provide better burst-handling at the cost of slightly lower out-of-burst performance; its value is currently fixed in the AutoRAID firmware, but it could also be determined dynamically.

2.4.3 Balancing: adjusting data layout across drives

When new drives are added to an array, they contain no data and therefore do not contribute to the system's performance. Balancing is the process of migrating PEXes between disks to equalize the amount of data stored on each disk, and thereby also the request load imposed on each disk. (Access histories could be used to balance the disk load more precisely, but this is not done in the current prototype.) Balancing is also a background activity, performed when the system has little else to do.

2.5 Workload logging

The performance delivered by a secondary storage system depends to a certain degree on the workload it is presented. This is true of HP AutoRAID, and doubly so of regular RAID arrays. Part of the uncertainty we faced while doing our performance work was the lack of a broad range of real, measured system workloads at the

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

disk I/O level that had been measured accurately enough. The number of such traces that have been gathered is relatively small; the number that are available to research groups other than those that acquired them is smaller still.

To help remedy this in the future, the HP AutoRAID array incorporates an I/O workload logging tool. When presented with a specially-formatted disk, the start and stop times of every externally-issued I/O request are recorded on it. Other events can also be recorded, if desired. The overhead of doing this is very small: the event logs are first buffered in the controller's RAM, and then written out in large blocks. The result is a faithful record of everything the particular unit was asked to do; it can be analyzed after the event, and used to drive simulation studies such as the kind we describe here.

2.6 Management tool

The product team also developed a management tool that can be used to analyze the performance of an HP AutoRAID array over a period of time. It operates off a set of internal statistics kept by the firmware in the controller, such as cache utilization, I/O times, disk utilizations, and so on. These statistics are relatively cheap to acquire and store, and yet can provide significant insight into the operation of the system. By doing an off-line analysis using a log of the values of these statistics, the tool can use a set of rule-based inferences to determine (for example) that for a particular period of high load, performance could have been improved by adding cache memory because the array controller was short of read cache.

3 HP AutoRAID performance results

A combination of prototyping and event-driven simulation was used in the development of HP AutoRAID. Most of the novel technology for HP AutoRAID is in the algorithms and policies used to manage the storage hierarchy. As a result, hardware and firmware prototypes were developed concurrently with event-driven simulations that studied design choices for algorithms, policies, and parameters to those algorithms.

The primary development team was based at the product division that designed, built, and tested the prototype hardware and firmware. They were supported by a group at HP Laboratories that built a detailed simulator of the hardware and firmware and used it to model alternative algorithm and policy choices in some depth. This organization allowed the two teams to incorporate the technology into products in the least possible time while still fully investigating alternative design choices.

In this section, we present measured results from a laboratory prototype of a controller embodying the HP AutoRAID technology. In the next, we present a set of comparative performance analyses of different algorithm and policy choices that were used to help guide the implementation of the real thing.

3.1 Experimental setup

The baseline HP AutoRAID configuration on which the data we report were measured is a 12-disk system with 16MB of controller data cache, connected to an HP 9000/897 system running release 10.0 of the HP-UX operating system [Clegg86]. The HP AutoRAID array was configured with 12 2.0GB 7200RPM Seagate ST32550 (Barracuda) disk drives.

To calibrate the HP AutoRAID results against external systems, we also include measurements taken (on the same host hardware, on the same days, with the same host configurations, number of disks, and type of disks, except as noted below) on two other disk subsystems:

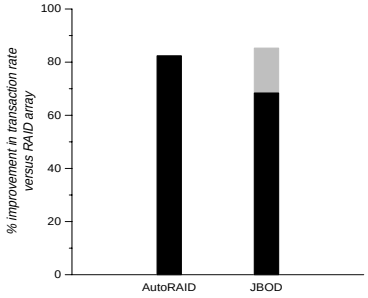


Figure 6. OLTP benchmark comparison of HP AutoRAID and non-RAID drives to a regular RAID array (percent improvement in transaction rate versus the RAID array result). Our estimate of the performance of the non-RAID drives (JBOD) using the same SCSI adapter as the other tests is shown in grey.

- A Data General CLARiiON® Series 2000 Disk-Array Storage System Deskside Model 2300 with 64MB front-end cache. (We use the term "RAID array" to refer to this system in what follows). This array was chosen because it is the recommended third-party RAID array solution for one of the primary customers of the HP AutoRAID product.
- A set of directly-connected individual disk drives, referred to here as "JBOD" (Just a Bunch Of Disks). This allows us to offer a comparison with a solution that provides no data redundancy at all. There were two configuration differences between the JBOD tests and the other tests:
  - Because no space is used for parity in JBOD, 11 disks were used rather than 12 to approximately match the amount of space available for data in the other configurations.
  - The SCSI adapter used was a single-ended card that also requires more host CPU cycles per I/O than the differential card used in the HP AutoRAID and RAID array tests. We estimate that this reduced the performance of JBOD by 10% on the OLTP test described below, but did not affect the micro-benchmarks because they were not CPU limited.

3.2 Performance results

Although we present mostly micro-benchmark results because they isolate individual performance characteristics, we begin with a macro-benchmark: running an OLTP database workload made up of medium-weight transactions against the HP AutoRAID array, the regular RAID array, and JBOD. The database used in this test was only 6.7GB, which allowed it to fit entirely in mirrored storage in the HP AutoRAID; working set sizes larger than available mirrored space are discussed below. For this benchmark, the RAID array's 12 disks were spread evenly across its 5 SCSI channels, the 64MB cache was enabled, and the default 2KB stripe-unit size was used. Figure 6 shows the result: HP AutoRAID significantly outperforms the RAID array, and has performance comparable to JBOD.

Data from the micro benchmarks are provided in Figure 7. This shows the relative performance of the two arrays for random and sequential reads and writes. (The workloads were provided by a single-process version of an internal benchmark known as DB; the working-set size for the random tests was 2GB.) We hypothesize that the poor showing of the cached RAID array on the random loads is due to the cost of searching the cache: the cache hit rate should be close to zero in these tests.

The HP AutoRAID array significantly outperforms the regular RAID array. The results shown for the RAID array are the best results obtained after trying a number of different array configurations: this was always RAID 5, 8KB cache page, cache on or off as noted.

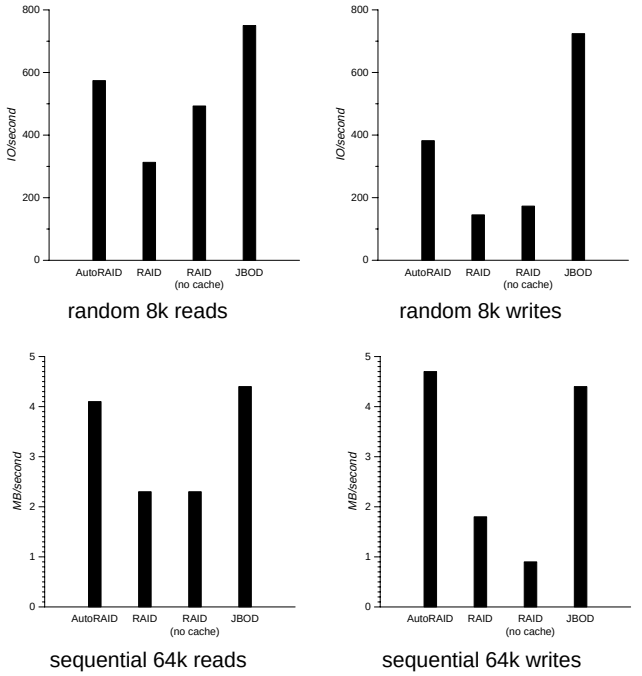


Figure 7. Micro-benchmark comparisons of HP AutoRAID, a regular RAID array, and non-RAID drives.

Results for RAID 3 were never better than the results shown. Indeed, this demonstrated the difficulties involved in properly configuring a RAID array: many parameters were adjusted (caching on or off, cache granularity, stripe depth and data layout), and no single combination performed well across the range of workloads examined.

With the working set within the size of the mirrored space performance is very good, as shown by Figure 6 and Figure 7. But if the working-set constraint is exceeded for long periods of time, it is possible to drive the HP AutoRAID array into a mode in which each update causes the target RB to be promoted up to the mirrored storage class, and a second one demoted to RAID 5. This behavior is obviously undesirable if the dominant workload pattern does not meet the working-set constraint. If the behavior occurs in practice, however, an HP AutoRAID device can be configured to avoid it by adding enough disks to keep all the active data in mirrored storage. If *all* the data were active, the cost-performance advantages of the technology would, of course, be reduced. Fortunately, it is fairly easy to predict or detect the environments that have a large working set and many updates, and to avoid them if necessary.

4 Simulation studies

The previous section provided insight into the overall, absolute performance of an HP AutoRAID, as measured on real hardware. In this section, we will illustrate several design choices that were made inside the HP AutoRAID implementation. To do so, we used trace-driven simulation.

Our simulator is built on the Pantheon[2] [Golding94, Cao94] simulation framework, which is a detailed, trace-driven simulation environment written in C++ using an enhanced version of the AT&T tasking package. Individual simulations are configured from the set of available C++ simulation objects using scripts written in the Tcl language [Ousterhout94], and configuration techniques described in [Golding94]. The disk models used in the simulation are improved versions of the detailed, calibrated models described in [Ruemmler94].

The traces used to drive the simulations are from a variety of systems, including: cello, a timesharing HP 9000 Series 800 HP-UX system; snake, an HP 9000 Series 700 HP-UX cluster file server; OLTP, an HP 9000 Series 800 HP-UX system running a database benchmark made up of medium-weight transactions (not the system described in section 3.2); a personal workstation; and a Netware server. We also used subsets of these traces, such as the /usr disk from cello, a subset of the database disks from OLTP, and the OLTP log disk. Some of them are for long time periods (up to three months), although most of our simulations used two-day subsets of the traces. Almost all contain detailed timing information to 1µs resolution. Several of them are described in considerable detail in [Ruemmler93].

We modelled the hardware of HP AutoRAID using Pantheon components (caches, buses, disks, etc.), and wrote detailed models of the basic firmware and of several alternative algorithms or policies for each of about 40 design experiments. The Pantheon simulation core comprises about 46K lines of C++ and 8K lines of Tcl, and the HP AutoRAID specific portions of the simulator added another 16K lines of C++ and 3K lines of Tcl.

Because of the complexity of the model and the number of parameters, algorithms, and policies that we were examining, it was impossible to explore all combinations of the experimental variables in a reasonable amount of time. We chose instead to organize our experiments into baseline runs and runs with one or a few related changes to the baseline. This allowed us to observe the performance effects of individual or closely-related changes, and to perform a wide range of experiments reasonably quickly. (We used a cluster of 12 workstations to perform the simulations; even so, a full run of all our experiments takes about a week of elapsed time.)

We performed additional experiments to combine individual changes that we suspected might strongly interact (either positively or negatively) and to test the aggregate effect of a set of algorithms that we were proposing to the product development team.

No hardware implementation of HP AutoRAID was available early in the simulation study, so we were initially unable to calibrate our simulator (except for the disk models). Because of the high level of detail of the simulation, however, we were confident that relative performance differences predicted by the simulator would be valid even if absolute performance numbers were not yet calibrated. We therefore used the relative performance differences we observed in simulation experiments to suggest improvements to the team implementing the prototype firmware, and these are what we present here. In turn, we updated our baseline model to correspond to the changes they made to their implementation.

Since there are far too many individual results to report here, we have chosen to describe a few that highlight some of the particular behaviors of the HP AutoRAID system. Please note that the measured data from the real hardware provide information about *absolute* performance, while the results in this section compare the *relative* performance of different simulated system configurations.

4.1 Relative performance results

4.1.1 Disk speed

Several experiments measured the sensitivity of the design to the size or performance of various components. For example, the

[2] The simulator used to be called TickerTAIP, but we have changed its name to avoid confusion with the parallel RAID array project of the same name [Cao94].

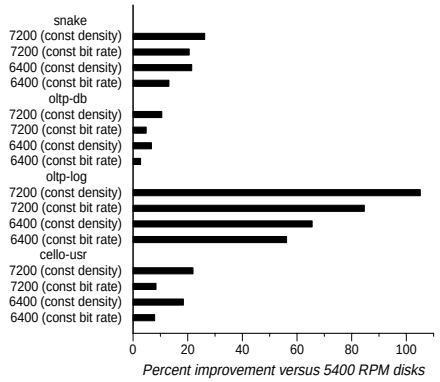


Figure 8. Effect of disk spin speed on overall performance.

system uses standard SCSI disks, so we wanted to understand the effects of buying more expensive, faster disks. The baseline disks held 2GB and spun at 5400 RPM. We evaluated four variations of this disk: spinning at 6400 RPM and 7200 RPM, keeping either the data density (bits per inch) or transfer rate (bits per second) constant. As expected, increasing the back-end disk performance generally improves overall performance, as shown in Figure 8. The results suggest that improving transfer rate is more important than improving rotational latency.

4.1.2 RB size

The standard AutoRAID system uses 64KB RBs as the basic storage unit. We looked at the effect of using smaller and larger sizes. For most of the workloads (see Figure 9) the 64KB size the best of the ones we tried: obviously the balance between seek and rotational overheads versus data movement costs is about right. (This is perhaps not too surprising: the disks we are using have track sizes of around 64KB, and transfer sizes in this range will tend to get much of the benefit from better mechanical delays.)

4.1.3 Data layout

Since the system allows blocks to be remapped, blocks that the host system has tried to lay out sequentially will often be physically discontiguous. To see how bad this could get, we compared the performance of the system when host LUN address spaces are initially laid out completely linearly on disk (as a best case) and completely randomly (as a worst case). Figure 10 shows the difference between the two layouts: there is a modest degradation in performance in the random case compared to the linear one. This

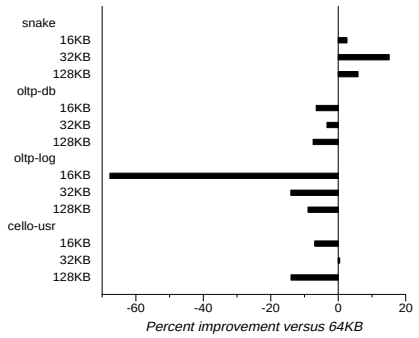


Figure 9. Effect of RB size on overall performance.

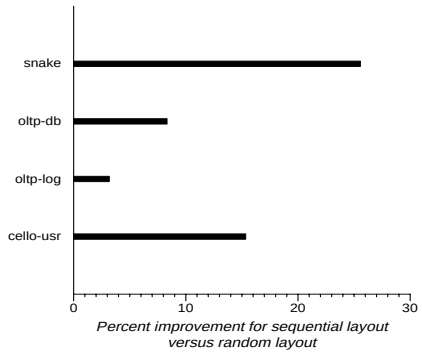


Figure 10. Sensitivity to data layout.

suggests that the choice of RB size is large enough to limit the impact of seek delays for sequential accesses.

4.1.4 Mirrored storage class read selection algorithm

When the front-end read cache misses on an RB that is stored in the mirrored storage class, the array can choose to read either of the stored copies. The baseline system selects the copy at random in an attempt to avoid making one disk a bottleneck. However, there are several other possibilities:

- strictly alternating between disks (alternate);
- attempting to keep the heads on some disks near the outer edge while keeping others near the inside (inner/outer);
- using the disk with the shortest queue (shortest queue);
- using the disk that can reach the block first, as determined by a shortest-positioning-time algorithm [Seltzer90, Jacobson91] (shortest seek).

Further, the policies can be "stacked", first using the most aggressive policy but falling back to another to break a tie. In our experiments, random is always the final fallback policy.

Figure 11 shows the results of our investigations into the possibilities. By using shortest queue as a simple load-balancing heuristic, performance is improved by an average of 3.3% over random for these workloads. Shortest seek performed 3.4% better than random, but is much more complex to implement because it requires detailed knowledge of disk head position and seek timing.

Static algorithms such as alternate and inner/outer sometimes perform better than random, but sometimes interact unfavorably with patterns in the workload and decrease system performance.

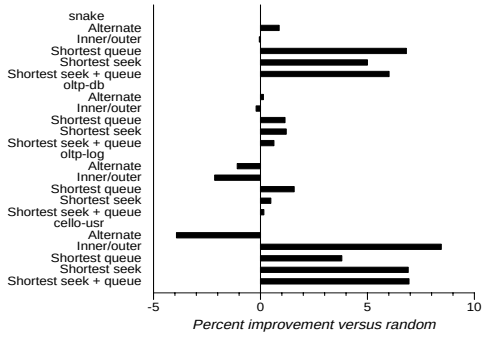


Figure 11. Effect of mirrored storage class read disk selection policy on overall performance.

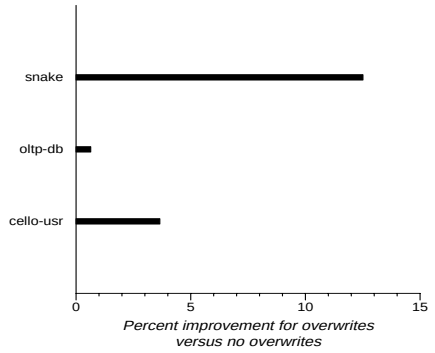


Figure 12. Effect of allowing write cache overwrites on overall performance.

We note in passing that these differences do not show up under micro-benchmarks (of the type reported in Figure 7) because the disks are typically always driven to saturation, and do not allow such effects to show through.

4.1.5 Write cache overwrites

We investigated several policy choices for managing the NVRAM write cache. The baseline system, for instance, did not allow one write operation to overwrite dirty data already in cache; instead, the second write would be blocked until the previous dirty data in the cache had been flushed to disk. As Figure 12 shows, allowing overwrites had a noticeable impact on most of the workloads. It had a huge impact on the OLTP-log workload, improving its performance by 432%! We omitted this workload from the graph for scaling reasons.

4.1.6 Hole-plugging during RB demotion

RBs are typically written to RAID 5 for one of two reasons: demotion from mirrored storage, or garbage collection. During normal operation, the system creates holes in RAID 5 by promoting RBs to the mirrored storage class. In order to keep space consumption constant, the system later demotes (other) RBs to RAID 5. In the default configuration, HP AutoRAID uses logging writes to demote RBs to RAID 5 quickly, even if the demotion is done during idle time; these demotions do not fill the holes left by the promoted RBs, giving the RAID 5 cleaner additional work. To reduce the work done by the RAID 5 cleaner, we allowed RBs demoted during idle periods to be written to RAID 5 using hole-plugging. This optimization reduced the number of RBs moved by the RAID 5 cleaner by 93% for the cello-usr workload and by 96% for snake, and improved mean I/O time for user I/Os by 8.4% and 3.2% respectively.

5 Summary

HP AutoRAID works extremely well, providing close to the performance of a non-redundant disk array across a range of workloads. At the same time, it provides full data redundancy, and can tolerate failures of any single array component.

It is very easy to use: one of the authors of this paper was delivered a system without manuals a day before a demonstration, and had it running a trial benchmark five minutes after getting it connected to his (completely unmodified) workstation. The product team has had several such experiences in demonstrating the system to potential customers.

The first product based on the technology, the HP XLR1200 Advanced Disk Array, is now available.

5.1 Principles and lessons learned

In the course of doing this work, we (re)learned several things. Some of them were obvious in hindsight (that's what hindsight is for), but others were slightly less so. Here's a short list of the second kind.

Ease of use is surprisingly important. New technology is taken up only slowly if it is difficult to use—and the performance sensitivity of traditional RAID arrays means that they suffer from this drawback. As we have shown, this need not be the case.

Dynamic adaptation to the incoming workload is a big win. It removes a considerable burden from the user, and provides much smoother degradation in performance as the workload changes over time. In fact, it is often a good idea not to ask users to specify their storage needs: most people don't know and don't care. (Even the highly-trained evaluators of the RAID array had considerable difficulty in configuring it to get maximum performance.)

The HP AutoRAID technology is not a panacea for all storage problems: there are workloads that do not suit its algorithms well, and environments where the variability in response time is unacceptable. Nonetheless, it is able to adapt to a great many of the environments that are encountered in real life, and it provides an outstanding general purpose solution to storage needs where availability matters.

The isolation barriers that result from standardized interfaces make it possible to deploy technology such as HP AutoRAID widely with little effort. The freedom to innovate provided by the SCSI command set interface is quite remarkable, given its genesis.

Software is the differentiator in the HP AutoRAID technology, not hardware. This may seem obvious, but remember that most people think of a disk array as hardware.

There were several instances where using real-life workloads gave different results than micro-benchmarks. For example, the differences we report for the mirrored storage class read algorithm were undetectable in the micro-benchmarks; because these benchmarks always saturated the physical disk mechanisms, they left no differences in queue length or seek distances to exploit. Furthermore, the burstiness that is common in real workloads, but not in micro-benchmarks, allows HP AutoRAID to do so well in adapting itself in the background—without periods of low activity, all rearrangements would have to be done in the foreground, which would contribute to lower performance and much higher performance variability.

HP AutoRAID invalidates the common mechanisms for benchmarks: as it is tested, its performance will tend to improve, so runs will not be repeatable in the normal sense.

There were several cases early on where the simulation team attempted a "fancier" resource usage model than the production system they were trying to model. (The most egregious of these was an attempt to use the same physical resources for data caching and back-end data movement.) Although this approach might perhaps have resulted in a small improvement in performance, it also resulted in horrendous deadlock problems. We re-learned that the virtues of simplicity often outweigh the cost of throwing a few additional hardware resources, such as extra memory, at a problem.

There were a few cases where our early simulations demonstrated classic convoy phenomena in the algorithms: a write that caused a migration, which slowed down writes, which bunched up and demanded a whole slew of migrations, which … Fixing these required careful attention to reserving sufficient free resources. In

addition, were reminded that it is sub-optimal to revert to a slower-than-normal write scheme when the number of outstanding writes exceeds some threshold, because this will simply exacerbate the problem. We still see proposals for new disk scheduling algorithms that have not yet understood this point.

Despite our growing range of real workload traces, we are painfully aware that we do not yet have a fully representative set. Also, traces are point measurements taken on today's systems—and so are only an approximation to the access patterns that will become prevalent tomorrow. Nonetheless, we believe strongly that measured traces exhibit properties that no synthetic stream is likely to. Despite their drawbacks, traces are much better tests of system behavior than synthetic loads or benchmarks for a controller as complex as HP AutoRAID.

5.2 Future work

What we have described in this paper is a subset of the complete HP AutoRAID design. The technology will continue to be enhanced by adding new functionality to it: given the implementation base established by the first development, this is an incremental task rather than a revolutionary one, so we hope for rapid deployment of several of these new features. In addition to the normal product-development performance tuning that takes place, two particular areas are likely to be idle-period detection and prediction, and front-end cache-management algorithms.

In addition, we have areas where we have plans to improve our processes. For example, we are pursuing better techniques for synthesizing traces with much greater fidelity to real life than is the current norm, and looking into the issues involved in replaying the current ones we have in a manner that allows us to experiment better with workload variations.

Finally, we are also considering how best to extend some of the HP AutoRAID ideas into tertiary storage systems (perhaps in the style of [Kohl93]).

Acknowledgments

We would like to thank our colleagues in HP's Storage Systems Division, who developed the product version of the HP AutoRAID controller, and were the customers for our performance and algorithmic studies. Many more people put enormous amounts of effort into making this program a success than we can possibly acknowledge directly by name; we thank them all.

Chris Ruemmler wrote the DB benchmark used for the results in section 3.3.

This paper is dedicated to the memory of our late colleague Al Kondoff, who helped establish the collaboration that produced this body of work.

References

[Akyurek93] Sedat Akyürek and Kenneth Salem. *Adaptive block rearrangement*. Technical report CS–TR–2854.1. Department of Computer Science, University of Maryland, November 1993.

[Baker91] 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.

[Blackwell95] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 277–88. Usenix Association, 16–20 January 1995.

[Burkes95] Terry Burkes, Bryan Diamond, and Doug Voigt. *Adaptive hierarchical RAID: a solution to the* RAID 5 *write problem*. Part number 5963–9151. Hewlett-Packard Storage Systems Division, Boise, ID, 6th July 1995.

[Burrows92] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. *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):2–9, October 1992.

[Cao94] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems* **12**(3):236–269, August 1994.

[Carson92] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. *USENIX Workshop on File Systems* (Ann Arbor, MI), pages 79–91, 21–22 May 1992.

[Cate90] Vince Cate. *Two levels of filesystem hierarchy on one disk*. Technical report CMU–CS–90–129. Carnegie-Mellon University, Pittsburgh, PA, May 1990.

[Chao92] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. *Mime: a high performance storage device with strong recovery guarantees*. Technical report HPL–92–44. Hewlett-Packard Laboratories, April 1992.

[Chen73] Peter Chen. Optimal file allocation in multi-level storage hierarchies. *Proceedings of National Computer Conference*, pages 277–82, 1973.

[Chen93] P. M. Chen and E. K. Lee. *Striping in a RAID level-5 disk array.* Technical report CSE–TR–181–93. University of Michigan, 1993.

[Chen94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, **26**(2):145–85, June 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.

[Cohen89] E. I. Cohen, G. M. King, and J. T. Brady. Storage hierarchies. *IBM Systems Journal*, **28**(1):62–76, 1989.

[DEC93] *POLYCENTER storage management for OpenVMS VAX systems.* Digital Equipment Corporation, November 1993. Product overview white paper.

[deJonge93] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: a new approach to improving file systems. *Proceedings of 14th ACM Symposium on Operating Systems Principles* (Asheville, NC), pages 15–28, 5–8 December 1993.

[Deshpande88] Milind B. Deshpande and Richard B. Bunt. Dynamic file management techniques. *Proceedings of 7th Institute of Electrical and Electronics Engineers Phoenix Conference on Computer and Communication* (Scottsdate, Arizona), pages 86–92, 16–18 March 1988.

[Dunphy91] Robert H. Dunphy, Jr, Robert Walsh, John H. Bowers, and George A. Rudeseal. *Disk drive memory.* United States patent 5 077 736, 31 December 1991, filed 13 Feb. 1990.

[English92] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. *Proceedings of USENIX Winter'92 Technical Conference* (San Francisco, CA), pages 237–51. Usenix, 20–24 January 1992.

[Epoch88] Mass storage: server puts optical discs on line for work stations. *Electronics*, November 1988.

[Ewing93] John Ewing. *RAID: an overview.* W I7004-A 09/93. Storage Technology Corporation, 2270 South 88th Street, Louisville, CO 80028-4358, December 1993. Available as http://www.stortek.com:80/StorageTek/raid.html.

[Floyd89] Richard A. Floyd and Carla Schlatter Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering*, **1**(2):238–47, June 1989.

[Geist94] R. Geist, R. Reynolds, and D. Suggs. Minimizing mean seek distance in mirrored disk systems by cylinder remapping. *Performance Evaluation*, **20**(1–3):97–114, May 1994.

[Gelb89] J. P. Gelb. System managed storage. *IBM Systems Journal*, **28**(1):77–103, 1989.

[Golding94] Richard Golding, Carl Staelin, Tim Sullivan, and John Wilkes. "Tcl cures 98.3%of all known simulation configuration problems" claims astonished researcher! *Proceedings of Tcl/Tk Workshop*, New Orleans, LA, June 1994. Available as Technical report HPL–CCD–94–11, Concurrent Computing Department, Hewlett-Packard Laboratories, Palo Alto, CA.

[Golding95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 201–12. Usenix Association, 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.

[Henderson89] Robert L. Henderson and Alan Poston. MSS-II and RASH: a mainframe Unix based mass storage system with a rapid access storage hierarchy file management system. *USENIX Winter 1989 Conference* (San Diego, California), pages 65–84. USENIX, January 1989.

[Jacobson91] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position.* Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991.

[Katz91] Randy H. Katz, Thomas E. Anderson, John K. Ousterhout, and David A. Patterson. *Robo-line storage: low-latency, high capacity storage systems over geographically distributed networks*. UCB/CSD 91/651. Computer Science Div., Department of Electrical Engineering and Computer Science, University of California at Berkeley, September 1991.

[Kohl93] John T. Kohl, Carl Staelin, and Michael Stonebraker. HighLight: using a log-structured file system for tertiary storage management. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 435–47, 25–29 January 1993.

[Lawlor81] F. D. Lawlor. Efficient mass storage parity recovery mechanism. *IBM Technical Discl. Bulletin*, **24**(2):986–7, July 1981.

[Majumdar84] Shikharesh Majumdar. *Locality and file referencing behaviour: principles and applications*. MSc thesis published as technical report 84–14. Department of Computer Science, University of Saskatchewan, Saskatoon, August 1984.

[McDonald89] M. Shane McDonald and Richard B. Bunt. Improving file system performance by dynamically restructuring disk space. *Proceedings of Phoenix Conference on Computers and Communication* (Scottsdale, AZ), pages 264–9. IEEE, 22–24 March 1989.

[McNutt94] Bruce McNutt. Background data movement in a log-structured disk subsystem. *IBM Journal of Research and Development*, **38**(1):47–58, 1994.

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

[Menon92] Jai Menon and Jim Kasson. Methods for improved update performance of disk arrays. *Proceedings of 25th International Conference on System Sciences* (Kauai, Hawaii), volume 1, pages 74–83, Veljko Milutinovic and Bruce Shriver, editors, 7–10 January 1992.

[Menon92a] Jai Menon and Dick Mattson. Comparison of sparing alternatives for disk arrays. *Proceedings of 19th International Symposium on Computer Architecture* (Gold Coast, Australia), pages 318–29, 19–21 May 1992.

[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.

[Miller91] Ethan L. Miller. *File migration on the Cray Y-MP at the National Center for Atmospheric Research*. UCB/CSD 91/638. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, June 1991.

[Misra81] P. N. Misra. Capacity analysis of the mass storage system. *IBM Systems Journal*, **20**(3):346–61, 1981.

[Ousterhout89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *Operating Systems Review*, **23**(1):11–27, Jan. 1989.

[Ousterhout94] John K. Ousterhout. *Tcl and the Tk toolkit*, Professional Computing series. Addison-Wesley, Reading, Mass. and London, April 1994.

[Park86] Arvin Park and K. Balasubramanian. *Providing fault tolerance in parallel secondary storage systems*. Technical report CS–TR–057–86. Department of Computer Science, Princeton University, November 1986.

[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.

[Patterson89] David A. Patterson, Peter Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). *Spring COMPCON'89* (San Francisco), pages 112–17. IEEE, March 1989.

[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.

[Ruemmler91] Chris Ruemmler and John Wilkes. *Disk shuffling*. Technical report HPL–91–156. Hewlett-Packard Laboratories, October 1991.

[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.

[SCSI91] Secretariat, Computer and Business Equipment Manufacturers Association. *Draft proposed American National Standard for information systems – Small Computer System Interface-2 (SCSI-2)*, Draft ANSI standard X3T9.2/86-109., 2 February 1991 (revision 10d).

[Seltzer90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of Winter 1990 USENIX Conference* (Washington, D.C.), pages 313–23, 22–26 January 1990.

[Seltzer93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *Proceedings of Winter 1993 USENIX* (San Diego, CA), pages 307–26, 25–29 January 1993.

[Seltzer95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Conference Proceedings of of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 249–64. Usenix Association, 16–20 January 1995.

[Sienknecht94] T. F. Sienknecht, R. J. Friedrich, J. J. Martinka, and P. M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, **20**(1–3):3–25, May 1994.

[Smith81] Alan Jay Smith. Optimization of I/O systems by cache disks and file migration: a summary. *Performance evaluation*, **1**:249–62. North-Holland, Amsterdam, 1981.

[STK95] *Iceberg 9200 disk array subsystem*. Storage Technology Corporation, 2270 South 88th Street, Louisville, Colorado 80028-4358, 9 June 1995. Product description available as http://www.stortek.com:80/StorageTek/iceberg.html.

[Taunton91] Mark Taunton. Compressed executables: an exercise in thinking small. *Proceedings of Summer USENIX*. (Nashville, TN), pages 385–403, 10–14 June 1991.