A 3-tier RAID Storage System with RAID1, RAID5 and compressed RAID5 for Linux*

K. Gopinath Nitin Muppalaneni N. Suresh Kumar Pankaj Risbood

Computer Science and Automation Dept.
Indian Institute of Science, Bangalore
gopi@csa.iisc.ernet.in

Abstract

This paper presents the design and implementation of a host-based driver (a "volume manager") for a 3-tier RAID storage system, currently with 3 tiers: a small RAID1 tier and larger RAID5 and compressed RAID5 (cRAID5) tiers. Based on access patterns ("temperature"), the driver automatically migrates frequently accessed data to RAID1 while demoting not so frequently accessed data to RAID5/cRAID5. The prototype system, called "Temperature Sensitive Storage" (TSS), provides reliable persistence semantics for data migration between the tiers using ordered updates or logging. Mechanisms are separated from policies through an API so that any desired policy can be implemented in trusted user processes. We also discuss the problems faced while moving from the original implementation on the Solaris platform to Linux. Finally, we present comparison of the performance of our design with comparable systems using striping or RAID5.

1 Introduction

The need for reliable, efficient, fast and easily manageable storage has dramatically increased because of the Web as web servers and database servers need to have these properties. The management cost nowadays is much higher than the actual storage cost, often by a factor of 4 to 7. Consider the case of a caching web proxy that maintains a large cache in persistent storage. But all the cached data is not useful; hit rates have been reported only in range of 30%. The performance of the caching proxies and database servers can be improved if it is possible to have storage on a device which allows fast retrieval of the frequently accessed data. At the same time, the storage should be reliable, i.e. it should be able to sustain disk failures without bringing the system down, a necessity for highly available applications. Further, even if the system crashes, it should be able to recover from the crash as soon as possible so that system down time is minimal. Another desirable feature is the efficient use of available storage, due to the huge amount of data that database and web servers may potentially handle.

Disk access patterns display good locality of reference [RW93], especially in non-scientific environments. For achieving cost-effective storage systems with terabytes of data, such locality can be exploited by using a multi-tiered storage system with different price-performance tiers that adapts to the access patterns by automatically migrating the data between the tiers.

This approach has a lot in common with memory caches. Like caches, we try to improve the performance using a small faster storage layered on top of a bigger, slower and relatively cheaper storage. But our problem is different from caches in that the caches need not provide reliable persistence semantics in the event of a system failure. Next, the latencies in our case can be much longer due to the use of lower speed devices such as disks when compared with memory caches. Also, in caches, the same data can occupy space in multiple tiers; in our case, data can be resident in only one tier. Our approach also has some similarities with hierarchical storage management (HSM) solutions. However, HSM is a more general storage system comprising of secondary (disks) and tertiary storage (tape), while our solution uses only secondary storage.

Our design currently has 3 tiers: declustered RAID1, RAID5 and compressed RAID5 (cRAID5). Redundant Arrays of Inexpensive Disks (RAID) is a technique to improve the reliability and performance of secondary storage. Of the various levels of RAID discussed in [CLG+94], RAID1 and RAID5 have become more popular due to ease of use and price/performance respectively. Mirroring or RAID1 maintains two copies of the same data and generally provides best performance and is easier to configure. Rotating parity scheme or RAID5 costs the least of all the RAID levels for the reliability and performance it provides. It suffers from poor small update performance and configuring RAID5 is more involved. Declustered RAID1 differs from RAID1 in that the data is striped across multiple disks. Two physical stripes constitute a RAID1 logical stripe with each stripe unit data being present in *two* different disks, thus

*This work has been supported by Veritas Software, Pune/MtView but they are not responsible for its contents. Nitin is currently at Veritas Software, Pune; Suresh at Lucent Technologies, Bangalore and Pankaj at Lucent Bell Labs, NJ

ensuring resilience to a single disk failure. cRAID5 is the same as RAID5 except that the data is compressed before writing. Parity is computed on the written (compressed) data; typically, the parity computation has to be done on the compressed data of more than one stripe. Random read access into a cRAID5 stripe is handled by decompressing just the compressed stripe (and not the full stripe as we have locking at the right granularity). However, writes are more involved which we discuss later.

1.1 Design Approaches for Multi-tier Storage Systems

A multi-tiered storage system can be implemented at various levels:

As an application This is the most flexible approach as the user application has complete control over data management. Applications like database servers are good candidates. In a file system environment that supports Data Management API (DMAPI) [DMAPI], an application can keep track of access patterns of files and implement a multi-tier storage. HSM implementations using DMAPI already exist.

Inside the file system Less flexible than the former, but may provide better performance than the former due to lower context switch overhead.

As a layered device driver (our approach) The device driver is layered on top of block storage media (generally disks). A given I/O request is divided into separate I/O requests each of which is issued to the device drivers of the underlying storage media it is configured to use. Our design currently has 3 tiers: declustered RAID1, RAID5 and compressed RAID5 (cRAID5), with future extensions for additional tiers like NVRAM. The data is migrated between the tiers depending on access patterns, so as to keep frequently accessed (hot) data in RAID1 tier and not so frequently accessed (cold) data in RAID5/cRAID5.

In the controller HP AutoRAID is a two-tier RAID storage array that uses RAID1 and RAID5. It is implemented at the controller level communicating with the host over the system bus. SCSI disks are connected to this controller through the controller's internal SCSI bus.

1.1.1 Motivation for a Layered Driver Approach

An application based approach would only improve that particular application while a file system based approach does not help configurations that use a block device directly such as a database configured to use raw devices.

There are both benefits and costs in either the controller (often called hardware RAID) or device driver (software RAID) approach. We briefly discuss them here (for a more detailed comparison, see [Veritas]):

Benefits of a Controller Based Approach

Lower I/O bus traffic As the host issues a logical I/O to the controller, the I/O bus will see much less traffic compared to a software approach. For instance, a mirrored configuration of software RAID will issue two separate I/Os, one for each copy.

Separate Hardware A controller approach uses a separate on-board processor to process I/O requests, thus not loading the host processor. In parity based configurations like RAID5, this can save the host some computing, but with very fast server processors nowadays, this is a minor issue. In addition, controllers may have on-board NVRAM to improve write performance. But all these add to the cost of the system.

Benefits of Device Driver Approach

Host processors easier to upgrade The problem of increased I/O bus traffic and load on the host processor need not always be that serious. Host processors are generally far more faster and are easier to upgrade than the ones used in controllers.

Controller level redundancy In controller based approaches, redundancy is provided against disk failures but providing redundancy for other hardware such as controller hardware, can increase the cost. In software based configurations this can easily be provided by connecting disks across different controllers. Also, this can lead to better performance as the load is distributed across multiple controllers.

Flexibility A controller based approach has limitations on the number of disks that it can manage. Once that limit is reached, any new disks have to go under another controller and so cannot be accessed by the former. Also, configuration flexibility is limited. A software approach can be easily customized for a new access pattern. As we shall show in the section on implementation, the mechanisms and policies can be cleanly separated in a software approach.

A device driver approach is much simpler from an infrastructural viewpoint and it is increasingly becoming important commercially as many vendors are now providing such solutions for managing multiple disks (Veritas VxVM, IBM & HP LVM, SGI XLV). Hence we have investigated the device driver approach. Implementing at the controller level was not considered seriously due to the difficulty of carrying out modifications at this level, lack of information, and also the time and cost involved in developing the skills and the cost of implementation in a university laboratory environment.

In section 2 we discuss related work. Section 3 explains the design. Section 4 covers the implementation, with specific details on the changes for Linux from our initial Solaris prototype. Performance study for the Solaris platform is presented in section 5. Section 6 draws conclusions and spells out further work.

2 Related Work

Loge [ES92] and Mime [CEJ$^+$92] disk controllers use a level of indirection to adaptively alter the physical location of data to improve performance. [AS95] present a device driver implementation of a related idea.

HP AutoRAID [WGSS95] is a firmware implementation of the idea. It is implemented at the controller level, communicating with the host over the SCSI bus. It has a separate on-board processor to do operations such as parity computation, maintaining various data structures, tracking access patterns and effecting migrations between the RAID1 tier and the RAID5 tier. It uses on-board NVRAM to improve writes and does log-structured updates to RAID5. It maintains logical to physical translation tables and two other tables for RAID1 and RAID5. The logical to physical translation makes the migrations of data between RAID1 and RAID5 transparent to the user.

[MK96] presents orthogonal placement of data to improve the performance of RAID5 in both normal and degraded mode.

[HD89] discusses chained declustered RAID1. Declustered RAID1 differs from RAID1 in that the data is striped across the disks and two physical stripes constitute a RAID1 logical stripe. Our driver's RAID1 tier is implemented as a more flexible version of declustered RAID1. Since our experimental setup consists of a single controller, we observed poorer performance for declustered RAID1 than RAID1. On a multi-controller configuration, it might provide better performance than RAID1.

[KL96] explains how compression algorithms can be used to predict future accesses with high probability by the use of access patterns and perform prefetching. But there is a sizable memory requirement for maintaining the required data structures, and an in-kernel implementation would pin down most of main memory. We show how it can be implemented in a user process[1] that uses the application interface provided by our driver to keep track of accesses and effect migrations.

Linux has an implementation for RAID personalities including RAID0, RAID1, RAID4, and RAID5 as *md* (multiple disk device driver). It can be called a device driver as it occupies a major number, but it actually never services any I/O requests. All the I/O requests are mapped to the respective underlying device driver even before they reach the strategy routine of *md*. The implementation is a kind of hack in the kernel (necessitating changes in the kernel code only for *md*), and thus does not follow the framework of a standard Linux device driver. As explained below, this has been necessary until 2.1 kernels as it was not possible to use concurrency amongst multiple devices managed by a single device driver.

3 Design of the Storage System

3.1 Design Principles

1. Use only commodity hardware that is available in typical systems.
2. Avoid storage media dependencies such as use of only SCSI or only IDE disks.
3. Keep the data structures and mechanisms simple.
4. Support reliable persistence semantics.
5. Separate mechanisms and policies with the former inside the driver and the latter in user level applications.

Our driver uses the host processor for performing operations like I/O processing and parity computation. As the driver runs in kernel mode with forced context switching turned off, increased load on the host processor can slow down the system, and result in poorer system response. This forced us to keep the data structures and mechanisms simple. We believe this tradeoff to be advantageous overall.

Our design does not assume any special hardware such as NVRAM or dedicated processor which are generally not available in typical workstation systems. But if NVRAM is available, it can be used (a ramdisk type driver is all that is required). Also we should be able to use the existing storage media; the driver can work with any devices with a block device driver. This keeps the design flexible but makes us unable to use device specific optimizations.

To guarantee reliable persistence semantics, we have investigated making changes to the state of a stripe using both *ordered updates* and using a separate logging device. State changes occur due to migrations. The second approach is especially suitable in the presence of cRAID5 in the system as many writes become read/modify/write operations. In addition, it speeds up RAID5 partial writes as well as mirrored (declustered) writes for RAID1. Our first implementation on Solaris 2.5.1 used only ordered writes as we had only 2-tiers in that system. Our next prototype on Solaris used both while the Linux prototype will also use both.

The ordered updates guarantee correct operation in the event of a system failure as the state changes, but there is a possibility for some physical storage to go unaccounted during these changes, which can be reclaimed by using a UNIX *fsck*-like program that we call *device-check*. Unlike fsck, it is not essential to run this program every time the system crashes. Also this program can be run on a live system. The ordered updates are explained further in the section on migrations.

We have provided enough hooks to implement polices outside the kernel. The driver provides an interface for applications[1] to access the data and services of the driver. The

[1] By the term application or user process in this paper, we mean root privileged processes, such as fsck. Storage devices are protected resources and are generally not directly accessible by unprivileged processes.

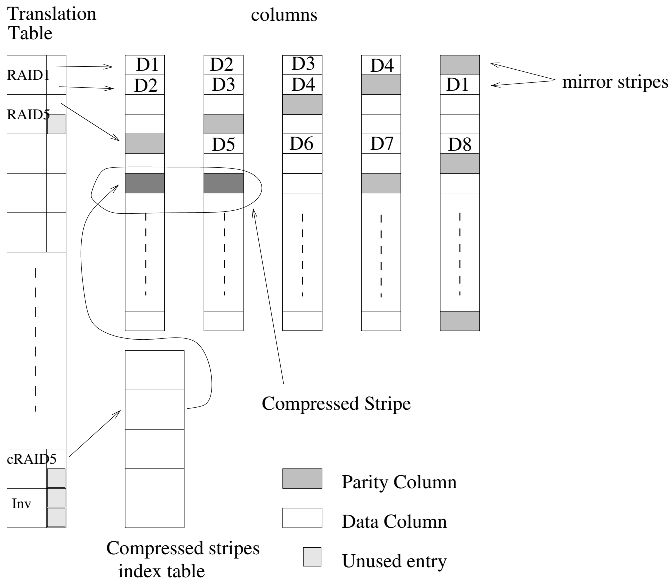

Figure 1: Storage Organization

device-check program, for instance, that is needed with the ordered write approach uses this interface.

The multi-tier storage is referred to also as temperature sensitive storage (TSS) as migrations enable frequently accessed ("hot" temperature) to reside in RAID1 and less frequently ones in RAID5/cRAID5 ("warm" and "cold").

3.2 Data Layout

3.2.1 Physical Storage

The physical storage is organized in RAID5 fashion as shown in the figure 1. The storage consists of a set of *columns*. We use the term columns to distinguish them from disks. A column is a block device which can be a single disk or can be a pseudo device built from multiple disks using a layered driver. The smallest column limits the size of the storage. A column is divided into contiguous regions called *stripe units*. A *stripe* is a formed by grouping one stripe unit from each column.

The size of the data portion of a stripe is N-1 times the stripe unit size (N being the number for columns), as one stripe unit from each stripe is used for parity. We use left-symmetric parity distribution scheme[LK91]. The *polarity* of a physical stripe is defined as the column number of its parity stripe unit. The polarity is used to ensure declustering for RAID1 stripes. These physical stripes act as backing store for logical stripes that are explained next.

3.2.2 Logical Storage

The logical storage can viewed as a collection of logical stripes. The logical to physical translation is done by the driver, so that the user's view of the data does not change even as the stripes undergo change of state. A logical stripe can be in any of the following states:

Invalid No backing store is allocated for this type of logical stripe. Initially all the logical stripes belong to this type.

RAID1 Two physical stripes of different *polarity* provide backing store for a declustered RAID1 logical stripe. Due to the way stripe units are numbered in a stripe, this ensures that the mirrors always come from different columns. The parity stripe units of the physical stripes backing a RAID1 logical stripe are left unused. This leads to some of the storage going unused but it keeps the data structures and the mechanisms simple.

RAID5 A single physical stripe provides backing store for a RAID5 logical stripe.

cRAID5 The data of this stripe is compressed and stored in a physical stripe. Since a compressed stripe is smaller than the uncompressed one, a full physical stripe is not necessary for providing the backing store for the compressed data. The unused region of the physical stripe can be backing store for some other compressed stripes. Since the compressed data is stored in RAID5 format, we can sustain single disk failures. Since the unit of allocation is at sub stripe level, an allocation bitmap is stored persistently on the disk.

An *allocation bitmap* is used to maintain the allocation status of physical stripes. The logical to physical translation table and the allocation bitmap are persistent structures stored on private partitions[2] which are read into the main memory at the time of loading the driver which is a loadable kernel module. When changes to these data structures need to be persistent, like logical stripe type or allocation status of a physical stripe, the on-disk copies of those specific entries are updated.

With out compression, it is sufficient to know if a physical stripe is acting as backing store for any logical stripe or not. Introducing compression requires us to be able to address storage in smaller units than stripes. For the sake of simplicity, we set the number of individually allocatable subunits within a stripe to a power of 2 (2^n) before the creation of device; the granularity of compression is thus set to *stripe size*/2^n. Once the device is created, this cannot be changed.

If a logical stripe is backed by a cRAID5 stripe, the translation table gives the index into the compression table where the information of the backing physical stripe is stored. The latter includes the physical stripe that contains the backing store for the logical stripe, the actual size after compression and the offset of the allocation unit in the physical stripe.

[2] Redundancy for the information on the private partitions remains part of future work.

3.3 I/O Processing

A given logical access is divided into separate physical accesses and issued to the underlying drivers. The given access is first broken at the stripe granularity as all stripes that the access covers need not be of the same type. Each of these accesses is handled depending on the type of the stripe.

A read to an invalid type stripe always succeeds. Backing physical stripe(s) are allocated on demand for writes making the stripe RAID5, and the operation is retried.

A write to a RAID1 stripe should update both the copies, whereas a read can be satisfied with either of the copies.

Reads and full stripe writes to a RAID5 stripe are handled in normal RAID5 fashion. During partial stripe writes, an attempt is made to migrate the whole stripe to RAID1 failing which the access is handled in RAID5 fashion.

Read in cRAID5 are handled by reading the stripe units of the compressed stripe. The data is uncompressed and the requested portion is copied to the target. Currently, no migrations are done as this will be left to policy modules (see below).

For write requests, the compressed data of the stripe is read, uncompressed and the old data overwritten by the new data. The data is then compressed and written in a new stripe or possibly in the original stripe itself. The latter is the best case when the number of physical blocks allocated to the new compressed stripe does not change. Otherwise, we need to find a new stripe that can accommodate this new compressed stripe and then release the storage in the old stripe. If compression is not good enough so that we need close to a full stripe itself for the write, the above read-modify-write cycle can be dispensed with in some cases if the write is for a full stripe. However, in all the above cases, no migrations are done, leaving the decisions to the policy modules.

3.4 Migrations

Migrations result when a logical stripe changes type. A logical stripe changes from invalid to RAID5 on a write to it. After the migration is completed, the I/O is retried (this time in RAID5 fashion). If the request is a partial stripe write, another migration is triggered to make it RAID1. Thus a partial write to an invalid type stripe ultimately results in its migration to RAID1. A full stripe write to an invalid stripe only makes it a RAID5 stripe. An alternative strategy is to move the stripe directly from invalid to RAID1 on an update whether it is full stripe request or a partial one. We chose the 2-stage approach as we have no information whether the data will remain hot after this write. If it remains hot, the policy mechanisms will migrate it to RAID1. It also keeps the implementation simple and it is the right strategy for large I/Os.

RAID1 to RAID5 migration usually happens when a RAID1 stripe is victimized to give one of its physical stripes to a currently invalid logical stripe to make it RAID5 or to a RAID5 stripe to make it RAID1.

Our strategy for migrations to/from cRAID5 is similar. First migrate the needed stripe to RAID5 (leaving the other stripes in cRAID5 with certain parts invalid) and migrate to RAID1 as needed. RAID5 to cRAID5 migrations typically take place through policy mechanisms when the data becomes cold.

We use software LRU to maintain access frequencies of stripes.

3.5 Application Interface

To keep the system flexible and the implementation simple, we have tried to offload the policy decisions to user level applications[3]. The driver provides interfaces using which an application (necessarily with root privileges) can access the driver data such as logical to physical translation table, stripe access information, physical stripe allocation bitmap and services such as initiating migrations, punching holes, etc.

4 Implementation

The first prototype has been implemented as a pseudo device driver on Sun Solaris 2.5.1 with two tiers of RAID1 and RAID5[4]. The next prototype added the cRAID5 layer. The current effort is reengineering it for Linux 2.2.5[5]. Since the Linux device driver interface is very different from Solaris, and given the rapidly changing internal interfaces in the Linux kernel since 2.0, this has required careful study. We will discuss some of these below (see Sec 4.2.1).

The implementation of TSS has been designed to be modular to allow each module can be tested and debugged independently. The implementation not only supports a unified, integrated TSS device but also supports RAID1, RAID5, declustered RAID1 devices also. The control and configuration of the devices is done through ioctl calls.

Each device has a *dev* structure associated with it, which contains the information about the device including its personality type, personality specific information, information of the underlying devices, and the function pointers corresponding to the entry points specific to the device personality. Figure 2 depicts the details of *dev* structure and associated data structures.

When any entry point like read, write or close is called on a device, it first executes the generic implementation as exported by the device. The generic code in turn calls the

[3]This is a common Unix approach, used for example, in fsck, X, to access kernel data through kmem, proc, etc

[4]The source of this is available under GNU GPL. Pl. send e-mail for the exact location.

[5]*The current status of the Linux implementation*: Though the prototype works correctly for the case of 5 "virtual disks" on one physical disk, it does not yet work properly for the case of 5 physical disks (the useful case).

personality specific function based on the type of the device. This design allows us to implement each personality separately and then integrate them together.

In the initialization module, a global array of pointers, to device structures is created. In the configuration ioctl, the generic device structure is created and its personality is specified, later the personality specific ioctl is used to configure the actual device parameters.

The I/O queue maintained by the driver is the same for all the devices controlled by it. When a request is in queue and `request_fn()` is called, it detaches the first request from the queue and passes this request structure to the personality specific strategy routine.

The repackaged I/O request is divided into stripe requests and all these are collected under first level stage I/O. Now the mapping from the logical to physical stripe is performed. For the declustered RAID1 and RAID5, the mappings are direct functions of the logical stripe number. For the integrated TSS device, *maptable* and *compression table* are used for this operation. Figure 3 gives the details of the way I/O staging is done at various levels in the data flow from the pseudo device to the actual device and the various abstractions involved.

We first take a look at how I/O is handled in Linux.

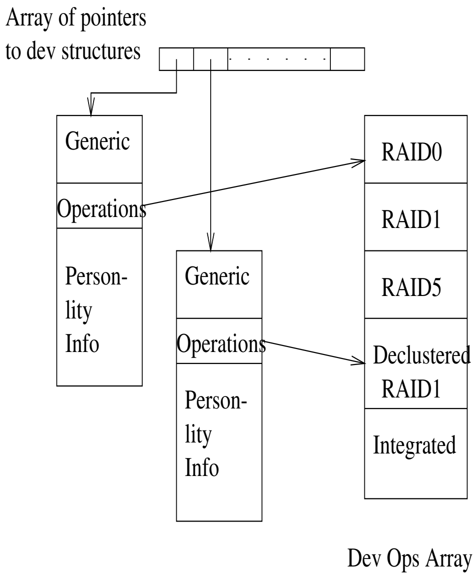

Figure 2: Device info structure

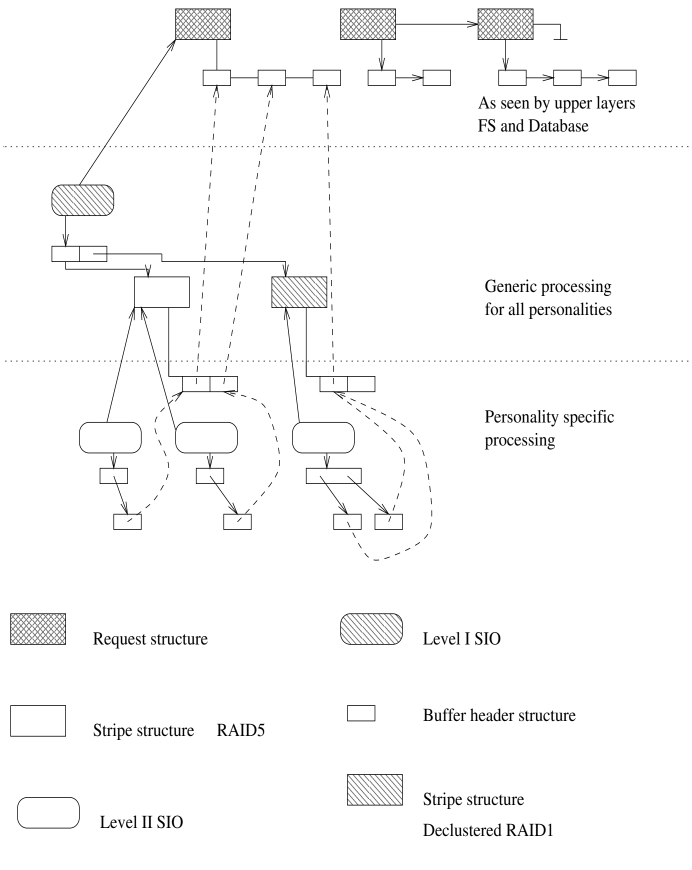

Figure 3: I/O staging Levels

4.1 I/O handling in Linux

Linux uses *request* structures to pass the I/O requests to the devices. All the block devices maintain a list of *request* structures. When a buffer is to be read or written, the kernel calls `ll_rw_block()` routine and passes it an array of pointers to buffer heads. `ll_rw_block()` routine in turn calls `make_request()` routine for each buffer. `make_request()` first tries to cluster the buffer with the existing buffers in any of the *request* structures present in the device queue. A *request* structure consists of a list of buffers which are adjacent on the disk. This clustering is performed only for the drivers compiled in the kernel and not for loadable modules. If clustering is possible, no new *request* structure is created, otherwise a new *request* is taken from the global pool of structures and initialized with the buffer and is passed to the `add_request()` routine. This routine applies the elevator algorithm using insertion sort based on the minor number of the device and the block number of buffer. If the device queue is empty, the kernel calls the strategy routine i.e. the `request_fn()` of the driver; otherwise, it is the responsibility of the driver to reinvoke it from the interrupt context (see Figure 4). Another requirement for `request_fn()` is that it cannot block as it needs to be called from the interrupt context.

To allow the accumulation of requests in the device queue, a plug is used. When the request comes in and the device queue is empty, the plug is put at the head of the device queue, and a task comprising of the unplug function is registered in the disk task queue. Thus the requests keep on accumulating for some time and then the task queue executes the unplug routine which removes the plug and calls the `request_fn()` to service the requests.

4.2 Problems for a Linux Implementation

4.2.1 Linux Device Driver Issues

As an example, the 2.0.x versions of Linux cannot use the concurrency provided by multiple disks if its device driver framework is strictly followed. Though concurrency exists at the hardware device level, if the same driver is used for all of them, the I/O requests for different devices will be serviced sequentially. A driver can have only one queue of device requests and hence all the requests for the devices controlled by the driver will be in the same queue. The processing of a new request is initiated only when the previous I/O is finished and in interrupt context. This problem has been eliminated in 2.2.x versions by allowing drivers to register a function which returns the pointer to the head of the queue in which the new request is to be inserted. Now a driver can maintain separate queues for each hardware device.

We face the following situations in implementing a layered device driver in Linux:

Blocking in Interrupt Context For interrupt driven block drivers, the strategy routine (`request_fn`) can be called from interrupt context but it cannot block. On Solaris, this is possible as it has interrupt threads. For a layered implementation, one needs to call the `ll_rw_block` routine from the `request_fn`, so that it can put the buffers in the request queue of the underlying device.

But `ll_rw_block` routine in Linux can block as it has a global array of request structures, and if all the slots in the array are filled then the function has to block. One solution could be to modify the `ll_rw_block` code so that if we cannot find a request structure, we return immediately and queue a task in schedule queue, to be executed later.

A better solution would be to make sure that we never need to call strategy routine in the interrupt context. This can be done by consuming all the requests queued to the device queue in a single invocation of the `request_fn`. This is so as the kernel calls the `request_fn` from process context only if the device queue is empty.

The solution to this problem is to design the `request_fn()` in such a way that it keeps on executing till all the requests in the device queue are exhausted. Thus it will always execute from the process context. One drawback of this scheme is that one process may have to delayed or blocked for I/O requested by some other process, but this is acceptable as the situation will occur only when all the request structures are exhausted which is likely to be infrequent. The pseudocode for `request_fn()` is as below:

```
tss_strategy() {
  while (1) {
    if (no request in queue) return
    remove first request from queue
    get tss dev corresp to minor#
      in request
    call personality specific strategy
    if (error in delegating I/O)
      call end_request with buffers
        not uptodate
  }
}
```

Fixed Size Buffer The buffer size for a device is fixed, unlike Solaris where we can have variable sized buffers. For example, to implement RAID5 efficiently, we need to distinguish between the full stripe write and partial stripe write as the latter involves a read-modify-write cycle. In Solaris, this is easier as one buffer can span across stripes. In Linux, each logical buffer is already split into smaller fixed sized buffers, so one has to rediscover the logical buffer to distinguish between the two cases and do the processing accordingly.

In addition, reporting of errors when they occur has to be at buffer granularity. We can keep track of errors only at the individual buffer and therefore cannot do error reporting at the stripe level.

end_request If we need to use multiple queues, then the current `end_request` does not work. We need a new implementation.

4.2.2 Lack of other support in Linux kernel

In addition to the above problems arising due to lack of infrastructure in the Linux kernel for layered device drivers, we have the following additional problems, due to lack of other required support in the Linux kernel. Firstly, there is a **cache consistency** problem that occurs in case of RAID5 writes. RAID5 writes are of two types. The full stripe writes completely bypass the buffer cache as I/O is done by creating only the buffer header structures with the data pointers appropriately set, and later releasing them. The partial stripe writes go through buffer cache as they involve a read-modify-write cycle. Thus the cache can become inconsistent. To eliminate this problem, buffers for the stripe undergoing a full stripe write need to be invalidated.

Second, Linux does not have an implementation for **condition variables** to allow atomic grabbing of a lock with condition checking. *ilock* had to be implemented to check for any overlap among the various I/Os being issued concurrently.

When a request comes in, the region that needs to be locked is calculated in terms of starting and ending sector. First, a region structure is created with this information, the global lock that protects the list is then acquired followed by disabling of the interrupts. This whole exercise ensures that any addition or deletion to or from the list is atomic. Now each region structure in the list is compared for any overlap with the new structure. If no overlap is found, a lock structure is allocated and initialized. The mutex lock in the lock structure is acquired and the reference count is set to 1. Now the list lock is released followed by reenabling of the interrupts.

If an overlap occurs, then the reference count of the associated lock structure is atomically incremented, the list lock is then dropped. Now the process does a down on the mutex of the lock structure corresponding to the overlapping region. Linux reenables the interrupts when the process tries to sleep. By exploiting this feature, we can do checking of overlap and wait in an atomic fashion.

When the process wakes up, it atomically decrements the reference count of the lock structure on which it was waiting. If it is zero, it frees the lock structure. Now it starts all over again to check for overlaps. Following is the pseudo code:

```
     create region struct for request
rep: grab the list lock
     disable the interrupts
     check for overlap
     if(overlap occurs) {
       release the list lock
       wait on mutex & enable interrupts
       atomically decrement the refcnt
       if zero, free lock structure
       goto rep
     } else {
       create a lock struct
       set its refcnt=1 & lock its mutex
       insert region struct in list
       release the list lock
       enable interrupts
     }
```

When an I/O is done, the associated lock structure's reference count is decremented in the interrupt context and tested for zero. If it is zero, the associated lock structure is freed. The region structure associated with the done I/O is also removed from the list and freed. The pseudo code is given below:

```
decr refcnt of associated lock struct
if (lock refcnt is zero)
  free the lock structure
else wake up those sleeping on mutex
free the region structure in list
```

4.2.3 Other problems not unique to Linux

One important problem is that there is no **API** between users of a device and its driver. The size of the TSS device is not fixed but varies dynamically as the organization of data changes among different storage personalities. If there exists a file system that can talk to underlying device through an appropriate interface and can dynamically

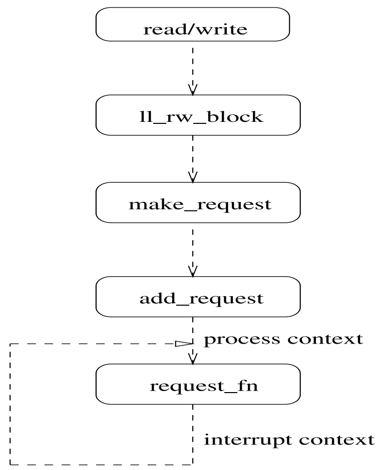

Figure 4: I/O flow in Linux

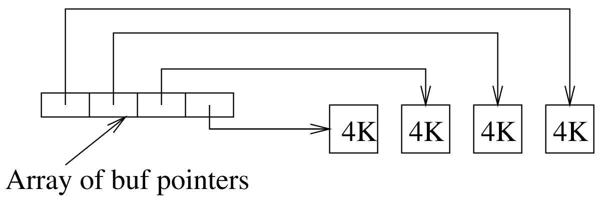

Figure 5: Data structure passed to `ll_rw_block()`

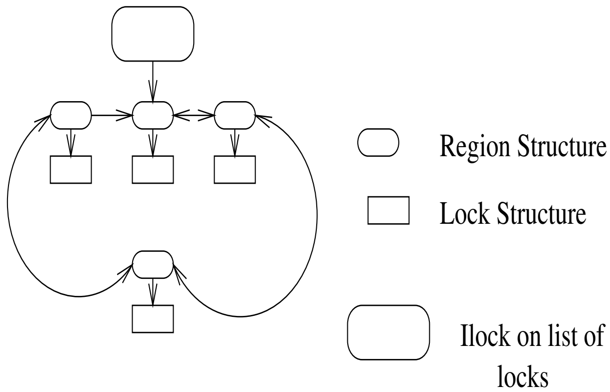

Figure 6: Interlock Structure

change the file system size, this feature of dynamic growth or shrinking feature of TSS can be exploited. None of the file systems available on Linux platform have such a facility (and most other FS on most OSes too!), so we still need to export a fixed size of the device (required at time of mount). During device configuration, the user needs to specify the fraction of device that can be under RAID1 and fraction that will be compressed. Using this information the device size is calculated and exported. Ambiguity still exists as the amount of compression achieved cannot be predicted in advance.

4.3 Changes to Kernel

Even though the device driver framework of the Linux has been strictly followed, there are a few changes that are still needed to the kernel:

Enable Clustering for TSS device Linux does clustering only for drivers compiled into the kernel and not for loadable modules. TSS relies on clustering to gain performance, otherwise all the stripe I/Os will be partial ones. So we need to enable clustering for the TSS driver in `make_request()` code.

Limiting the number of requests For each request structure that gets queued to the TSS device queue, there will be requests queued in the underlying device queue. So if all the request slots are consumed by the TSS device, it will lead to deadlock, as processing these requests require a free request structure . To avoid such deadlock, the number of request structures that can be captured by the TSS device is set to half of the total possible request structures in the system.

Removal of Plugging Plugging of the device queue is done in Linux to allow for the accumulation of the requests in the queue. For a pseudo device this accumulation doesn't make sense as this will be done again at the underlying device level. Thus `ll_rw_block()` code needs to be changed to bypass plugging for the TSS device.

4.4 Data Structures

Maptable The maptable is a data structure used for logical to physical translation, and maintaining access information. This is read into the main memory at driver loading time. Whenever an entry changes its type, it is flushed to stable storage but access information updates do not result in flushing. The entries of this data structure are 64bit long, containing fields for the type of the stripe, the physical stripes backing the stripe, the access information and an advisory bit to note if there is an access currently active on the stripe (see figure 8). The size of the maptable is equal to the number of logical stripes in the configuration.

Allocation Bitmap This is used to maintain the allocation status of physical stripes. Since the unit of allocation is smaller than the size of the stripe with cRAID5, we have a bit for each allocation unit of storage.

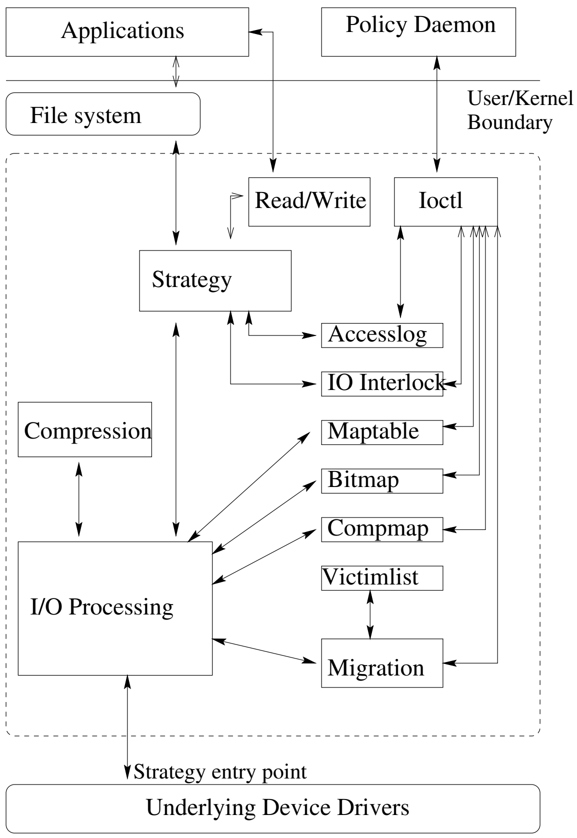

Figure 7: Implementation Model

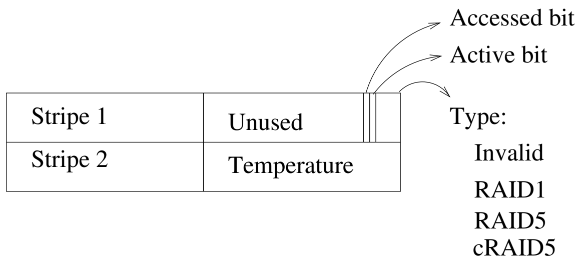

Figure 8: Maptable Entry

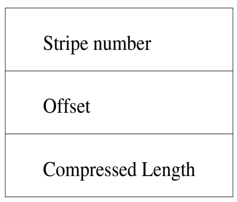

Figure 9: Compression Table Entry

Compression Table This table stores the metadata of each of the compressed stripe. For each compressed stripe, we need to know the stripe number which provides the backing store for the compressed data. Also, since the allocation of backstore is not done in units of bytes but in some units of sectors, we need to store the exact number of bytes which the compressed stripe occupies. This is required for the decompression algorithm. Also since the allocation is done in units smaller than the stripe size, we need to know which units are occupied by this compressed data. But since the allocation is contiguous, we need to store the offset of the first unit within the stripe. We can always calculate the number of units, from the size of the compressed stripe. Thus the structure of the compressed table entry is as shown in the figure 9.

Access Log When enabled, all the accesses to the device driver are logged in an in-memory access log which is a circular array of access entries. Each entry consists of two fields; the starting sector number and the size of the access in number of sectors. This can be enabled on a per instance basis. This access log can be accessed from an application to keep track of the accesses to the driver.

4.5 I/O Handling

On an access, the region spanned by it is locked. This is required as the type of the stripe should not change when an access is in progress. The access is then subdivided into separate subaccesses at the stripe granularity. These are then grouped under a *StageIO* and issued to the underlying storage drivers. The StageIO maintains a count of the number sub I/Os. In UNIX, whenever a block I/O request completes, `biodone()` is called for the request in the interrupt context which results in a call to the routine pointed to by the `b_iodone` field of the I/O request data structure (`struct buf`). The count is decremented in this calling routine and when count becomes zero, the parent I/O is signalled as complete by calling `biodone()` on it.

In Linux, as discussed above in Section 4.2.1, we have to come up with a few workarounds for a similar effect.

I/O Interlocking is not critical for non redundant storage but for redundant storage schemes like RAID it is essential for correct operation. For RAID1, we need to interlock the I/O region as two concurrent updates can leave the two copies (the data and the mirror) out of sync.

For RAID5/cRAID5, the updates need not be overlapping either to result in wrong operation. If two update accesses come to the same stripe, they need to be serialized as both will be updating the parity (in addition to the data). In addition, in cRAID5, the interlock has to be set at the logical stripe level so that atomicity of the write is maintained in case of simultaneous accesses to the compressed stripe.

In addition, we need to make sure that the type of a logical stripe does not change when an I/O is in progress on it.

4.6 Further Details of Linux I/O Processing

Processing of the stripe level I/O now depends on the type of the stripe, as explained below.

4.6.1 Declustered RAID1 I/O Handling

For a write request, corresponding to each buffer in the stripe, two buffer heads are created. For a read request only one buffer head per buffer is needed. These buffers are grouped under a second level stage I/O and are issued to the underlying devices to which the logical buffer maps.

4.6.2 RAID5 I/O Handling

I/O handling depends on whether it is a partial stripe I/O or full stripe I/O, so we discuss them separately.

Full stripe I/O Parity is computed from the data, and then data and parity both are written by breaking the stripe I/O into second level stage I/Os corresponding to each buffer in the stripe. We also need to invalidate the cache buffers corresponding the stripe undergoing I/O before writing the data to disk (see section 4.2.2).

Partial stripe I/O Partial stripe writes are a bit more involved than full stripe writes. To improve performance, a partial stripe write is handled in two ways:

1: Read-Modify-Write: This type of I/O handling is used when the size of update is less than half the stripe length. The buffers in stripe units that are going to be overwritten are read, then the parity is computed by xoring the old data, old parity and new data. The new parity and data are written to the disk by creating second level stage I/Os for each of the buffer involved.

2: Reconstruction-Writes: This type of I/O handling is used when the size of update is more than half the stripe length. Stripe units that are not going to be written are read in from the disk. The parity is computed by xoring the data read and data to be written. The new parity and data are written in similar way to Read-Modify-Write I/O.

4.7 Implementation of Migration

Here we discuss only the ordered write strategy for migrations between RAID1 and RAID5 for ensuring reliable persistence semantics[6]. We are currently investigating policies for migrations to/from cRAID5, mostly in a prefetching/caching framework[CFKL95, KTPB96].

In ordered write strategy, the updates are performed in a well defined order. We now explain this ordering of events and show how it does not result in wrong operation in the event of a crash.

Invalid to RAID5

[6] We omit discussion of the logging approach as it has been currently implemented on Solaris only.

1: The allocation bitmap is checked for a free physical stripe. If a free physical stripe is found, it is marked allocated and the entry of the bitmap is flushed to stable storage. The physical stripe is returned to the caller.

2: If no free physical stripes are left, a RAID1 stripe is victimized migrating it from RAID1 to RAID5 (this conversion itself involves multiple steps which are explained next), and one physical stripe is returned to the caller.

3: The maptable entry of the stripe is marked RAID5 and flushed to stable storage. The access is retried.

If the system fails between steps 1 and 3 or between 2 and 3, some storage may go unaccounted but does not result in any incorrect operation. This storage can be recovered online using the *device-check* program (see the section on application interface).

RAID1 to RAID5

1: An interlock is set on the stripe so that no one can access this stripe.

2: Full stripe data is read in, the parity is computed and the parity stripe unit of the physical stripe to be retained is updated.

3: The maptable entry of the stripe is marked RAID5 and flushed to stable storage. The physical stripe currently unused is returned to the caller.

Steps 1 and 2 do not change the structural state information of the system in any persistent way. If the system fails just after completing step 3 (marked RAID5, metadata flushed), one physical stripe goes unaccounted but cannot result in incorrect operation. The lost stripe can be recovered by running *device-check*.

RAID5 to RAID1

In the current implementation, a partial stripe write access to a RAID5 stripe triggers this migration. A more general scheme might wait till the stripe is accessed frequently enough and then attempt migration. To make transition from RAID5 to RAID1, a stripe requires a physical stripe whose polarity is different from the physical stripe currently used.

1: The allocation bitmap is searched for a free physical stripe whose polarity differs from the currently used physical stripe.

2: If the required physical stripe cannot be found in the free list, a RAID1 stripe is identified and victimized, migrating it to RAID5.

3: The full stripe data is read in and the physical stripe obtained from the victimization is updated with this data.

4: The maptable entry of the stripe is marked RAID1 and flushed to stable storage.

Steps 1 and 2 change the structural state information, but only one of them do it as step 2 is attempted only when step 1 fails. Step 3 does not change the structural information in any persistent way. If the system crashes before step 4, one physical stripe goes unaccounted which can be recovered by using *device-check*.

4.8 Optimizations to Enhance Performance

4.8.1 Victim list Management

The best candidate for victimization is a RAID1 stripe that has not been accessed for the longest time. But we found searching through the entire maptable for finding such a *perfect* victim to be extremely time consuming loading the host CPU. The problem is even more serious for a kernel module as kernel threads are not preempted when they run out of their time slice (even in Solaris 2.5.1, a kernel thread is preempted only if a higher priority thread becomes runnable), affecting all other processes and drastically reducing the system response. To reduce the amount of searching, we implemented *victim lists*.

A victim list is a list of RAID1 stripes that are sorted in the increasing order of access frequency. This list is consulted to find a victim but only as a hint. The stripe is again checked to see if it still remains a RAID1 stripe, or if it had any accesses after the (re)construction of the list, or if there is an access currently active against it. If any of these conditions is true, the entry is dropped and the next entry in the list is checked.

When the list is exhausted, it is reconstructed by searching through the maptable populating the list with RAID1 stripes that do not have their access bit set, and do not have any accesses active against them.

In the current implementation, victim list reconstruction is done by a kernel thread. This thread sleeps waiting for requests and, on being woken up with a request, does the reconstruction and sleeps again till the next request. The size of the list is fixed at 4096. A victim can be sought in either blocking and non-blocking mode. In the blocking mode, on failing to find a victim, the calling thread is suspended till the reconstruction of the victim list is done. This mode is used for invalid-RAID5 migrations which can not continue till a victim is obtained. In non-blocking mode, if no victim can be found, the victim list reconstruction thread is signalled with a request and the call immediately returns failure. This mode is used by RAID5-RAID1 migrations.

4.9 Interface to Applications

We have implemented an interface for use by privileged applications to access the data and services of the driver. The motivation for introducing this interface was to implement only mechanisms inside the kernel and offload policy decisions to user level applications. These are implemented as `ioctl()`s.

Get Maptable Address Using this `ioctl()`, an application can get the kernel virtual address of the maptable and the number of entries in it. The application can then perform an `open()` on `/dev/kmem` and `mmap()` the region into its address space. As the access frequency information is also kept in the maptable entries, the application can keep track of how often a stripe has been accessed without making any further calls to the kernel.

Get Bitmap And Maptable Using this `ioctl()`, an application can read copies of the allocation bitmap and the maptable frozen at some point in time. The driver locks the entire maptable (by setting an interlock that spans the entire device), and then the maptable and the bitmap are copied into the user buffer and the lock on the maptable is released. Using this information, an application can compare the allocation bitmap against the maptable to find if any of the physical stripes have gone unaccounted which can happen when the system fails when a migration is in progress. The application (eg. *device-check*) can then fix this by using another `ioctl()`. Since the temperature information is also available in the maptable, the application can know the temperature without making further calls to the kernel.

Get Compressed Table Address Using this ioctl, the user application can access the compressed table and hence can get information about every compressed stripe, such as compressed length and physical stripe backing for this stripe.

Get Access log Using this `ioctl()`, an application can get the kernel virtual address of the access log and the number of entries in it. The application can then call `open()` on `/dev/kmem` and `mmap()` the region into its address space and keep track of the accesses without making any more calls to the kernel.

Migrate Given a RAID1 stripe and a RAID5 stripe, this `ioctl()` migrates the RAID1 stripe to RAID5 and vice versa. This `ioctl()` along with the former can be used by an application to keep track of access patterns and perform prefetching. The application can maintain its data structures in user space which do not pin memory down. To reduce the system call overhead, multiple requests can be grouped together and passed as a list. The request is failed if any of the stripes has already changed the type.

Age The maptable can be aged using this `ioctl()`. This way we implement software LRU for maintaining the access frequencies of stripes. The *accessed* bit is ORed with the *temp* field after it is shifted to right by one bit. The *accessed* bit is then cleared. Aging is also performed from within the driver during victim list reconstruction if sufficient RAID1 stripes are not found to fill the list.

Sync Meta Data This results in all meta data structures being updated to stable storage.

Punch Hole A given RAID5, cRAID5 or RAID1 stripe is marked an invalid stripe and the backing physical stripes are released.

Migrations from/to cRAID5 These ioctls try to migrate RAID1/RAID5 to/from cRAID5.

cRAID5 Move This ioctl allows us to move the cRAID5 stripe from one physical stripe to another physical stripe as backing store. This ioctl can be used by an user level application to take policy decisions for compaction.

Make	Quantum Fireball 1280S	
Formatted Size	1,281,982,464B	
Drive Config	Disks	2
	Heads	4
	Tracks per surface	4,142
	Sectors per track	95-177
	Bytes per sector	512
Perf Specs	Average seek(ms)	<11
	Rotational speed(RPM)	5,400
	Avg. rotational latency(ms)	5.56
	Internal Data Rate(MB/sec)	5.8-10.4
	Cache buffer size(KB)	128

Table 1: The specifications of disks used

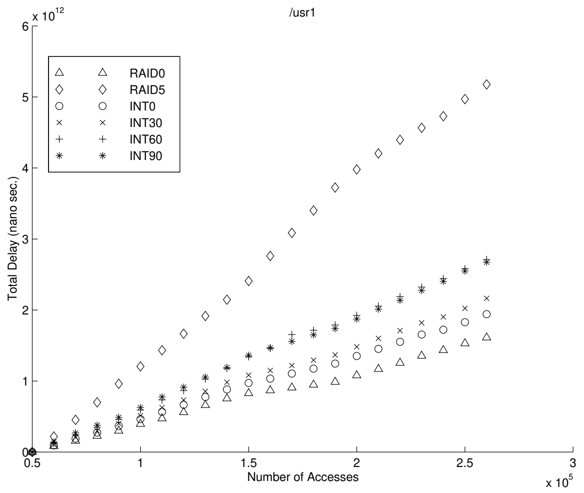

Figure 10: Total Delay vs. Number of Accesses for /usr1 disk

5 Experimental Results

We present experimental results on the Solaris platform with 2 layers (RAID1 and RAID5). We expect to provide results for Linux with 3 layers (RAID1, RAID5, cRAID5) by end of this year.

Setup The Experimental setup consists of a Sun SPARC5 workstation, with 32MB RAM. Five 1.2GB disks are connected to the machine over a 10Mbps SCSI bus. The specifications of the disks is given in Table 1.

Results and Analysis We have used HP disk traces [RW93] to evaluate the performance of our driver in comparison to RAID5 and RAID0. The trace that we used has been generated on a departmental server (snake) with accesses for two filesystems /usr1 and /usr2.

Both the integrated driver and RAID5 use 5 disks. RAID0 configuration uses only 4 disks, as it does not maintain any parity. The integrated driver is configured with 25 percent of the physical storage under RAID1. The stripe length for all configurations is 64 sectors.

Before applying the traces to our driver, we mounted a

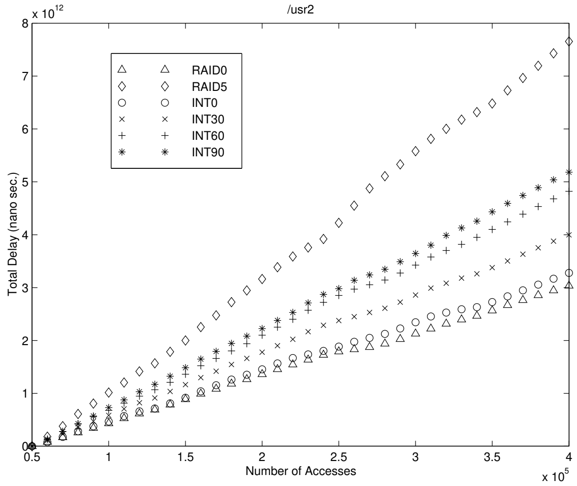

Figure 11: Total Delay vs. Number of Accesses for /usr2 disk

Disk#	Average Access times (ms)					
	RAID0	int0	int30	int60	int90	RAID5
/usr1	7.70	9.25	10.32	12.87	12.73	24.37
/usr2	8.67	9.36	11.42	13.77	14.81	21.87

Table 2: The average device access times

filesystem on it and populated it to various degrees. The first 50000 accesses were used as warmup, as the way we laid out the filesystem and populated it need not reflect the use of the traced system. The results given are for the remaining accesses. In all the figures and tables, the integrated driver is denoted by *INTx* where *x* is the degree (percentage) to which the device was populated.

Figure 10 and figure 11 show the total delay against the number of accesses for the two traces. The integrated driver performed 30-50 percent better than RAID5 even when populated to 90 percent.

Table 2 shows the average access times for the integrated device populated to various degrees along with RAID0 and RAID5. The fresh device (just ran *mkfs*) performed far better than the one that was populated. This was expected as initially all the stripes are of invalid type and writes result in the allocation of physical stripes that are contiguous which results in less seeking.

The traces exhibit high degree of locality. Figure 12 and figure 13 show the number of migrations against the number of accesses for the traces on an integrated device populated to 90 percent. The number of migrations remained very small compared to the number of accesses (<1%). Despite such high hit rates (>99%), the integrated driver access times increased with population, probably due to the following reasons:

Suboptimal data placement The present implementation has a very simple data placement policy. We simply select the next RAID1 stripe on the victim list and steal one of its

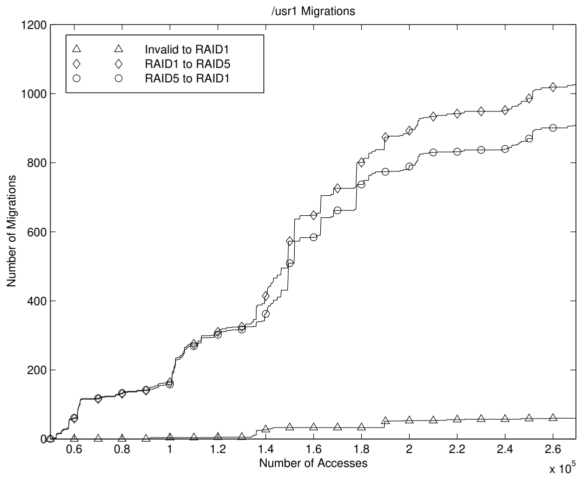

Figure 12: Number of Migrations vs. Number of Accesses for /usr1 disk

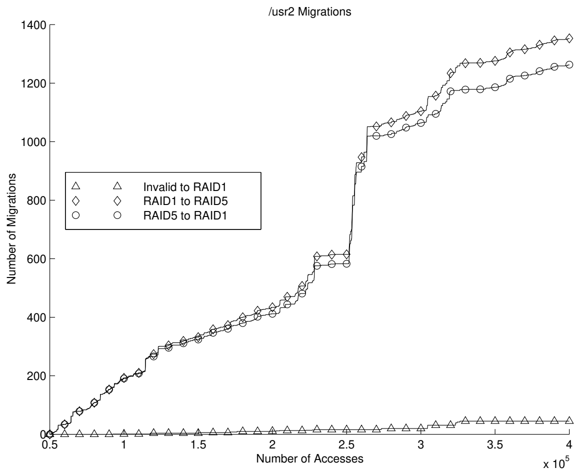

Figure 13: Number of Migrations vs. Number of Accesses for /usr2 disk

physical stripes when a RAID5 stripe or an invalid stripe needs to be migrated. A better policy should take seek distances into consideration.

High migration cost The current implementation does not maintain a pool of free physical stripes. Once the entire physical space is used up, any more migrations need to demote existing RAID1 stripes which involves updating parity, and writing metadata synchronously. This severe penalty on migrations can be reduced by making sure that there exist enough free physical stripes at all times.

6 Conclusions and Future Work

We have designed and implemented a two-tier RAID storage system using RAID1 and RAID5. We will be finishing the implementation of a 3-tier system (RAID1, RAID5, cRAID5) on Linux by end of this year. We also intend to present a evaluation of the Linux prototype at the same time.

Adding an NVRAM layer to improve write performance needs to be investigated. Prefetching and replacement of stripes across tiers is another area[CFKL95, KTPB96].

7 Acknowledgements

We thank John Carmichael, then at Veritas Software Corp., for suggesting that we look into this area, Fred van den Bosch of Veritas for his help and interest, S. Roy of Veritas for providing valuable technical assistance while implementing the driver, John Wilkes of HP Labs for providing the traces, Vaishnav Malay for his help in deciphering the traces and Kameshwari for suggestions.

References

[AS95] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.

[LK91] E.K.Lee and R.H.Katz. Performance Consequences of Parity Placement in Disk Arrays *4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), Palo Alto, CA, April 1991.*

[CEJ+92] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A high performance parallel storage device with strong recovery guarantees. *Technical Report HPL-CSP-92-9, Hewlett-Packard Laboratories*, November 1992.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, Kai Li. Implemenation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling

[KTPB96] Tracy Kimbrel, et. al. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. USENIX 2nd Symposium on OSDI'96

[CLG+94] 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):145–185, June 1994.

[DMAPI] Data Management Interface Group. Interface specification for the Data Management Application Programming Interface (DMAPI), version 2.1. March 1995.

[ES92] R. M. English and A. A. Stepanov. Loge: A self-organizing disk controller. *Proceedings of the Winter 1992 USENIX Conference*, pages 237–251, January 1992.

[HD89] Hui-I Hsiao and David J. DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. *Technical Report CS TR 854, University of Wisconsin, Madison*, June 1989.

[KL96] Thomas M. Kroeger and Darell D. E. Long. Predicting File System Actions from Prior Events. *Proceedings of 1996 USENIX Technical Conference, San Diego, CA*, pages 319–328, January 1996.

[MK96] Kazuhiko Mogi and Masaru Kitsuregawa. Hot mirroring: A method of hiding parity update penalty and degradation during rebuilds for RAID5. *Proceedings ACM SIGMOD 96, Montreal, Canada*, pages 183–194, 1996.

[RW93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of the Winter 1993 USENIX Conference, San Diego, CA*, pages 405–420, January 1993.

[Veritas] Veritas Software Inc. Enhancing Hardware RAID with Veritas Volume Manager. Available at http://www.veritas.com.

[WGSS95] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 96–108, December 1995.