Linux Based Temparature Sensitive Storage : Design, Implementation and Policy Framework

Venkatesh P

31 December 2000

Abstract

Temperature Sensitive Storage (TSS) is a data storage architecture, which keeps track of the access frequencies of the data and uses this information to manage the storage efficiently and effectively. The term *temperature* refers to the access frequency of the data.

TSS provides a hierarchical storage architecture, with frequently accessed data residing in a storage device with a faster access time, while less frequently accessed data placed in a device which is more storage efficient. The hierarchy is a 3-tier architecture consisting of Declustered RAID 1 (RAID1), RAID 5 (RAID5) and Compressed RAID 5 (CRAID5) storage devices. The data is dynamically transferred between these tiers, depending on their access frequencies.

TSS along with fast access and efficiency of storage, also provides reliability of data by incorporating redundancy in system through the use of RAID architecture.

Suitable policies have been implemented as user level daemon process, independent of the low level implementation of the TSS driver. Such policies make use of certain `ioctl()` calls provided by the TSS device driver.

In this paper we discuss the design, implementation and policy framework of Linux based TSS, and its performance analysis done with respect to static RAID systems[1].

1 Introduction

With the massive computerization and the falling price of storage devices, the focus has shifted from simple data storage towards fault-tolerant, efficient and easily manageable storage devices. The increase in the number of Database servers and Web servers has further added to this requirement.

Consider a typical Web server, storing some public domain information, like software, music etc. Naturally, the information on the web server will not be equally popular, and hence will not be uniformly accessed. The desirable feature of the storage device storing such an information is the efficient use of available storage. This is an important requirement mainly due to the huge amount of data that the web servers usually handle. At the same time the storage should be reliable. i.e. It should be able to sustain disk failures without bringing the system down. This requirement is a must for highly available application servers, where system should be up and running all the time. Further even if the system crashes it should be able to recover from the crash as soon as possible and the system down time should be minimal.

In this paper we present a storage device architecture TSS, which provides a hierarchical storage device consisting of RAID1, RAID5 and CRAID5 tiers. TSS gives a unified view of a single storage device, and the data migrates from one tier to another depending on the access frequency of the data. Thus the frequently accessed data resides in fastest storage (RAID1), while the not so frequently accessed data slowly moves to a storage which is not as fast but is storage efficient (RAID5) and the least frequently accessed data moves to a highly storage efficient and slow tier (CRAID5). Thus the scheme optimizes between performance and storage efficiency. TSS implements different storage personalities as different RAID levels, making the storage reliable and can sustain single disk failures without bringing the system down. Data consistency across the system crash is taken care of using the data logging mechanism.

The functionality provided by TSS matches the requirement of the typical Web server. Further, the dynamic nature of the data placement in different tiers, can take the advantage of *locality of reference* of the data accesses by typical applications. The policy for the dynamic data migration and maintenance of data at different tiers is kept separate from the low level implementation of TSS. The policies are implemented as user level daemon process, using the `ioctl()` calls

[1] static raid systems here refer to simple RAID1 and RAID5 systems

provided by the TSS device. This also has the advantage of having different policies for different situations, depending on the type of application and the availability of the resources.

Section 2 of this paper lists out the previous work done in this area. The major design issues are discussed in Section 3 and Section 4 gives an overview of TSS implementation. Section 5 reports the results of the trace driven simulation, followed by conclusions and future work in Section 6.

2 Previous Work

RAID (Redundant Array of Independent/Inexpensive Disks) systems guard against single disk failures by incorporating redundancy in the storage. There are different types of basic RAID systems from RAID0 to RAID5, of which RAID1 and RAID5 are the most common ones. RAID1 is a mirroring system while RAID5 has rotating parity. RAID1 is fastest as it allows parallel access but has a storage overhead of 100%. RAID5 is storage efficient but gives poor performance because of the Read-Modify-Write cycle involved in partial stripe writes. RAID5 performance drops drastically in degraded mode due to reconstruction of data involved for the failed disk from the data in other disks and parity[1].

Linux also has an implementation for RAID0, RAID1 and RAID5. This is the multiple device driver (*md*). This design claims to be a device as it occupies a major number but the actual implementation is a hack in `ll_rw_block()` code. *md* does mapping of requests from the *md device* to the actual device so the actual request never reaches the *md device*[2].

Another approach to the efficient storage management is Hierarchical Storage Management(HSM). It is a scheme that uses secondary storage (disks) and tertiary storage (tapes) for data storage and retrieval. This design doesn't apply to our system as we only consider disks.

In the past, work has been done on *Adaptive Hierarchical RAID* by M. Nitin [3] [4], and later was extended to include CRAID5 by Suresh N[5]. Their work was done on Solaris 2.5.1 as a pseudo layered device driver.

The work of Linux based TSS, *Temperature Sensitive Storage for Linux* was started by Pankaj Risbood[6]. The current work extends the TSS on linux, using the framework provided by [6]. The current work involved -

- The design and implementation of CRAID5, Integrated Device and the data logging.
- Development of policy framework and implementation of some sample user level policy routines.
- Test and configuration routines for TSS.
- Debugging the RAID1 and RAID5 part of TSS device driver.
- Performance Analysis of TSS on linux.

3 Design Details

3.1 Pseudo Device Driver Layer approach of TSS

The most important design issue with TSS is where to implement such a functionality. The possible alternatives are :
- *Application Level*
- *File System Level*
- *Device Driver Level*
TSS is implemented in *Device Driver level* as a pseudo device driver. A device driver will export the view of a single device while actually implementing TSS internally on a number of devices. Refer [3] for further details.

3.2 High Level Design

Figure 1 shows the overview of the TSS device. The TSS device can be divided into the a lower level pseudo device driver and a set of user level routines. The pseudo device driver consists of -

- **RAID1 -** The driver code, which handles the request for RAID1 kind of device.
- **RAID5 -** The driver code, which handles the request for RAID5 kind of device.
- **CRAID5 -** The driver code, which handles the request for CRAID5 kind of device. Here, a lossless compression algorithm, with least overheads like mod, multiply operations, is selected for the compression/decompression of the data.
- **Integrated Device -** The driver code, which provides a unified view of the multiple RAID devices under it. It also takes care of Victimization and Promotion of stripes. The promotion and victimization can be triggered by the demand for extra stripe at the device level, or by the requests from the user level.

- **Data logging -** The data logging is used to keep the data consistent across crashes, in the case of RAID5 and CRAID5. This is done by using an additional partition for logging, and then synchronously logging each of the request before doing any writes to the data disks, and synchronously committing the log entry after the write is complete in all the data disks.

- **Metadata Persistence -** For the data to be consistent in the TSS device, the metadata about the logical stripes, their mapping to the physical stripes, information about the compressed stripes etc, needs to be persistent across reboots. This is taken care of by using a separate private partition, which is used to hold the metadata of the TSS device.

- **Performance Measuring Driver (ganak) -** To keep track of the number of accesses and the time taken by the access of TSS device driver, one more pseudo device driver (ganak) is developed. This gets requests from user level routines, maintains some counters, and in turn does the read/write on the TSS driver.

  The advantage of this approach is the User-Kernel switching time will not be taken into account in the performance measures of the device drivers.

The user level routines for TSS device consists of -

- **Device Configuration -** These are a set of routines which are used to configure the TSS device. They can be used to specify both general device information and the personality specific information. The configuration information is parsed using `lex` and `ioctl()` calls used to communicate the config information to the TSS device driver.

- **Policy Enforcement -** The management of stripes at different RAID levels depending on the temperature, is done by the Policy Enforcement routines. These routines run as a daemon process, at user level. They collect the information about the stripes using suitable `ioctl()` calls, analyses the information and takes the decision for dynamic stripe management, by suitably promoting/victimizing the stripes.

  This routine is fully independent of the low level driver. Hence, we can have different policy enforcing routines for different applications. Also, multiple policy enforcing routines can be running at the same time, monitoring different characteristics of the logical stripes.

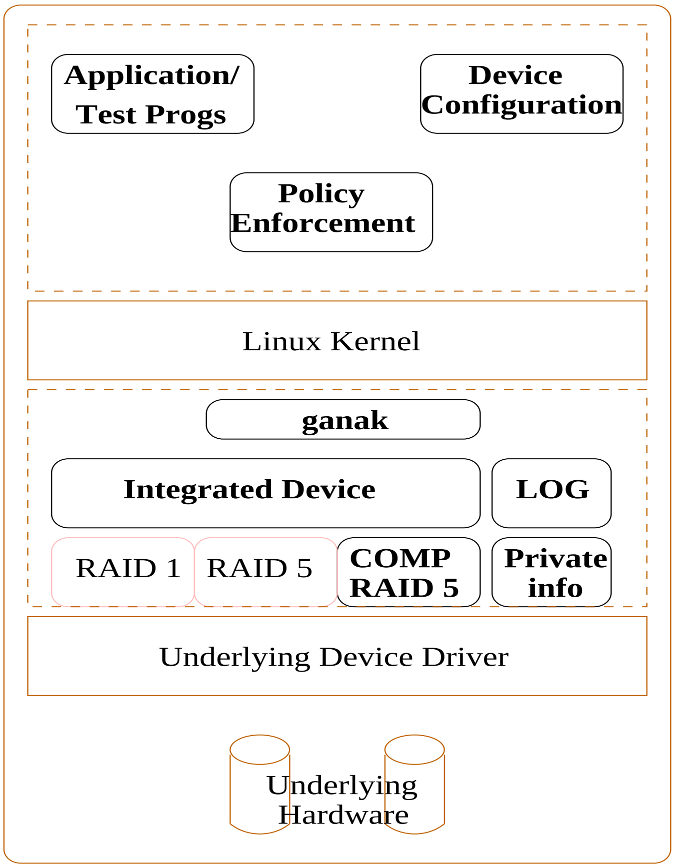

Figure 1: TSS High Level Design

- **Application/Test Programs -** Some user level routines are the application programs to test the functionality of the TSS device. This includes routines to read the trace and apply on the TSS device. This again uses the `read()/write()/ioctl()` interface of the TSS device.

3.3 Data Layout

The underlying storage is organized in RAID5 fashion as shown in figure 2. The storage consist of a set of columns which may be separate disks or partitions of disk. The smallest column limits the size of the TSS device. Each column is divided into contiguous regions called stripe units. A stripe is comprised of one stripe unit from each column. If there are N columns the number of data stripe units is N-1 and one stripe is kept for parity. We define the *polarity* of the stripe as the index of the column that contains the parity information. The stripes on the actual storage devices are the physical stripes and they provide the backing store for the logical stripes which the TSS device exports to the upper layers.

The logical stripe to physical stripe mapping is maintained by a Maptable. Any logical stripe can be mapped to a -

1. **Invalid** stripe i.e the logical stripe is not backed by any physical stripe. At device creation time all stripes will be of this type.

2. **RAID1** physical stripe in which case two physical stripes are needed to back it. The two physical stripes must have different *polarity* otherwise the failure of one disk (other than the parity disk) make the data irrecoverable. The parity stripe units in case of RAID1 are left unused this leads to a wastage of space which is 1/N the fraction of the total space under RAID1 storage of the total storage available. But this makes the design simple.

3. **RAID5** stripe which needs only one physical stripe.

4. **CRAID5** stripe, here the physical stripe that backs a logical stripe is not a full stripe but a part of it. Hence the unit of allocation has to be at a sub stripe level.

   If a logical stripe is backed by a CRAID5 stripe the *maptable* will give the index into the *compression table* where information of the backing physical stripe is stored. The *compression table* gives 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.

The free stripe information is stored in a bitmap format in *bitmap table*. As the CRAID5 addresses in sub-stripe granularity, the bitmaps also, should at the sub-stripe granularity. i.e., if the stripe is stored in sub parts for the CRAID5 storage, then a bit in the *bitmap table* corresponds to $1/4^{th}$ of a stripe.

3.4 RAID1 and RAID5

The design of RAID1 and RAID5 modules in TSS uses many data abstractions along with appropriate data structures. All of the structures used have been designed in the previous work of TSS on Linux, and fully explained in [3]. Further, I/O handling framework in Linux and its impact on the design of TSS is described in [6].

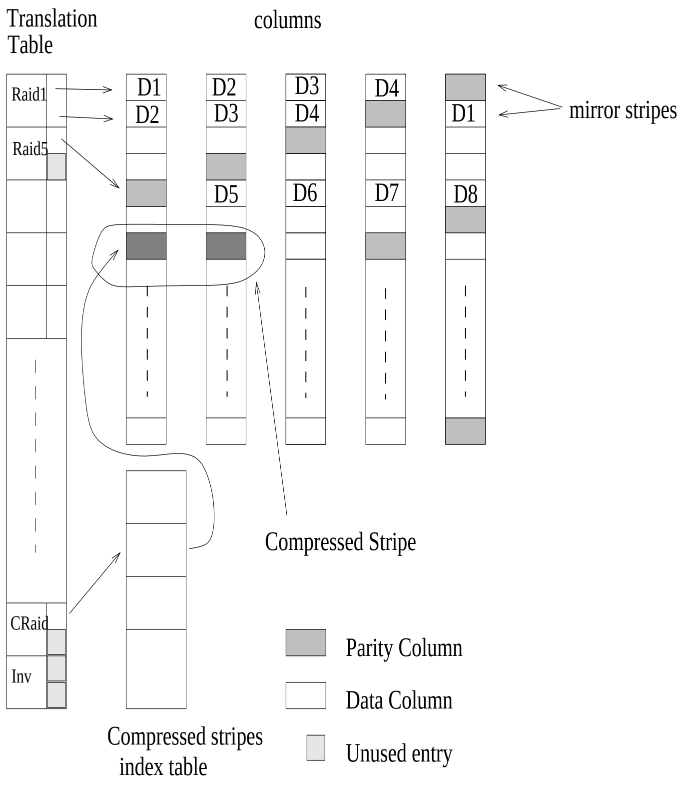

Figure 2: Data Layout

The data abstraction terms used in RAID1 and RAID5 are briefly explained here for the sake of completeness -

Stage I/O: I/O staging is a mechanism by which an I/O is divided into a number of sub I/Os. Stage I/O or *sio*, as it is called, allows this breakup to be done in a unified way.

There are two levels of I/O staging that occurs in our design, first when the request structures passed to the device strategy routine are broken down at the stripe granularity, and second when a stripe is broken down into buffers for issue to the underlying device.

Stripe: In TSS, the data is striped across the columns which could be disks or partitions of a disk. Each stripe in TSS could potentially be of a different personality, so to incorporate this a generic stripe structure has been designed. A stripe structure represents a sub I/O at the first level of I/O staging. It has information about the logical stripe number, the type of physical stripe backing it, the physical stripe number, etc.

Stripe is the level at which the mapping from the logical to the physical domain occurs. Stripe I/O handling completely depends on the physical stripe backing it.

Interlock: In TSS, an I/O in form of a request structure is spawned into multiple sub I/Os that are issued to the underlying device for handling. Between the time an I/O has been delegated to the underlying device and isn't completed, another I/O may come in which overlaps with the already delegated I/O, then it must wait for the first I/O to finish.

The meaning of overlap is different for different personalities. For RAID1, overlap occurs if the two I/Os have at least a buffer in common. For RAID5, the overlap occurs if the I/Os have a stripe in common. To avoid inconsistency, *interlock* structure is used to protect the region over which an I/O spans in the logical domain, so that the overlapping I/Os are not issued in parallel.

Signature: A *signature* is stored in the first sector of the private disk. This, along with some magic numbers which are used for identification, contains the information about where other metadata can be found on the disk, like the beginning block number of the *Maptable*, *bitmaps* etc.

Besides the *signature* the *Maptable*, *bitmap* and *compression table* are also stored persistently on the private disk.

3.5 CRAID5

The design of CRAID5 follows the RAID5 design framework, with the compression/decompression added in the data handling at the stripe level. The mapping between the logical stripe to the physical sub-stripe, is done through *compression table* look-up. And the physical sub stripe is read/written through the compression/decompression as filter using the normal RAID5 framework at the stripe level.

The issues in the Design of CRAID5 device are :

- For the CRAID5 device, the physical stripe also need to be locked when the write on stripe is in progress so that write of some other unrelated logical stripe, residing in the same stripe due to compression factor, should not affect the parity information of the physical stripe.

  Thus CRAID5 device needs two levels of locking. One at the logical stripe level (ilock) and the other at the physical stripe level (craid5 ilock).

- The compression algorithm to be used should not be memory and CPU time intensive. The kernel memory is a scarce resource. The compression and decompression occur inside kernel so we cannot use any algorithm that uses, floating point arithmetic as kernel doesn't save floating point registers during mode switch, or any heavy operation, like mod, as during this no activity other than interrupt servicing is done.

- During a write to a compressed stripe, the compressed length may vary. This has to be properly handled, by moving the physical stripe to some other suitable physical location, and then freeing the current physical location.

- No algorithm can promise a good compression on all sorts of data. Thus on sometimes there may not be any compression possible, and such cases has to be handled properly. Presently, such stripes are stored with no compression, and stripe length in the compression table indicating full stripe. This information is used during future reads/writes of the stripe. This can be extended future versions of TSS, to cover the cases where compression ratio is not worth the effort involved in compression and decompression.

3.6 Integrated Device

The design of the integrated TSS device follows the other RAID device design framework,up-to the stripe level Reads/Writes. After this level, the *map table* to map physical to logical stripe and the corresponding RAID device stripe I/O is called.

The issues in the Design of integrated TSS device are :

- Initially all the stripes will be invalid. Whenever some READ/WRITE happens to a particular stripe, it should move to RAID1 or RAID5 stripe depending on the availability of the free stripes. If enough free stripes are not available, then some victimization should be done to create the space.

- During victimization/promotion of a stripe, there should not be any I/O happening on the stripe. In case of promotion of CRAID5 or victimization into CRAID5 there should not be any I/O on the physical stripe also.

- Promotion from RAID5 to RAID1, a free stripe of different polarity should be used as otherwise, failure of one disk will lead to the data loss.

- To avoid the data inconsistency during the promotion/victimization the changes are to be carried out in an orderly manner. Say for a RAID5 promotion, find a suitable free stripe, copy the data, then change the mapping and the type in the map table in that order. This kind of ordered modification should be followed during all promotions/victimizations.

- During victimization of RAID5 to CRAID5, if there is no other free stripe, then we can victimize onto the same stripe itself. But this can lead to inconsistent data, in case there is a crash, with an incomplete victimization. There is no logging done for victimization/promotion, and hence data is not consistent. The same sort of situation arises when we are promoting CRAID5 stripe in the same physical stripe. To avoid such situation, one stripe is marked as *reserved* and used in above mentioned situations. Thus when there is no other free strip during RAID5 victimization, we move the compressed data to *reserved* stripe, then mark the stripe as CRAID5 and then reclaim the previous physical stripe as the new *reserved* stripe, in that order.

3.7 Failure Handling

As TSS uses the RAID storage in all the 3 tiers, it can provide fault tolerance against a single disk failure. If an I/O on any one of the underlying devices fail, then TSS device should switch to the degraded mode, and avoid all the I/Os to the particular device, and use the redundancy in TSS device instead to handle the I/Os. If more than one of the underlying disks fail then the TSS device should exit gracefully.

Further, when the degraded disk is replaced by a working disk, the new disk has to be *synch*ed with the other disks.

In TSS, when any I/O on a particular device fails, that device has to be noted, and if this happens regularly, then the device has to be marked as bad and all the I/O to the disk to be avoided, until the device is reconfigured.

In case of failure of data read through a RAID1 stripe, the data should be read through the alternate stripe; in case of failure of data read in RAID5 and CRAID5, the data has to be reconstructed using the other blocks in the stripe and the parity information.

The buffer handling routines that can fail, due to a bad disk are `getblk()`, `bread()`, `ll_rw_block()` (`uptodate` flag after `wait_on_buffer()`). Proper error handling in the calls to these routines are required to handle the TSS disk failures. The code for handling of failure condition is not included in the current TSS version, due to the shortage of time.

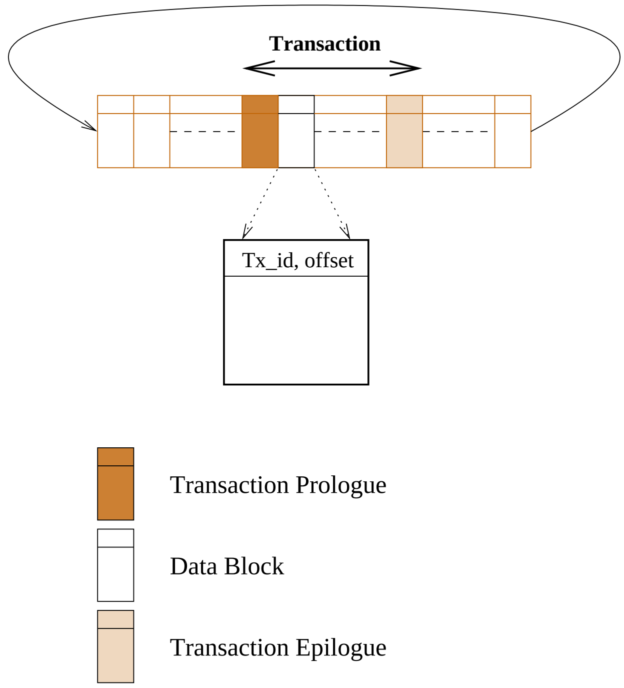

Figure 3: Design of the Log

3.8 Design of Logging

Data logging consists of logging the data along with the stripe information in a log device synchronously and then writing to the actual device. After the actual write is complete again a commit of the transaction is done on the log device. The design of the log is as in figure 3, and is explained in detail in [5].

The issues in logging are :

- Should the data logging be only targeting the data consistency or can try to improve performance, by delaying the writes to the disks, similar to a log structured file system. The former approach was chosen in TSS, mainly due to its simplicity and ease of addition into the existing TSS device driver.

- At what level should the logging be incorporated in the existing TSS device driver. The possible options were -
  request level - The log written, as soon as a write request comes to the TSS device driver. The advantage of this approach is less logging overhead, as there will be a single prologue and epilogue for a request. There are a few disadvantages in this approach. Firstly, a single log en-

try spans across multiple stripes and can map to different personalities and will complicate the recovery process. Secondly, the size of the log data can vary widely, depending on the request.
stripe level - The log written, after the write requests are broken into stripes. This has the advantage that the log records refer to a single stripe, of a single personality and the maximum size of the log record is fixed. On the disadvantage side, a request may be split into multiple records, and it works at the stripe granularity.
physical block level - The log written, just before the data is written into the physical blocks. Advantage with this approach is that the log records are of equal size. But the major disadvantage is that the more than essential data is logged on to the log device. Say, for a RAID1 stripe the data is logged w.r.t. both the physical stripes in place of one log entry in the previous cases. This method also complicates the recovery process.

Considering the advantages and the disadvantages of the three approaches, the stripe level logging was chosen as a optimal method for TSS.

- Another issue was, when should the recovery process be initiated. An `ioctl()` can be provided to trigger the recovery mechanism, which can be called through an application program, in case of crash of the TSS device.

- If the recovery is done in a straight forward manner, using the normal stripe write mechanisms, there is a possibility of data corruption. This case arises due to the use of Read-Modify-Write(RMW) cycles in RAID5 device. The problem is: if a part of a write is committed before the crash, it leaves the stripe at inconsistent state. If RMW cycle is used during recovery, then some inconsistent data may be read and used during the recovery, that results in permanent inconsistent data on the disk.

  This problem can be solved, if reconstructing writes are used in place of RMW cycles during the recovery. Since, in reconstructing writes the data will be reconstructed from the whole stripe information, partially committed data will not affect the recovery, and the data will be consistent.

- The next issue is what should be done in case of failures in accessing the log device. Logging is used for crash recovery, and we cannot have any kind of redundancy in the log device to avoid the failure of the device, as this will essentially reduce the speed of TSS device access, as the logging synchronously. So, failure in log device is not considered as a critical error, and it is up-to the administrator to replace the faulty log device.

- Another issue is of easy enabling/disabling of the log device. A feature has been provided in TSS, enabling the setting/resetting the log device through an `ioctl()` call.

3.9 Policy Framework and Policies Implemented

Policies and mechanisms for the migrations of the stripes are separated, with various mechanisms for migrations provided by the low level TSS driver in the kernel. The policies are imposed over the TSS device, by separate user level programs, that run as daemons, with suitable permissions. These policy routines communicate with the low level mechanisms through the `ioctl()` calls.

The low level TSS device, does take some policy decisions, in the extreme situations like when there is no free stripe for a particular I/O operation. This policy decisions at the kernel level could not be avoided, as there may be no policy running at the user level forcing the kernel to take the decisions.

The TSS device driver maintains *temperature* and *access* fields for each stripe. *Access* field indicates whether the stripe has been accessed in the recent past. *Temperature* field accumulates the accesses over a period of time. The *aging* is the process of updating the *temperature* of all the stripes, depending on its *access* field. *Temperature* of a stripe is reset during the migration of that stripe.

Following are the different mechanisms provided by the TSS driver that can be used by policy routines.

- Get/Set the temperature of a particular stripe
- Reset temperatures of all the stripes
- Do an aging on the stripes
- Get the number of stripes of each kind of personality
- Get the maximum temperature of any kind of personality
- Victimize/promote a particular stripe, from a particular RAID level.
- Victimize/promote the best possible stripe, from a particular RAID level.

It is important to note that, the information provided by the kernel about the stripes in the above mechanisms are as of that particular instance, and may not be true at a later point in time, when the user level application may use the information. There is no lock held in the kernel across requests. This approach was chosen as the policy enforcement is more to optimize the performance and not a strict guideline.

The advantages of having the policies separate from the low level device driver are the following :

- Different policies can be implemented depending on the kind of application and the hardware.
- None, one or more policies can be running at the same time

Following are the kinds of policies that can be implemented on the TSS device :

1. **Watermark Policy -** In this policy, there are lower and upper watermarks fixed on the three kinds of stripe personalities. Periodic checks are done to check whether the number of stripes of all the personalities confirm to the corresponding watermarks. If not, migrations of the stripes are performed, so as to confirm to the watermarks. Say if we have 20-25% lower watermark and 70-75% upper watermark on the RAID1 personality. This means that at any point in time, there should be a minimum of 20% RAID1 stripes. If it is not so, then promotions are done from RAID5 stripes, to have 25% of stripes at RAID1. Similarly, whenever there is more than 75% RAID1 stripes, there number is reduced to less than 70% through the victimization of the stripes.

   This policy is simple to implement, but has the disadvantage of being static. It does the migrations of the stripes based on temperature.

2. **Modified Watermark Policy -** In this policy, along with the regular migrations due to watermark, some additional migrations can be done depending the temperature. i.e, before doing the regular watermark migrations, additional migrations of the stripes in top and bottom 10% temperature bracket, with considerably high and low temperature, respectively, are done. This adds the dynamism to the Watermark Policy.

3. **The Second Chance Algorithm -** A policy similar to second chance algorithm in demand paging can be implemented on the TSS device. The stripes that are frequently accessed are given a second chance, and the rest are victimized to a higher level personality.

4. **Five Minute Rule -** The Five-Minute Rule is a way of organizing hierarchical memory, using the performance and the cost considerations, proposed in [9]. The idea originally proposed for organizing memory hierarchy can be modified and applied to the TSS device as well.

   The basic idea of Five minute rule is as follows. Frequently accessed data should be in main memory, while it is cheaper to store infrequently accessed data on disk. But this statement is little vague. What does this frequently mean? Five minute rule says that frequently means five minutes. The basis for this is the following formula

   $$Frequency = \frac{(\frac{Cache\_Cost}{Byte} - \frac{Memory\_Cost}{Byte}) * Object\_Bytes}{Object\_Access\_Per\_Sec\_Cost}$$

   This formula can be modified as below and used in the hierarchical TSS device.

   $$Frequency\_between\_RAID\_levels\_A\_and\_B =$$

   $$\frac{(\frac{RAID\_A\_Cost}{byte} - \frac{RAID\_B\_Cost}{byte}) * Object\_Bytes}{(RAID\_A\_acc\_per\_sec\_cost - RAID\_B\_acc\_per\_sec\_cost)}$$

   The access_per_sec cost for individual RAID devices can be got from the simulations and can be used here. The frequency values determined as above can be used by the policy routine managing the device and can trigger migrations depending on these frequencies.

5. **Predictive Policies -** More complex and intelligent policies that does the disk access prediction using the information about past disk access patterns can also be implemented using the basic policy framework provided by the TSS device driver.

Of the above policies, *Watermark Policy* and *Five Minute Rule* have been implemented on the TSS device.

3.10 Changes to Kernel 2.2.5

Even though the device driver framework of the Linux is strictly followed there are a few changes that are needed in the kernel [2].

- Linux does clustering only for drivers compiled in 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 `ll_rw_block()` code.

- 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 then it will lead to deadlock, as processing these requests requires free request structure which are not there. To avoid such deadlock, the number of request structures that can be captured by the TSS device is reduced to half of the total possible request structures in the system.

- Plugging of the device queue is done in Linux to allow for the accumulation of the requests in the queue. [6] suggested that plugging should be disabled for TSS device. But, disabling plugging for TSS device, can result in poor performance, as proper clustering will not happen, which will result in severe penalty in case of RAID5 and CRAID5 I/Os. This observation was confirmed by the simulation runs of the TSS.

4 Implementation

4.1 RAID1 and RAID5

The implementation details of RAID1 and RAID5 I/O handling is discussed in detail in [3] [6], and are not included here for the sake of brevity.

4.2 CRAID5

In CRAID5 read handling, corresponding compressed physical stripe data is read, the data uncompressed and requested part of the data is returned. Similar thing is done in case of writes also. The key issue: as the location and the size of the data requested and the actual data accessed from the disk are different, we need to get a new set of buffer blocks while reading or writing. The buffer blocks are got for the underlying devices and data read/written into them. Thus it involves data copy inside kernel, apart from compression and/or uncompression, and thus making the I/O on the CRAID5 slower.

Any compression algorithm which uses less amount of memory and CPU time, and that do not use floating point operation can be used in TSS. In this implementation LZRW1 compression algorithm [10] is used.

CRAID5 I/O requests are split into stripe level in a similar way as RAID1 and RAID5. The handling of the requests after splitting at the stripe level is described below.

4.2.1 CRAID5 read stripe

During a read to the CRAID5 device -

1. Look-up the compression table and get the physical stripe and the compression length of the stripe.
2. If no compressed stripe exists (read before writes), return zeros.
3. Otherwise, hold the craid5 interlock for the physical stripe.
4. Read the full stripe data onto a separate buffer.
5. Uncompress the data
6. Copy the required portion of the data onto the buffers
7. Release the craid5 interlock for the physical stripe
8. Mark the buffers uptodate and return.

4.2.2 CRAID5 write stripe

Full stripe and partial stripe writes are handled in different ways. During a full stripe write

1. Copy the full stripe data to be written onto separate buffer.
2. Compress the data.
3. Check whether the new compressed stripe can fit in its existing place.
4. If yes, get the craid5 ilock on the physical stripe, call the RAID5 write stripe, release the lock, update the compression table return.
5. Otherwise, get a new location for the compressed stripe.
6. Obtain the craid5 ilock on the physical stripe.
7. Call the RAID5 write stripe.
8. Release the lock.
9. Update the compression table and return.

Partial stripe write involves Read Uncompress Modify Compress Write cycle, and is a combination of the steps of full stripe read and full stripe write.

4.3 Integrated Device

Integrated device I/Os are also broken down onto stripe level I/Os. Then the specific I/O routine called, depending on the personality information in the map table. If the personality of the stripe is invalid, then that particular stripe is made into a RAID1 or RAID5, in that order of priority. Also, the access bit is set in the stripes after the I/Os.

4.4 Migration Mechanisms

Migration mechanisms are implemented in the TSS device driver and can be invoked using `ioctl()` calls. Mutual exclusion in stripe I/O is ensured during the Migrations by the use of ilocks over the logical stripes. In case of migrations into/outof CRAID5 stripe craid5 ilock over the physical stripe is also used. Metadata is kept consistent during migrations by using ordered writes. The updation to maptable is done in the end, after all the other updations. Also, care is taken not to overwrite the existing data, to have the consistency of the data in case of crashes during migrations.

4.4.1 RAID1 to RAID5 Victimization

It involves selecting one of the physical stripes of the RAID1 personality, and calculating parity for that stripe. The parity is written onto the disk and the maptable updated to indicate the stripe as a RAID5 stripe.

4.4.2 RAID5 to CRAID5 Victimization

It involves reading the compressed stripe, uncompressing it, getting a free stripe and writing the uncompressed data onto it. Finally, maptable is modified to indicate the change in the personality.

4.4.3 RAID5 to RAID1 Promotion

It involves getting a free stripe of different polarity, than the existing physical stripe. The data from the RAID5 physical stripe is read in and written onto the free stripe using the RAID1 type of writes. Finally, the stripe is marked as RAID1 stripe.

4.4.4 CRAID5 to RAID5 Promotion

It involves reading in the compressed stripe, uncompressing it, getting a free stripe and writing the uncompressed data onto it. Finally, the stripe is marked as RAID5 stripe.

4.5 Performance Optimizations / Improvements

Following possible optimizations/improvements in the basic TSS device driver has been added.

- The major delaying factor in the disk I/O are queuing delay, rotational and seek time. Two different optimizations were thought of to reduce the seek time during the RAID1 I/Os. One approach is placing the two physical stripes of the single logical stripe, next to each other, as far as possible. This can be advantageous in case of full stripe writes, as write has to go through both of the physical stripes. An alternative, entirely different approach for RAID1 stripe placings is also thought of and implemented. In this approach, we place the two physical RAID1 stripes halfway across in the memory. Here we are trying to get the continuous RAID1 stripes to be placed consecutively on the physical memory. This can reduce the seek time, as during the reads only one of the physical stripes will be accessed. These optimizations has to be enabled separately with some compile time parameters.
- The recent versions of the linux kernels have fast parity calculation routines, built-in. linux-2.2.5 has an optimized assembly code for the parity calculations, and that was used in the implementation of TSS device. 2.2.12 or later kernels has optimized these routines further, and can be used in the future versions of TSS.
- During the RAID1 reads, only one of the physical stripes will be accessed. TSS device has an option to route the read requests to any of the two options. Here an optimization to balance the load on all the disks is implemented. A set of counters, one corresponding to each device, is maintained and is incremented every time a read is performed on that disk. This information is used during the future read requests, and read requests are balanced across the devices. This again has to be enabled with a compile time option.
- The implementation of the TSS is planned to be released for the linux community, for the general usage and future enhancements. So the coding had to be modular and there should be proper utilities for using the TSS. For this reason a simple tss-config utility is developed using flex. The tss-config utility can parse the config file, and set up the TSS device, based on the configuration information.

5 Performance Measures

5.1 ganak Device Driver

The performance of the TSS device is measured with a series of I/Os on the TSS device. This can be done through the `read()/write()` system call to the TSS device. But, the problem with this approach is that the measure can be skewed, as it includes User-Kernel mode change and the measure heavily depends on the system load.

One way to avoid the above problems is to do the performance measure at the kernel level rather than at the user level. One more pseudo device driver will sit on top of the TSS device driver, and send I/O request to the TSS driver, and measure the time taken for the requests. The time measure granularity is in jiffies(10ms).

5.2 Simulation Data

Disk access in a typical workstations have some access patterns, in which data is accessed. In order to use such access patterns in our simulation, the disk access traces from HP-UX were used. A subset of HP disk traces [11], were used in simulation contains disk access generated by HP-UX file systems, on a system not running scientific applications.

The particular trace used in the simulation is from **Snake**, a departmental server at UCB Berkeley. The trace contains 0.7 million accesses. The main use of the system was for compilation and editing. It contains more writes(56.2%) than reads(43.8%), and most of the accesses were either 2, 4, or 16 sectors long. Each snake trace contains collection of accesses to 3 disks. disk 5 (/usr), disk 6 (/var) and disk 22 (root and swap). Only traces of disk 5 and disk 6 were used for the performance study.

Further, in order to reduce the simulation time, *Trace Compaction* (as in [4]) was done. Trace compaction reduced the time between the accesses by a factor of 1000. It was found in [4], that the trace compaction has little impact on the results, but speeds up the performance study process.

5.3 Simulation Set-up

Performance Measure was taken on the system *anant*, with linux-2.2.5 kernel and Red Hat Linux release 6.0. The machine is Pentium 75MHz board, with 32MB RAM. The 4 SCSI Disks, 1.2GB capacity each (Model - Quantum Fireball 1280S), were connected using 10Mbps scsi bus with Adaptec controller (Model - Adaptec AHA-294X SCSI host adapter) and used to configure different devices for the TSS device. Simulation was done when the system was on light load (load avg < 1).

Type of Device	Disk No.	Avg. I/O time per block[2] (ms)	
		with plugging	without plugging
RAID1	5 (/usr)	14.737	25.036
	6 (/var)	10.854	20.076
RAID5	5 (/usr)	18.479	48.609
	6 (/var)	12.321	33.505

Table 1: Effect of plugging on TSS

5.4 Simulation Results and Analysis

The effect of plugging in the kernel on the TSS device can be seen by comparing the RAID1 and RAID5 device performance with plugging enabled and disabled in the kernel. This results are summarized in Table 1. This clearly indicates that, plugging increases the performance of TSS. This can be attributed to reason that clustering of the requests happen due to plugging, and this clustering increases the performance of TSS.

The following trace simulation run results were taken with plugging enabled. The total number of I/Os in each run was around 4,00,000 blocks. Performance results of TSS with different configurations is summarized in Table 2. Simulation of TSS was also done with different policies and the results summarized in Table 3.

From table 2 we find that the access times in RAID0 is the least, and CRAID5 is the highest. This result is on the expected lines. RAID0 is nearly twice as fast as RAID1, mainly because RAID0 is free of any redundancy and it does simple reads and writes. Notable result here is RAID5 to RAID1 ratio in terms of access time is around 1.25 and in terms of space is 0.5. In other words RAID5 is consuming half the amount of space as RAID1, with a access time degrading by 125CRAID5 access time is around twice that of RAID5, and uses around 60migrations to the corresponding RAID types were done before the actual simulation.

These comparative figures, gives nice indications for setting the policy parameters in Integrated TSS

[2]All access times include queuing delay, seek time and rotational time

Disk No.	Percentage of Writes	Type of Device	Avg. Write Time per Block (ms)	Avg. I/O Time per Block (ms)	Space used wrt actual data
5 (/usr)	83.17	RAID0	8.283	7.147	100%
		RAID1	17.203	14.737	250%
		RAID5	21.904	18.479	125%
		CRAID5	66.506	57.136	87.2%
6 (/var)	59.83	RAID0	7.302	4.671	100%
		RAID1	17.345	10.854	250%
		RAID5	20.024	12.321	125%
		CRAID5	56.444	35.688	87.5%

Table 2: Performance of RAID devices through TSS

Type of Device	Disk No.	Percentage of Writes	Avg. Write Time per Block (ms)	Avg. I/O Time per Block (ms)	Space used wrt actual data[3]
TSS with watermarks	5 (/usr)	83.17	34.019	29.642	166%
	6 (/var)	59.83	36.833	24.342	165%
TSS with 5 min rule	5 (/usr)	83.17	23.030	20.834	169%
	6 (/var)	59.83	36.992	24.812	182%

Table 3: Performance of *cold* TSS device

device. On the basis of its performance the RAID5 should form the majority of the stripes in the integrated device. The number of RAID1 and CRAID5 can be around the same, and can vary depending on other things like available free space in the device.

The results of the integrated TSS device simulation, with different policies, is summarized in Table 3.

The simulation with watermark policy was carried out with the following watermarks :
RAID1 - Lower Watermark 10-15%, Upper Watermark 35-40%
RAID5 - Lower Watermark 20-25%, Upper Watermark 75-80%
CRAID5 - Lower Watermark 10-15%, Upper Watermark 35-40%

The simulation with the five minute rule was done with RAID1-RAID5 transition frequency as 1 minute and RAID5-CRAID5 transition frequency as 5 minutes.

The results shows that the integrated device takes around 70% of the space as RAID1. The relatively higher access times of the integrated device, can be attributed to the large amount of migrations that are involved, as initially all stripes were in invalid state (cold simulation). Significantly less access times can be expected if the simulation runs are repeated with proper warming up of the TSS device.

5.5 Notes on the Trace Simulation

The trace simulation may not be an exact reproduction of a disk access pattern on a typical linux machine because

- The traces were taken on HP-UX machine. But the disk access pattern in linux machine may be different as it depends on plugging/clustering of the requests by the operating system.

- The ganak device driver cannot call the request function of the underlying TSS driver due to the nonstandard request interface in linux. The workaround of calling `ll_rw_block()` in the ganak driver was used in the implementation. This works reasonably well with WRITE requests as it calls the TSS driver's request function. But, the READ requests at the TSS device may be satisfied by the buffer cache itself. The comparison of both the Write Performance and Overall Performance can be done to minimize this effect.

[3] The space consumed by integrated TSS varies dynamically and the figures provided here are the space consumed at the end of the simulation run

6 Conclusions and Future Work

6.1 Conclusions

- Linux device driver interface is not very suitable for implementing a layered device. Even with the module interface, we had to make changes in the kernel, for our module to run properly. This shows the need for some better, standard interfaces for the layered device drivers in linux.
- TSS can provide the storage with all the required features: reliability, speed and efficiency of storage.

6.2 Future Work

- Implementation of the Failure Handling routines in the TSS device.
- More intelligent policy routines such as predictions can be implemented, using the past disk access information.
- The hot plugging of disks in the TSS device.
- Provide provision for spare disks to be used in case of failure of an existing disk.

References

[1] Mark Holland. *On-Line Data Reconstruction in Redundant Disk Arrays.* PhD thesis, Carnegie Melon University, 1994.

[2] The Linux Kernel Sources version 2.2.5.

[3] K Gopinath, Nitin Muppalaneni, N Suresh Kumar and Pankaj Risbood. *A 3-tier RAID Storage System with RAID1, RAID5 and compressed RAID5 for Linux.* FREENIX Track Proceedings, 2000 USENIX Annual Technical Conference, June 21-23, 2000 - San Diego, California.

[4] Nitin Muppalaneni. *Adaptive hierarchical RAID.* Master's thesis, Indian Institute of Science, 1998.

[5] Suresh Kumar Nelluru. *Temperature sensitive storage.* Master's thesis, Indian Institute of Science, 1998.

[6] Pankaj Risbood *Temperature Sensitive Storage for Linux* Master's thesis, Indian Institute of Science, 1999.

[7] Alessandro Rubini. *Linux Device Drivers.* O'Reilly, 1998.

[8] Michael Beck et al. *Linux Kernel Internals.* Addison-Wesley, 1998.

[9] Jim Gray and Andreas Reuter. *Transaction Processing : concepts and techniques.* Morgan Kaufmann, 1993.

[10] R N Williams. *An Extremely Fast Ziv-Lempel Data Compression Algorithm* Data Compression Conference 1991 (DCC'91), 8-11 April , 1991, Snowbird, Utah, pp.362-371, IEEE.

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