# Design, Implementation and Policy Framework for a Linux Based Temperature Sensitive Storage

Venkatesh Pallipadi
*Indian Institute of Science, Bangalore*
venkatesh.pallipadi@alumnus.csa.iisc.ernet.in
K Gopinath
*Indian Institute of Science, Bangalore*
gopi@csa.iisc.ernet.in

## Abstract

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

TSS is a hierarchical storage architecture with frequently accessed data residing in a storage device with faster access time, while less frequently accessed data placed in a device which is more storage efficient. The hierarchy is currently a 3-tier architecture consisting of declustered RAID 1 (RAID1), RAID 5 (RAID5) and compressed RAID 5 (cRAID5) storage devices. The data is dynamically moved 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 the system through the use of RAID architecture.

Simple policies for dynamically managing the data have been implemented as user level daemon processes. They are independent of the low level implementation of the TSS driver. They make use of certain `ioctl()` calls provided by the TSS device driver.

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

## 1 Introduction

With massive computerization and falling price of storage devices, the focus has shifted from simple data storage towards fault-tolerant, efficient and easily manageable storage devices. The increasing importance of highly available web servers, database servers and data mining is another need for such devices.

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. As web servers usually handle large amounts of information, a desirable feature of a storage device storing such information is the efficient use of available storage. 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 that should be up and running without interruptions. 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.

Other useful and desirable properties are the providing of an abstraction of large virtual disks that can be split into smaller ones as needed (while moving towards the goal of attribute based storage), the ability to mix old and new disks with migration from older (less reliable?) (slower?) disks to newer ones automatically and the ability to perform well on write-dominated traffic. The last one is quite important as large caches filter out most of the reads leaving more writes than reads in the traffic for the storage device.

In this paper, we present a Linux based Temperature Sensitive Storage (TSS) that provides a hierarchical storage consisting of RAID1, RAID5 and cRAID5 tiers. TSS gives a unified view of a single storage device with data migrating from one tier to another depending on the access frequency of the data. Thus the frequently accessed data resides in faster 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 but slow tier (cRAID5). Thus the scheme optimizes between performance and storage efficiency. TSS implements different storage personalities as different RAID levels that can sustain single disk failures without bringing the system down. Data consistency across system crashes is taken care of using data logging. Because of the poor performance of partial writes in RAID5 and cRAID5, the hierarchy provides different performance and storage efficiencies while providing approximately the same reliability across the three layers. For read-dominated traffic, RAID1 and RAID5 do not provide the difference in performance that makes the hierarchy useful.

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 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 discusses the previous work done in this area. The major design issues are discussed in Section 3 and Section 4 gives some details about TSS implementation. Section 5 reports the results of the trace driven execution on our experimental setup, followed by conclusions and future work in Section 6.

## 2 Previous Work

Linux 2.2.5 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*[1].

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 does not apply to our system as we only consider disks.

Work on TSS was carried out on Solaris by M. Nitin [3] (RAID1+5) and Suresh N[4] (also cRAID5). A study of the issues in moving TSS to Linux and a prototype implementation was carried out by Pankaj Risbood[5]. The combined work has been reported in [2].

## 3 Design of TSS

### 3.1 Pseudo Device Driver Approach

The TSS functionality can be implemented at either the application level, file system level or device driver level. Considering the relative advantages and disadvantages (see [2] for further details), we have chosen to implement our TSS at the *device driver level*, as a pseudo device driver. The TSS device driver will export the view of a single device while actually implementing TSS internally on a number of devices.

### 3.2 High Level Design

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

- **RAID1, RAID5 and cRAID5 -** The driver code that handles the requests to corresponding RAID devices.
- **Integrated Device -** The driver code that provides a unified view of the multiple RAID device types under it. It also takes care of victimization (RAID1 to RAID5 for example) and promotion (RAID5 to RAID1 for example) of stripes.
- **Data logging -** Data logging is used to keep data consistent across crashes. This is done by using an additional partition for logging, synchronously logging each 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 Measurement 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) has been developed. This gets requests from user level routines, maintains some counters, and in turn does the read/write on the TSS driver.

The user level routines for the TSS device support:

- **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 configuration 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, analyse the information and take decisions for dynamic stripe management, by suitably promoting / victimizing the stripes.
- **Application/Test Programs -** Some user level routines (as application programs) are available to test the functionality of the TSS device. These include routines to read the trace and apply requests to the TSS device. These again use the `read()/write()/ioctl()` interface of the TSS device.

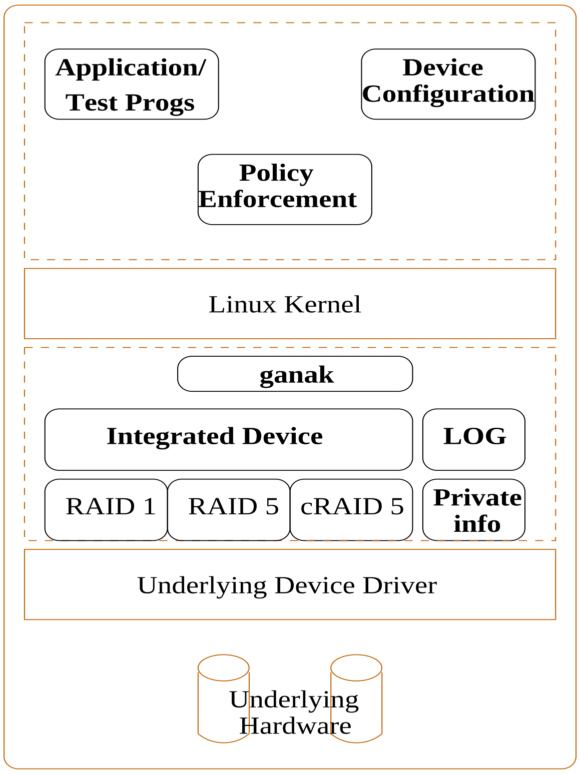


Figure 1: TSS High Level Design

### 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 a 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 unit is kept for parity. We define the *polarity* of the stripe as the index of the

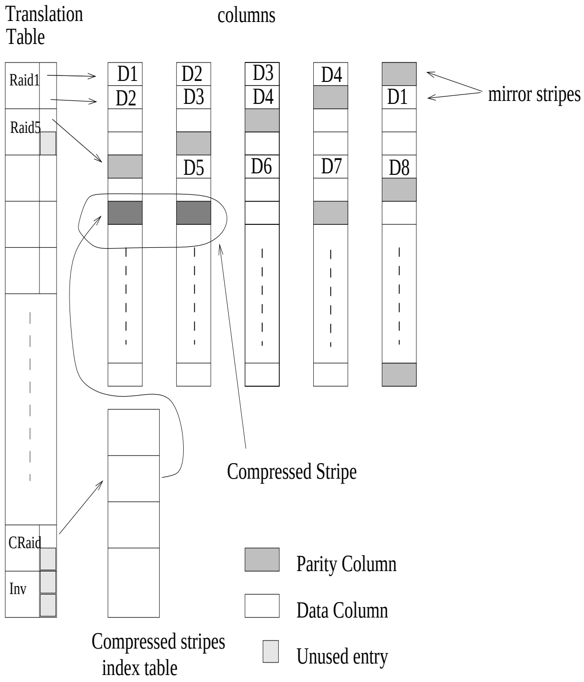


Figure 2: Data Layout

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 - 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 - The logical stripe is mapped to two physical stripes. The two physical stripes must have different *polarity* otherwise the failure of one disk (other than the parity disk) makes 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 RAID1 storage of the total storage capacity of the device. But this makes for a simpler design.

3. **RAID5** stripe - The logical stripe needs only one physical stripe.

4. **cRAID5** stripe - 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* gives 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 physical stripe is smaller than a RAID5 physical stripe due to the use of compression, the bitmaps also should at the sub-stripe granularity. i.e., if the full physical stripe is allocated in terms of equal sub-parts (say, a maximum of 4) 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. The structures and detailed design have been given in detail in [2].

### 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 a *compression table* look-up.

Some of the issues in the design of cRAID5 device are:

- For the cRAID5 device, the physical stripe also needs to be locked when the write on stripe is in progress, so that a write of some other unrelated logical stripe, residing in the same stripe due to compression, should not affect the parity information of the physical stripe.
- The compression algorithm to be used should not be memory and CPU time intensive. Com-

pression and decompression occur inside the kernel so we cannot use any algorithm that uses floating point arithmetic (as the Linux kernel does not save floating point registers during mode switch).

- During a write to a compressed stripe, the new compressed length is usually different. 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 good compression on all data. Sometimes there may not be any compression possible and such cases have to be handled properly. Presently, such stripes are stored with no compression with the stripe length in the compression table indicating a full stripe. This information is used during future reads/writes of the stripe. This can be extended (in future versions of TSS) to cover the cases where the resulting compression ratio is not worth the effort involved in compression and decompression (as specified by an user).

## 3.6 Integrated Device

The design of the integrated TSS device follows the RAID device design framework in [2], upto the stripe level I/O. After this level, the *map table* is used and the corresponding RAID device stripe I/O is then called.

Some 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.
- To promote from RAID5 to RAID1, a free stripe of different polarity should be used. Otherwise, a failure of one disk will lead to the data loss.
- To avoid data inconsistency during promotion / victimization, the changes have to be ordered. For example, during a RAID5 promotion, we have to find a suitable free stripe, copy the data, change the mapping and then the type in the map table in this order.
- If there is no free stripe to victimize a RAID5 stripe to cRAID5, one can victimize it onto the same stripe itself. But this can lead to inconsistent data in case there is a crash during such a victimization. A similar situation arises when we promote cRAID5 stripe in the same physical stripe. To avoid such situations, one stripe is marked as *reserved* and used in above mentioned situations. Thus when there is no other free stripe during a RAID5 victimization, we move the compressed data to the *reserved* stripe, mark the stripe as cRAID5 and then reclaim the previous physical stripe as the new *reserved* stripe, in this 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 gives soft errors, that device should to be noted. If this happens regularly, then the device has to be marked as bad and all I/O to the disk is to be avoided, until the device is reconfigured.

Currently work is in progress on providing suitable failure handling mechanisms for TSS.

## 3.8 The Logging Option in TSS

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, a commit of the transaction

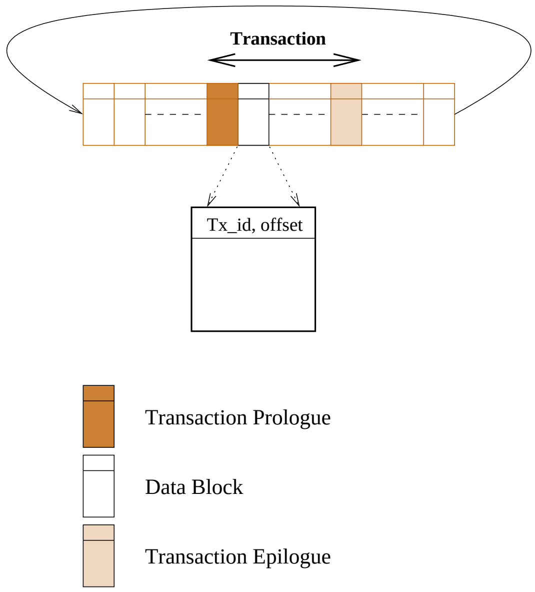


Figure 3: Design of the log

is done on the log device. The design of the log is depicted in figure 3.

Some of the issues in logging are:

- Should the data logging be only targeting the data consistency or 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 at request level, at stripe level and at physical block level. Considering the advantages and the disadvantages of the three approaches, the stripe level logging was chosen as a suitable method for TSS.

- When should the recovery process be initiated? An `ioctl()` was 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 a RAID5 device. The problem: if a part of a write is committed before the crash, it will leave the stripe in an inconsistent state. If a RMW cycle is used during recovery, then some inconsistent data may be read and used during the recovery that would result 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. In reconstructing writes, data is reconstructed from the whole stripe; partially committed data will not affect the recovery and the final data will be consistent.

- What should be done in case of failures in accessing the log device? Logging is used for crash recovery but any redundancy in the log device to avoid the failure of the device is likely to be costly.

### 3.9 Policy Framework

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 extreme situations such as when there is no free stripe for a particular I/O operation. These policy decisions at the kernel level cannot 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. *Temperature* field accumulates the accesses over a period of time. *Access* field indicates whether the stripe has been accessed in the recent past. The *aging* is the process of increasing the *temperature* of all recently accessed stripes, depending on its *access* field. *Uptodate* field also indicates whether the stripe has been accessed in recent past and is used by *degrade* process which reduces the stripe temperature using exponential degradation. We need separate *access* and *uptodate* field, as the aging and degradation can happen at different frequencies. *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
- Do a temperature degradation 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 instant 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 policies that were 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 conform to the corresponding watermarks. If not, migrations of the stripes are performed, so as to conform to the watermarks.

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

2. **Five Minute Rule -** The Five Minute Rule is a way of organizing hierarchical memory, using the performance and the cost considerations, as proposed in [8]. The idea originally proposed for organizing memory hierarchy can be modified and applied to the TSS device as well. Adapting the original formula[8],

   $$\frac{(\frac{Cache\_Cost}{Byte} - \frac{Mem\_Cost}{Byte}) * Obj\_Bytes}{Obj\_Access\_Per\_Sec\_Cost}$$

   that gives the time interval in memory before demotion to disk if there are no accesses in that interval, for the hierarchical TSS device, we get the time interval for RAID levels A and B as

   $$\frac{(\frac{RAIDA\_Cost}{byte} - \frac{RAIDB\_Cost}{byte}) * Obj\_Bytes}{(RAIDA\_acc/s\_cost - RAIDB\_acc/s\_cost)}$$

   The acc/s_cost for individual RAID devices can be got from the simulations and 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.

## 4 Implementation

TSS is currently implemented on the Linux-2.2.5 kernel. The following optimizations / improvements has been done on the basic TSS device driver.

- One of the performance bottlenecks of TSS is the latency due to the synchronous writes over the underlying devices. These writes can be made asynchronous and any failures in the asynchronous writes can be handled by using the existing data logging for the writes. With this modification to the TSS, the cleanup after the completion of asynchronous writes are carried out by a separate cleanup thread.

  These asynchronous writes soon ran into problems of deadlock. The possible reasons for this deadlock could be:

  - Too many asynchronous I/Os consume all available kernel memory and no kernel memory is available for SCSI cmd blocks, etc.

- A large number of asynchronous requests may be generated, which the lower level SCSI device may not be able to handle.
- A large increase in the request queue size, which the underlying SCSI device may not be able to handle.
- Some mysterious bug in TSS code itself!

A simple workaround for the problem has been to limit the number of asynchronous I/Os issued by the TSS device. A maximum of 8 asynchronous requests avoids this deadlock problem.

- The major delays in disk I/O are queuing delay, rotational delay and seek time. Two different optimizations have been investigated to reduce the seek time during the RAID1 I/Os. One approach places the two physical stripes of the single logical stripe, next to each other, if possible. This can be advantageous in case of full stripe writes, as a write has to go through both of the physical stripes. An alternative approach for RAID1 stripe placement is to put the two physical RAID1 stripes halfway apart in the disk. This can reduce the seek time, as during the reads only one of the physical stripes needs to be accessed. These optimizations have to be enabled separately with 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 that has been used in the implementation of TSS device. Linux 2.2.12 or later kernels have optimized these routines further (using xmm registers), and could be used in future versions of TSS.
- During RAID1 reads, only one of the physical stripes will be accessed. The TSS device has an option to route the read requests to any of the two stripes to balance the load on the disks. 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 future read requests and read requests are balanced across the devices. This again has to be enabled with a compile time option.

### 4.1 Changes to Kernel 2.2.5

Even though the device driver framework of the Linux has been attempted to be strictly followed, a few changes have been necessary in the kernel[1]:

- 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.
- Each request structure that gets queued to the TSS device queue will result in requests queued in the underlying device queues. 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 used by the TSS device is set to half of the total possible request structures in the system.

## 5 Performance Evaluation

### 5.1 *ganak* Device Driver

The performance of the TSS device can be measured with a stream of I/Os on it. This can be done through the `read()/write()` system call to the TSS device. However, the problem with this approach is that the measure can be skewed, as it includes user-kernel mode change costs and is also dependent heavily on the system load.

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

### 5.2 Benchmark Data

For our performance evaluation of the Linux TSS system, a subset of HP disk traces [10] have been used. It contains disk accesses generated by HP-UX file systems on a system not running scientific applications.

Further, in order to reduce the time taken to execute a trace through the system, *Trace Compaction* (as in [3]) has been done. Trace compaction reduces the time between the accesses by a factor of 1000. It was found in [3] that the trace compaction has little impact on the results but speeds up the performance evaluation process. Also, a part of the actual trace containing about 70,000 accesses was used for the trace simulation.

### 5.3 Experimental Set-up

Performance measures have been taken on a system *anant* with Linux-2.2.5 kernel and Red Hat Linux release 6.0. The machine is a Pentium 75MHz with 32MB RAM. The 3 SCSI Disks, 1.2GB capacity each (model - Quantum Fireball 1280S), were connected using 10Mbps SCSI bus with Adaptec controller (Model - Adaptec AHA-2940 SCSI host adapter) and used to configure different devices for the TSS device. Experiments were done when the system was on light load (load avg $< 1$).

### 5.4 Experimental Results and Analysis

#### 5.4.1 Synchronous and Asynchronous writes

The effect of asynchronous writes on the performance of TSS device can be seen by comparing the RAID1, RAID5 and cRAID5 performance with synchronous and asynchronous writes (with max 8 async writes at a time). This results are summarized in Table 1.

The results indicate a significant improvement in performance of TSS, due to asynchronous writes in case of RAID1 and a moderate improvement in case of RAID5 and cRAID5. This improvement is achieved due to the reduction of waiting time through the use of asynchronous writes. In case of asynchronous writes, we have a separate thread running, which does the cleanup for each of the request. The time taken by this cleanup operation is not included in the results, as they happen asynchronously and they do not form a part of response time for the requests.

| Type of Device | Avg. write time per block (ms) | | Avg. I/O time per block (ms) | |
|---|---|---|---|---|
| | sync writes | async writes | sync writes | async writes |
| RAID1 | 17.23 | 8.79 | 12.66 | 7.43 |
| RAID5 | 21.98 | 19.32 | 14.87 | 13.47 |
| cRAID5 | 94.87 | 74.36 | 65.90 | 54.00 |

Table 1: Effect of asynchronous writes on the performance of TSS (trace data from 6 [/var])

All of the following trace runs have been taken with async writes enabled in TSS.

#### 5.4.2 RAID1 physical stripe placement

Two different physical stripe allocation polices have been tried out on a RAID1 device. One policy allocates successive physical stripes for a logical RAID1 stripe, if possible. Another policy allocates physical stripes of a logical stripe separated be some distance, if possible. The performance of TSS under the two policies is summarized in Table 2.

The results show that allocating the physical stripes apart performs better. This can be due to the fact that, with physical stripes away from one another, the successive logical stripes can be successive physical stripes also. Due to this the clustering can happen across the stripes at the lower level drivers (SCSI) both during reads and writes.

All of the following trace runs have been taken with two RAID1 stripes allocated half-way across on the physical devices.

#### 5.4.3 Performance results of TSS device

Performance results of TSS with different configurations is summarized in Table 3. Trace runs of TSS was also done with different policies and the results summarized in Table 4. Note that we are not comparing our results with the existing RAID

| Type of Device | Type of I/O | Avg. time per block (ms) | |
|---|---|---|---|
| | | cont. placement | fixed dist. placement |
| RAID1 | READ | 5.42 | 4.94 |
| | WRITE | 8.79 | 8.45 |
| | Avg. I/O | 7.43 | 7.04 |

Table 2: Effect of physical stripe placement on RAID1 device TSS (trace data from 6 [/var])

in Linux, as our main aim here is to consider the advantages/disadvantages of TSS with/without policies and not comparing the different static RAID systems.

From Table 3 we find that the access times in RAID0 is the least, and cRAID5 is the highest. This result is on the expected lines. Notable result here is RAID5 to RAID1 ratio in terms of access time is around 2 and in terms of space is 0.5. cRAID5 access time is around four to five times that of RAID5, and uses around 60% of the space used by RAID5.

RAID0 is just 1.2 times faster than RAID1, mainly because RAID0 does not have the benefit of asynchronous writes in it. RAID1 performance matches with that of RAID0, even with additional redundancy and logging features in it.

RAID5 access times is nearly twice that of RAID1. This can be attributed to:

- RMW cycles for short writes
- RAID5 has an additional parity calculation time compared to RAID1.
- Due to asynchronous writes, there is no time spent waiting for the disks to complete the writes. So, the additional write request in case of RAID1 does not reduce performance.

cRAID5 access times is much higher than RAID5. This can be attributed to:

- Compression / Decompression of the whole stripe needed even for small data requests.
- cRAID5 involves lot of copying from buffer cache to page cache, during compression / decompression. This is due to the fact that the page cache and buffer cache are separate in Linux-2.2.5.

These comparative figures gives good indications for setting the policy parameters in an integrated TSS device. On the basis of performance, the RAID1 and RAID5 parts should have around the same number of stripes and cRAID5 a smaller number of stripes. This also can vary depending on other things like available free space in the device.

The trace runs with watermark policy was carried out with the following watermarks:
**RAID1 -** Lower Watermark 20-30%, Upper 50-60%
**RAID5 -** Lower Watermark 25-35%, Upper 55-65%
**cRAID5 -** Lower Watermark 5-10%, Upper 20-25%

The trace run with the five minute rule[8] was done with RAID1-RAID5 transition frequency as 20 seconds and RAID5-cRAID5 transition frequency as 350 seconds. The actual values got from five-minute rule formula have not been used in case of trace runs, as the execution does not happen in real time but compacted in time. So the actual RAID1-RAID5 and RAID5-cRAID5 transition frequencies from five-minute rule had to be reduced by a large factor.

TSS with watermarks has access time more than that of normal RAID5. This is mainly due to the static nature of watermark policy. If all watermarks are conforming, and no new blocks are accessed, then it does not do any more migrations, and the temperatures of the stripes are not used at all. This can result in a severe performance penalty.

TSS with five-minute rule has access time in between that of RAID1 and RAID5. This improvement in performance compared to watermarks is mainly due to dynamic stripe placements that happen on a continuous manner depending on the temperature.

## 6 Conclusions and Future Work

### 6.1 Conclusions

- TSS can provide storage with the required features of efficiency of storage, speed and reliability.

| Disk No. | Percentage of Writes | Type of Device | Read Time per Blk (ms) | Write Time per Blk (ms) | I/O Time per Blk (ms) | Space overhead |
|---|---|---|---|---|---|---|
| 5 (/usr) | 59.83 | RAID0 | 4.40 | 7.17 | 6.70 | 100% |
| | | RAID1 | 5.89 | 8.95 | 8.44 | 250% |
| | | RAID5 | 4.94 | 20.91 | 18.22 | 125% |
| | | cRAID5 | 29.74 | 106.81 | 93.84 | 87.2% |
| 6 (/var) | 83.17 | RAID0 | 4.30 | 6.98 | 5.90 | 100% |
| | | RAID1 | 4.94 | 8.45 | 7.04 | 250% |
| | | RAID5 | 4.76 | 19.32 | 13.47 | 125% |
| | | cRAID5 | 23.70 | 74.36 | 54.00 | 87.5% |

Table 3: Performance of RAID devices through TSS

| Disk No. | Percentage of Writes | Type of Device | Read Time per Blk (ms) | Write Time per Blk (ms) | I/O Time per Blk (ms) | Space overhead [2] |
|---|---|---|---|---|---|---|
| 5 (/usr) | 59.83 | TSS with watermarks | 6.41 | 28.03 | 24.39 | 205% |
| | | TSS with 5 min rule | 6.06 | 13.35 | 12.12 | 237% |
| 6 (/var) | 83.17 | TSS with watermarks | 7.77 | 26.32 | 18.87 | 190% |
| | | TSS with 5 min rule | 5.43 | 14.53 | 10.87 | 225% |

Table 4: Performance of TSS device with policies

- Performance of TSS device with simple policies is in between RAID1 and RAID5 performance. More intelligent policies, like one using predictions, are likely to achieve better results.
- The Linux device driver interface is not yet very suitable for implementing a layered devices. Even with the module interface, we had to make changes in the kernel for our module to run.

### 6.2 Future Work

- Implementation of the failure handling routines in the TSS device.
- More intelligent policies such as predictions can be implemented, using information about past disk accesses.
- Extend the design of TSS in a networked or SAN environment.
- Port TSS onto linux 2.4 and release TSS as open source to the Linux community for general use and future enhancements.

**Acknowledgments:** 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 and John Wilkes of HP Labs for providing the traces used in our experiments. Financial support from Veritas Software, Pune is also gratefully acknowledged.

## References

[1] The Linux Kernel Sources version 2.2.5.

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

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

[4] Suresh Kumar Nelluru. *Temperature Sensitive Storage.* Master's thesis, Indian Institute of Science, 1998.

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

[6] A. Rubini. *Linux Device Drivers.* O'Reilly, 1998.

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

[8] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[9] 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.

[10] Chris Ruemmler and John Wilkes. *UNIX Disk Access Patterns.* Proceedings of the Winter 1993 USENIX Conference, San Diego, CA, January 1993.

.

[1] Static RAID systems here refer to the standard RAID1 and RAID5 systems.

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