Implementation of a Fault-Tolerant Real-Time Network-Attached Storage Device

Ashish Raniwala, Srikant Sharma, Anindya Neogi, Tzi-cker Chiueh
Experimental Computer Systems Laboratory
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{ashish, srikant, neogi, chiueh}@cs.sunysb.edu
tel +1-516-632-8436
fax +1-516-632-8334

Abstract

Phoenix is a fault-tolerant real-time network-attached storage device (NASD). Like other NASD architectures, Phoenix provides an object-based interface to data stored on network-attached disks. In addition, it features many functionalities not available in other NASDs. Phoenix supports both best-effort reads/writes and real-time disk read accesses required to support real-time multimedia applications. A standard cycle-based scan-order disk scheduling algorithm is used to provide guaranteed disk I/O performance. Phoenix ensures data availability through a RAID5-like parity mechanism, and supports service availability by maintaining the same level of quality of service (QoS) in event of single disk failures. Given a spare disk, Phoenix automatically reconstructs the failed disk data onto the spare disk while servicing on-going real-time clients without degradation in service quality. Phoenix speeds up this reconstruction process by dynamically maintaining additional redundancy beyond the RAID5-style parity on the unused space left on the disks. Phoenix attempts to improve the reliability of the disk subsystem by reducing its overall power consumption, using active prefetching techniques in conjunction with disk low-power modes. This paper describes the design and implementation details of the first Phoenix prototype.

1 Introduction

An emerging network file system architecture, called *Network-Attached Storage Device* (NASD) architecture, separates the processing of metadata such as access permission check and file directory lookup, from actual data movement between disks and client machines. Storage devices that are directly attached to the network off-load the data movement processing burden from network file servers, and thus improve the overall system scalability. This architecture contrasts with the conventional network file-systems in which there is no separation of metadata processing and data-storage. In NASD architecture, clients still send their access requests to network file servers, which after necessary checks and translations

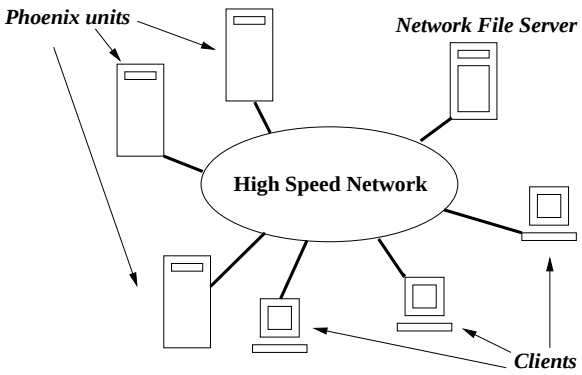

Figure 1: *Instead of attaching disk drives to the backplanes of the network file server machines, the NASD architecture uses storage devices that can be directly attached to high-speed LANs, and thus is able to exploit the aggregate bandwidth on the LAN for data transfers between disks and client machines.*

return cryptographically secure object capabilities. From this point on, clients use object capabilities to directly access the data residing on network-attached storage devices without involving network file servers. By distributing the bandwidth-intensive data transfer function across the network, the NASD architecture becomes more scalable than traditional server-attached storage architecture, both with the number of client machines as well as with the increasing link speed of the LANs. Figure 1 shows how NASD devices interact with client machines directly. *Phoenix* units constitute the storage system part of the NASD architecture. The complete NASD architecture is realized by augmenting *Phoenix* units with a file-server.

Phoenix is a Linux-based network-attached storage device built from off-the-shelf PC hardware, Fast Ethernet adapter and a set of Ultra-SCSI disks. *Phoenix* supports the following features:

- An object-based SCSI-like API.
- Bandwidth-guaranteed disk access, which is essential to real-time multimedia applications, *e.g.* MPEG streams in video-server applications.
- Both real-time disk reads and best-effort disk reads/writes.
- QoS guarantees that remain valid across single disk failures, specifically reconstruction of the contents of the failed disk onto a new disk while maintaining QoS for the existing streams.
- Utilization of unused space on disks to speed up the reconstruction process.
- Active prefetching and use of disk low-power mode to reduce disk failure probability.

This paper presents the detailed design and implementation decisions that went into the construction of the first *Phoenix* prototype. Section 2 reviews related projects in the area of NASD. Section 3 describes the data access interface which *Phoenix* provides to its client

applications. Section 4 presents an overview of the design of *Phoenix* and its major architectural features. Section 5 discusses in detail the implementation of the *Phoenix* prototype under Linux. Section 6 discusses optimization features implemented in *Phoenix*. Section 7 reports performance measurements from the first *Phoenix* prototype. Finally, we conclude with a summary of major innovations in *Phoenix* and an outline of the planned work in Section 8.

2 Related Work

One of the early systems that adopted the idea of network-attached storage device is the RAID-II system built at Berkeley [1]. The focus of this work was to address the bus/memory bandwidth limitations of the disk array's host machine, by moving data directly between the network and the disks with minimal host involvement. Katz [2] discussed the concept of network and channel-based storage systems where networking and storage access are tightly integrated as a single entity. van Meter [3] provided a survey on the research areas of network-attached peripherals and the impacts of such devices on operating system design. Petal [4] uses a set of block-level storage servers that collectively manage a large set of physical disks to provide clients the abstraction of distributed virtual disks that tolerate and recover from disk, server and network failures. Frangipani [5] is a distributed file system that is built on top of Petal's distributed virtual disk service to provide scalable network file service. GFS [17] aims at providing a serverless file-system that integrates network attached storage and fibre-channel-based storage area network. This setup provides client computers full access to all storage devices on the network resulting in higher data availability.

The idea of separating high-level file system processing from low-level storage management opened up the possibility of customized optimization for file metadata processing and file data movement. The NASD project at CMU [6, 7] focused on the reduction of the file server load by providing clients an object-based access interface, which is more general and flexible than the file-based and block-based interfaces supported by file systems and disk devices, respectively. This project also addressed the important security issues in the NASD architecture. More recently, projects at U.C. Berkeley [8], CMU [9] and University of Maryland / U.C. Santa Barbara [10] all explored the idea of performing a limited form of computation inside disk drives to improve the overall system performance by reducing the data traffic between disk devices and clients. Similar ideas have been used to improve the efficiency of the disk storage system itself rather than that of the clients, for example, HP's AutoRAID system [11].

There have been several real-time storage server projects such as SUNY Stony Brook's SBVS [12], Microsoft's Tiger server [14], Starlight's StarWorks [13], and IBM's Tiger Shark parallel file system [15]. All the above systems took the more traditional network file system architecture rather than the NASD architecture. Some of these enhanced their scalability by deploying a clustered system architecture, but all data transfers had to go through the file servers.

Power management by reducing disk power consumption has been studied for mobile computers [20, 21], however, the primary goal there is to extend the battery life. Similar ideas applied to NASD can reduce heating effects by optimizing power consumption, potentially increasing the reliability of the disk subsystem [22].

Phoenix is heavily influenced by SBVS in terms of its overall architecture and internal design. It is one of the first, if not the first, NASDs that support fault-tolerant real-time object-based accesses. It provides high level of service availability as well as data availability. It also attempts to improve the reliability of the overall system by use of prefetching techniques. In addition, it supports both *server push* and *client pull* file accesses to accommodate the requirements of distributed multimedia applications.

3 Data Access Interface

The programming abstraction exposed to the clients by a *Phoenix* device is a set of logically contiguous objects whose internal structure such as disk layout is completely hidden from user applications. Clients may create, delete, access and modify objects. Each object has associated attributes like object-id, size, etc. The mapping from files and directories to objects is performed by a separate machine that serves as a network file server.

Phoenix supports both best-effort and real-time bandwidth-guaranteed disk accesses. The clients specify the data items of interest via a tuple: a unique object identifier, a block offset within the object and the number of blocks. For real-time disk accesses, an additional parameter, the bandwidth requirement in terms of 4K blocks/sec, must be specified. Clients can access data in either the *client-pull* or *server-push* mode. In the *server-push* mode, data may build up and thus exhaust buffers on the client side due to software/hardware glitches or mismatches in disk/network bandwidth scheduling granularities. To address this problem, *Phoenix* supports a general `skip` command interface with which a client application could request the *Phoenix* server not to send any data for N cycles, where N is a user-supplied parameter.

Table 1 summarizes the list of commands supported by *Phoenix*. `createsp` is used to create *special objects* at installation time and is the only one that cannot be done remotely. Executing this command is similar to creating a partition table on a fresh disk. Special objects maintain metadata information about a *Phoenix* device and the objects it contains. User objects are created and deleted with `create` and `delete`. Attributes of an object are set and read with `setattr` and `getattr` commands. All clients, real-time as well as non-real-time, use the `read` command to read data objects. The `type` parameter can have values `server-push`, `client-pull` and `best-effort`, denoting the mode of data access. For real-time clients in the client-pull mode, `read` command performs just the initial set-up for reads. To actually read the data in the object, they use the `pull` command. Data is written to an object using the `write` command. An object's size has to be declared in advance and cannot be changed dynamically. However, this restriction is not important

Command	Parameters	Return Value
`createsp/create`	attributes, perms	objid/status
`delete`	objid, perms	status
`read`	objid, offset, range, rate, perms, type	strmid/status
`write`	objid, offset, range, perms, data	status
`getattr`	objid, perms	attributes/status
`setattr`	objid, attribute name, value	status
`pull`	strmid, range, perms	data/status
`skip`	strmid, cycles, perms	status
`getdeviceinfo`	perms	deviceInfo
`shutdown/bootup`	perms	status

Table 1: *The set of commands supported by* Phoenix*, their arguments and return values.*

because a conventional file is organized as a chain of objects with new objects added on file growth. Commands `shutdown` and `bootup` perform the remote shutdown and bootup of a *Phoenix* system.

4 Phoenix System Architecture

4.1 Basic Design

In *Phoenix* each storage object is striped across a software-controlled disk array in a sequentially interleaved fashion, with a RAID-5 style of parity to protect data against single disk failures. Two *special objects* keep the metadata about a *Phoenix* device and individual objects on the device. The *DeviceInfo* object contains the device type, capacity, free space, block size, permissions, the starting location and size of the *ObjectList* object, etc. The *ObjectList* object contains a list of attributes for each object striped on the disk array including its size, starting offset, permissions, etc. *Phoenix* uses a fixed stripe unit size of 4 KBytes, which is independent of objects and the requested access rates to them.

Phoenix uses a cycle-based disk scheduling algorithm to provide disk bandwidth guarantees. In each I/O cycle, *Phoenix* retrieves from disks an amount of data for each real-time stream corresponding to its bandwidth reservation. Within an I/O cycle, initially real-time disk access requests are serviced in the scan order based on blocks accessed from the disks, and then the best-effort access requests are served in a partial scan order (explained in section 5.2). This ordering reduces the disk head seek overhead, simplifies the scheduling of non-real-time accesses and also makes it possible to perform I/O cycle utilization measurements required for admission control. A fixed percentage of the I/O cycle is reserved for best-effort traffic to guarantee that best-effort requests never starve. An explicit dynamic measurement-based statistical admission control, similar to the one used in SBVS, ensures that *Phoenix* can admit as many requests as possible while meeting the QoS guarantees to its clients. To maintain the continuity of data flow, *Phoenix* employs a double buffering scheme where the disk subsystem fills up one set of buffers with data while the other set is being emptied out onto the network.

4.2 Failure-Tolerant Real-Time Disk Service

An innovative feature of *Phoenix* is its ability to maintain the QoS guarantee to real-time clients across single disk failures. In contrast, conventional disk arrays put more emphasis on data availability and render the disks' service unavailable during the failure recovery period. *Phoenix*, on the other hand, continues to provide guaranteed disk bandwidth to real-time applications by treating reconstruction-related disk accesses as best-effort traffic.

Disk failures are detected by associating a timeout with each request issued to the disk array. On failure detection, *Phoenix* switches to *failure* mode. In failure mode, the reads which should be served by the failed disk are redirected to the corresponding block on the parity disk. After reading a complete stripe group, *Phoenix* re-builds the block on the failed disk through parity. The parity computation leads to an increase in the I/O cycle time. However, the parity computation is partially overlapped with the disk accesses to improve the performance.

4.3 Failure Recovery

While in failure mode, *Phoenix* sends periodic SCSI *inquiry* commands to detect the existence of spare disk. On successful detection, a switch is made to the *recovery* mode. To shorten the recovery phase, *Phoenix* denies all best-effort access requests in this mode. During the recovery period, a dummy stream called *reconstruction stream* is started to reconstruct the data of the failed disk onto the spare disk by making use of parity. Disk I/Os associated with the service of the client real-time streams also computes the portions of data on the failed disk using parity. The question is whether to write such computed data back to the spare disk (called the *piggyback* approach) or not (called the *non-piggyback* approach). Experiments with both the approaches were conducted and finally the *piggyback* approach was chosen for implementation [19]. The *piggyback* approach reuses the efforts involved in servicing real-time streams to do the disk reconstruction.

5 Implementation

5.1 Hardware Components

The first *Phoenix* prototype has been implemented on a PentiumPro 200-MHz PC with 128 MBytes of physical memory. The prototype has a 1-GByte IDE disk to hold the *Phoenix* kernel, swap space, and basic utilities programs. In addition, it is connected to an array of five Seagate ST34371W 4-GByte Ultra Wide SCSI disks physically mounted within an external disk case via an Adaptec 2940 Ultra-Wide SCSI adapter sitting on a 33-MHz PCI bus. Data is striped across the SCSI disk array, with a striping unit of 4 KBytes (which is also the minimum retrieval size for all disk accesses) and one of the disks designated as the parity disk. The prototype is connected to a switched Fast Ethernet through an Intel PRO/100+ PCI adapter.

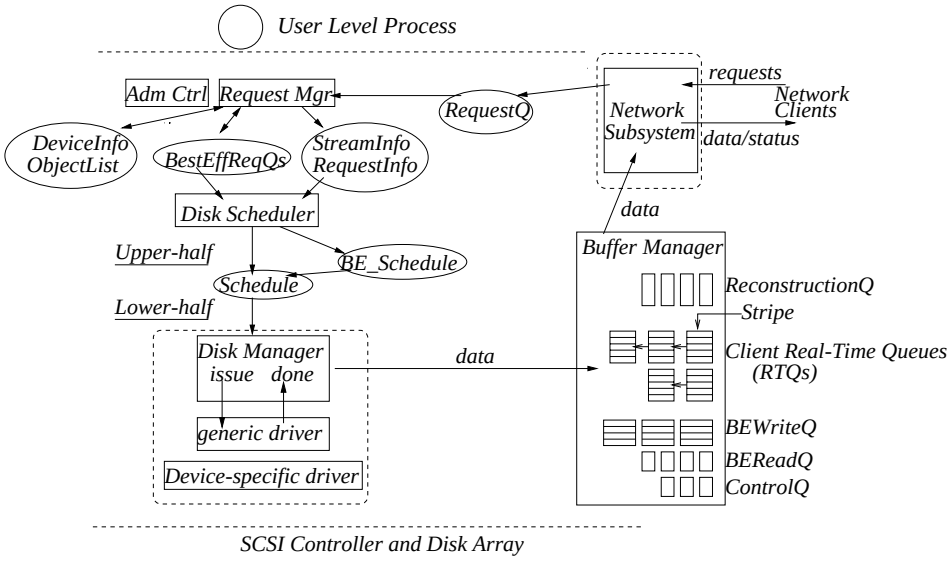

Figure 2: *Software Architecture of Phoenix depicting various modules, data structures and interactions between them. The arrows depict the basic data flow.*

5.2 Software Architecture

The *Phoenix* kernel is based on Linux 2.2.12. The interaction between the *Phoenix* subsystem with the Linux kernel is limited to memory management for allocation of buffers, scheduling of timers, kernel socket code for the network subsystem, and the generic SCSI controller driver for sending SCSI commands to the disk. The device-specific portion of the SCSI driver was left untouched. Because of the modular software architecture, it is expected that porting *Phoenix* to other hardware/OS platforms and Linux versions should be relatively straightforward.

The software architecture of the *Phoenix* kernel is shown in figure 2. *Phoenix* kernel code is activated by a startup user-level program that makes a system call with some configuration parameters. From this point onwards, *Phoenix* remains in the kernel mode. The kernel consists of a timer-driven upper-half which comprises the disk scheduler, the request manager and the admission controller, and the disk interrupt-driven lower-half comprising the low-level disk manager. The buffer manager supports other subsystems. The network subsystem is a timer-driven module that is invoked once every network cycle [16] to send data to the clients, and to accept new requests from the clients.

To implement cycle-by-cycle disk scheduling, the upper-half is invoked once every I/O cycle to prepare the disk schedule for every disk and initiate the lower-half to start disk request processing. Thereafter, the lower-half issues the next disk request from the SCSI callback function once the previous request finishes, until the access requests for all the disks are completed.

Since the disk scheduler can not determine in advance the number of non-real-time requests

to be scheduled for service, it prepares a separate schedule, called best-effort-schedule, for these requests. Once the lower-half is done with the real-time requests for an I/O cycle, it invokes the best-effort scheduler to dynamically schedule requests from the best-effort schedule. Best-effort access requests that remain unserviced at the end of the current I/O cycle are processed in later I/O cycles. New best-effort requests are added to the best-effort schedule in scan order after these left-over requests. The ordering of left-over requests is not altered to avoid their starvation. Thus, the set of requests arriving within an I/O cycle are put in scan order and those arriving across I/O cycles are put in FIFO order. This ordering is termed as *partial scan order*.

5.2.1 Data Structures

`StreamInfo` and `RequestInfo` lists maintain the information regarding the on-going real-time streams, and best-effort read and write requests. The `DeviceInfo` and the `ObjectList` structures are in-memory copy of the *DeviceInfo* and *ObjectList* stored on the disks (refer to section 4.1). `RequestQ` is used by the network subsystem to queue new client requests. The corresponding `ReplyQ` is the `ControlQ` of the buffer manager. After processing the best effort requests, the request manager queues them up in the `BestEffReqQs` which are then picked up by the disk scheduler. `Schedule` is the schedule prepared by the upper-half to be used by the lower-half in the next disk I/O cycle. `BE_Schedule`, also prepared by the upper-half, is used to hold the best-effort requests scheduled to be sent to the disks. The various queues maintained by the buffer manager are discussed in section 5.2.5. The network subsystem, the upper-half and the lower-half all are executed from bottom halves of the timer or disk interrupt service routines. Since no two bottom-halves can execute concurrently, the consistency of any shared data structure among bottom-half processing modules is guaranteed.

5.2.2 Admission Control

The admission control module implements a measurement-based statistical admission control algorithm to determine whether to admit a new real-time stream. The module exports `admit_stream()` function which uses the following equation to predict the total service time after admitting the new $(N+1th)$ stream based on the past service time measurements for the on-going N streams.

$$\begin{aligned} \texttt{Pred\_Service}_{N+1} &= \texttt{Current\_Service}_N + \texttt{Std\_dev}_N + \texttt{Increase\_Seek\_Time} + \\ &\quad (\texttt{Current\_Service}_N * (\texttt{Requested\_Rate}/\texttt{Total\_Rate}_N)) \end{aligned}$$

$\texttt{Std\_dev}_N$ is the standard deviation from the current service time (averaged over past few I/O cycles), $\texttt{Current\_Service}_N$, for N streams. `Increase_Seek_Time` is the increase in the seek time per I/O cycle if the new stream is admitted. $\texttt{Total\_Rate}_N$ is the summation of rates of all the on-going real-time streams. If the predicted service time for $N+1$ streams is less than the I/O cycle share reserved for real-time streams, then the new stream is admitted, otherwise it is rejected. While *Phoenix* is operating in the failure or recovery mode, the admission control simply rejects all new stream/best-effort read/write requests.

5.2.3 Request Manager

This module exports the function `process_requests()`, which takes client requests from the `RequestQ` and processes them based on their types. Table 1 lists all the possible requests. All requests are first validated. `createsp`, `create`, `delete` and `setattr` involve updating the `DeviceInfo` and `ObjectList` data structures. `getattr` and `getdeviceinfo` just access these structures for sending information to the clients. Best-effort `read` and `write` requests are processed and queued in the `BestEffReqQs` list and an entry is made in the `RequestInfo` list. Client `Write` requests are broken into stripe group writes and for each such stripe group write, `OLD_DATA_READ` and `OLD_PARITY_READ` requests are put into the `BestEffReqQs`. These reads in turn trigger the actual writes. The parity block is read to keep it updated with new block writes.

Every real-time `read` request is validated by the admission control and then an entry is inserted into the `StreamInfo` list. `pull` and `skip` simply update a counter in the `StreamInfo` structure, which is periodically checked by the network subsystem to decide whether to send data to the client or not. `shutdown` closes down *Phoenix* by writing the in-core copy of the `DeviceInfo` and `ObjectList` structures to the disks, cleaning up all required data structures. `Bootup` initializes *Phoenix* by reading the disk-copy of these data structures into memory.

5.2.4 Disk Scheduler

The disk scheduler exports the function `update_schedule()` which prepares the next I/O cycle's disk schedule to be served by the lower-half. The disk scheduler first puts the real-time requests in the `Schedule` data structure. It reads `StreamInfo` structure to retrieve the rate and current pointer information for on-going real-time streams. For each real-time stream, the disk scheduler schedules `2*data_rate-unconsumed_buf_size` amount of reads (rounded off to complete parity groups). To reduce disk seek overhead, the disk scheduler tries to use a retrieval size as close to the maximum retrieval size (64 KBytes) as possible.

Unlike real-time requests, the exact number of non-real-time requests which will make the I/O cycle utilization optimal can not be pre-determined. To handle this, the disk scheduler fills up enough non-real-time requests in a separate `BE_schedule` in a partial scan order. When all real-time requests scheduled in an I/O cycle are completed, the disk scheduler invokes `get_next_BE_request()` to get the next best-effort request from the `BE_schedule` into the `Schedule`. This allows the lower-half to get as many non-real-time requests as it can serve and thus keep the I/O cycle optimally utilized.

In failure mode, the disk scheduler shifts the requests which should be served by the failed disk to the parity disk. It also puts `inquiry` commands in the disk schedule to probe pre-configured I/O locations to detect if a spare disk is available. On detection of a spare disk, the system switches to *recovery* mode. In the recovery mode, the disk scheduler schedules reads associated with data reconstruction, which in turn trigger reconstruction-related

writes. The ranges of the disk blocks which the current streams are accessing are stored in the AutoReconstRanges and are termed as active blocks. The reconstruction of active blocks is piggybacked with the continual service of real-time streams. The disk scheduler schedules stripe-group reads for reconstruction of inactive blocks (not accessed by the existing real-time clients). Once the reconstruction of such inactive blocks is over and that of active blocks is not complete, the reconstruction stream starts with the reconstruction of the active blocks.

5.2.5 Buffer Manager

Figure 2 shows various queues maintained by the buffer manager. Each on-going stream has an RTQ structure, which is allocated using allocate_rtq(). This points to the linked list of the data buffers (shown in figure). Each node in the list can store a complete parity group, i.e., num_disks*retrieval_size bytes. These buffers are allocated using bmgr_get_buffer(). RTQ[0] is a special stream used to store the data read by the reconstruction stream. Write requests are allocated write buffers linked in the BEWriteQ using bmgr_allocate_write_buffer(). Each such buffer has 4 parts - OLD_DATA, NEW_DATA, OLD_PARITY and NEW_PARITY. Best-effort reads are allocated a data buffer linked in the BEReadQ. ReconstructionQ stores the reconstruction data to be written to the spare disk. Both the best-effort read buffers and the reconstruction buffers are allocated using bmgr_get_buffer() routine. ControlQ stores the control messages for the clients.

5.2.6 Disk Manager and Generic SCSI Driver

The disk manager exports the dmgr_start() function, which is called by the upper-half to trigger the next disk I/O cycle. This function issues the first set of requests to all the disks and then immediately returns. The completion of these requests is indicated by a call to scsi_done() which is the main part of this module. scsi_done() issues the next request and processes the reply received from the disk.

Issuing a disk request involves getting the next request from the disk schedule based on diskid and slotid of the reply, constructing the next command to be sent to the disk, and then sending out the actual disk request (with an associated timeout to detect possible disk failures). **Processing a reply** involves checking the reply for error_codes, switching to failure mode in case of disk failure detection, and then performing further reply processing termed as *post-processing*. Switching to failure mode involves schedule recomputation for the current I/O cycle where the real-time reads scheduled for failed disk are shifted to the parity disk. The next request is issued just before the *post-processing* stage so as to overlap the post-processing (e.g. parity computation, etc.) with the next disk access. The post-processing constitutes queueing up the required buffers with the buffer manager, updating relevant data structures, performing parity computations if required and queueing up further disk requests.

A subtle change was made to the disk request issue scheme described above based on reconstruction experiments. The request processing rate is not uniform across the disks. Thus, some disk might have processed more reconstruction-read requests than other ones, leading to accumulation of buffers. To avoid this buffer accumulation, *Phoenix* tries to balance the number of reads across the disks. In effect, in every I/O cycle, the first set of requests to the disks is sent out in the reverse order of the length of the disks' pending requests queues.

Another technique used to ensure uniform disk service progress is to slow down the *leading disk*. The *leading disk* is defined as the disk which has processed maximum number of requests from its schedule. This *leading disk* keeps changing dynamically as an I/O cycle proceeds. No more requests are sent to this disk as long as it remains the *leading disk*. Since 4 disks are sufficient to optimally utilize an UltraWide SCSI bus, not scheduling the fifth disk does not have significant impact on performance. In a typical setting, the number of *leading disks* can be determined based on supported SCSI bus bandwidth, the total number of disks and the difference between the processing rates of the disks. The main concept is to avoid request scheduling for the disks which are going faster than the other disks without affecting the overall performance.

5.2.7 Network Subsystem

The function `do_net_io_cycle()` of the network subsystem is invoked once every network cycle. It looks at the `StreamInfo` and `RequestInfo` structures, dequeues data from the buffer manager and sends it to the clients. It also sends them the new control messages queued in the `ControlQ` of the buffer manager. It processes the new client requests and puts them in the `RequestQ`. This module works by making socket-layer system calls from within the kernel to send out UDP packets over the network. This subsystem is relatively independent of the rest of the system and can be fairly easily replaced by other real-time network subsystems, e.g., Rether [16].

6 Optimization Features

6.1 Dynamic Replication to Reduce Reconstruction Time

To reduce the data reconstruction time, *Phoenix* employs a *dynamic replication* scheme that uses unutilized storage space in the disk array to *mirror* a part or all of the utilized portion of the disk array. The extent of the disk array's utilized portion that gets replicated depends on the size of the unused space. Here, a disk array consists of three parts *viz.* utilized & mirrored (UTM), utilized & parity-protected (UTP), and unutilized & mirrored (UUM) (figure 3). Both the UTM and UUM are reconstructed via $1:1$ reads and writes, whereas the UTP portion is reconstructed via $(N-1):1$ reads and writes, where N is the parity group size, plus a parity computation. The mirroring scheme chosen, called *Declustered Replication*, distributes the replication for each disk across all other disks to increase read parallelism. To minimize seek overheads, replication unit is chosen to be the disk's maximum retrieval size (64 KBytes).

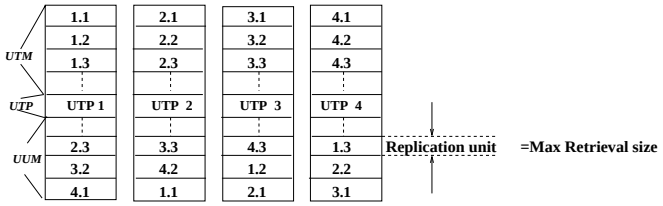

Figure 3: *Dynamic Declustered Replication. Each disk in the array is partitioned into 3 parts: UTM, which is reconstructed from the mirrored copy replicated across other disks, UTP, which is reconstructed from parity, and UUM, which stores mirror copies of other disks' UTM portions. UTM and UTP together represent the part of a disk that is being utilized.*

To implement dynamic replication, additional replication writes are scheduled for client writes to maintain the replication consistency. During the reconstruction phase, the disk scheduler tries to use the existing mirror copy to reconstruct the data. `ReconstructionQ` is used to temporarily store this data. The reconstruction of UTM and UTP is done in parallel to ensure optimal performance. Also, excessive UTM reads in short time may lead to write buffer accumulation and are therefore thwarted appropriately. Reconstruction related measurements on *Phoenix* prototype indicate significant performance gains achieved by the use of this approach. The benefits are expected to increase further as the number of disks in the parity group increases.

6.2 Active Prefetching to Lower the Power Consumption

To reduce the probability of disk failures due to overheating [22], *Phoenix* tries to reduce the overall power consumption of the disk subsystem. *Phoenix* employs an *active prefetching* technique by exploiting real-time applications' regular data access patterns. Rather than leaving the unused bandwidth in each I/O cycle wasted, *Phoenix* uses the spare bandwidth to prefetch data for each real-time stream, in order to skip some I/O cycles every once in a while. In these skipped I/O cycles, *Phoenix* puts the disks in the *low-power mode* and thus lowers the power consumption of the disk array. Switching between *low-power* and *normal* operating modes involves only electronic components rather than mechanical parts [18]. Therefore, mode switching power consumption is negligible as compared to power saving achieved. Consequently, active prefetching can ensure that the power consumption of a *Phoenix* device is proportional to the number of active streams being serviced at that time.

As *Phoenix* switches to the low-power mode, the upper-half no longer remains timer-driven. When the disk manager is done with its I/O cycle, it invokes the upper-half directly. The scheduler now schedules read requests for all the streams making sure that streams are prefetched fairly. When enough data is accumulated, the disk manager puts the disks in *low-power mode* and does not invoke the upper-half. The network subsystem keeps consuming the data and when the data level falls below a certain threshold, the network subsystem invokes the upper-half.

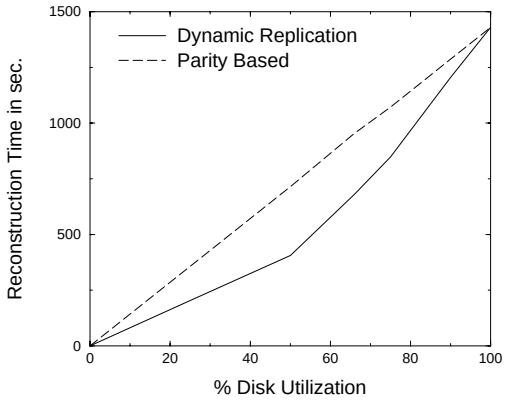

Figure 4: Variation of reconstruction time with increasing disk utilization.

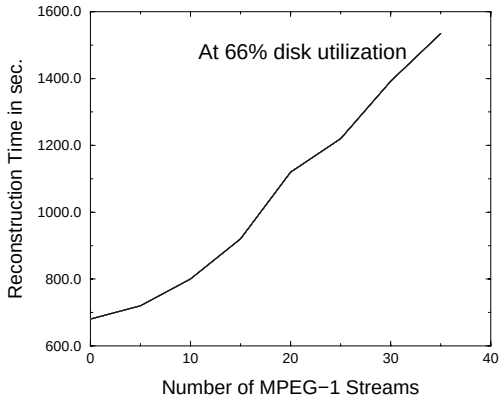

Figure 5: Variation of reconstruction time with increasing number of active streams

7 Performance Measurements

The throughput of the system was measured in terms of the number of MPEG-1 streams it can support. *Phoenix* supports a maximum of 52 streams in normal mode, 42 streams in failure mode and 36 streams in reconstruction mode. Thus, *Phoenix* services a maximum of 36 guaranteed RT streams across failures/reconstruction and an additional 16 guaranteed RT streams but not across failures.

Figure 4 shows the variation of raw reconstruction time (no streams in the system) as the disk utilization increases. A significant gain in reconstruction performance suggests use of *dynamic replication*. Reconstruction up to 50% disk utilization is totally based on replication and then onwards, reconstruction uses parity as well as mirrored data.

Figure 5 shows the variation of reconstruction time as the number of MPEG-1 streams increases. The length of these streams is constant and the streams are uniformly spread across the disks. The disk utilization is kept at 66% (equal UTM and UTP portions). When there are no client streams, the reconstruction is solely due to the reconstruction stream. The reconstruction time is minimum in this case. The reconstruction time increases with the number of real-time clients because of seek and request processing overheads.

Simulations were done to gauge the potential reduction in power consumption, and thus the increased reliability that can be achieved using active prefetching. The fraction of total time available for keeping the disks in low-power mode is shown in figure 6. As the number of streams reduces, the power consumption can also be reduced almost linearly.

8 Conclusions

This paper described in detail the design and implementation of a Linux-based network attached storage device, which exports an object based API, supports real-time reads and

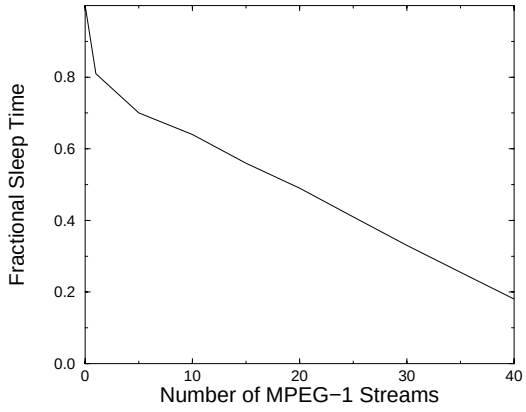

Figure 6: *Variation of* fractional sleep time *with increasing number of streams.*

best-effort reads/writes, provides uninterrupted real-time disk service in the event of a single disk failure, performs on-line disk reconstruction while using the active real-time streams, exploits full disk bandwidth and disk space all the time to speed up failed disk reconstruction, and increases the reliability of the disk subsystem by reducing its power consumption. Extensive measurements on the first *Phoenix* prototype were made to validate the design decisions (described in [19]). In future, the prototype will also be integrated with a real-time network subsystem [16] and a file-system.

References

[1] Drapeau A.L.; et. al., "RAID-II: a high-bandwidth network file server," Proc. of the 21st Annual Intl. Symp. on Computer Architecture, p. 234-44, Chicago, IL, 1994.

[2] Katz R.H., "High-performance network and channel-based storage," Proceedings of the IEEE, vol.80, no.8, p. 1238-61, Aug. 1992.

[3] Van Meter R., "A brief survey of current work on network attached peripherals," Operating Systems Review, vol.30, no.1, p. 63-70, Jan. 1996.

[4] Lee E.K.; Thekkath C.A., "Petal: distributed virtual disks," 7th International Conference on Architectural Support for Programming Languages and Operating Systems, p. 63-70, Cambridge, MA, Oct. 1996.

[5] Thekkath C.A.; Mann T.; Lee E.K., "Frangipani: a scalable distributed file system," 16th ACM Symposium on Operating Systems Principles, p. 224-37, Saint Malo, France, Oct. 1997.

[6] Gibson G.A.; et. al., "A cost-effective, high-bandwidth storage architecture," Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, p. 92-103, San Jose, CA, Oct. 1998.

[7] Gibson G.A.; et. al., “File server scaling with network-attached secure disks,” 1997 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 97), p. 272-84, Seattle, WA, June 1997.

[8] Keeton K.; Patterson D.A.; Hellerstein J.M., “A case for intelligent disks (IDISKs),” SIGMOD Record, vol.27, no.3, p. 42-52, Sept. 1998.

[9] Riedel E.; Gibson G.; Faloutsos C., “Active storage for large-scale data mining and multimedia,” Proceedings of the 24th VLDB Conference, New York, NY., Aug. 1998.

[10] Acharya A.; Uysal M.; Saltz J., “Active disks: programming model, algorithms and evaluation,” Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, p. 81-91, San Jose, CA, Oct. 1998.

[11] Wilkes J.; Golding R.; Staelin C.; Sullivan T., “ The HP AutoRAID hierarchical storage system,” ACM Trans. on Computer Systems, vol.14, no.1, p. 108-36, Feb. 1996.

[12] Chiueh T.; Vernick M.; Venkatramani C., “Integration of Real-Time I/O and Network Support in Stony Brook Video Server,” IEEE Network Magazine, April 1999.

[13] Tobagi F.A.; Pang J.; Baird R.; Gang M., “Streaming RAID - a disk array management system for video files,” Proceedings of First ACM International Conference on Multimedia, p. 393-400, Anaheim, CA, Aug. 1993.

[14] Bolosky W.J.; Fitzgerald R.P.; Douceur J.R., “Distributed schedule management in the Tiger video fileserver,” 16th ACM Symposium on Operating Systems Principles, p. 212-23, Saint Malo, France, Oct. 1997.

[15] Haskin R.L., “ Tiger Shark-a scalable file system for multimedia,” IBM Journal of Research and Development, vol.42, no.2, p. 185-97, March 1998.

[16] Venkatramani C.; Chiueh T., “Design, Implementation, and Evaluation of A Software-Driven Real-Time Ethernet Protocol,” ACM SIGCOMM, 1995.

[17] Perslan K. W.; et. al., “A 64 Bit, Shared Disk File System for Linux,” IEEE Mass Storage Systems Symposium, March 15-18, 1999, San Diego, California.

[18] “IBM Travelstar 6GT,” http://www.storage.ibm.com/hardsoft/diskdrdl/prod/6gtprod.htm.

[19] Neogi A.; Raniwala A.; Chiueh T., “Phoenix: A low-power fault-tolerant network-attached storage device,” ACM Multimedia, 1999.

[20] Li K.; Kumpf; Horton P.; Anderson T., “A quantative analysis of disk drive power management in portable computers,” Proc. of the 1994 Winter USENIX, p. 279–291.

[21] Douglis F.; Krishnan P.; Marsh B., ”Thwarting the power-hungry disk,” Proc. of the 1994 Winter USENIX Conference, Jan. 1994.

[22] Herbst G., “IBM’s drive temperature indicator processor (drive-TIP) helps ensure high drive reliability,”
http://www.storage.ibm.com/hardsoft/diskdrdl/technolo/drivetemp/drivetemp.htm