# An On-Line Reorganization Framework for SAN File Systems*

Shahram Ghandeharizadeh, Shan Gao
Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
shahram,sgao@cs.usc.edu

Chris Gahagan, Russ Krauss
BMC Software Inc.
2101 CityWest Blvd.
Houston, TX 77042, USA
chris_gahagan,russ_krauss@bmc.com

**Abstract**

While the cost per megabyte of magnetic disk storage is economical, organizations are alarmed by the increasing cost of managing storage. Storage Area Network (SAN) architectures strive to minimize this cost by consolidating storage devices. A SAN is a special-purpose network that interconnects different data storage devices with servers. While there are many definitions for a SAN, there is a general consensus that it provides access at the granularity of a block and is typically used for database applications.

In this study, we focus on SAN switches that include an embedded storage management software in support of virtualization. We describe an On-line Re-organization Environment, ORE, that controls the placement of data to improve the average response time of the system. ORE is designed for a heterogeneous collection of storage devices. Its key novel feature is its use of "time" to quantify the benefit and cost of a migration. It migrates a fragment only when its net benefit exceeds a pre-specified threshold. We describe a taxonomy of techniques for fragment migration and employ a trace driven simulation study to quantify their tradeoff. Our performance results demonstrate a significant improvement in response time (order of magnitude) for those algorithms that employ ORE's cost/benefit feature. Moreover, a technique that employs bandwidth of all devices intelligently is superior to one that simply migrates data to the fastest devices.

## 1 Introduction

Organizations are alarmed by the increasing cost of *managing* storage [31]. These costs include expenses associated with the human operators who manage disk storage, and the lost productivity when data is unavailable. Data might be unavailable for several reasons. Failure of the disk subsystem containing the referenced data is one. Another might be the load imposed on the system which results in formation of hot spots and bottlenecks, preventing it from responding in a timely manner, causing the user to perceive the data as being unavailable.

Storage Area Network (SAN) architectures strive to minimize the cost of managing storage by consolidating storage devices in a centralized place. They also promise to increase the productivity and effectiveness of human operators by providing detailed information about the system, advanced notification of when the storage subsystem is not meeting the performance requirements of an application (or filling up), suggestions on how to improve system performance, etc. A SAN is a special-purpose network that interconnects different kinds of data storage devices with servers. It may consist of multi-vendor storage systems, storage

*A shorter version of this paper appeared in the *Tenth East-European Conference on Advances in Databases and Information Systems*, Thessaloniki, Hellas, September 3-7, 2006.

management software and network hardware. It provides block-level access to the data[1]. This is important for database management systems that implement the concept of a transaction and its ACID properties: Atomic, Consistent, Isolation, and Durable [22]. The commercial arena offers SAN solutions in a variety of hardware configurations, e.g., Fibre Channel, iSCSI, Intel's Infiniband, etc.

The focus of this study is on SANs that include an embedded storage management software in support of virtualization. This software includes a file system that separates storage of a device from the physical device, i.e., physical data independence. Virtualization is important because it enables a file to grow beyond the capacity of one disk (or disk array). Such embedded file systems constitute the focus of this study. We investigate ORE, a framework that enables these embedded devices to incorporate new devices and populate them intelligently. Moreover, ORE migrates fragments from one device to another with the objective to minimize the average response time of the system. It incorporates the availability requirements of data and controls its placement to meet this objective.

ORE consists of three steps: monitor, predict, and migrate. The first step gathers data about the environment. The second predicts what data to migrate (if any) to which node. Finally, migrate schedules and performs the data migration. Its novel feature is its use of "time" to quantify the benefit and cost of a migration. It migrates a fragment only when its net benefit exceeds a pre-specified threshold, zero in this study. We describe a taxonomy of techniques with and without this feature. Our performance results indicate that this feature enhances average response time significantly. This is because it (a) migrates those fragments that impose the highest load to the fastest disks, and (b) assigns fragments that are referenced together to different devices, minimizing the formation of queues.

In order to realize acceptable throughput, we assume the distribution of blocks is based on a large striping unit [21, 28, 35, 7, 18, 6]. This means (a) the fraction of a file assigned to a disk is larger than a block size and (b) a block is almost always assigned to a single device. The focus of our framework is to improve average response time with inter-block (instead of intra-block) parallelism.

We use a trace-driven performance evaluation study to compare this framework with alternative reorganization algorithms. Our performance results demonstrate the superiority of the proposed algorithm. The rest of this paper is organized as follows. Section 2 details related work and how this study is novel. Section 3 details our target environment. Based on this, we details our re-organization framework in Sections 4 and 5. Section 6 contains a comparison of this algorithm with its alternatives and different parameter settings. Brief conclusions and future research directions are described in Section 7.

## 2 Related Work

There exists a large body of excellent research on parallel databases [8, 10, 36, 26], code mobility and distributed computing in network attached storage [27, 2, 24], cluster computing [5], redundant arrays of inexpensive disks [37, 17, 11], availability with network attached storage [1], multi-disk architectures [25], and video-on-demand systems [9, 15]. While different elements of this vast literature relate to ORE, it is impossible to enumerate each in this paper. Instead, we focus on research that relates directly to an embedded SAN device and on-line re-organization of data.

The Petal [25] file system is one of the early studies to describe virtual disks. While it does not explicitly describe a SAN or an embedded file system, it employs the concept of "storage servers" that resemble a SAN embedded file system. It outlines an addressing scheme for these servers to map logical block references to physical disk address spaces. It employs chain-declustering [23, 18] for high availability and dynamic load balancing. We adapt their concept to a SAN embedded file system and describe ORE as a novel extension that decides what fragment to migrate to improve response time.

[1] A SAN may also provide the key element of a Network Attached Storage (NAS) system, namely, manage data at the granularity of a file.

On-line data re-organization has been studied by the COMFORT [28, 29, 33] and SNOWBALL [32] projects. Our work is novel for several reasons. First, there is a conceptual difference: We assume magnetic disks are inexpensive and small enough to justify their presence in an embedded device for SAN switches. This enables our framework to collect and maintain trace data on how requests utilize disks in order to make decisions that improve average response time dramatically. A key assumption here is that past request patterns resemble future access patterns. Second, we focus on how to migrate fragments amongst a *heterogeneous* collection of disks. Our performance results demonstrate that a simple extension[2] of the algorithms described in either COMFORT or SNOWBALL does **not** result in the best possible performance.

# 3 Target Environment

In our assumed environment, there are $K$ storage devices. Each storage device $d_i$ has a fixed storage capacity, $C(d_i)$, and an average bandwidth, $BW(d_i)$. With one or more applications that consume $B_{total}$ bandwidth during a fixed amount of time, ideally, each disk must contribute a bandwidth proportional to its $BW(d_i)$:

$$Fairshare(d_i) = B_{total} \times \frac{BW(d_i)}{\sum_{i=1}^{K} BW(d_i)} \quad (1)$$

The bandwidth of a disk is a function of the average requested block size ($\beta$) and its physical characteristics [19, 6]: seek time, rotational latency, and transfer rate (tfr). It is defined as:

$$BW(d_i) = tfr \times \frac{\beta}{\beta + (tfr \times (seek\ time + rotational\ latency))} \quad (2)$$

Given a fixed seek time and rotational latency, BW($d_i$) approaches disk transfer rate with larger block sizes ($\beta$).

There are $F$ files stored on the underlying storage. The number of files might change over times, causing the value of $F$ to change. A file $f_i$ might be partitioned into two or more fragments. Its number of fragments is independent of the number of storage devices, i.e., $K$. Fragments of a file may have different sizes. Fragment $j$ of file $f_i$ is denoted as $f_{i,j}$. In our assumed environment, two or more fragments of a file might be assigned to the same disk drive[3].

# 4 ORE: A Three Step Framework

ORE consists of 3 logical steps: monitor, predict, and migrate. It partitions time into fixed intervals, termed *time slices.* During **monitor**, it constructs a profile of the load imposed by each file fragments per time slice. During **predict**, it performs two tasks. First, it computes what fragments to migrate from one disk to another in order to enhance system performance. Second, it identifies when in the future the migration should be performed so that it does not interfere with the current system load. Once an idle time arrives and there are candidates to migrate, ORE enters the **migrate** phase and changes the placement of these fragments. Below, we detail each of these steps.

[2]This simple extension would be the EVEN policy of Section 5.1 that is several orders of magnitude slower than either EVEN$_{C/B}$ or PYRAMID$_{BW,C/B}$ proposed in this study.

[3]Some studies require each fragment of a file to be assigned to a different disk drive [29].

### 4.1 Monitor

During each time slice, ORE constructs a profile of the load imposed on each disk drive and the average response time of each disk $d_i$. The load imposed on disk drive $d_i$ is quantified as the bandwidth required from disk $d_i$. It is the total number of bytes retrieved from $d_i$ during a time slice divided by the duration of the time slice. The average response time of $d_i$ is the average response time of the requests it processes during the time interval.

This process produces two tables, FragProfiler and DiskProfiler, that are used by the other two steps. FragProfiler table maintains the average block request size, heat, and load imposed by each fragment $f_{i,j}$ per time slice. DiskProfiler table maintains the following metadata for each disk drive $d_i$ per time slice: its heat, load, standard deviation in system load, average response time, average queue length, and utilization.

### 4.2 Predict

During this stage, ORE predicts what fragments to migrate to enhance response time. Section 5 describes a taxonomy of algorithms that can be employed for this step. In Section 6, we quantify the tradeoff associated with these alternatives.

### 4.3 Migrate

Fragment migration might be performed in two possible ways. With the first, the fragment is locked in exclusive mode while it is migrated from $d_{src}$ to $d_{dst}$. This simple algorithm prevents updates while the fragment is migrating. It is efficient and easy to implement. However, the data might appear to be unavailable during the reorganization process. Due to this limitation, we ignore this algorithm from further consideration.

The second, allows concurrent updates against two copies of the migrating fragment: (a) one on $d_{src}$, termed primary, and (b) the other on $d_{dst}$, termed secondary. The secondary copy is constructed from the primary copy of the fragment. All read requests are directed to the primary copy. All updates are performed against both the primary and secondary copy. The migration process is a background task that is performed based on availability of bandwidth from $d_{src}$. It assumes some buffer space for staging data from primary copy to facilitate construction of its secondary copy. This buffer space might be provided as a component of the embedded device. Depending on its size, the system might read and write units larger than a block. Moreover, it might perform writes against $d_{dst}$ in the background depending on the amount of free buffer space. Once the free space falls below a certain threshold, the system might perform writes as foreground tasks that compete with active user requests [4].

## 5 Predict: Fragments to Migrate

Predict, the second step of our framework (see Section 4.2), may utilize a taxonomy of techniques for choosing what fragments to migrate. In this section, we detail two classes of greedy algorithms, each with a different objective:

1. **EVEN** strives to distribute the load uniformly across the disks by migrating fragments to minimize the difference between (a) the disk that has more than its fair share of system load, and (b) the disk that has less than its fair share.

2. **PYRAMID** maximizes the utilization of fastest disks by (a) organizing disks in a vertical hierarchy with the fastest disk appearing at the top of the pyramid, and (b) migrating fragments that impose the greatest load to the top of this hierarchy.

When a configuration consists of groups of disks with approximately the same bandwidth, ORE constructs clusters with each cluster containing the same bandwidth. This does not impact the design of EVEN. However, it modifies PYRAMID to become a hybrid approach. Same as before, PYRAMID organizes the clusters in a hierarchy with the cluster containing the fastest disk type appearing at the top of the hierarchy. Fragments that impose the highest load migrate to the top of the pyramid. Within a cluster, fragments are migrated using EVEN because the disks that constitute a cluster are of the same type.

To illustrate, assume a configuration with 100 disks of type A, each offering a bandwidth of 10 megabytes per seconds, and 2 disks of type B, each offering 100 megabytes per second. The first cluster, termed $C_1$, offers an aggregate bandwidth of 1000 megabytes per second. The second cluster, termed $C_2$, offers an aggregate bandwidth of 200 megabytes per second. This does not impact EVEN because it continues to treat each disk individually, migrating fragments to approximate an even distribution of workload. However, PYRAMID organizes these two clusters in a hierarchy with $C_2$ as at the top layer because it contains the fastest disks. It migrates fragments that impose the highest load to $C_2$. Within each cluster, it employs EVEN to approximate an even distribution of workload across the disks of that cluster.

In the following, we detail each approach and its variations.

## 5.1 EVEN: Constrained by bandwidth

At the end of each time slice, EVEN computes the fair-share of system load for each disk drive. Next, it identifies the disk with (a) maximum positive load imbalance, termed $d_{src}$, and (b) minimum negative load imbalance, termed $d_{dst}$. (The concept of load imbalance is formalized in the next paragraph.) Amongst the fragments of $d_{src}$, it chooses the one with a load closest to the minimum negative load of $d_{dst}$. It migrates this fragment from $d_{src}$ to $d_{dst}$. This process repeats until either there are no source and destination disks or a new time slice arrives.

The maximum positive load imbalance pertains to those disks with an imposed load greater than their fair share. For each such disk $d_i$, its $\delta^+(d_i)$ = load($d_i$)-Fairshare($d_i$). Positive imbalance of $d_i$ is defined as $\frac{\delta^+(d_i)}{Fairshare(d_i)}$. EVEN identifies the disk with highest such value as the source disk, $d_{src}$, and migrates its fragments to those disks with a negative load imbalance.

We define the minimum negative load imbalance for those disks with an imposed load less than their fair share. For each such disk $d_i$, its $\delta^-(d_i)$ equals load($d_i$)-Fairshare($d_i$). Negative imbalance of $d_i$ is $\frac{\delta^-(d_i)}{Fairshare(d_i)}$. The disk with the smallest negative imbalance[4] is the destination disk, $d_{dst}$, and EVEN migrates fragments to this disk. EVEN identifies those fragments of $d_{src}$ with an imposed load approximately the same as $\delta^-(d_{dst})$ and migrates them to $d_{dst}$.

### 5.1.1 EVEN$_{C/B}$: Constrained by Bandwidth with Cost/Benefit Consideration

EVEN$_{C/B}$ extends EVEN, see Section 5.1, by quantifying the benefit and cost of each candidate migration from $d_{src}$ to $d_{dst}$. Section 5.3 describes how the system quantifies the cost and benefit of each candidate migration because it is general purpose and used by the PYRAMID variation of Section 5.2.3. EVEN$_{C/B}$ sorts candidate migration based on their net benefit, i.e., benefit - cost, migrating those that provide the greatest savings first. After each migration, the cost of each candidate migration is re-computed (because the migration might have changed this value) and the list is resorted. Section 6 shows this algorithm provides significant response time enhancements when compared with EVEN.

[4]Given two disks, $d_1$ and $d_2$ with negative imbalance of -0.5 and -2.0, respectively, $d_2$ has the minimum negative load imbalance.

## 5.2 PYRAMID

PYRAMID migrates fragments with the highest load to the fastest disk drives. It constructs layers of storage devices and assigns the fastest to the top of the hierarchy. Fragments migrate up and down the pyramid based on their imposed load. We describe 3 variations of this algorithm. The performance results of Section 6 demonstrate the superiority of the last design, PYRAMID$_{BW,C/B}$.

### 5.2.1 PYRAMID$_{SP}$: Constrained by Space

PYRAMID$_{SP}$ migrates highest load fragments to the fastest disks until their storage capacity is exhausted. If the database size is smaller than the total storage capacity of devices then disks that constitute the lowest layer of this hierarchy might be completely un-utilized. For example, in Figure 1.a, the database is small enough to fit on two disks, causing PYRAMID$_{SP}$ to render disk 3 idle.

Its details are as follows. At the end of each time slice, this algorithm sorts fragments based on their imposed load. It maintains a sorted list of disks based on their available bandwidth. Next, it computes which fragments should reside on which disk by exhausting the storage capacity of disks at the highest layer. This is the target placement. PYRAMID$_{SP}$ compares this with the current placement and computes a collection of migrations to transform the current placement to the target placement. It migrates those that impact fragments with the highest load first. It terminates when (a) the new placement is realized or (b) a new time slice arrives.

By exhausting the storage capacity of fast disks, a migration with PYRAMID$_{SP}$ might translate into multiple migrations. For example, assume the storage capacity of disks 1 and 2 are exhausted in Figure 1.b. In order to switch the place of two equi-sized fragments, say 1.1 and 3.2, the system might perform 3 migration: migrate fragment 1.1 from disk 1 to disk 3, migrate fragment 3.2 from disk 2 to disk 1, and migrate fragment 1.1 to disk 2. Of course, this can be prevented as long as main memory is sufficiently large to hold either fragment 1.1 or 3.2 and partially written fragments can be restored in the presence of failures.

When a configuration consists of groups of disks with each disk group providing similar bandwidth, this algorithm constructs clusters of disks. Migration of fragments across the clusters is the same as before. Within a cluster, PYRAMID$_{SP}$ employs EVEN to migrate fragments from one disk to another, see Section 5.1.

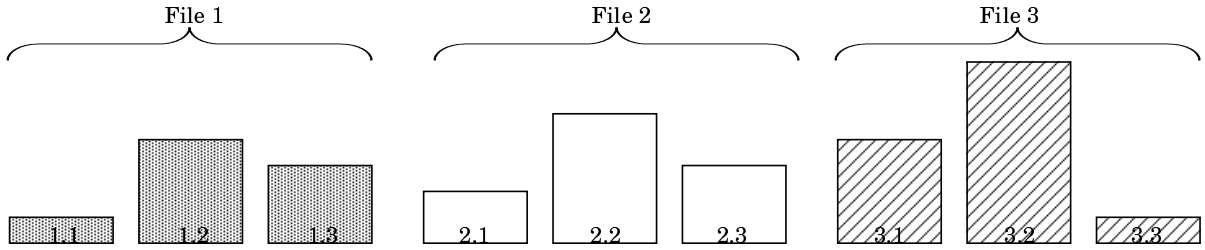
### 5.2.2 PYRAMID$_{BW}$: Constrained by Bandwidth

PYRAMID$_{BW}$ migrates fragments with the highest load to the fastest disk drives with the objective to utilize the bandwidth of each layer in the hierarchy. Thus, even if the database is small enough to fit on the fastest disk, this algorithm utilizes the storage capacity of each layer. The amount of data assigned to each layer is proportional to its bandwidth.

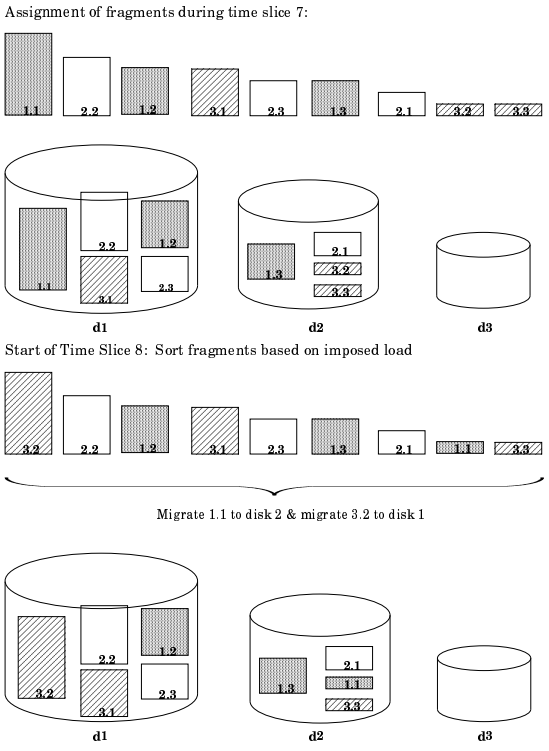
Its detail is as follows. At the end of each time slice, PYRAMID$_{BW}$ sorts fragments and disks based on their imposed load and bandwidth, respectively. Next, using the bandwidth of each disk, it estimates the fraction of load that should be assigned to each disk to exhaust its bandwidth. This identifies which fragments should reside on which disk drive. If the current assignment realizes this new placement then the algorithm terminates. Otherwise, it migrates fragments to realize the new placement. It terminates when (a) the new assignment is realized or (b) a new time slice arrives. Figure 1.b shows this algorithm when the system detects an increase in the heat of fragment 3.2, motivating its migration from disk 3 to disk 1.

### 5.2.3 PYRAMID$_{BW,C/B}$: Constrained by Bandwidth with Cost/Benefit Consideration

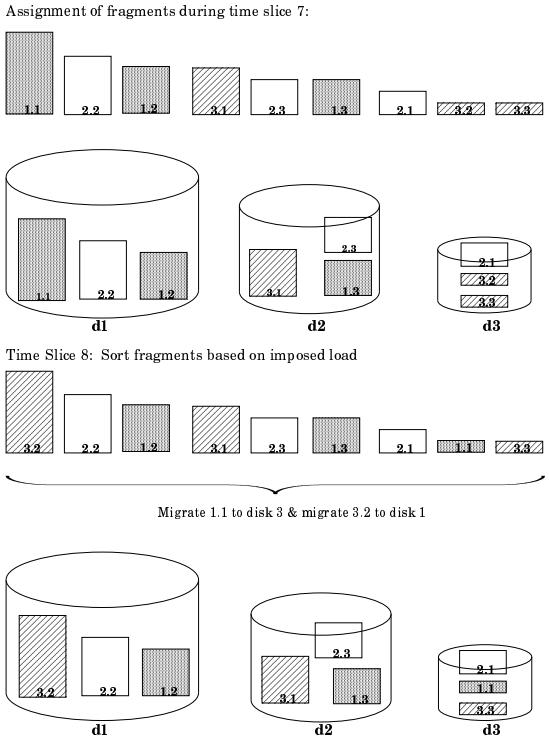
This algorithm extends PYRAMID$_{BW}$ by computing the cost and benefit of each candidate migration. It sorts these candidates based on their net benefit, performing those that provide greatest savings first. Note




1a. Logical view of 3 files, each partitioned into 3 fragments. Vertical height is the imposed load of each fragment at the end of time slice 8.




1b. PYRAMID$_{SP}$, space constrained.




1c. PYRAMID$_{BW}$, bandwidth constrained.

Figure 1: PYRAMID Migration.

that once it migrates the first candidate in the list, the benefit of the other migrations might change. Thus, it re-computes the benefit of each candidate after performing one migration.

### 5.3 Evaluating Benefit and Cost of a Migration

This section describes an approach to quantify the benefit and cost of migrating a fragment $f_{i,j}$ from $d_{src}$ to $d_{dst}$. Its unit of measurement is time, i.e., milliseconds. The cost of migrating a fragment is the total time spent by $d_{src}$ to read the fragment and $d_{dst}$ to write the fragment.

The benefit of migrating $f_{i,j}$ is measured in the context of previous time slices. ORE hypothesizes a virtual state where $f_{i,j}$ resides on $d_{dst}$ and measures the improvement in average response time. In essence, it estimates an answer to the following question: "What would be the average response time if $f_{i,j}$ resided on $d_{dst}$?" By comparing this with the observed response time, we quantify the benefit of a migration. Of course, this number might be a negative value which means that there is no benefit to performing this migration. Note this methodology assumes the past access patterns resemble future access patterns.

We start by describing a methodology to estimate a response to the hypothetical "what-if" question. Next, we formalize how to compute the benefit. Subsequently, we present how much space is required to implement our methodology.

Our methodology to estimates a response to the "what-if" question assumes ORE is previewed to all block requests issued to the SAN switch and the status of each storage drive. ORE maintains one additional piece of information, namely the duration of overlap between two fragments, termed OVERLAP($f_{i,j}$, $f_{k,l}$). This information is maintained for each time slice. It estimates how long two requests referencing fragments $f_{i,j}$ and $f_{k,l}$ overlap with each other in time. It is used to detect correlations between requests referencing fragments $f_{i,j}$ and $f_{k,l}$. If there is a high correlation then these fragments should be assigned to different disks to minimize the impact of queuing delays. Below, we present a formal definition of OVERLAP.

In order to define OVERLAP and describe our methodology, and without loss of generality, assume that we are answering the "what-if" question in the context of one time slice. To simplify the discussion further, assume a homogeneous collection of disk drives. (This assumption is removed at the end of this section.) The average system response time, $RT_{avg}$, is a function of average response time observed by requests referencing each fragment. Assuming $F$ files, each partitioned into at most $G$ fragments, it is defined as:

$$RT_{avg} = \frac{\sum_{i=1}^{F} \sum_{j=1}^{G} RT_{avg}(f_{i,j})}{F \times G} \tag{3}$$

The average response time of a fragment, $RT_{avg}(f_{i,j})$, is the sum of its average service time, $S_{avg}(f_{i,j})$, and wait time, $W_{avg}(f_{i,j})$:

$$RT_{avg}(f_{i,j}) = S_{avg}(f_{i,j}) + W_{avg}(f_{i,j}) \tag{4}$$

$S_{avg}(f_{i,j})$ is a function of the disk it resides on and average requested block size. For each fragment, as detailed in Section 4.1, ORE maintains the average requested block size in the FragProfiler table. Thus, given a disk drive $d_{dst}$ and a fragment $f_{i,j}$, ORE can estimate $S_{avg}(f_{i,j})$ if $f_{i,j}$ resided on $d_{dst}$ (using the physical characteristics of $d_{dst}$).

To compute $W_{avg}$, note that each request has an arrival time, $T_{arvl}$. For each fragment $f_{i,j}$ residing on disk $d_i$, we maintain when the requests referencing $f_{i,j}$ will depart the system, $T_{depart}$. $T_{depart}$ is estimated by analyzing the wait time of the request in the queue of $d_i$. Upon the arrival of a request referencing fragment $f_{k,l}$, we examine all those fragments with a non-negative $T_{depart}$. For each, we set OVERLAP($f_{k,l}$, $f_{i,j}$,$T_{arvl}$) to be the difference between $T_{arvl}(f_{k,l})$ and $T_{depart}(f_{i,j})$: OVERLAP($f_{k,l}$, $f_{i,j}$,$T_{arvl}$)= Max(0, $T_{depart}(f_{i,j}) - T_{arvl}(f_{k,l})$). For a time slice, OVERLAP($f_{k,l}$, $f_{i,j}$) is the sum of the individual OVERLAP($f_{k,l}$, $f_{i,j}$,$T_{arvl}$) where $T_{arvl}$ is during the time slice. In our implementation, we maintained

1. the load imposed by fragment $f_{i,j}$ on $d_{src}$ is termed $load(f_{i,j}, d_{src})$.
2. the load imposed by fragment $f_{i,j}$ on $d_{dst}$ after migration is $load(d_{dst}) + load(f_{i,j}, d_{src})$.
3. Number of accesses processed by disk $d_{src}$ is $Access_{src}$
4. Number of accesses processed by disk $d_{dst}$ is $Access_{dst}$
5. Look-up the average response time of $d_{src}$ prior to migration, termed $RT_{src,before}$
6. Look-up the average response time of $d_{dst}$ prior to migration, termed $RT_{dst,before}$
7. Estimate the average response time of $d_{src}$ after migration, termed $RT_{src,after}$
8. Estimate the average response time of $d_{dst}$ after migration, termed $RT_{dst,after}$
9. Total response time savings of $d_{src}$ after migration is:
   $Savings_{src}$=$(Access_{src,after} \times RT_{src,after}) - (Access_{src,before} \times RT_{src,before})$.
10. Total response time savings of $d_{dst}$ after migration is:
    $Savings_{dst}$=$(Access_{dst,after} \times RT_{dst,after}) - (Access_{dst,before} \times RT_{dst,before})$.
11. Benefit of migrating $f_{i,j}$ is $Benefit(f_{i,j})$=$Savings_{src} + Savings_{dst}$.

Figure 2: Pseudo-code to compute the benefit of a candidate migration.

OVERLAP($f_{k,l}, f_{i,j}$) as an integer that is initialized to zero at the beginning of each time slice. Upon the arrival of a request referencing $f_{k,l}$, we increment OVERLAP($f_{k,l}, f_{i,j}$) with OVERLAP($f_{k,l}, f_{i,j}$,$T_{arvl}$). This minimizes the amount of required memory.

OVERLAP($f_{k,l}, f_{i,j}$) defines how long requests referencing $f_{k,l}$ wait in a queue because of requests that reference $f_{i,j}$. Assuming that $f_{i,j}$ and $f_{k,l}$ are the only fragments assigned to disk $d_i$ and the system processes #Req($f_{k,l}$) requests that reference $f_{k,l}$, the average wait time for these requests is:

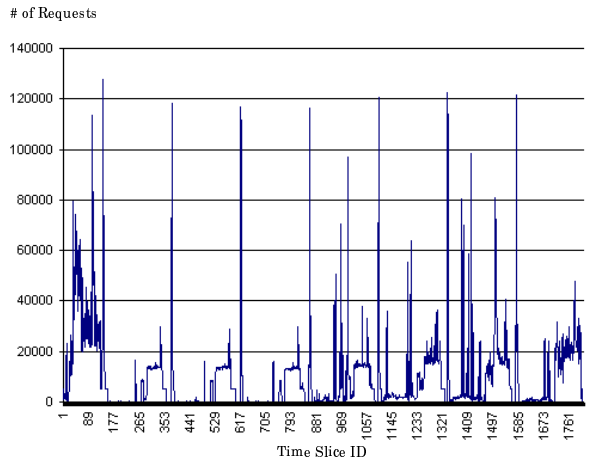
$$W_{avg}(f_{k,l}) = \frac{OVERLAP(f_{k,l}, f_{i,j}) + OVERLAP(f_{k,l}, f_{k,l})}{\#Req(f_{k,l})} \quad (5)$$

It is important to observe the following two details. First, self OVERLAP is also defined for a fragment $f_{k,l}$, i.e., OVERLAP($f_{k,l}$, $f_{k,l}$). This enables ORE to estimate how long requests that reference the same fragment wait for one another. Second, this paradigm is flexible enough to enable ORE to maintain OVERLAP($f_{k,l}, f_{i,j}$) even when $f_{k,l}$ and $f_{i,j}$ reside on different disks. ORE uses this to estimate a response time for a hypothetical configuration where $f_{i,j}$ migrates to the disk containing $f_{k,l}$. Third, ORE can estimate the response time of a disk drive for an arbitrary assignment of fragments to disks using Equation 3.

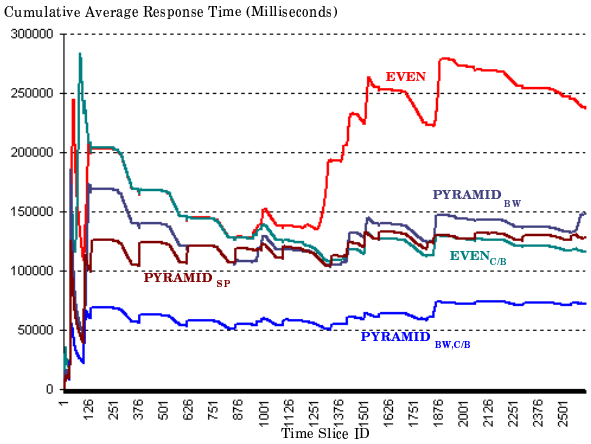
Based on Equation 4, there are two ways to enhance response time observed by requests that reference a fragment, $f_{k,l}$. First, migrate $f_{k,l}$ to a faster disk for an improved service time, $S_{avg}$. Second, migrate a fragment $f_{i,j}$ away from those disks whose resident fragments have a high OVERLAP($f_{k,l}, f_{i,j}$).

Figure 2 shows the pseudo-code to estimate the benefit of migrating $f_{i,j}$ from $d_{src}$ to $d_{dst}$. ORE may compute this for $N$ previous time slices where $N$ is an arbitrary number. The only requirement is that the embedded device must provide sufficient space to store all data pertaining to these intervals.

Given $G$ fragments, in the worst case scenario, the system maintains $\frac{G^2 + G}{2}$ integer values. For example with a 1000 fragments (G=1000) and a 32 bit integer representation, in the worst case scenario, the system would store 4 megabytes of data per time slice. In our experiments, the amount of required storage was significantly less than this. With the 80-20 rule, we expect this to hold true for almost all applications. In Section 7, we describe how ORE can employ a circular buffer to limit the size of trace data that it gathers from the system.




3a. No. of requests as a function of time




3b. Cumulative average response time

Figure 3: Pattern of request arrival and system performance

In our experiments, see Section 6, the maximum percentage error observed by our methodology was 23% when estimating the average response time of a request.

# 6 Performance Evaluation

We used a trace driven simulation study to quantify the performance of the proposed on-line re-organization algorithm. We start with a brief overview of the trace driven simulation model. Next, we present the obtained results and our observations.

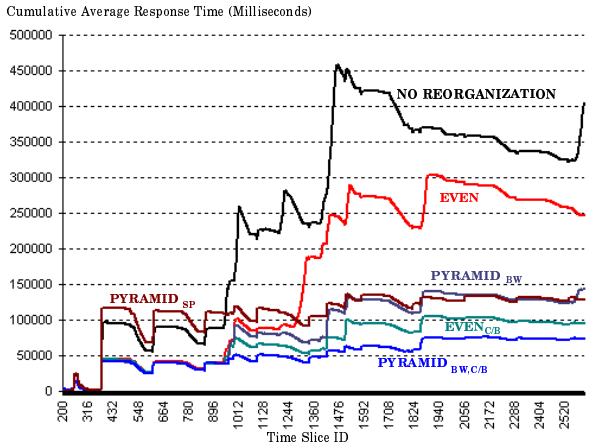
The traces were gathered from a production Oracle database management system on a HP workstation configured with 4 gigabyte of memory, and 5 terabytes of storage devices (283 raw devices). The database consisted of 70 tables and is 27 gigabyte in size. The traces were gathered from 4 pm, April 12 to 1 pm April 23, 2001. It corresponds to 23 million operations on the data blocks. The file references are skewed with approximately 83% of accesses referencing 10% of the blocks. Moreover, accesses to the tables are bursty as a function of time. This is demonstrated in Figure 3a, where we plot the number of requests to the system as a function of six minute intervals, termed time slices.

We used the Java programming language to implement our simulation model. It consists of 3 class definitions:

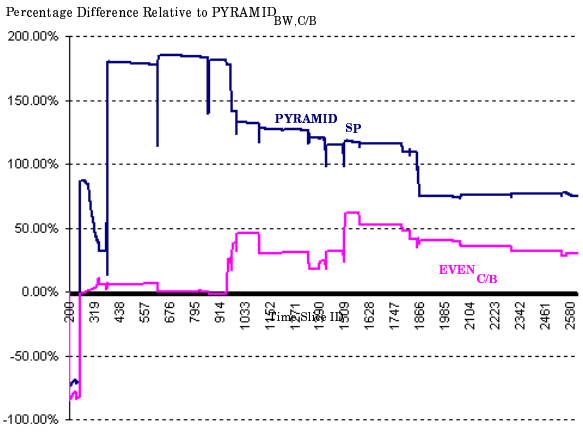
1. Disk: This class definition simulates a multi-zone disk drive with a complete analytical model for computing seeks, rotational latency, and transfer time. When a disk object is instantiated, it reads its system parameters from a database management system. Hence, we can configure the model with different disk models and different number of disks for each model. A disk implements a simplified version of the EVEREST file system.

2. Client: The client generates requests for the different blocks by reading the entries in the trace files.

3. SAN Switch: This class definition implements a simplified SAN switch that routes messages between the client and the disk drives. The file manager is a component of this module. This module services each request generated by a client.




4a. Cumulative average response time




4b. Relative to PYRAMID$_{BW,C/B}$

Figure 4: Cumulative average response time starting with time slice 200

(a) File Manager: The file manager controls and maintains the placement of data across disk drives. It maintains the assignment of different files and their fragments across the disk drives. Given a request for a block of a file, this module locates the fragment referenced by the request and resolves which disk contains the referenced data. It consults with the file system of the disk drive to identify the appropriate cylinder and track that contains the referenced block.

The file manager implements the 3-step re-organization algorithm of ORE, see Section 4, and its alternative policies detailed in Section 5.

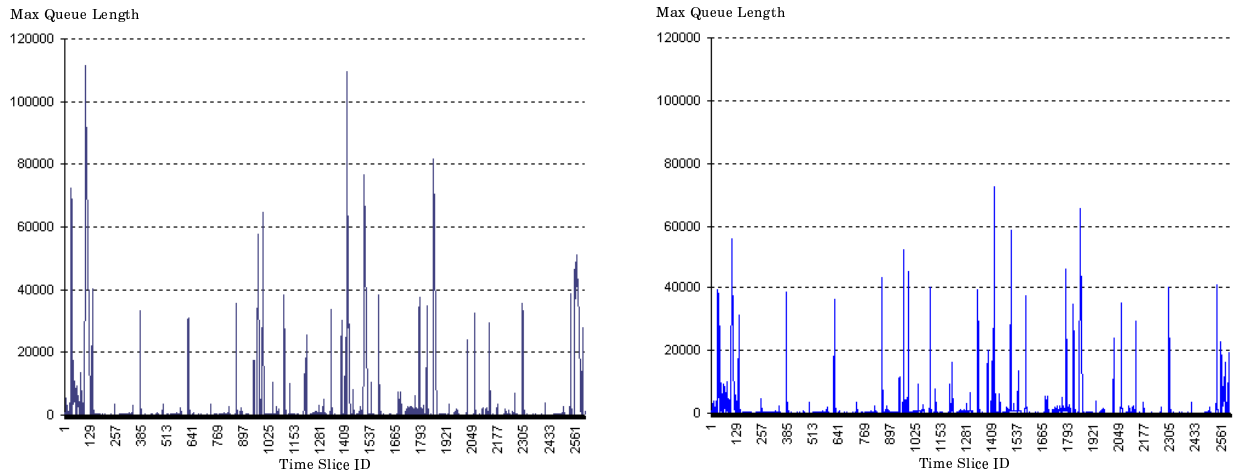
We conducted experiments with both a large configuration consisting of 283 raw devices that corresponds to the physical system that produced the traces and smaller configurations. The smaller configurations are faster to simulate. The performance results presented in this paper are based on one such configuration consisting of 9 disk drives. It consisted of 3 different disk models, forming 3 clusters of homogeneous disks: $C_1$, $C_2$, and $C_3$. Each cluster consisted of 3 disk drives. These disks correspond to those introduced in the late 2000, late 1998 and early 1997. Each disk in $C_1$ has a storage capacity of 180 gigabytes with a transfer rate of 40 megabytes per second (MB/sec). These were modeled after the high density, Ultra160 SCSI/Fibre-Channel disks introduced by Seagate in late 2000. Each disk in $C_2$ has a storage capacity of 60 gigabytes with a transfer rate of 20 MB/sec. Each disk in $C_3$ has a storage capacity of 20 gigabytes with a transfer rate of 4 MB/sec. These two models are similar to those described in [38].

We also analyzed different block sizes. The experimental results presented here are based on 128 kilobyte blocks. In Section 7, we summarize our observations based on smaller block sizes.

All experiments described in Section 6.1 start with the same data placement, namely, a random distribution of fragments across all 9 disks.

### 6.1 Performance Results

Figure 3b shows the performance of alternative predict techniques using the trace. The x-axis of this figure denotes time, i.e., six minute time intervals termed time slices. The y-axis is the cumulative average response time. It is computed as follows. For each time slice, we compute the total number of requests and the sum of all response times till the end of that time slice. The cumulative average response time is the ratio of these two numbers, i.e., $\frac{total\ response\ time}{total\ requests}$. If during a time slice, no requests are issued then the cumulative average response time remains constant. This explains the flat portions of each curve in Figure 3b.




5a. PYRAMID$_{BW}$ 5b. PYRAMID$_{BW,C/B}$

Figure 5: Maximum queue length as a function of time

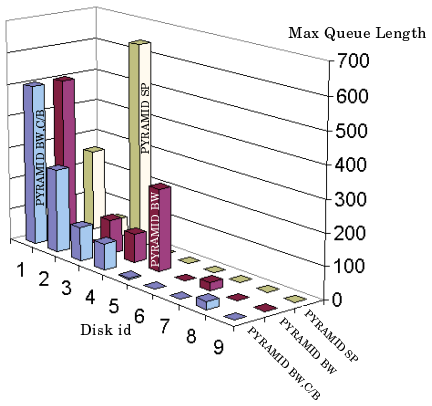
Figure 3b reflects the performance of the system with a cold start, i.e., a random distribution of fragments across the disks. In Figure 4a, we show the cumulative average response time starting with time slice 200, i.e., 20 hours into the simulation. These results demonstrate the superiority of EVEN$_{C/B}$ and PYRAMID$_{BW,C/B}$ when compared with the other strategies. It is important to observe that the y-axis of Figure 4a ranges from 12 milliseconds to 16 minutes (1000 seconds). The maximum response time is very high given that (a) the average service time of the slowest disk is 68 milliseconds, and (b) the average utilization of the available bandwidth (for all 9 disks) is less than 10%. The high response time is because of the bursty nature of request arrivals, see Figure 3a. There are time slices that observe more than 100,000 requests in a matter of seconds. These requests reference a few tables and access the same disk to form long queues. The wait-time in these queues explains the high response times.

EVEN$_{C/B}$ and PYRAMID$_{BW,C/B}$ use OVERLAP to identify those fragments that are referenced together and migrate them to different disks. This minimizes the length of queues observed at each disk drive. Figure 5 shows the highest observed queue length as a function of time slice for PYRAMID$_{BW}$ and PYRAMID$_{BW,C/B}$. In general, using the concept of cost/benefit to migrate fragments reduces the maximum queue lengths by more than a factor of 2.

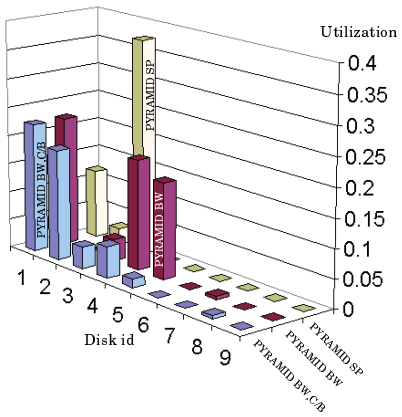
With our configuration, the database is small enough to fit on one of the three 180 gigabyte disks. PYRAMID$_{SP}$ migrates all file fragments to the 3 fastest disks by the $200^{th}$ time slice, rendering the six slower disks idle. Next, it uses EVEN to migrate the fragments amongst these 3 disk drives. Similar to EVEN and PYRAMID$_{BW}$, it observes queues longer than those seen by EVEN$_{C/B}$ and PYRAMID$_{BW,C/B}$, although it benefits from the speed of the fast disks to service these requests quickly (12 millisecond service time versus 18 and 68 milliseconds with the other two disk models).

Figure 6 shows the maximum queue length and utilization of the system as a function of each disk for one of the time slices that observes a burst of requests. The "Disk id" axis of this figure shows the individual disk drives, starting with the fastest, i.e., disks 1, 2, and 3 belong to cluster $C_1$. In this figure, PYRAMID$_{SP}$ utilizes the fastest disks 1, 2 and 3 only. It observed a queue longer than that seen by PYRAMID$_{BW}$ or PYRAMID$_{BW,C/B}$.

The fact that the performance of PYRAMID$_{SP}$ is not as good as EVEN$_{C/B}$ and PYRAMID$_{BW,C/B}$ highlights the following interesting observation: migrating data to fast disks does not result in superior performance. A re-organization strategy that assigns a proportional amount of system workload to each




6a. Maximum queue length




6b. Average utilization

Figure 6: Maximum queue length and utilization of each disk for one time slice

device intelligently is a superior alternative.

Figure 6 also shows that PYRAMID$_{BW,C/B}$ utilizes disks 4, 5 and 8. It is important to note that even though it utilizes disk 4 more than disk 3, the maximum queue length for disk 4 is smaller. The same observations apply to EVEN$_{C/B}$. This shows minimizing the queue lengths provides enhanced response times.

# 7 Conclusion and Future Research Directions

This paper introduced ORE as a 3-step reorganization framework for embedded SAN file systems. We described several algorithms that decide what fragments to migrate to which disk. We employed a trace driven simulation study to quantify their performance trade-offs. Two algorithms, namely, EVEN$_{C/B}$ and PYRAMID$_{BW,C/B}$ provide the best cumulative average response time. There are several reasons for this. First, they migrate fragments with a high load to the faster devices. Second, they use the concept of OVERLAP to migrate two fragments that are referenced simultaneously to different devices in order to minimize the queuing delays. While we do not claim that these observations apply to all SAN applications, we expect them to hold true for those applications that issue requests in a bursty manner, exhibit a skewed data access pattern (80-20 rule [19]) with future access patterns resembling past access patterns.

The block size ($\beta$) impacts the behavior of ORE greatly. A small block size, e.g., 2 kilobyte, reduces bandwidth of fast disks dramatically because seek and rotational delays dominate the transfer time, see Equation 2. With the nine disk configuration of Section 6, a 2 kilobyte block size would force ORE to treat all disks as identical.

An intelligent technique that utilizes bandwidth of all devices (PYRAMID$_{BW,C/B}$) is superior to one that simply migrates the data to the fastest devices (PYRAMID$_{SP}$). In our experiments, PYRAMID$_{BW,C/B}$ controls placement of data with the objective to minimize the likelihood of simultaneous requests referencing the same device, reducing formation of bottlenecks and hot spots.

The results presented in this paper are very promising and we plan to extend ORE in several ways. First, ORE should consider the availability requirement of a file $f_i$ when placing it across devices. For example, $f_i$ may specify that its mean-time-to-data-loss, $MTTDL(f_i)$, should exceed 200,000 hours, $MTTDL_{min}(f_i)$ = 200,000 hours. Assuming physical disk drives fail independent of one another, each disk has a certain

failure rate [38, 30, 16], termed $\lambda_{failure}$. Its mean-time-to-failure (MTTF) is simply:$\frac{1}{\lambda_{failure}}$. When a file (say $f_j$) is partitioned into $n$ fragments and assigned to $n$ disks (say $d_1$ to $d_n$) then the data becomes unavailable in the presence of a single failure[5]. Hence, it is defined as follows [38, 30, 16]: $MTTDL(f_i) = \frac{1}{\sum_{i=1}^{n} \lambda_{failure}(d_i)}$. For example, if the MTTF of disk A and B is 1 million and 2 million hours, respectively, then the MTTDL of a file with fragments scattered across these two disks is 666,666 hours. This is important because ORE may not be able to spread the fragments of a file across all devices. Moreover, ORE might be forced to place those files with a high availability requirement on the newer disks with a high MTTF characteristics.

Second, we intend to investigate the design of an online capacity planner that consumes the maximum response time requirements of an application, detects when the system is not meeting this requirement, and suggests changes to the configuration to meet the specified response time. This capacity planner would be a component of the embedded device. It can detect when the response time requirement is being violated because it observes all request arrivals and departures. It can suggest hardware changes because the first step of ORE, monitor, gathers important details on how the resources are used.

# 8 Acknowledgments

We wish to thank Anouar Jamoussi and Sandra Knight of BMC Software for collecting and providing traces used in this study. We also thank William Wang, Sivakumar Sethuraman, and Dinakar Yanamandala of USC for assisting with the implementation of our simulation model. This research was made possible by an unrestricted cash gift from BMC Software Inc., a fellowship award from the Annenberg Center for Communication, and NSF research grant IIS-0307908.

# References

[1] K. Amiri, G. Gibson, and R. Golding. Highly Concurrent Shared Storage. In *Proceedings of the International Conference on Distributed Computing Systems*, April 2000.

[2] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamice Function Placement for Data-Intensive Cluster Ccomputing. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[3] T. Anderson, Y. Breitbart, H. Korth, and A. Wool. Replication, Consistency, and Practicality: Are These Mutually Exclusive? *Proceedings of ACM SIGMOD*, 27, 1998.

[4] W. Aref, I. Kamel, T. Niranjan, and S. Ghandeharizadeh. Disk Scheduling for Displaying and Recording Video in Non-Linear News Editing Systems. In *Proceedings of Multimedia Computing and Networking Conference*, 1997.

[5] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.

[6] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered Striping in Multimedia Information Systems. In *Proceedings of ACM SIGMOD*, pages 79–90, 1994.

[5]There has been a significant amount of research on construction of parity data blocks and redundant data, see [38] that focuses on this for heterogeneous disks. This topic is beyond the focus of this study. In this paper, we control the placement without constructing redundant data.

[7] P. M. Chen and D. A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News*, page 322, 1990.

[8] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proceedings of ACM SIGMOD*, pages 99–108, 1988.

[9] Asit Dan and Dinkar Sitaram. An Online Video Placement Policy Based on Bandwidth to Space Ratio (BSR). In *Proceedings of ACM SIGMOD*, pages 376–385, 1995.

[10] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

[11] A. L. Drapeau, K. W. Shirrif, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, K. Lutz, D. A. Patterson, E. K. Lee, P. H. Chen, and G. A. Gibson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 234–244, 1994.

[12] S. Ghandeharizadeh, D. Ierardi, and D. Kim. Placement of Data in Multi Zone Disk Drives. In *Proceedings of the Second International Baltic Workshop on Databases and Information Systems*, 1996.

[13] S. Ghandeharizadeh, D. Ierardi, and R. Zimmermann. An Algorithm for Disk Space Management to Minimize Seeks. *The Computer Journal*, 57:75–81, 1996.

[14] S. Ghandeharizadeh, D. Ierardi, and R. Zimmermann. Management of Space in Hierarchical Storage Systems. In M. Arbib and J. Grethe, editors, *A Guide to Neuroinformatics.* Academic Press, 2001.

[15] S. Ghandeharizadeh, S. Kim, W. Shi, and R. Zimmermann. On minimizing startup latency in scalable continuous media servers. In *Multimedia Computing and Networking*, Feb 1997.

[16] G. Gibson. Redundant Disk Arrays: Reliable, Parallel Secondary Storage, 1991.

[17] G. A. Gibson and D. A. Patterson. Designing Disk Arrays for High Data Reliability. *Journal of Parallel and Distributed Computing*, 17(1–2):4–27, /1993.

[18] L. Golubchik and R. R. Muntz. Fault Tolerance Issues in Data Declustering for Parallel Database Systems. *Data Engineering Bulletin*, 17(3):14–28, 1994.

[19] J. Gray and G. Graefe. The 5 Minute Rule, Ten Years Later. In *SIGMOD Record*, volume 26, 1997.

[20] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of ACM SIGMOD*, pages 173–182, 1996.

[21] J. Gray, B. Horst, and M. Walker. Parity Striping of Disk Arrays: Low Cost Reliable Storage with Acceptable Throughput. In *Proceedings of the VLDB Conference*, pages 152–162, September 1990.

[22] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques.* Morgan Kaufmann, 1992.

[23] H. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[24] M. K. Lakhamraju, R. Rastogi, S. Seshadri, and S. Sudarshan. On-Line Reorganization in Object Databases. In *Proceedings of ACM SIGMOD*, pages 58–69, 2000.

[25] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[26] M. L. Lee, M. Kitsuregawa, B. C. Ooi, K. Tan, and A. Mondal. Towards Self-tuning Data Placement in Parallel Database Systems. In *Proceedings of ACM SIGMOD*, pages 225–236, 2000.

[27] D. Petrou, K. Amiri, G. Ganger, and G. Gibson. Easing the Management of Data-Parallel Systems via Adaptation. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[28] P. Scheuermann, G. Weikum, and P. Zabback. "Disk Cooling" in Parallel Disk Systems. *Data Engineering Bulletin*, 17(3):29–40, 1994.

[29] P. Scheuermann, G. Weikum, and P. Zabbak. Data Partitioning and Load Balancing in Parallel Disk Systems. *VLDB Journal*, 7(1), 1998.

[30] D. P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

[31] A. Veitch, E. Riedel, S. Towers, and J. Wilkes. Towards Global Storage Management and Data Placement. Technical Report HPL-SSP-2001-1, Hewlett Packard Laboratories, March 2001.

[32] R. Vingralek, Y. Breitbart, and G. Weikum. Snowball: Scalable Storage on Networks of Workstations with Balanced Load. *Distributed and Parallel Databases*, 6(2):117–156, 1998.

[33] G. Weikum, C. Hasse, A. Moenkeberg, and P. Zabback. The COMFORT Automatic Tuning Project, Invited Project Review. *Information Systems*, 19(5):381–432, 1994.

[34] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., 2000. IEEE Computer Society Technical Commitee on Distributed Processing.

[35] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 96–108, Copper Mountain, CO, 1995. ACM Press.

[36] K. Wu, P. Yu, J. Chung, and J. Teng. A Performance Study of Workfile Disk Management for Concurrent Mergesorts in a Multiprocessor Database System. In *Proceedings of the VLDB Conference*, pages 100–110, September 1995.

[37] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, and T. Anderson. Trading Capacity for Performance in a Disk Array. In *Symposium on Operating Systems Design and Implementation*, October 2000.

[38] R. Zimmermann and S. Ghandeharizadeh. HERA: Heterogeneous Extension of RAID. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June 2000.