DISTRIBUTED RAID SYSTEM: A UNIQUE USE FOR REED SOLOMON CODING

By

NATHANIEL PAUL LEWIS

Bachelor of Science

University of Wisconsin – Madison

Madison, Wisconsin

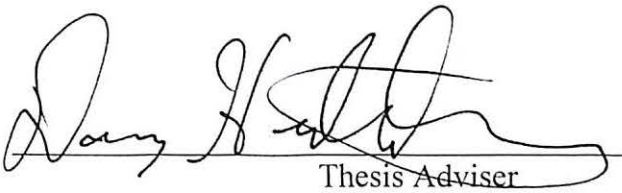
2001

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 2004

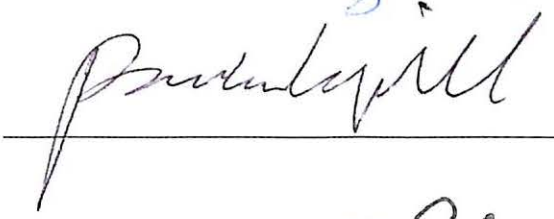
DISTRIBUTED RAID SYSTEM: A UNIQUE

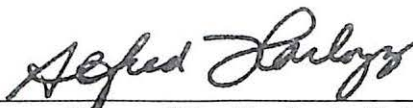
USE FOR REED SOLOMON CODING

Thesis Approved:



Thesis Adviser





Dean of the Graduate College

Preface

This study was conducted to investigate the usability of a locally distributed RAID system in a way that utilizes previously underutilized workstation storage space.

I sincerely thank my advisor, Professor Douglas R. Heisterkamp, for his guidance and support in completion of this research.

Table of Contents

List of Tables

Table Page

List of Figures

Figure Page

Nomenclature

IDE	Integrated Drive Electronics
LAN	Local Area Network
RAID	Redundant Array of Inexpensive/Independent Disks
SCSI	Small Computer Standard Interface

I. Introduction

Traditional studies of Redundant Array of Inexpensive/Independent Disk (RAID) systems generally concern a single central server with a stack of disks directly attached to the server via a disk bus such as Small Computer Standard Interface (SCSI) or Integrated Drive Electronics (IDE) as shown in Figure 1 [7, 13, 14, 20]. Their focus is on reliable storage methods that can survive a single or even double disk failure. Such systems are very common, and are regularly implemented in both hardware and software. These systems can be costly in comparison to non-redundant systems, and upgrades generally require the replacement of all of the disks in the system with larger disks. In many Local Area Network (LAN) implementations, when the central storage becomes full, an upgrade of the central storage or purge of data is required; workstation storage is often vastly underutilized.

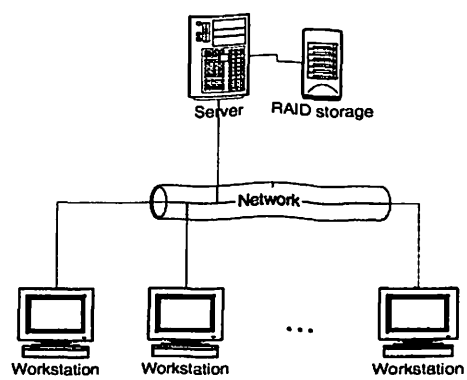


Figure 1: Typical LAN storage system

Research has been done on distributed storage systems, utilizing the storage capabilities of a network of computers as a single volume [4, 8, 12, 16, 28, 29, 30]. Most of this

research has focused on distributed storage for large clusters of computers which make up a supercomputer or a set of reliable, networked storage servers. Workstations in the can be unreliable and uniquely problematic in the hands of users. Prior studies that utilize workstation storage in a distributed manner have focused on file mirroring techniques which yield a low percentage of usable disk space and often no guarantee on availability [1, 5].

II. Background

2.1 RAID Background Information

Disk arrays are a way to increase performance and reliability over a single disk. By spreading data over multiple disks, called striping, disk arrays improve performance by simultaneously utilizing multiple disks and presenting them as a single RAID volume. However, since more disks are involved, this can decrease reliability unless redundant disks are included in the array to tolerate failures. Many options are available for the RAID storage as described by the most common RAID levels. Note that all RAID levels are implemented at the block level, such that any file system could be used on top of the RAID volume [7].

2.2 RAID 0 (Striping)

RAID 0 spreads the data across all of the disks to achieve a performance increase using as few as 2 disks. Given n disks, the data is split up into blocks of equal size and written such that block b is stored on drive $b \bmod n$. Thus, consecutive blocks are always written to different disks, in order. A series of n consecutive blocks across the n disks is called a stripe. Both read and write performance is increased especially for large accesses. Despite the name, RAID 0 offers no redundancy; if any of the disks fail, the data on the array cannot be completely recovered [7, 13].

2.3 RAID 1 (Mirroring)

RAID 1 is the simplest form of redundancy in which data written to one disk is also written to another so there are always two copies of the data. Writes are done simultaneously to both disks so the performance is similar to that of a single disk. By combining RAID 0 with RAID 1, mirroring can be achieved with any even number of disks and also benefit from the performance increases associated with striping. If any disk fails, the disk's mirror is used to retrieve the corresponding data [7, 13, 20].

2.4 RAID 4 (Parity)

RAID 4 can be considered as a RAID 0 array with one additional disk, therefore requiring at least 3 disks. The extra disk is used to store parity information for each of the stripes. Read performance is similar to that of RAID 0, but write performance can be much slower because the parity disk must be updated on every write and is therefore a bottleneck. If any disk fails, the other stripe information is combined with the information on the parity disk to calculate the failed drive's data [7, 13, 20].

2.5 RAID 5 (Interleaved Parity)

RAID 5 works in exactly the same way as RAID 4 except that the parity information is interleaved across all of the disks to eliminate the single bottleneck parity disk. Since RAID 5 offers a performance increase over RAID 4 with the same redundancy, RAID 4 is almost never implemented in practice [7, 13, 20].

2.6 Related Works Review

There have been several studies and implementations of distributed storage systems. The focus of such studies has been decentralizing storage and utilizing the capacity of several reliable networked servers. These systems generally rely on replication techniques similar to RAID 1 or parity to distribute the data similar to RAID 5, often built on top of a custom file system.

Coda: Coda is a distributed file system developed at Carnegie Melon University by the systems group of M. Satyanarayanan in the School of Computer Science. It is based off several central servers and replication. It also allows for a cached, disconnected operation allowing offline use of the files on the system [28].

Intermezzo: Intermezzo is also developed at Carnegie Melon University, and was inspired by the Coda project. Like Coda, it is based off central servers and replication and also allows for offline use and synchronization, but is implemented with a simpler design [4].

Lustre: The same group that develops Intermezzo (Cluster File System) is also working on a system called Lustre. Lustre is object based and abandons block-based file systems. It is oriented toward a new storage paradigm but is still based on a large number of fairly reliable nodes with uses on today's largest clusters [8].

Frangipani/Petal: Frangipani is a file system designed to be used on top of Petal, which is a distributed, block-based storage system [16]. It is designed for a set of central servers and can only sustain a single failure [30].

Sistina GFS: Sistina GFS is a commercial file system designed for cluster systems similar to those targeted by Lustre, a large number of fairly reliable nodes [29].

Berkley xFS: xFS is a serverless distributed file system designed to distribute data over cooperating workstations. It uses parity information to guarantee no individual node is the single point of failure. The goal of the system is a high performance, scalable storage system based on workstations connected via a very fast network like ATM or Myrinet. Since it is based off parity for replication, the system can only sustain a single failure [1]. The system also appears to be unfinished and has not been updated in several years.

Serverless Distributed File System: Bolosky et al. at Microsoft Research have published a paper on a serverless distributed file system based on replication of files amongst the peer nodes. By making multiple replicas of each file and distributing them amongst the client machines, the system provides high availability and high reliability. The system is unique in that it does not assume a mutual trust among the client computers, eliminating the need for central administration [5].

Google FS: Ghemawat et al. have developed a file system for the Google search services designed to run on inexpensive commodity hardware while maintaining high

performance. It is based off a master server and uses block level replication for redundancy. It has been used successfully to store hundreds of terabytes of data across more than a thousand nodes [12].

2.7 Reed-Solomon Coding

Reed-Solomon coding extends beyond the traditional RAID levels to allow for multiple simultaneous failures. James S. Plank gives an excellent description of Reed-Solomon coding in [24], and even briefly describes a system similar to the proposed system, but only for checkpointing [22, 23], not for a general purpose storage system. Plank defined the problem domain of Reed-Solomon coding as follows:

> Let there be n storage devices, $D_1, D_2, \ldots, D_n$, each of which holds k bytes. These are called the *"Data Devices."* Let there be m more storage devices $C_1, C_2, \ldots, C_m$, each of which also holds k bytes. These are called the *"Checksum Devices."* The contents of each checksum device will be calculated from the contents of the data devices. The goal is to define the calculation of each C_i such that if any m of $D_1, D_2, \ldots, D_n, C_1, C_2, \ldots, C_m$ fail, then the contents of the failed devices can be reconstructed from the non-failed devices [24].

First the Data and Checksum devices must be split up into consecutive words of length w, where w is generally 8 or 16 bits. Thus, D_i consists of words $d_{i,1}, d_{i,2}, \ldots, d_{i,k/w}$, and C_i consists of words $c_{i,1}, c_{i,2}, \ldots, c_{i,k/w}$. To simplify, the second subscript can be dropped and Reed-Solomon encoding can be defined in terms of blocks of length w, $d_1, d_2, \ldots, d_n$, $c_1, c_2, \ldots, c_m$.

Per Plank's correction in [25], it is necessary to construct a dispersal matrix, B, such that:

- It is an $(n+m)\times n$ matrix.
- The $n\times n$ matrix in the first n rows are the identity matrix.
- Any submatrix formed by deleting m rows of the matrix is invertible.

The dispersal matrix B can be created by starting with a Vandermonde matrix. Define the $n+m$ by n Vandermonde matrix V such that $v_{i,j} = j^{i-1}$:

$$V = \begin{bmatrix} 0^0 & 0^1 & 0^2 & \cdots & 0^{n-1} \\ 1^0 & 1^1 & 1^2 & \cdots & 1^{n-1} \\ 2^0 & 2^1 & 2^2 & \cdots & 2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (n+m-1)^0 & (n+m-1)^1 & (n+m-1)^2 & \cdots & (n+m-1)^{n-1} \end{bmatrix}$$

By definition, V has the property that any submatrix formed by deleting m rows is invertible [17]. V can then be converted to the desired B and still retain this property by performing column-wise Gaussian elimination on the first n rows, such that $V' = B = \begin{bmatrix} I \\ Z \end{bmatrix}$, where I is the $n \times n$ identity matrix and Z is the remaining lower m rows of B forming an $m \times n$ matrix. Now construct the vectors $D = [d_1, d_2, ..., d_n]^T$ and $C = [c_1, c_2, ..., c_m]^T$. C is then calculated by the matrix-vector multiplication $ZD=C$.

Recovery also employs the B matrix defined above and involves Gaussian elimination as follows. Define $n+m$ vector $E = \begin{bmatrix} D \\ C \end{bmatrix}$. Thus $BD=E$. Now suppose t devices fail such that $t \leq m$. Now define B' and E' from B and E by removing the rows corresponding to the failed devices. Now $B'D = E'$ and D can be determined using Gaussian elimination, which is guaranteed to succeed due to the linearly independent properties of B, and thus all Data Devices can be recovered. Any failed Checksum Devices can then be recovered from the $ZD=C$ equation above [24].

It should also be noted that if a data word changes from d_j to d_j' then $c_i' = c_i + z_{i,j}(d'_j - d_j)$. Thus, the new checksum data can be calculated using the old checksum data, an item from the Z matrix and the difference between the old data and the new data. Lastly, while the above equations are guaranteed to work with infinite precision mathematics, a Galois Field with 2^w elements must be utilized for all calculations. Thus addition and subtraction are replaced by the XOR operation while multiplication and division involve a table of logarithms [24].

It should also be noted that the particular variation on Reed-Solomon Coding used here is denoted by Plank as RS-Raid, which is only used as an erasure code algorithm. A complete, much slower implementation of Reed-Solomon Coding like the one found in [15] and described in [3] and [21] can handle not only erasures but also errors, and is often used for media-storage such as CD-ROMs as well as applications in Forward Error Correcting [26,31].

III. Problem Domain

This thesis investigates the feasibility of a distributed storage system to utilize the unused disk storage on a LAN of workstations, creating a network storage system that scales in capacity with the number of workstations. Since workstations are essentially controlled by the user, they are inherently unreliable, as the user may reboot or shutdown the machine at any time. As such, a system based off the storage of workstations must be able to handle a very large percentage of simultaneous failures.

Most of the existing distributed storage systems reviewed above would be unsuitable for such an environment since they can generally only handle a small number of simultaneous failures and are designed for use on fairly reliable nodes. The serverless distributed file system by Bolosky et al. at Microsoft research is an exception, and was designed to run on workstations. However, it is also based on the file replication which is the most inefficient in terms of storage space to provide redundancy. The system is also based off a custom file system requiring custom client software.

III. Hypothesis

It is hypothesized that a distributed RAID system can be developed for a group of unreliable storage devices by utilizing Reed-Solomon coding to provide configurable redundancy to handle multiple simultaneous failures. Furthermore, such a system can provide adequate performance similar to that of a local RAID system. For the purposes of testing the system, adequate performance is defined as reading and writing data with a sustained data throughput at least 50 percent that of a local RAID system, and a burst throughput at least 90 percent that of a local RAID system. Such performance would be considered a reasonable trade-off for the reduced cost of implementing a distributed RAID system on existing workstations versus a local RAID system.

IV. Foundations

5.1 Overview

This distributed storage system was designed with a medium-sized LAN in mind, such as an educational institution's computer labs, or a medium to large business. Each workstation will have a block of storage allocated for use by the distributed system. Further, each block will be designated as either a Data Device or a Checksum Device. The Reed-Solomon encoding algorithm allows an arbitrary number of Checksum Devices to be designated. For each failed Data Device, the data from a single Checksum Device, along with the data from the rest of the Data Devices is needed to calculate the missing data. Thus any combination of Data Devices and Checksum Devices can fail simultaneously, as long as the number of simultaneous failures does not exceed that of the number of Checksum Devices.

These devices will be made available only to a central coordinating server via a network. The central server will create the logical volume from the devices and handle all read and write requests, as well as the Reed-Solomon calculations. Since the system is block-based, any file system could be implemented on top of the logical volume. Additionally, the logical volume could be made available to any client on the network via existing protocols such as NFS, or SMB.

Figure 2 shows the physical network connections of the system, and Figure 3 shows the logical flow of data during a typical read or write. Notice that all of the data must travel through the coordinating server where the Reed-Solomon coding takes place. Also note that a RAID Data Workstation could also make a read or write request to the logical volume, but the flow of data would remain the same.

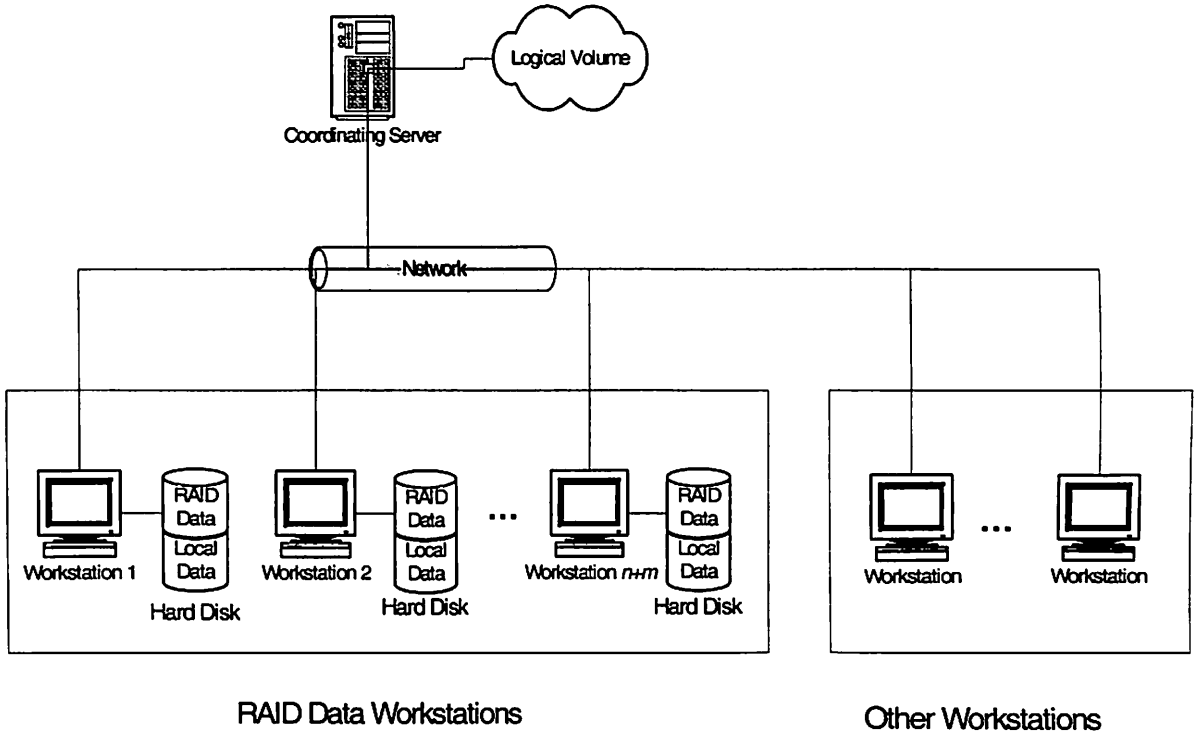


Figure 2: Proposed distributed RAID system.

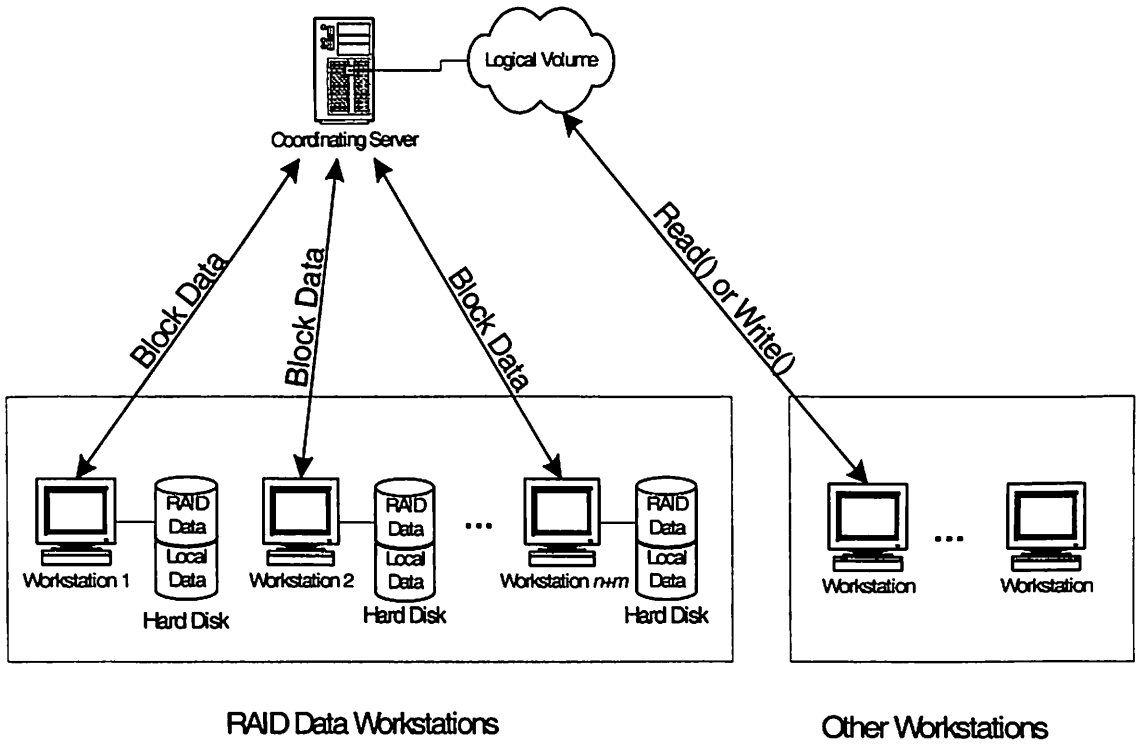


Figure 3: Logical flow of data in the proposed RAID system.

5.2 Proposed Approach

To fully analyze such a proposed distributed storage system, several aspects of the system must be examined. First, both read and write performance must be evaluated in the situation where all nodes are online, as well as when one or more nodes are offline. If performance is deemed adequate when the system is in a stable state, the performance during rebuild must be considered, after a node goes offline then comes back online, similar to the analysis done in [14]. The ability and associated performance considerations could also be examined when a node is added or removed permanently from the system.

5.3 Introductory Setup

The first step in describing a RAID system is to designate a stripe size. That is, given data of size S, it must be split into s equal size segments $S_1,S_2,S_3,\ldots,S_s$. The resulting s segments must then each be split up into n equal size blocks, each one stored sequentially on each of the n Data Devices. Thus, S_i is split up into $(S_{i,1},S_{i,2},\ldots,S_{i,n})$. The result thus far is exactly equivalent to Raid 0, striping. To add fault tolerance, for every segment $S_1,S_2,S_3,\ldots,S_s$, we must designate m additional blocks of calculated, redundant data (R) to be placed sequentially on the Checksum Devices. Note that if m equals 1, and the checksum algorithm was parity instead of Reed-Solomon coding, it would be equivalent to a RAID 4 system.

	n Data Devices				m Checksum Devices			
Data Segments	D_1	D_2	...	D_n	C_1	C_2	...	C_m
S_1:	$S_{1,1}$	$S_{1,2}$	...	$S_{1,n}$	$R_{1,1}$	$R_{1,2}$	...	$R_{1,m}$
S_2:	$S_{2,1}$	$S_{2,2}$	...	$S_{2,n}$	$R_{2,1}$	$R_{2,2}$	...	$R_{2,m}$
⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋱	⋮
S_s:	$S_{s,1}$	$S_{s,2}$	...	$S_{s,n}$	$R_{s,1}$	$R_{s,2}$	...	$R_{s,m}$

Table 1: The relationship between Data Devices, Checksum Devices, and Data Segments.

Now, if only a partial segment is needed, and the appropriate Data Devices are available, that data may be retrieved directly from the Data Devices with no computation. However, if one of the Data Devices is not available, the entire segment must be

reconstructed from the remaining blocks from the Data Devices, plus one block from a corresponding Checksum Device for each missing Data Device.

For example, if there are 50 data devices (1 through 50) and 50 Checksum Devices, but Data Devices 46 through 50 of the data devices are unavailable (offline) at the moment. A read is being processed by the central coordinating server that requests Segment *i*, blocks 25 through 50. This request cannot be processed directly since Data Devices 46 through 50 are offline. Thus, the entire Segment *i* must be reconstructed on the central coordinating server. The server must request 50 blocks from 50 different devices to reconstruct the Segment *i*. The beauty of Reed-Solomon coding allows the coordinating server to request the blocks from *any* 50 unique devices, there is no need to request blocks 1-45 from Data Devices 1-45, and then checksum data from 5 of the Checksum Devices. To maximize throughput, the coordinating server would need to be designed to make such requests to the 50 least used Devices when reconstructing an entire segment.

This system allows for very configurable redundancy, which means reliability is customizable, at the expense of storage space and computation time. If the reliability of the workstations to be used as Devices in the system is carefully analyzed prior to implementation, the reliability of the Distributed RAID system can be configured as needed by determining *n* and *m* to maximize storage space while maintaining an appropriate level of redundancy.

Note that stripe size (the size of a block of a segment $S_{i,j}$) can greatly affect performance. For example, if the stripe size is too small, incoming requests will be more likely to require more than one stripe to fulfill and therefore the coordinating server will need to send out a large number of requests to Data Devices and Checksum Devices to fulfill each incoming request, and the required overhead for each packet of data could become excessive. On the other hand, if the stripe size is too large, in the case where an entire segment must be reconstructed, a very large amount of data may need to be requested from the Data Devices and the Checksum Devices in order to fill a relatively small incoming request. For the distributed RAID system proposed, stripe size will need to be determined on a system by system basis. The type of data the system holds, the request pattern of clients, the network throughput, the network latency, and the availability of the Data Devices will each play a role in determining stripe size.

5.4 Experimental Methodology

Since no distributed system shares the same goals as the system proposed, the system cannot fairly be tested against any such system. However, the proposed system is essentially an alternative for a single centralized server utilizing a few large drives in a RAID 5 configuration, and thus will be used as the comparison system during experimentation. Though the two systems are not congruent, it is a valid comparison, as it is the same comparison system administrators would make if deciding which system to implement. The single, central RAID 5 system may be faster, but if the proposed system can at least remain competitive, it becomes a viable alternative for system administrators

since a distributed RAID system would utilize otherwise unused storage located on workstations, and thus, could be cheaper and scale better to the users' needs.

The distributed RAID test system was comprised of a number of workstations connected to a central server via Network Block Devices (NBD). The central server is responsible for the Reed-Solomon calculations. For comparison, the same central server was also equipped with several drives locally and configured for RAID 5 via Linux software RAID.

To measure array performance, a disk benchmark utility called IOzone was run on both systems to highlight the strengths and weaknesses of each. IOzone performs a wide variety of tests on a range of parameters to determine a system's abilities in terms of file size access, cache performance and limitations, etc [6]. Specifically, IOzone was used to measure sustained throughput and burst throughput while varying the request (record) size to simulate various usage types. IOZone is a popular tool that has been used extensively to test various storage systems [1, 11, 16]. While there are efforts to improve mass storage system benchmark tools [19], IOZone remains one of the most common, and is well suited for this application.

VI. Experimental Results

6.1 Environment

The coordinating server was a dual 533Mhz Xeon processor system with 512MB of RAM, and six 7200RPM 9GB Ultra-SCSI hard disks for local testing. The system was connected to seven other machines through NBD via a switched 100Mbit network for distributed testing.

The Linux Software Multi-Device (MD) driver served as a basis for both the local and distributed testing. The Linux Kernel version 2.6.2 was employed with the RAID5 block driver as well as a custom RS-RAID driver written with the assistance of [27] and [9]. A portion of the RS-RAID driver can be found in Appendix B. Due to limitations in the Linux kernel and MD architecture, the stripe size was fixed at 4KB. However, since the NBD protocol has very little overhead, this was not a problem in the distributed system.

The custom RS-RAID driver was based on the work of Evan Danaher [10], then modified to be highly optimized for w=8 bits, including a pre-built lookup table for the Galois field multiplication and division. The interface to the MD driver was then based off the existing RAID5 and RAID6 drivers, expanding the number of parity devices to m versus the one of RAID5 or two of RAID6.

For all tests, IOZone was used to measure throughput on increasing file sizes from 64 kilobytes to 1 gigabyte in order to exceed the coordinating server's RAM size and thus internal disk cache so that disk-level performance could be measured. Thirteen tests (writer, rewriter, reader, rereader, random read, random write, backward read, record re-write, stride read, fwrite, fre-write, fread, and fre-read) were performed in each scenario. However, only the read and write tests were analyzed, as they clearly show the burst and sustained throughput for reads and writes. The tests were performed with a range of record sizes from 4k to 16M, though record sizes of 32K and lower were not tested with file sizes 32M or higher to save time. All of the test arrays were created with a chunk size of 32KB and formatted as a single ext2 file system with a block size of 4KB.

6.2 RAID 5 Control Test

Figures 4 and 5 show the baseline RAID 5 test, to which all other tests can be compared. This is the "normal" speed that a server disk system would operate. All 6 local disks were utilized during this test.

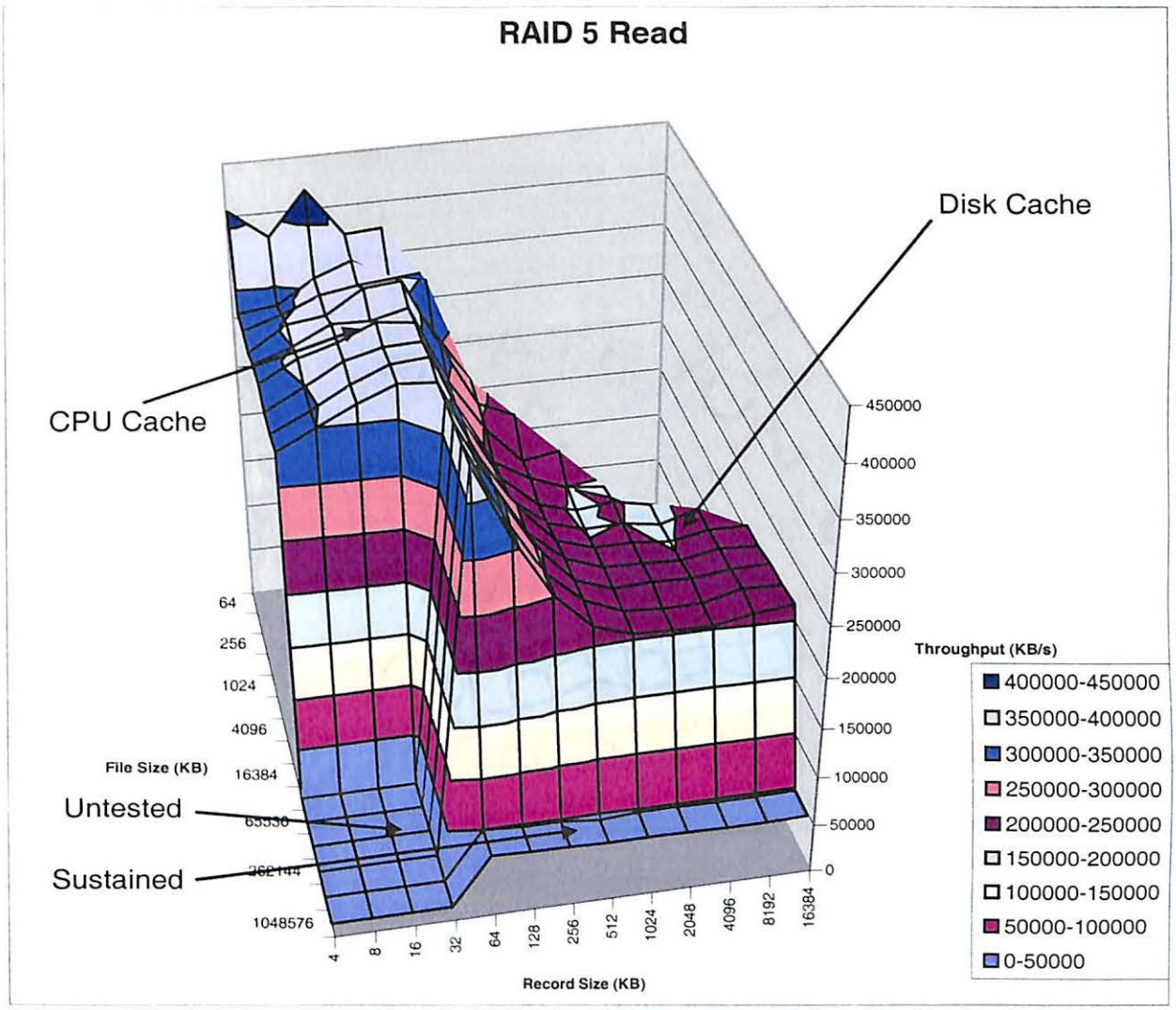


Figure 4: Local RAID 5 IOZone Results (Read)

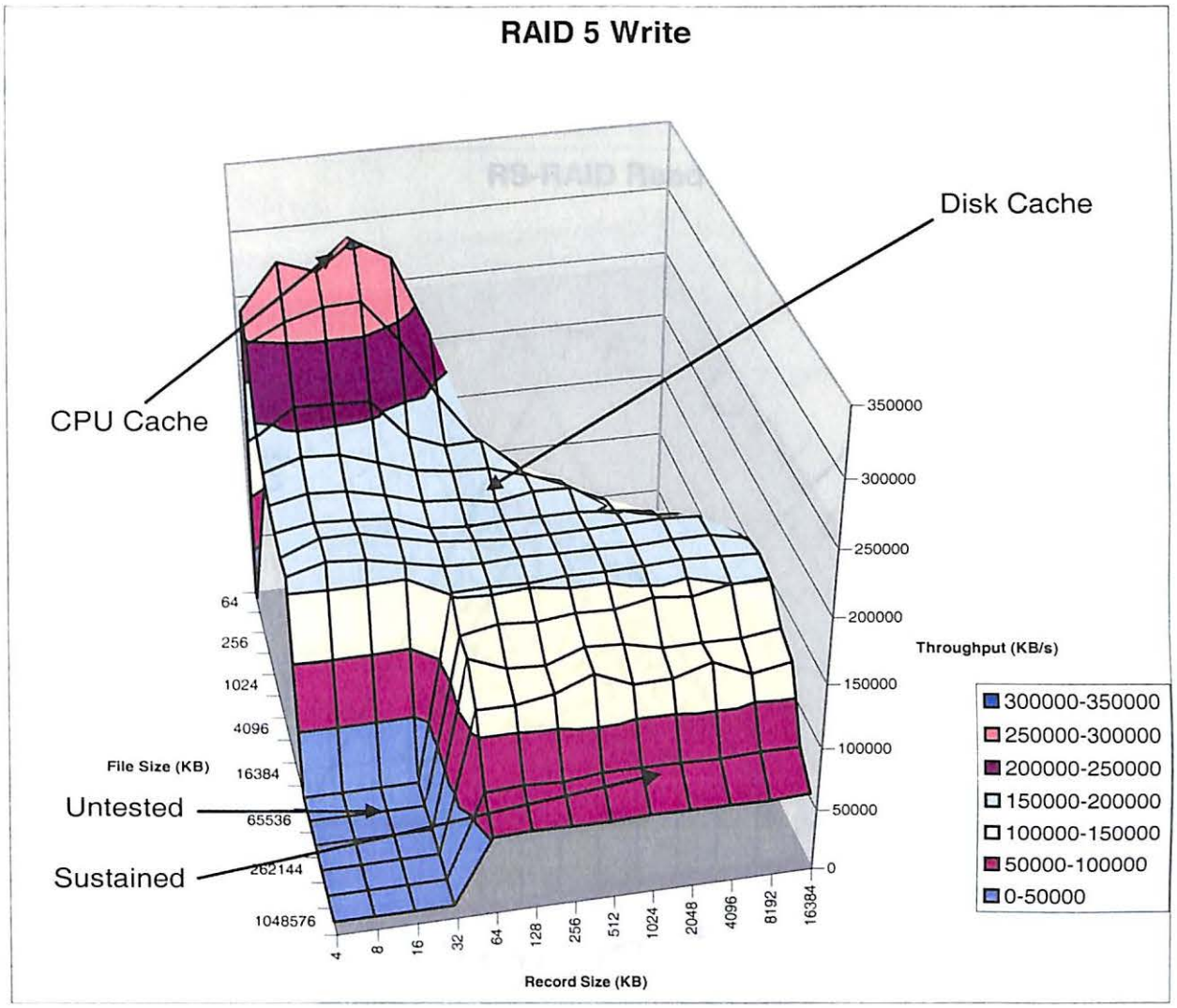


Figure 5: Local RAID 5 IOZone Results (Write)

The effects of the CPU cache and Disk Cache can clearly be seen with a burst speed well above 300MB/sec, but the last test with the 1GB file size shows sustained data rate of approximately 50MB/sec for both reads and writes.

6.3 Distributed Reed-Solomon Test, Partially Degraded

The first Reed-Solomon RAID test was a distributed test with 5 of the 7 distributed nodes such that n=3, m=4. This test shows how the RS-RAID algorithm can perform with 2 of

the 3 data disks failed, but with all 4 parity disks in tact, about 30% failure. While it taxed the CPU heavily, it did not max it out the entire time.

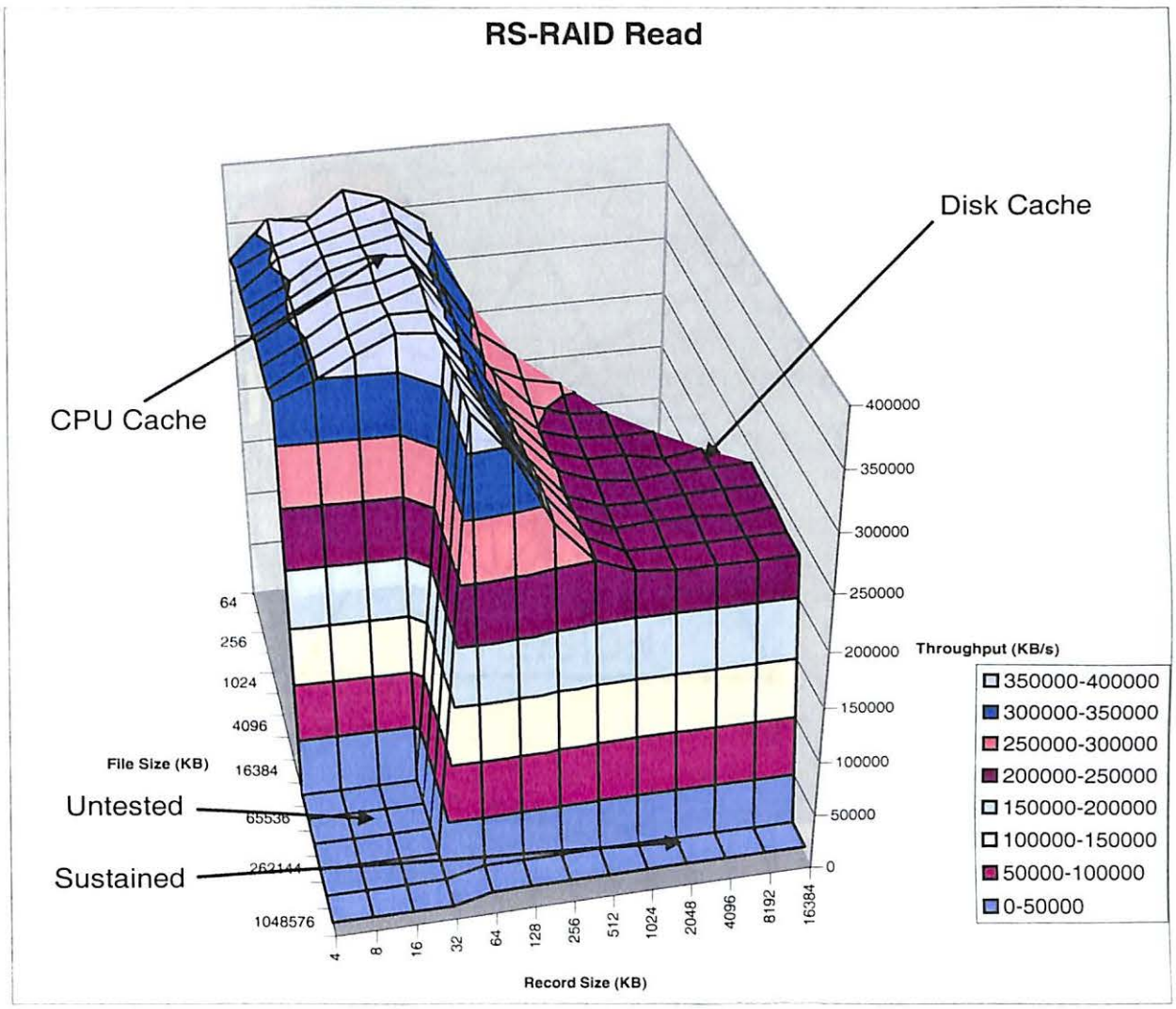


Figure 6: Distributed Reed-Solomon, Some Degraded IOZone Results (Read)

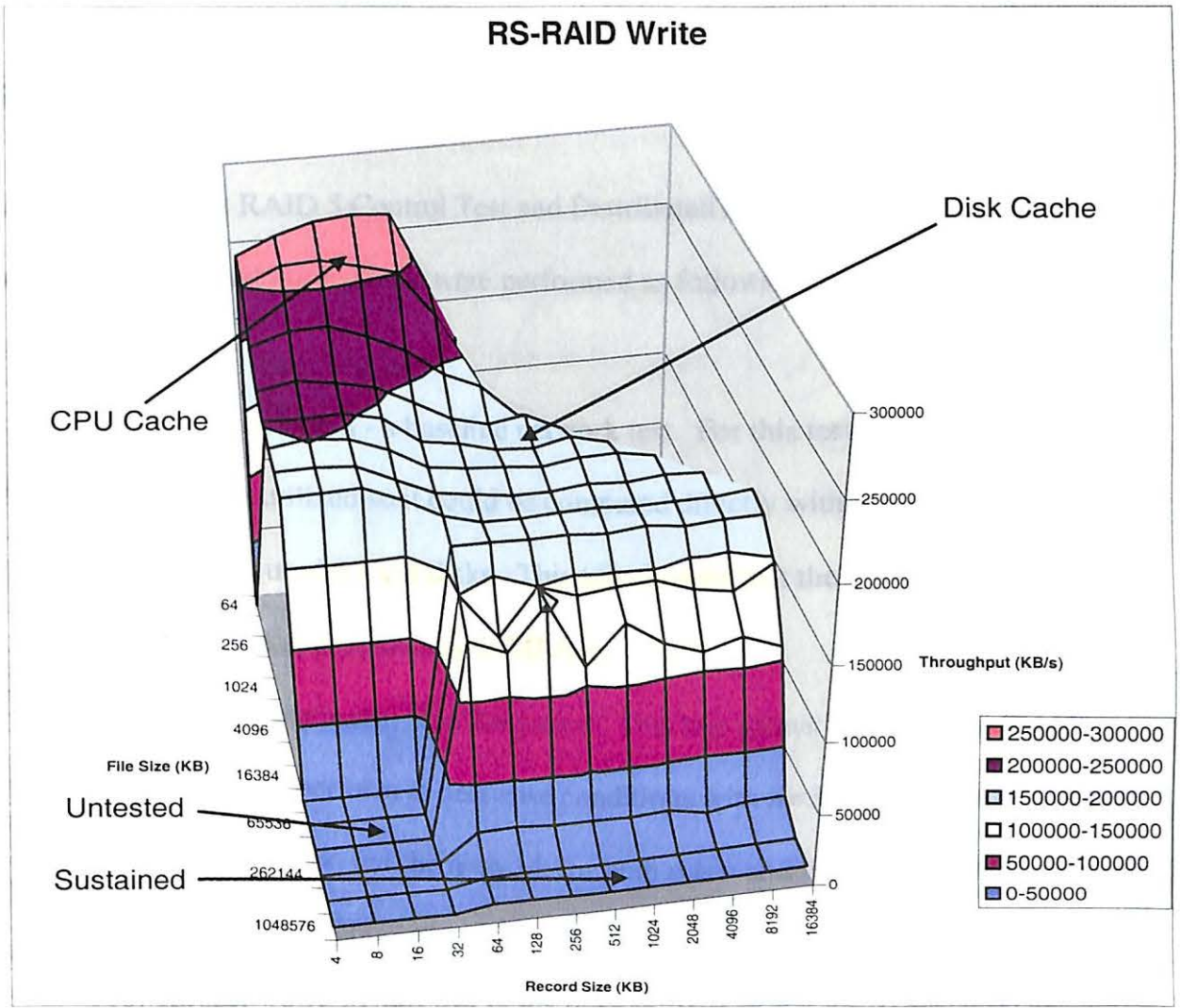


Figure 7: Distributed Reed-Solomon, Some Degraded IOZone Results (Write)

It is clear from Figures 6 and 7 that RS-RAID can compete with RAID5 in burst throughput, but this is not surprising as the burst throughput is almost entirely dependent on the operating system's disk cache as well as the CPU cache; the underlying storage mechanism is irrelevant. However, it appears that the RS-RAID test shown above was limited by the 100Mbit switched network with a sustained data rate for both reads and

writes around 10MB/sec. Thus, more tests were performed to get a better idea of how RS-RAID and RAID 5 compare.

6.4 Additional Tests

In addition to the RAID 5 Control Test and Distributed RS-RAID, Partially Degraded tests above, four additional tests were performed as follows:

- RAID5 Distributed - a baseline network test. For this test, 6 of the distributed nodes were utilized so it could be compared directly with the RAID5 Control Test, which used 6 local disks. This test showed that the network can be a bottleneck at approximately 10MB/sec.
- RS-RAID Distributed, Non-Degraded. This test shows how the RS_RAID algorithm can perform at best-case conditions with $n=3$ and $m=4$ with no failed devices. It maxed out the network for both reads and writes while still consuming a fair amount of CPU time. Reads had a sustained throughput of approximately 10MB/sec, while writes, due to the need to write to the parity devices, sustained approximately 5MB/sec.
- RS-RAID Distributed, Fully Degraded. This test shows how the RS-RAID algorithm can perform in worst-case scenario with $n=7$, $m=7$, with exactly n devices operational, 3 data and 4 parity. Since RS-RAID is a processor-intensive algorithm, this test was designed to see if the CPU could become the limiting factor in a fairly small system. It heavily taxed the CPU, with the network no longer the limiting factor.

- RS-RAID Local, Partially Degraded – In a typical implementation, it is expected that the distributed RS-RAID would usually work at some level of degradation but not the maximum. This test shows how well the algorithm performs assuming the network is not a limiting factor. This test maxed out at least 1 CPU for the duration of the test. It is a direct comparison for RS-RAID Distributed, Partially Degraded. It shows the maximum throughput of the algorithm, on that central server, with n=3, m=4, and 2 data devices failed.

Figures 8 and 9 below are graphs of the sustained data rate, using the 1GB file size over the various record sizes of all 6 tests. Note that record size did not significantly affect sustained data rate for any of the tests.

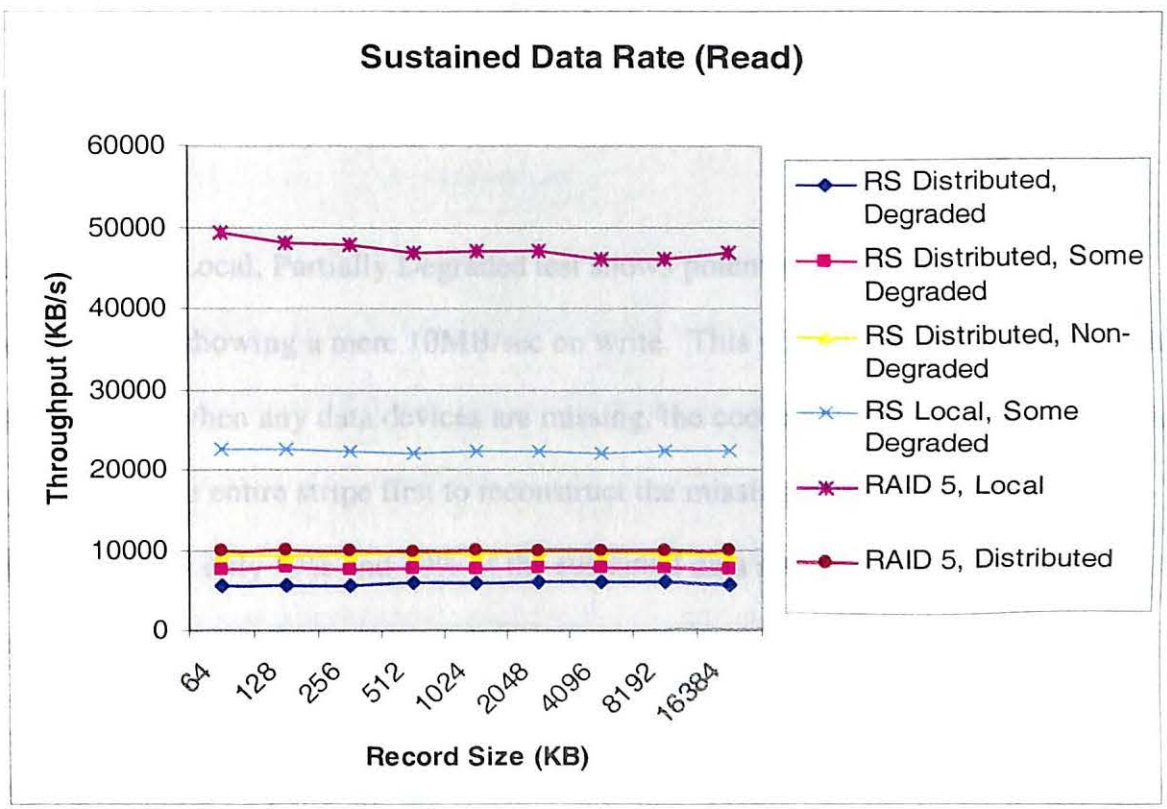


Figure 8: Sustained Data Rate IOZone Results (Read)

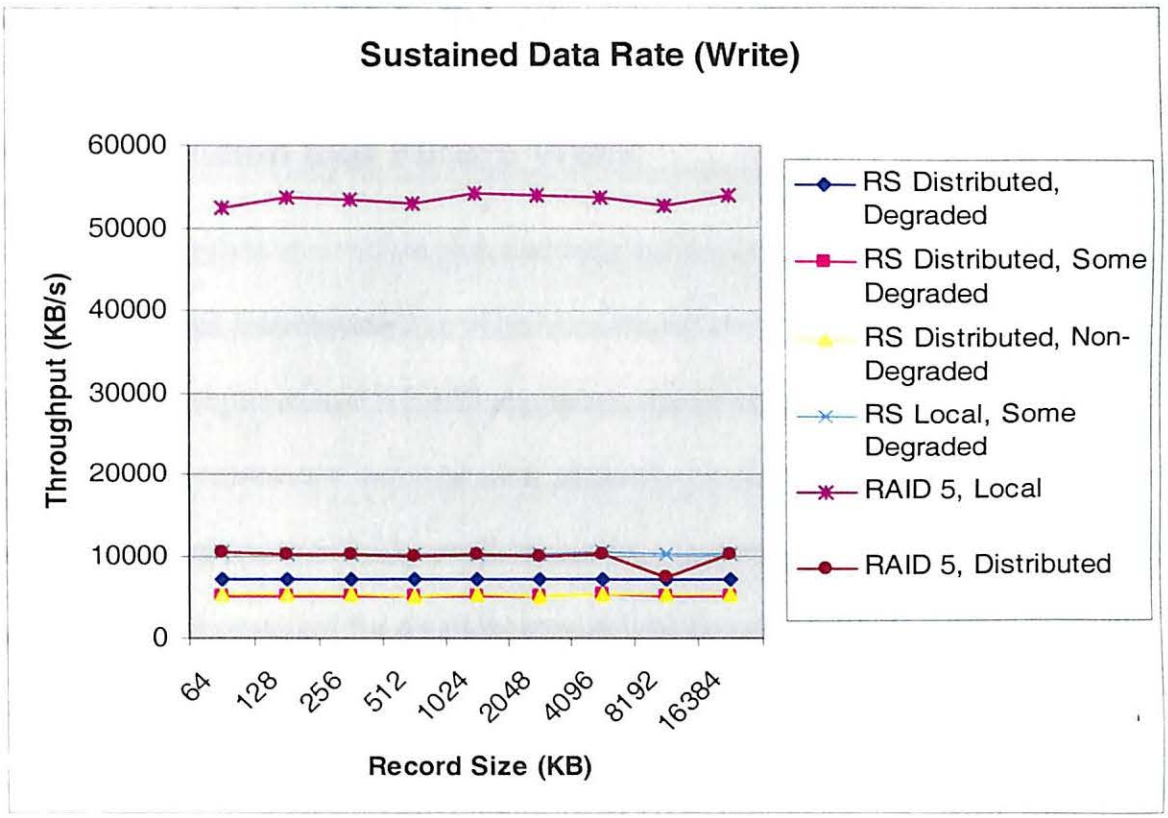


Figure 9: Sustained Data Rate IOZone Results (Write)

The RS-RAID Local, Partially Degraded test shows potential, reaching 22MB/sec on reads, but only showing a mere 10MB/sec on write. This poor write performance is due to the fact that when any data devices are missing, the coordinating server almost always needs to read the entire stripe first to reconstruct the missing data then do the write. However, this is only 47% and 19% of the sustained data rate for reads and writes achieved by the local RAID5 array, respectively.

The Distributed tests (both RS and RAID5) also shows that in a real-world distributed system, a switched 100Mbps network would not be nearly fast enough to handle a

distributed RAID system; 1000Mbps would certainly be necessary to match the 22MB/sec achieved by the RS-RAID Local test.

VII. Conclusion and Future Work

7.1 Experimental Conclusion

Despite a highly optimized RS-RAID algorithm, the Gaussian elimination of data recovery and matrix-vector multiplication of checksum data calculations proved to be a significant bottleneck with fairly small values for n and m when testing for sustained throughput. Higher values for n and m, as would be found in a medium-size business or large educational computer lab would certainly produce similar or worse sustained throughput due to the $O(n^2)$ nature of the matrix calculations. Clearly, distributed RAID via Reed-Solomon coding is not a viable alternative to a local RAID 5 system for general use where sustained throughput is a concern.

However, such a distributed RAID system may have other uses. Such a system could be used as a backup system for a local disk array in lieu of an expensive tape backup system. Its distributed nature is ideal for a backup system, especially if the nodes of the distributed system are in different physical structures, providing additional means for disaster recovery. A distributed RAID system may also have uses in an environment where burst throughput is important, but sustained throughput is not. One such example is a large database, where a small number of records are accessed very frequently.

7.2 Future Work

Transaction Log: Though it was not tested in this experiment, it is apparent that the slow performance of a rebuild of a node after it comes back online would be detrimental to the system, especially considering the frequency of expected node failures. (i.e. a user rebooting a workstation would require that node to be rebuilt.) To counteract this situation, a transaction log could be added, located on the central coordinating server. It would simply maintain a list of writes over a given period of time. Thus, if a node is down for a short period of time, that is, less than the amount of time covered by the transaction log, the central server can retrieve the changed data from the other nodes and send the node the writes that occurred during its offline period. This would prevent the need for a full synchronization.

Central Disk Cache: In addition to the operating system's built-in disk cache in memory, it is possible to consider a disk cache, stored on one of the central server's local disks. This would create another performance plateau between that of the memory cache and the distributed storage, thus increasing average performance, but would not change the sustained throughput of the system.

Distributed Processing: With a custom client, it may be possible to speed up the system by utilizing each node's processor to facilitate the Reed-Solomon coding, particularly on writes. A write to a block would consist of sending the new data to the data device, plus the difference between the old and new data to the checksum devices $(d'_j - d_j)$, which

would then read their own checksum data for that stripe and recalculate the checksum using $c_i' = c_i + z_{i,j}(d'_j - d_j)$ as mentioned above.

Interleaved Parity: To simplify coding, testing, and debugging, the system was set up analogous to RAID 4, with designated Data Devices and Checksum Devices. In order to alleviate the bottleneck of needing to write to the same Checksum Devices for every write, checksum data could be interleaved amongst the $n + m$ disks. This would improve performance in the same way that RAID 5 improves over RAID 4.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz and A. Sussman. *Workshop on I/O in Parallel and Distributed Systems.* Proceedings of the fourth Workshop on I/O in Parallel and Distributed Systems: Part of the Federated Computing Research Conference, Philadelphia, PA, May 1996, pages 15-27.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang. *Serverless Network File Systems.* ACM Transactions on Computer Systems, Volume 14, Issue 1, February 1996, pages 41-79.

[3] E. Berlekamp, *Algebraic Coding Theory,* McGraw-Hill, New York, 1968.

[4] P. Braam, M. Callahan and P. Schwan. *The InterMezzo File System.* The Perl Conference 3.0, O'Reilly Open Source Convention, Monterey, CA, August 1999.

[5] W. Bolosky, J. Douceur, D. Ely and M. Theimer. *Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs.* ACM SIGMETRICS Performance Evaluation Review, June 18-21, 2000, pages 34-43.

[6] D. Capps, *Iozone Filesystem Benchmark.* http://www.iozone.org/. Date of access: November 15, 2003, Date of creation: unknown, Date of update: April 29, 2003.

[7] P. Chen, E. Lee, G. Gibson, R. Katz and D. Patterson. *RAID: High-Performance, Reliable Secondary Storage.* ACM Computing Surveys, Volume 26, Issue 2, June 1994, pages 145-185.

[8] Cluster File Systems, Inc. *Lustre: A Scalable, High-Performance File System.* http://www.lustre.org/docs/whitepaper.pdf, Date of Access: November 2003, Date of creation: November 11, 2002, Date of update: None.

[9] J. Corbet. *Porting Device Drivers to the 2.6 Kernel.* http://lwn.net/Articles/driver-porting/, Date of Access: February 2004, Date of Creation: October 2003, Date of update: January 2004.

[10] E. Danaher. *Distributed Storage.* http://www.tjhsst.edu/~edanaher/techlab/index.php, Date of access: February 2004, Date of creation: September 2003, Date of update: January, 2004.

[11] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller and L. Reuther. *The SawMill Multiserver Approach.* Proceedings of the Ninth Workshop on ACM SIGOPS European Workshop, Kolding, Denmark September 2000, pages 109-114.

[12] S. Ghemawat, H. Gobioff and S. Leung. *The Google File System.* Proceedings of the nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, October 2003, pages 29-43.

[13] J. Hennesy and D. Patterson. *Computer Architecture A Quantitative Approach, Third Edition.* Morgan Kaufmann Publishers, San Francisco, CA, 2003.

[14] H. Kari, H. Saikkonen, N. Park and F. Lombardi. *Analysis of Repair Algorithms for Mirrored-Disk Systems.* IEEE Transactions on Reliability, Volume 46, Issue 2, June 1997, pages 193-200.

[15] P. Karn. *Forward Error Correcting Codes.* http://www.ka9q.net/code/fec/, Date of update: August 2003, Date of access: February 2004, Date of creation: unknown.

[16] E. Lee and C. Thekkath. *Petal: Distributed Virtual Disks.* Proceedings of the seventh international conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, October 1996, pages 84-92.

[17] F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes, Part I.* North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[18] D. Michail, Flouris, P. Evangelos and Markatos. *The Network RamDisk: Using remote memory on heterogeneous NOWs.* Cluster Computing, Volume 2, Issue 4, 1999, pages 281-293.

[19] E. Miller, *Towards Scalable Benchmarks for Mass Storage Systems.* Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, College Park, MD, September 1996, pages 515-528.

[20] D. Patterson, G. Gibson and R. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID).* Proceedings of the 1988 ACM SIGMOD Conference on Management of Data, Chicago, IL, June 1988, pages 109-116.

[21] W. Peterson and E. Weldon, Jr. *Error-Correcting Codes, Second Edition.* The MIT Press, Cambridge, Massachusetts, 1972.

[22] J. Plank and K. Li. *Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations.* Proceedings of the twenty-fifth International Symposium on Fault-Tolerant Computing, Pasadena, CA, June 1995, pages 351-360.

[23] J. Plank, *Improving the Performance of Coordinated Checkpointers on Networks of Workstations Using RAID Techniques.* Proceedings of the fifteenth Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, Canada, October 1996, pages 76-85.

[24] J. Plank. *A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems.* Software Practice & Experience, Volume 27, Number 9, September 1997, pages 995-1012.

[25] J. Plank and Y. Ding. *Note: Correction to the 1997 Tutorial on Reed-Solomon Coding.* Technical Report UT-CS-03-504, University of Tennessee, April, 2003.

[26] M. Purser. *Introduction to Error Correcting Codes.* Artech House, Boston, 1995.

[27] A. Rubini and J. Corbet. *Linux Device Drivers, Second Edition.* O'Reilly & Associates, Inc., 2001.

[28] M. Satyanarayanan. *Coda: A Highly Available File System for a Distributed Workstation Environment.* IEEE Transactions on Computers, Volume 39, Number 4, April 1990, pages 447-459.

[29] Sistina Software, Inc. *Sistina GFS.* http://www.sistina.com/downloads/datasheets/GFS_datasheet.pdf, Date of access: October 31, 2003, Date of creation: unknown, Date of update: unknown.

[30] C. Thekkath and T. Mann, E. Lee. *Frangipani: A Scalable Distributed File System.* Sixteenth Symposium on Operating Systems Principles, St. Malo, France, December 1997, pages 224-237.

[31] S. Wicker and V. Bhargava. *Reed-Solomon Codes and Their Applications.* IEEE Press, New York, 1994.

Appendix A – IOZone Test Result Tables

The results of the IOZone read and write tests can be found below. The results of the 11 other tests are available from the OSU Computer Science department.

RAID 5, Local

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	413089	385446	426528	380881	382990								
128	331692	336046	358611	371977	349805	344911							
256	332492	345058	359529	365164	368339	311813	257789						
512	329481	353098	358061	379517	382919	315863	253580	238029					
1024	336964	350081	367415	366225	374678	299775	246918	210739	216582				
2048	337563	357169	368204	383089	378632	296207	234136	213197	199630	196149			
4096	329602	349340	369312	377890	379015	298696	237919	206586	196979	201526	194583		
8192	328417	346266	366253	380386	383252	295165	238750	209144	198733	200161	195056	206931	
16384	331484	350100	367411	381404	378522	304897	250872	212809	201253	202296	198859	205924	205640
32768					380510	307311	247580	216140	202946	202549	201140	208482	205946
65536					380826	313872	255109	216658	203942	205871	204745	209055	209638
131072					381645	309521	251706	219703	204520	201967	202516	211914	210402
262144					382364	310159	251824	216911	205024	203739	202774	209621	213868
524288					49972	47655	47891	43924	46889	47398	46687	46881	47515
1048576					49281	48205	47935	46797	47099	47080	46177	46077	46957

RAID 5 Local IOZone Test Results (Read) in KB/sec

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	4363	244285	271259	293630	241598								
128	259678	294195	283834	303991	288219	226545							
256	248088	262559	270606	271759	238831	192191	155146						
512	191405	211569	210777	209062	177844	166771	167545	144510					
1024	178085	187755	186009	176311	166452	162255	160700	152129	144085				
2048	168171	174683	172957	166098	157368	155057	149936	154810	154333	141564			
4096	162565	167663	166058	163624	152426	149778	150660	150727	151027	148994	145185		
8192	159822	171578	171779	164656	158021	155077	152440	152895	153675	154881	155652	153979	
16384	162203	170083	167813	161313	155313	153709	155006	154309	154633	153853	155475	155410	149243
32768					151679	152505	151381	153924	153637	154681	156319	154949	153517
65536					135745	141091	138893	142135	141332	147210	143874	150076	149644
131072					136427	124834	123130	134012	132249	125338	125732	125011	128917
262144					115337	115145	120084	131776	119201	120068	128812	115565	120966
524288					69104	70822	69240	72630	72200	72172	71030	72709	72027
1048576					52364	53680	53432	52906	54293	53867	53621	52666	53830

RAID 5 Local IOZone Test Results (Write) in KB/sec

RS-RAID Distributed, Partially Degraded

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	326411	357688	353477	376580	367741								
128	333364	349813	360461	372084	379746	358508							
256	329497	354553	364127	372643	378700	351206	299096						
512	329891	346894	367573	373207	380100	337715	290097	266396					
1024	331293	351642	370619	379264	377288	338061	277053	261088	254344				
2048	334372	353226	370283	374330	383377	335952	287805	254154	243551	241341			
4096	330669	348005	367747	373656	384386	327731	280125	249194	239490	237476	238097		
8192	327757	347797	365014	379189	378714	338609	283175	249087	238980	238890	231924	236851	
16384	334537	348535	366302	380210	376989	335991	283592	250569	238688	234798	237945	238222	238576
32768					379214	332842	283801	247148	235305	234583	237693	237633	237129
65536					380388	336370	287409	249334	234896	237198	237069	234040	237046
131072					380492	339727	288177	250355	234606	234097	233943	234133	233918
262144					380613	343933	284501	250262	237379	234001	234082	233779	233844
524288					7968	7665	7642	7940	7780	7961	7961	7717	7732
1048576					7718	7725	7712	7649	7611	7712	7575	7693	7643

RS-RAID Distributed, Partially Degraded IOZone Test Results (Read) in KB/sec

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	837	232774	255044	253905	264564								
128	258057	268908	275827	278295	276478	227375							
256	250464	261494	262822	257042	250718	220884	181161						
512	223962	231991	233046	224164	212881	195045	184897	163370					
1024	207669	212535	209192	203864	191580	178518	172655	176522	164950				
2048	200842	200605	201855	196411	178661	175104	168809	167321	172814	167127			
4096	196574	200470	198181	188695	175058	172137	170566	168213	172324	172929	168664		
8192	194903	195601	194003	182812	172068	168736	165578	165364	168472	166886	169200	167755	
16384	192185	193098	193930	183693	171189	164872	164567	166090	167096	165111	165218	165048	167219
32768					166283	166224	165427	165968	164353	166584	163315	164603	164535
65536					149390	119613	149778	141958	144638	145894	136913	132593	147465
131072					135510	125526	154228	110834	136546	120131	111201	117945	107992
262144					20562	21726	19932	20525	19364	20774	15432	19577	20127
524288					6410	7034	6990	6892	6872	6839	6956	6773	7002
1048576					5021	5014	5068	5099	5118	5145	5199	5069	5127

RS-RAID Distributed, Partially Degraded IOZone Test Results (Write) in KB/sec

RS-RAID Distributed, Fully Degraded

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	331730	320132	355790	367571	359458								
128	329863	349624	355652	364668	369888	348708							
256	330339	345530	364665	370431	370416	345028	286696						
512	330095	343375	364179	373995	368348	331620	266656	269904					
1024	329260	351641	341670	376334	363642	329058	275047	260620	250799				
2048	332689	337288	354147	373591	368949	328042	282244	255521	242708	244537			
4096	328075	347734	365844	369573	369777	337064	280144	253119	242037	240106	240671		
8192	331660	350624	364135	371671	376108	328361	282668	251713	239476	238987	236319	239580	
16384	330382	355571	368288	380263	377545	329784	286058	251605	236472	235368	236053	235392	235061
32768					378505	335092	282417	247587	235410	238091	238238	235324	234702
65536					379546	334437	286866	251093	235246	237649	237360	234859	237414
131072					379431	334643	286585	247378	235525	237283	237085	237019	237264
262144					386030	335458	286566	250367	235472	234366	234864	236548	235009
524288					5514	5495	5629	5579	5648	5580	5673	5559	5543
1048576					5660	5655	5657	5746	5798	5848	5758	5706	5607

RS-RAID Distributed, Fully Degraded IOZone Test Results (Read) in KB/sec

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	223047	255020	262362	262393	239664								
128	246100	271214	275323	266728	280736	227768							
256	245208	248540	256779	243117	246629	213671	179663						
512	220025	228458	226946	217602	206862	180155	176253	163577					
1024	208763	203901	201856	202210	185610	177966	172275	174242	165079				
2048	202891	207643	200351	194862	181542	172738	173617	174091	173516	166368			
4096	198392	202631	200292	188027	176903	168920	171108	169129	170156	171971	166321		
8192	194271	195606	195582	184293	173140	168293	165317	165917	166219	166247	168761	165588	
16384	191897	193543	192977	181353	168053	165991	165893	166370	166771	164678	167525	164989	166761
32768					166410	164891	164452	163236	162951	163668	165855	163754	166105
65536					157254	155422	152298	130826	114778	105047	156091	157136	113479
131072					85197	73804	156880	139833	141365	80843	128296	84897	63531
262144					27270	27588	27408	26914	27724	27071	27375	29490	28472
524288					9646	9697	9724	9732	9930	9655	10149	10006	10032
1048576					7144	7159	7060	6978	7030	6994	7053	7008	7058

RS-RAID Distributed, Fully Degraded IOZone Test Results (Write) in KB/sec

RS-RAID Distributed, Non-Degraded

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	326502	355376	369866	374283	376517								
128	329002	359597	358655	376355	376440	324958							
256	330318	348291	364205	362063	361613	326122	299762						
512	328423	345937	365708	372116	373421	324652	292056	268193					
1024	330421	339635	367408	377033	377718	324046	266389	247349	249884				
2048	329894	348419	354630	378775	374473	310117	283733	254506	242800	245179			
4096	326714	351435	365223	368180	367356	321711	284723	252137	241394	236557	241154		
8192	329763	351815	365387	376348	376437	321633	281009	246762	235788	234997	239190	239315	
16384	329128	352094	365445	379181	372770	325945	283312	246820	237897	237590	234906	237825	237900
32768					380753	325311	283085	246866	234911	234361	236987	234556	234598
65536					381202	320158	283716	250278	237261	237079	236888	234139	234198
131072					375551	325984	279460	250231	234702	236965	233799	234658	236818
262144					363316	326036	279353	247065	234654	233975	233984	237013	236791
524288					9214	9165	9317	9220	9374	9109	9236	9215	9280
1048576					9323	9223	9269	9246	9243	9247	9266	9176	9288

RS-RAID Distributed, Non-Degraded IOZone Test Results (Read) in KB/sec

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	218414	246124	257122	246209	236164								
128	255990	269467	280100	286386	259116	212602							
256	252960	270619	271495	267783	248290	209836	184438						
512	221451	227246	230534	217968	204474	191764	184776	166779					
1024	209192	214588	213328	200978	189522	179393	169763	171728	162979				
2048	199067	201495	200707	188806	178492	174459	170014	167895	170609	165949			
4096	194437	202201	200215	189577	177331	168941	166564	170516	168566	169424	168524		
8192	193321	195616	194662	182975	173196	165464	166065	165836	165685	168062	166092	167207	
16384	192671	195370	194139	182545	170709	164339	162359	163674	166277	164155	164004	166208	163496
32768					168708	162145	161492	163077	162920	164842	166023	163759	163777
65536					142632	157865	130143	133474	131763	143848	143922	153174	143155
131072					110626	113002	113964	112715	155340	116666	119537	116409	117450
262144					19327	18444	18266	21269	21020	20264	21306	23203	18868
524288					6878	6738	6734	7110	6975	6955	7280	6857	6885
1048576					5272	5299	5215	5148	5331	5086	5322	5251	5292

RS-RAID Distributed, Non-Degraded IOZone Test Results (Write) in KB/sec

RS-RAID Local, Partially Degraded

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	344014	359554	254865	387620	349527								
128	339510	342290	365595	382076	377564	333399							
256	324489	355992	356492	368406	377567	325269	283532						
512	327557	346656	369396	367798	381811	318808	284904	248419					
1024	329902	348289	373051	379688	374129	327158	285471	255748	249945				
2048	334693	349788	366956	380958	383021	325183	274566	249148	244889	244770			
4096	329575	348063	365939	374888	374473	325365	282656	248632	241479	237711	240968		
8192	332145	349921	371756	378976	376209	320727	282024	248822	239105	238022	235665	238799	
16384	330876	353157	366221	370821	376998	321028	278100	250030	236043	237301	234831	234896	238392
32768					378230	318943	278159	249581	237492	232089	236607	237025	237560
65536					363241	317970	281802	246268	234281	232188	236592	236739	237211
131072					377090	323107	277036	246188	234168	235836	236298	236571	237072
262144					378020	325034	278001	246558	234524	233282	236220	233704	233918
524288					22683	22579	22378	22524	22432	22339	22292	22531	22309
1048576					22570	22459	22368	22133	22278	22168	22111	22239	22196

RS-RAID Local, Partially Degraded IOZone Test Results (Read) in KB/sec

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	219929	266674	265616	274785	265581								
128	258113	277648	275286	293517	271124	218443							
256	243560	269172	265846	266690	253977	212780	186731						
512	225850	226959	230834	218437	206197	189425	181623	163111					
1024	205830	214629	212758	196844	187441	176247	174181	174472	163681				
2048	199514	201953	201613	191814	178302	170071	169452	170822	171050	168283			
4096	196113	200068	196621	185557	175642	169003	167909	168034	168998	169339	167054		
8192	195695	195080	192454	181914	171775	166565	164878	165585	165682	166119	166341	164759	
16384	189106	191332	189572	178937	169764	164802	164045	164565	164301	167067	167323	166248	166033
32768					170599	163917	165185	165729	163868	166520	166276	164152	164782
65536					157028	161251	158938	161064	151395	161163	159286	160406	154668
131072					129377	135829	130649	125388	130490	127141	128240	131448	131641
262144					40024	39802	38510	39985	39022	38187	40103	39091	40888
524288					13657	13759	13573	13578	13518	13552	13636	13610	13649
1048576					10271	10156	10153	10171	10153	10172	10260	10201	10140

RS-RAID Local, Partially Degraded IOZone Test Results (Write) in KB/sec

RAID 5, Distributed

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	333286	359390	370070	372253	361430								
128	322509	347746	362596	358627	356645	351678							
256	331171	347839	365200	370459	379879	345992	297679						
512	326305	348082	367786	376215	376188	328837	293735	245568					
1024	333121	347601	364160	367021	376198	328724	286117	245618	250429				
2048	331125	349248	365518	372023	375028	329947	277466	246123	235426	245977			
4096	335135	348302	366136	372970	372159	330431	283951	247343	239755	238611	237378		
8192	328507	348327	366432	371181	370546	333998	277731	244332	237207	239517	236626	236996	
16384	331237	347729	365616	378575	373500	326505	281918	250152	236267	238907	238861	238409	238423
32768					372638	326556	287696	248917	234542	237488	235031	237811	235096
65536					377781	334186	284319	249378	234895	234649	234677	234769	236867
131072					380845	329286	284434	246939	237419	234690	237178	234669	234370
262144					383315	329108	284474	247112	237934	234705	237304	237118	237173
524288					9775	9852	9942	9695	9666	9961	10029	9992	9765
1048576					9761	9838	9798	9723	9770	9738	9693	9740	9755

RAID 5, Distributed IOZone Test Results (Read) in KB/sec

File Size (KB)	Record Size (KB)												
	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
64	216961	257049	266722	263374	257060								
128	261255	280053	264959	280141	260713	221875							
256	252732	264479	266964	270024	241058	219738	181175						
512	223097	231472	228887	223384	201497	192113	181111	164899					
1024	210876	217785	211783	200942	186793	178829	174417	174359	162156				
2048	200806	207393	203860	196056	178073	171281	172475	169592	173413	156491			
4096	197911	205529	203004	190087	172848	169234	170233	167054	171409	171516	170092		
8192	191805	196639	192476	185234	171685	166727	167433	168455	166220	166884	169557	167669	
16384	190370	194679	193289	182924	172055	165924	163877	164797	164025	164474	167944	165446	167231
32768					167372	165262	162452	163265	165241	165978	164346	164571	166631
65536					153741	162457	162872	141804	150584	161951	165595	165889	163765
131072					147031	144372	121078	148503	164872	141949	119890	147306	119749
262144					43141	36096	39740	41005	43162	40496	41993	42233	40095
524288					14152	13582	13178	14450	13662	13628	14172	14260	13692
1048576					10372	10105	10232	9911	10148	9814	10148	7281	10060

RAID 5, Distributed IOZone Test Results (Write) in KB/sec

Appendix B – Reed-Solomon RAID Source Code

Selected portions of the RS-RAID driver can be found below. The full source is available from the OSU Computer Science department.

rs_raid.h

```
#ifndef _RS_RAID_H
#define _RS_RAID_H

#include <linux/raid/md.h>

//.............REED SOLOMON PARAMETERS.................
#define MAX_DISKS       256
#define MAXM 25
#define MAXN 25


//w of 8 has been chosen so the division and multiplication
//table optimization can be used.  If w were larger, the tables would
//be too big to fit in memory

typedef u8 RS_WORD;

#define NUMBITS 8
#define PRIME 0x1D

#define MSB (1 << (NUMBITS - 1))
#define SIZE (1 << NUMBITS)

//The Galois field operators
#define MUL(a,b) (conf->mulTable[(a)][(b)])
#define DIV(a,b) (conf->divTable[(a)][(b)])
#define ADD(a,b) ((a)^(b))
#define SUB(a,b) ((a)^(b))

//..........END REED SOLOMON PARAMETERS.................


#define NR_STRIPES              256
#define HASH_PAGES_ORDER        0
#define HASH_PAGES              1
#define   IO_THRESHOLD                  1
#define STRIPE_SIZE             PAGE_SIZE
#define STRIPE_SECTORS                  (STRIPE_SIZE>>9)
#define STRIPE_SHIFT                    (PAGE_SHIFT - 9)
#define NR_HASH                         (HASH_PAGES * PAGE_SIZE / sizeof(struct stripe_head *))
#define HASH_MASK               (NR_HASH - 1)

/*
 * Stripe state
 */
#define STRIPE_ERROR                    1
#define STRIPE_HANDLE                   2
#define   STRIPE_SYNCING                3
#define   STRIPE_INSYNC                 4
#define   STRIPE_PREREAD_ACTIVE         5
#define   STRIPE_DELAYED                6

/* Flags */
#define   R5_UPTODATE           0       /* page contains current data */
#define   R5_LOCKED 1           /* IO has been submitted on "req" */
#define   R5_OVERWRITE          2       /* towrite covers whole page */
/* and some that are internal to handle_stripe */
#define   R5_Insync 3           /* rdev && rdev->in_sync at start */
#define   R5_Wantread           4       /* want to schedule a read */
#define   R5_Wantwrite          5
#define   R5_Syncio 6           /* this io need to be accounted as resync io */

/*
```

```
 * Write method
 */
#define RECONSTRUCT_WRITE       1
#define READ_MODIFY_WRITE       2
/* not a write method, but a compute_parity mode */
#define    CHECK_PARITY                 3
/* Additional compute_parity mode -- updates the parity w/o LOCKING */
#define UPDATE_PARITY           4


#define mddev_to_conf(mddev) ((rs_raid_conf_t *) mddev->private)

struct disk_info {
        mdk_rdev_t              *rdev;
};

struct stripe_head {
        struct stripe_head          *hash_next, **hash_pprev; /* hash pointers */
        struct list_head            lru;                      /* inactive_list or handle_list */
        struct rs_raid_private_data   *raid_conf;
        sector_t                      sector;                 /* sector of this row */
        int                                 pd_idx;                   /* parity disk index */
        unsigned long                       state;                    /* state flags */
        atomic_t                      count;                  /* nr of active thread/requests */
        spinlock_t                          lock;
        struct r5dev {
                struct bio              req;
                struct bio_vec      vec;
                struct page             *page;
                struct bio              *toread, *towrite, *written;
                sector_t        sector;                         /* sector of this page */
                unsigned long       flags;
        } dev[1]; /* allocated with extra space depending of RAID geometry */
};

struct rs_raid_private_data {
        struct stripe_head  **stripe_hashtbl;
        mddev_t                         *mddev;
        struct disk_info    *spare;
        int                                 chunk_size, level;

        int                                 raid_disks, working_disks, failed_disks;
        int                                 max_nr_stripes;

        struct list_head    handle_list; /* stripes needing handling */
        struct list_head    delayed_list; /* stripes that have plugged requests */
        atomic_t                preread_active_stripes; /* stripes with scheduled io */

        char                                cache_name[20];
        kmem_cache_t                    *slab_cache; /* for allocating stripes */
        /*
         * Free stripes pool
         */
        atomic_t                active_stripes;
        struct list_head    inactive_list;
        wait_queue_head_t   wait_for_stripe;
        int                                 inactive_blocked;       /* release of inactive stripes blocked,
                                                                     * waiting for 25% to be free
                                                                     */
        spinlock_t                          device_lock;

        /**REED SOLOMON TABLES AND PARAMETERS****/
        int                     n,m;
        RS_WORD                                 divTable[SIZE][SIZE];
        RS_WORD                                 mulTable[SIZE][SIZE];
        RS_WORD                                 encodematrix[MAXM][MAXN];
        RS_WORD                                 overallmatrix[MAXN + MAXM][MAXN];
        RS_WORD                                 decodematrix[MAXN][MAXN];
        //don't really need these after init, but they're only 256 bytes each.
        RS_WORD                                 gflog[SIZE], gfalog[SIZE];

        struct disk_info    disks[0];
};

typedef struct rs_raid_private_data rs_raid_conf_t;


#endif
```

rs_raid.c (Reed-Solomon encoding and decoding parts only)

```
/**********REED SOLOMON FUNCTIONS**********/
/* The basic multiplication and division functions, used to construct the tables.
static inline RS_WORD _mul(RS_WORD a, RS_WORD b, rs_raid_conf_t *conf)
{
   return (a==0 || b==0)?0:conf->gfalog[(conf->gflog[a] + conf->gflog[b]) % (SIZE - 1)];
}

static inline RS_WORD _div(RS_WORD a, RS_WORD b, rs_raid_conf_t *conf)
{
   return (a==0)?0:conf->gfalog[(conf->gflog[a] - conf->gflog[b] + SIZE - 1) % (SIZE - 1)];
}

static void fillMulTable(rs_raid_conf_t *conf)
{
   int i,j;

   for(i=0;i<SIZE;i++)
      for(j=0;j<SIZE;j++)
         conf->mulTable[i][j]=_mul(i,j,conf);
}

static void fillDivTable(rs_raid_conf_t *conf)
{
   int i,j;
   for(i=0;i<SIZE;i++)
      for(j=0;j<SIZE;j++)
         conf->divTable[i][j]=_div(i,j,conf);
}



/*This function sets up the gflog and gfalog tables, which are the Galois Field
*tables used in this algorithm.  See Plank, Appendix A for more information */
static void setupMultTable(rs_raid_conf_t *conf)
{
   RS_WORD curVal = 1;
   RS_WORD curLog;

   conf->gflog[0] = -1;
   for(curLog = 0; curLog < SIZE - 1; curLog++)
   {
      conf->gflog[curVal] = curLog;
      conf->gfalog[curLog] = curVal;

      /* This code differs slightly from Plank's code in Figure 4 due to the fact
       *there are no extra bits to play with, so the shift must occur only when
       *necessary */
      if(curVal & MSB)
         curVal = ((curVal ^ MSB) << 1) ^ PRIME;
      else
         curVal <<= 1;
   }

}


/* This function sets up the overall matrix as defined in Plank's revision paper (1997).  Note
 *that the combined I/V matrix defined in the original paper does not work.
static void setupTransform(rs_raid_conf_t *conf)
{
   int n=conf->n;
   int m=conf->m;

   int i, j, k;
   RS_WORD fact;

   for(i = 0; i < m + n; i++)
   {
      conf->overallmatrix[i][0] = 1;
      for(j = 1; j < n; j++)
         conf->overallmatrix[i][j] = _mul(i, conf->overallmatrix[i][j - 1],conf);
   }

   /* reduce to UID - CODE!

   for(i = 0; i < n; i++)
   {
      fact = conf->overallmatrix[i][i];
      for(j = i; j < m + n; j++)
         conf->overallmatrix[j][i] = _div(conf->overallmatrix[j][i], fact,conf);

      for(j = 0; j < n; j++) if(i != j)
      {
         fact = conf->overallmatrix[i][j];
         for(k = i; k < m + n; k++)
            conf->overallmatrix[k][j] = SUB(conf->overallmatrix[k][j], _mul(conf->overallmatrix[k][i], fact,conf));
      }
   }

   for(i = 0; i < m; i++)
      for(j = 0; j < n; j++)
```

```
        conf->encodematrix[i][j] = conf->overallmatrix[i + n][j];

}

static int encode(rs_raid_conf_t *conf, void** ptrs)//struct StripeBuffer *sb, int n, int m)
{

   int n=conf->n;
   int m=conf->m;

     local, constant buffers are the "fastest" way
   RS_WORD inVec[MAXN], outVec[MAXM];
   int i, j, k;
   RS_WORD **buffer = (RS_WORD**) ptrs;  //sb->buffer; //local copy of buffer pointer

   for(i = 0; i<STRIPE_SIZE *chunkSize; i++)
   {

      It seems silly to make a copy of the data like this, but trust me, it is faster this way.
      for (j = 0; (j < n); j++)
      {
         inVec[j]=buffer[j][i];
      }

      memset(&outVec[0],0,sizeof(RS_WORD)*m);

        multiply encodematrix by the inVec to get outVec
      for(j = 0; j < m; j++)
         for(k = 0; k < n; k++)
            outVec[j] = ADD(outVec[j], MUL(conf->encodematrix[j][k], inVec[k]));

        Again, seems silly to make a copy of the data, but this is faster.
      for(j = 0; j < m; j++)
         buffer[j+n][i]=outVec[j];
   }

   return 0;
}

static int decode(struct stripe_head *sh, void** ptrs)
{
   rs_raid_conf_t *conf=sh->raid_conf;
   int n=conf->n;
   int m=conf->m;

   RS_WORD** buffer=(RS_WORD**) ptrs;

   int i, j, k, l, r;
   int newvalid=0;
   RS_WORD factor, tmp;  *outbuffer[MAXN], inbuffer[MAXM];

   RS_WORD augmentedmatrix[MAXN][MAXN];  //the augmented part is the stripebuffer itself!

   //First, fill the preaugmented matrix with data from the overallmatrix according to sb->valid.


   j=0;
   for(i=0;i<(n);i++)
   {
      if(test_and_set_bit(R5_UPTODATE, &sh->dev[i].flags))
         memcpy(augmentedmatrix[i],conf->overallmatrix[i],sizeof(RS_WORD)*n);
      else  //this data row not valid, find a checksum row to use.
      {
         k=0;
         for(;j<m;j++)
         {
            if(test_bit(R5_Insync, &sh->dev[n+j].flags))
            {
               memcpy(augmentedmatrix[i], conf->overallmatrix[n+j],sizeof(RS_WORD)*n);
                 also copy the data in the stripebuffer
               memcpy(buffer[i],buffer[n+j],STRIPE_SIZE/*sizeof(RS_WORD)*chunkSize*/);

               j++;  //this row's been used, don't use it again.
               k=1;
               break;
            }
         }
         if(k==0)
         {
            BUG();  no more valid checksum rows found.
            return  1;
         }
         newvalid++;

      }
   }

   Now the state of this is as follows:
      sb->buffer contains all the data we'll need in rows 0 through N-1
      the augmented array is an nxn matrix we'll be doing the gaussian elimination on

   for(j = 0; j < n; j++) {
      if(!augmentedmatrix[j][j])
      {
         for(r = j + 1; !augmentedmatrix[r][j]; r++)
```

```
            if(r>=n) return 0;     something bad happened, r got too big.
            swap a row.    Not exactly a quick procedure, but not done often enough to worry about.
          for(k = 0; k < n; k++) {
            tmp = augmentedmatrix[j][k];
            augmentedmatrix[j][k] = augmentedmatrix[r][k];
            augmentedmatrix[r][k] = tmp;
          }

          for(k=0; k<STRIPE_SIZE *chunkSize* ;k++) {
            tmp = buffer[j][k];
            buffer[j][k] = buffer[r][k];
            buffer[r][k] = tmp;
          }
        }

        factor = augmentedmatrix[j][j];
        if(factor!=1)     no need to divide by 1
        {
          for k = 0; k < n; k++)
            augmentedmatrix[j][k] = DIV(augmentedmatrix[j][k], factor);
          for(k = 0; k < STRIPE_SIZE *chunkSize* ; k++ )  //The augmented part of the matrix
            buffer[j][k] = DIV(buffer[j][k], factor);
        }

        for(k = 0; k < n; k++
        {
          if(j != k) {
            factor = augmentedmatrix[k][j];
            if(factor!=0)     no need to multiply by 0, then subtract 0 from self...
            {
              for(l = j; l < n; l++)
                augmentedmatrix[k][l] = SUB(augmentedmatrix[k][l], MUL(factor, augmentedmatrix[j][l]));
              for(l=0; l<STRIPE_SIZE*chunkSize*;l++)  //do an entire stripe at once!
                buffer[k][l] = SUB(buffer[k][l], MUL(factor, buffer[j][l]));
            }
          }
        }
      }

      return newvalid;
    }



    static int compute_blocks(struct stripe_head *sh)//, int dd_idx1, int dd_idx2)
    {
      rs_raid_conf_t *conf = sh->raid_conf;
      int i, count, disks = conf->raid_disks;
      void* ptrs[MAX_DISKS];


      count = 0;
      i = 0; /*d0_idx
      do {
        ptrs[count++] = page_address(sh->dev[i].page);
        i = rs_raid_next_disk(i, disks);
      } while ( i != 0  *d0_idx*  );

      DK("REED SOLOMON DECODING HERE.
      return decode(sh,ptrs);  /*decode takes care of setting the uptodate bit.
    }

    static void compute_parity(struct stripe_head *sh, int method)
    {
      rs_raid_conf_t *conf = sh->raid_conf;
      int i, pd_idx = sh->pd_idx,  *qd_idx, d0_idx,*/ disks = conf->raid_disks, count;
      struct bio *chosen;

      void *ptrs[MAX_DISKS];

      /*qd_idx = rs_raid_next_disk(pd_idx, disks);
      /*d0_idx = rs_raid_next_disk(qd_idx, disks);

      PRINTK("compute_parity, stripe %llu, method %d\n",
        (unsigned long long)sh->sector, method);

      switch(method) {
      case READ_MODIFY_WRITE:
        BUG();        * READ MODIFY WRITE N/A for RAID-6 */
      case RECONSTRUCT_WRITE:
      case UPDATE_PARITY:
        for (i= conf->n; i--;  ) {
          if ( i != pd_idx && i != qd_idx && sh->dev[i].towrite ) {
            chosen = sh->dev[i].towrite;
            sh->dev[i].towrite = NULL;
            if (sh->dev[i].written) BUG();
            sh->dev[i].written = chosen;
          }
        }
        break;
      case CHECK_PARITY:
        BUG();        * Not implemented yet */
      }
```

```
for (i = disks; i
   if (sh->dev[i].written) {
      sector_t sector = sh->dev[i].sector;
      struct bio *wbi = sh->dev[i].written;
      while (wbi && wbi->bi_sector < sector + STRIPE_SECTORS) {
         copy_data(1, wbi, sh->dev[i].page, sector);
         wbi = r5_next_bio(wbi, sector);
      }

      set_bit(R5_LOCKED, &sh->dev[i].flags);
      set_bit(R5_UPTODATE, &sh->dev[i].flags);
   }

count = 0;
i = 0;
do {
   ptrs[count++] = page_address(sh->dev[i].page);
   i = rs_raid_next_disk(i, disks);
} while

encode(conf,ptrs);

   STRIPE_SIZE, ptrs);

switch(method) {
case RECONSTRUCT_WRITE:
   for(i=conf->n;i<conf->raid_disks;i++)
   {
      set_bit(R5_UPTODATE, &sh->dev[i].flags);
      set_bit(R5_LOCKED, &sh->dev[i].flags);
   }
     set_bit(R5_UPTODATE, &sh->dev[pd_idx].flags);
   //set_bit(R5_UPTODATE, &sh->dev[qd_idx].flags);
   //set_bit(R5_LOCKED,    &sh->dev[pd_idx].flags);
     set_bit(R5_LOCKED,    &sh->dev[qd_idx].flags);
   break;
case UPDATE_PARITY:
   for(i=conf->n;i<conf->raid_disks;i++)
   {
      set_bit(R5_UPTODATE, &sh->dev[i].flags);
   }
   //set_bit(R5_UPTODATE, &sh->dev[pd_idx].flags);
     set_bit(R5_UPTODATE, &sh->dev[qd_idx].flags);
   break;
 }
}
```

#1

VITA

Nathaniel Paul Lewis

Candidate for the Degree of

Master of Science

Thesis: DISTRIBUTED RAID SYSTEM: A UNIQUE USE FOR REED SOLOMON CODING

Major Field: Computer Science

Biographical:

Education: Graduated from Neenah High School, Neenah, Wisconsin in June 1998; received Bachelor of Science degree in Computer Science from the University of Wisconsin – Madison, Madison, Wisconsin in May 2001. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2004.

Experience: Employed as a software developer by Epic Systems Corporation, Madison, Wisconsin from June 2001 to August 2002; employed by Seagate, Oklahoma City, Oklahoma summer 2003; employed by Oklahoma State University, Computer Science Department as a teaching assistant, August 2003 to present.

Professional Memberships: Association for Computing Machinery.