A Highly Reliable GPU-Based RAID System

Matthew L. Curry
University of Alabama at Birmingham

A HIGHLY RELIABLE GPU-BASED RAID SYSTEM

by

MATTHEW L. CURRY

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
BIRMINGHAM, ALABAMA

2010

A HIGHLY RELIABLE GPU-BASED RAID SYSTEM

MATTHEW L. CURRY

COMPUTER AND INFORMATION SCIENCES

ABSTRACT

In this work, I have shown that current parity-based RAID levels are nearing the end of their usefulness. Further, the widely used parity-based hierarchical RAID levels are not capable of significantly improving reliability over their component parity-based levels without requiring massively increased hardware investment. In response, I have proposed $k+m$ RAID, a family of RAID levels that allow m, the number of parity blocks per stripe, to vary based on the desired reliability of the volume. I have compared its failure rates to those of RAIDs 5 and 6, and RAIDs 1+0, 5+0, and 6+0 with varying numbers of sets.

I have described how GPUs are architecturally well-suited to RAID computations, and have demonstrated the Gibraltar RAID library, a prototype library that performs RAID computations on GPUs. I have provided analyses of the library that show how evolutionary changes to GPU architecture, including the merge of GPUs and CPUs, can change the efficiency of coding operations. I have introduced a new memory layout and dispersal matrix arrangement, improving the efficiency of decoding to match that of encoding.

I have applied the Gibraltar library to Gibraltar RAID, a user space RAID infrastructure that is a proof of concept for GPU-based storage arrays. I have integrated it with the user space component of the Linux iSCSI Target Framework, which provides a block device for benchmarking. I have compared the streaming workload performance of Gibraltar RAID to that of Linux md, demonstrating that Gibraltar RAID has superior RAID 6 performance. Gibraltar RAID's performance through $k+5$ RAID remains highly competitive to that of Linux md RAID 6. Gibraltar RAID operates at the same speed whether in degraded or normal modes, demonstrating a further advantage over Linux md.

DEDICATION

This thesis is dedicated to my family and friends who have supported me personally through the process of completing this work. I wish I could individually list all who were there providing words of encouragement and inspiration, but their names are too numerous to list. Friends and family from my earlier life have remained supportive, and colleagues in the University of Alabama at Birmingham graduate program and the Sandia National Laboratories summer internship program have provided me a much wider network of like-minded and similarly ambitious supporters and friends. I am thankful for them all.

ACKNOWLEDGEMENTS

I have had the great fortune of having some very understanding and accommodating superiors on this project. My advisor, Dr. Anthony Skjellum, allowed me to complete much of this work in New Mexico to strengthen collaborations between our group and Sandia National Laboratories. Lee Ward, my mentor for my work at Sandia, provided financial support for this project and invaluable guidance in general during my stay in Albuquerque. I feel that my professional life has been dramatically improved by this arrangement, and I am grateful that all involved were immediately on board for my unusual suggestion.

Although I did have the best of associates, this work would not have been possible without the support of funding agencies. This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000. This work was also supported by the National Science Foundation under grant CNS-0821497.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASIC	application-specific integrated circuit
BER	bit error rate
CPU	central processing unit
DAS	direct-attached storage
ECC	error correction code
GPU	graphics processing unit
GPGPU	general purpose computation on GPUs
HPC	high performance computing
HRAID	hierarchical RAID
I/O	input/output
JBOD	just a bunch of disks
MRAID	multi-level RAID
MTBF	mean time between failures
MTTDL	mean time to data loss
MTTF	mean time to failure
NAS	network attached storage
RAID	redundant array of independent disks
URE	unrecoverable read error

CHAPTER 1

Introduction

Redundant arrays of independent [1] disks (RAID) is a methodology of assembling several disks into a logical device that provides faster, more reliable storage than is possible for a single disk to attain [**84**]. RAID levels 5 and 6 accomplish this by distributing data among several disks, a process known as striping, while also distributing some form of additional redundant data to use for recovery in the case of disk failures. The redundant data, also called parity, are generated using erasure correcting codes [**63**]. RAID levels 5 and 6 can drastically increase overall reliability with little extra investment in storage [**20**].

RAID can also increase performance because of its ability to parallelize accesses to storage. A parameter commonly known as the chunk size or stripe depth determines how much contiguous data are placed on a single disk. A stripe is made up of one chunk of data or parity per disk, with each chunk residing at a common offset. The number of chunks within a stripe is known as the stripe width. For a particular RAID array, the number of chunks of parity is constant. RAID 5 is defined to have one chunk of parity per stripe, while RAID 6 has two. If a user requests a read or write of a contiguous block of data that is several times the size of a chunk, several disks can be used simultaneously to satisfy this request. If a user requests several small, random pieces of data throughout a volume, these requests are also likely to be distributed among many of the disks.

RAID has become so successful that almost all mass storage is organized as one or more RAID arrays. RAID has become ingrained into the thought processes of the enterprise storage community. Hardware RAID implementations, with varying levels

[1]This acronym was formerly expanded to redundant arrays of inexpensive disks, but has changed over time.

of performance, are available from many vendors at many price points. Alternatively, software RAID is also available out of the box to the users of many operating systems, including Linux [**116**]. Software RAID is generally viewed as trading economy for speed, with many high-performance computing sites preferring faster, hardware-based RAIDs. While software RAID speeds do not compare well with those of hardware RAID, software RAID allows a wider community to benefit from some of the speed and reliability gains available through the RAID methodology.

Software RAID is economically appealing because hardware RAID infrastructure is expensive, thus making hardware RAID impractical in a large number of applications. Figure 1 shows the results of a 2009 survey of costs for a petabyte in raw disk capacity, software RAID infrastructure, and hardware RAID infrastructure from several storage vendors [**73**]. The least expensive hardware-based RAID solution, the Dell MD1000, is nearly eight times as expensive as building servers to use with Linux md, the software RAID implementation included with the Linux operating system. While the md servers and those with hardware RAID are somewhat different, the cost disparity is mostly attributable to the cost of hardware RAID controllers.

The currently popular RAID levels are beginning to show weakness in the face of evolving disks. Disks are becoming larger, and their speeds are increasing with the square root of their size. This implies that, when a larger disk fails, the mean time to repair (MTTR) will be much larger than for a smaller disk. Further, the incidence of unrecoverable read errors (UREs) is not changing, but is becoming more prevalent during a RAID's rebuild process. UREs are manifested as a disk's failure to retrieve previously stored contents of a sector. RAID 6 is capable of tolerating up to two simultaneous media failures during a read operation, whether they are disk failures or UREs. RAID 6 exists because RAID 5 has been shown to be inadequate, as double disk failures do occur. This indicates that RAID 6 will not be able to maintain data integrity when encountering increased numbers of UREs during a rebuild.

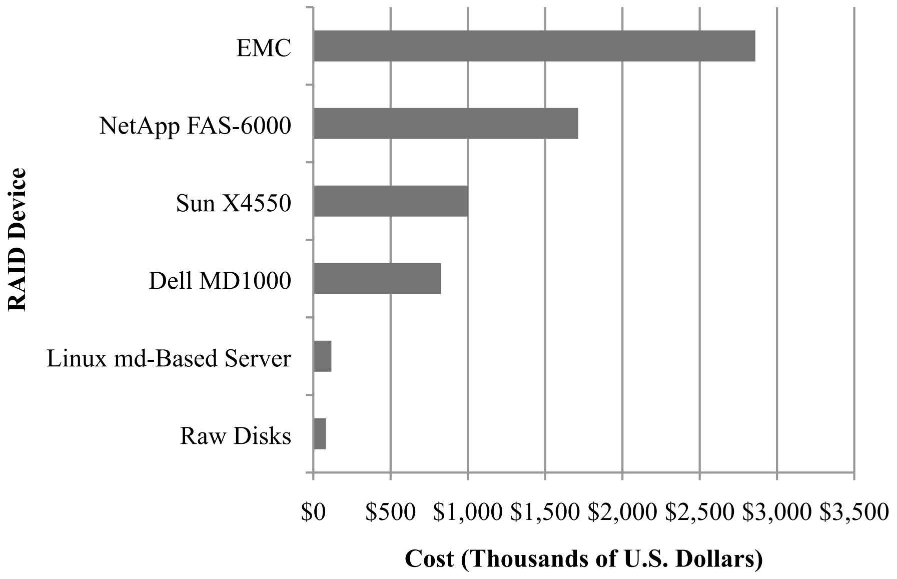


Figure 1. Cost for One Petabyte of Storage from Several Vendors
Adapted from "Petabytes on a Budget: How to Build Cheap Cloud Storage" by Tim Nufire, September 1, 2009, BackBlaze Blog (`http://blog.backblaze.com`). Copyright 2009 by BackBlaze, Inc. Adapted with permission.

Hierarchical RAID (HRAID) was introduced to improve the reliability of traditional RAID infrastructure [**6**]. By treating RAID arrays as individual devices, new RAID arrays can be composed of several smaller RAID arrays. This improves the reliability of RAID by increasing the total amount of parity in the system. While HRAIDs do protect efficiently from disk failures, UREs present a different kind of problem.

Chapter 3 contains an extensive analysis of RAID reliability, with particular attention paid to risks associated with UREs. It demonstrates that hierarchical RAID levels do not significantly increase protection against risk of data loss, and that a new strategy is required. It describes a new family of RAID levels, $k + m$ RAID, that can significantly reduce risk of data loss. These RAID levels are similar to RAID levels 5 and 6, as they are all parity based. However, $k + m$ RAID allows the amount of storage dedicated to parity to be customized, with m being tunable to determine the number of disks that may fail in a RAID set without data loss. For example, $k + 2$ RAID is functionally equivalent to RAID 6. Increasing m beyond two can allow for drastically increased reliability.

One algorithm that stands out as being directly applicable to $k+m$ RAID is Reed-Solomon coding. Reed-Solomon coding has the disadvantage of being computationally expensive. Other codes are less expensive, but have their own limitations. For example, EVENODD [**10**] and its variants are specialized to certain values of m. Others, like tornado codes [**17**], are inefficient in the amount of storage used and the amount of work required for small updates. Current hardware RAID controllers do not implement functionality similar to $k+m$ RAID. However, as Reed-Solomon coding is efficiently performed in hardware, controllers can be manufactured to provide these new levels. This advance would incur the same costs that make current hardware RAID 6 controllers expensive. More economical and flexible software RAID is required for many scenarios.

Software RAID controllers can be modified to provide $k+m$ RAID with Reed-Solomon coding today, but would likely operate more slowly than current RAID 6 when using $m > 2$. The most widely used CPUs, x86 and x86-64, do not have vector instructions that can be used to accelerate general $k+m$ Reed-Solomon coding. Such acceleration is required to approach the peak processing power of current CPUs, so much of the computation power will not be used. This situation is already apparent in some software implementations of RAID 6, but will be exacerbated by the increased computational load required: $k+m$ RAID requires $O(m)$ computations per byte stored, implying that $k+3$ RAID requires 50% more computations than RAID 6.

One solution to this problem is to look toward a growing source of compute power available in the commodity market: Graphics processing units (GPUs). GPUs are devices intended to accelerate processing for demanding graphics applications, such as games and computer-aided drafting. GPUs manufactured for much of the last decade have been multi-core devices, reflecting the highly parallel nature of graphics rendering. While CPUs have recently begun shipping with up to twelve cores, NVIDIA's GeForce 480 GTX has recently shipped with 480 cores per chip.

Applications that are easily parallelized can often be implemented with a GPU to speed up computation significantly.

This work shows that Reed-Solomon coding in the style of RAID is a good match for the architecture and capabilities of modern GPUs. Further, a new memory layout and a complementary matrix generation algorithm significantly increase the performance of data recovery from parity, or decoding, on GPUs. In fact, the operations can be made to be nearly identical to parity generation, or encoding, yielding equivalent performance. RAID arrays are widely known to suffer degraded performance when a disk has failed, but these advances in a GPU RAID controller can eliminate this behavior.

A tangible contribution of this work is a practical library for performing Reed-Solomon coding for RAID-like applications on GPUs, the Gibraltar library, which is described in Chapter 5. This library can be used by RAID implementations, or applications that share RAID's style of parity-based data redundancy. The Gibraltar library uses NVIDIA CUDA-based GPUs to perform coding and decoding. There are over 100 million GPUs capable of running this software installed world-wide [**65**], implying that a wide population can apply the findings from the Gibraltar library's creation.

While a practical library for Reed-Solomon coding is important, a view into future viability of RAID on multi-core processors and GPUs is necessary. Design choices that would benefit Reed-Solomon coding can be at odds with those that benefit other popular applications, and vice versa. Chapter 5 describes projected performance for theoretical devices that have varied design parameters. Further, the impending merge of conventional CPUs and GPUs [**4**] points to a significant change in the performance characteristics of many general-purpose GPU (GPGPU) applications because of the elimination of PCI-Express bus use as well as increased data sharing. This chapter addresses these concerns as well.

A new RAID controller that targets streaming workloads, Gibraltar RAID, has been prototyped around the Gibraltar library. It serves as a proof of concept for the capabilities of the Gibraltar library. It is tested according to streaming I/O patterns as direct-attached storage (DAS) and network-attached storage (NAS) for up to four clients. This demonstrates the applicability of this controller. Linux md's RAID 6 performance has been compared to Gibraltar RAID's performance in configurations $2 \geq m \geq 5$ with identical I/O patterns. Benchmarks equally emphasize normal mode operation, where no disks have failed, and degraded mode operation, where at least one disk has failed. Gibraltar RAID's performance has proven superior over Linux md for all values of $2 \leq m \leq 5$.

Gibraltar RAID's applications extend beyond conventional RAID. Because it is software-based, one can quickly modify it to support significant flexibility in its operation. For example, Gibraltar RAID can support large arrays composed of several smaller arrays residing on other machines, an organization known as multi-level RAID (MRAID) [**104**]. Chapter 7 provides several examples of alternative organizations and policies enabled by Gibraltar RAID. These variations can be exceedingly expensive with a hardware implementation. These storage configurations can allow for full data center reliability, enabling an inexpensive means of eliminating single points of failure with software RAID techniques. Further applications of the library are also explored in Chapter 7.

The data being read and written by users are subject to processing with a GPU, providing another potential benefit to the software nature of the library. Extra storage operations that can benefit from GPU computation, like encryption, deduplication, and compression, can be integrated into the storage stack. This amortizes the transfer costs associated with GPU computation by allowing multiple computations to be performed on data with a single transfer. GPUs supporting multiple simultaneous kernels have recently been introduced, allowing such operations to be pipelined efficiently.

In summary, this work details a RAID methodology and infrastructure that improve on existing RAID implementations in multiple dimensions. First, this methodology provides a high degree of flexibility in balancing performance, storage utilization, and reliability in RAID arrays. Second, a software infrastructure is described that has improved speed and capabilities over Linux md, allowing for NAS and DAS that can take advantage of more capable networks. Finally, flexibility in application of Gibraltar RAID and the Gibraltar library allows for their use in many situations that are decidedly similar to RAID but differ in details. Further, extra storage computations may be integrated into storage stack when beneficial. This work has the potential to impact the economics of high-performance storage, allowing for more storage per dollar and faster capability improvements than is possible with custom ASIC-based solutions.

CHAPTER 2

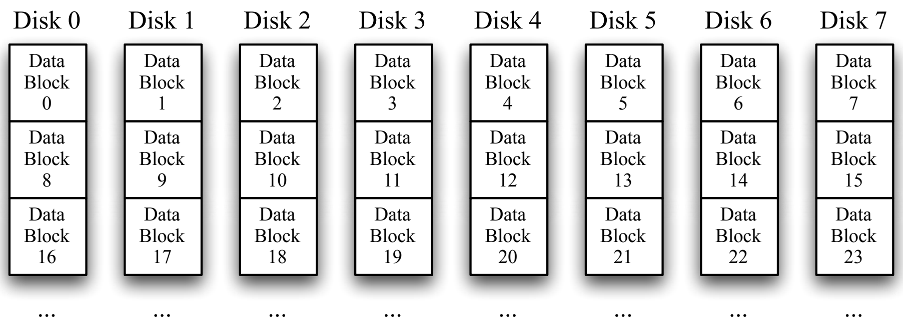
LITERATURE REVIEW

This work lies at the intersection of three main subject areas: RAID, erasure coding, and GPU computing. Other interesting work in fault-tolerant storage is in traditional file systems and network file systems. This chapter provides an overview of history and efforts in all of these areas.

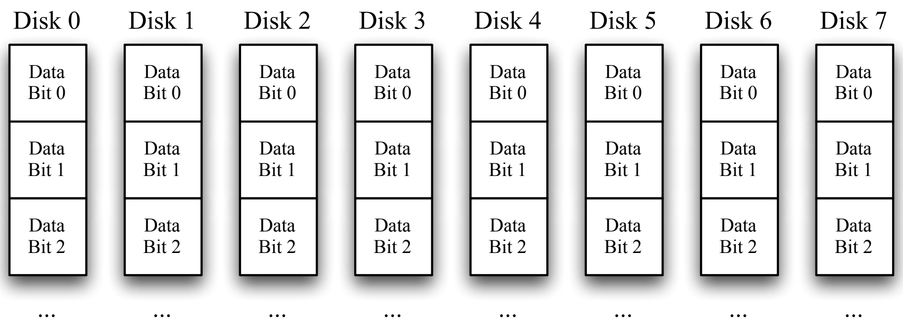
1. RAID

The introduction of RAID formalized the use of striping, mirroring, parity, and error correction codes (ECC) to increase reliability and speed of storage systems composed of many disk drives [**84**]. The original RAID levels numbered only 1-5, with RAID 0 and RAID 6 later added to the standard set of RAID levels [**20**]. A list of the salient characteristic of each level, as originally defined and characterized [**20, 84**], follows.

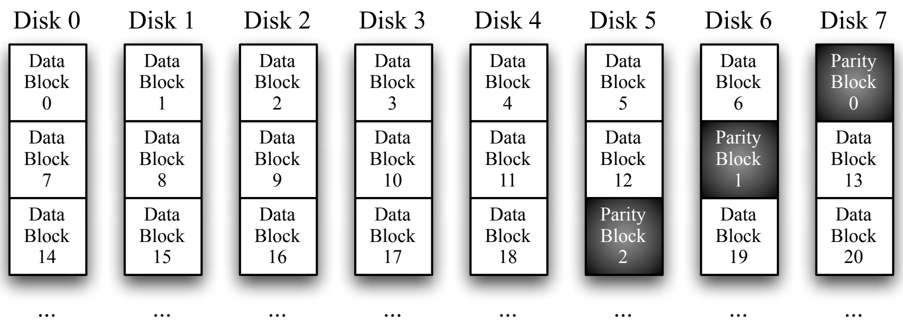
- RAID 0 (Figure 2a) stripes data among all disks with no measures for fault tolerance included. This level has the highest possible write bandwidth, as no redundant information is written.
- RAID 1 (Figure 2b) mirrors data between all of the disks in the volume. This level has the highest possible read bandwidth, as several mirrors can be tasked simultaneously.
- RAID 2 (Figure 3a), a bit-striping level (i.e., striping across devices with a block size of one bit), uses Hamming codes to compute error correction information for a single bit error per stripe. This level is no longer used, as this level of error correction is typically present within modern disk drives. Striping at the bit level requires all disks in an array to be read or written for



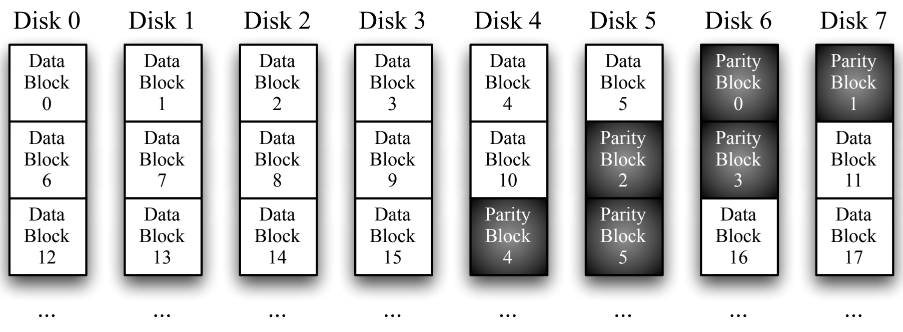
(a) RAID 0



(b) RAID 1

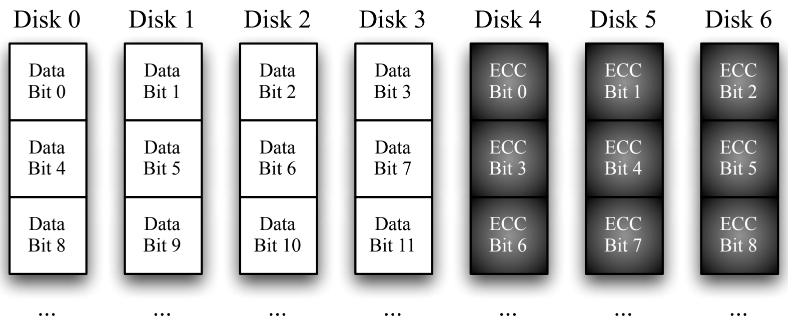


(c) RAID 5

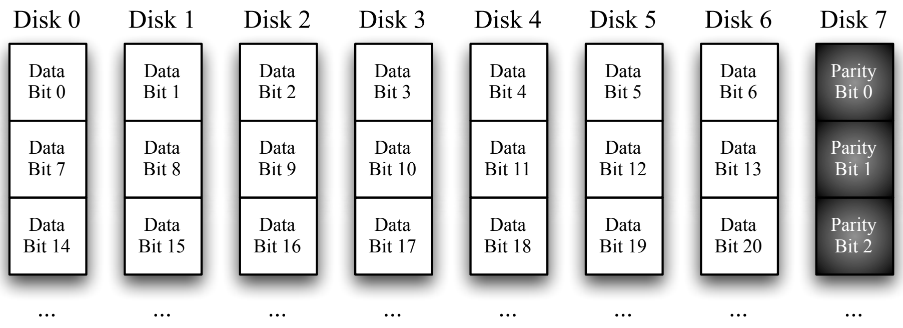


(d) RAID 6

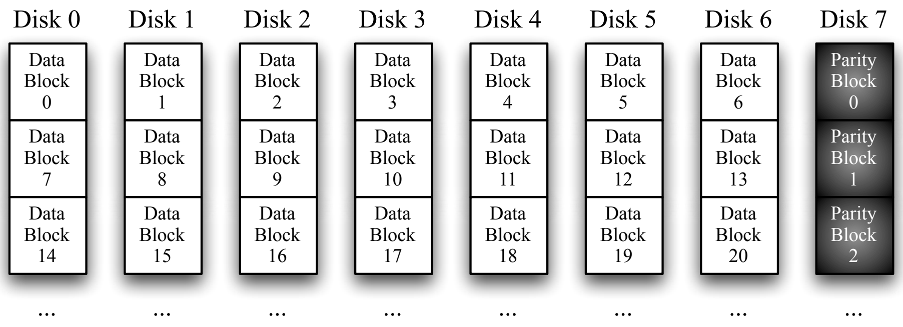
Figure 2. Sample Configurations of RAID Levels in Common Use Today [**20, 84**]



(A) RAID 2



(B) RAID 3

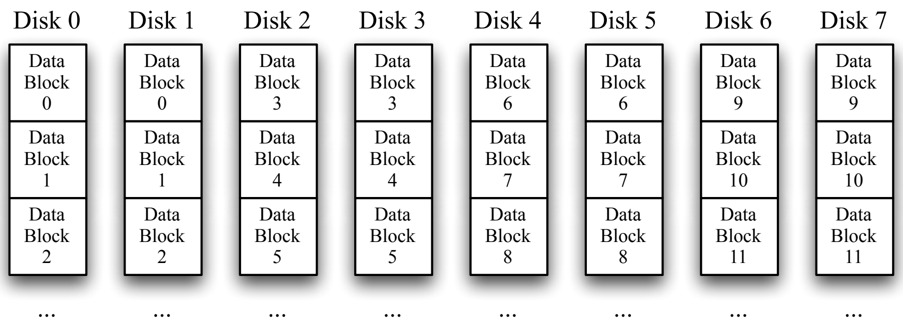


(C) RAID 4

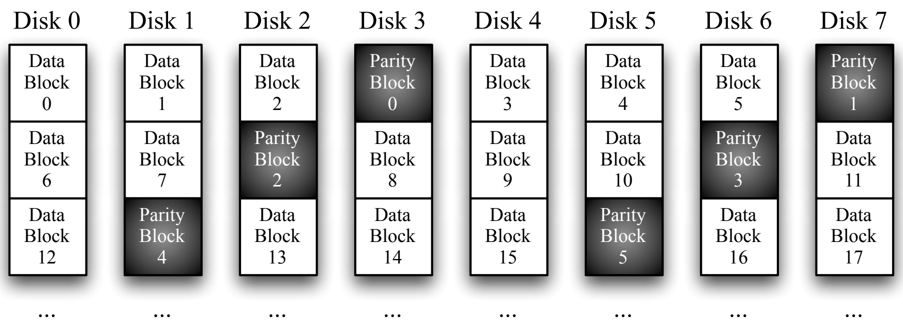
FIGURE 3. Sample Configurations of Original RAID Levels Not in Common Use Today [**84**]

most accesses, reducing potential parallelism for small reads and writes. The number of ECC disks required is governed by the equation $2^m \geq k + m + 1$, where k is the number of data disks and m is the number of ECC disks [**42**].

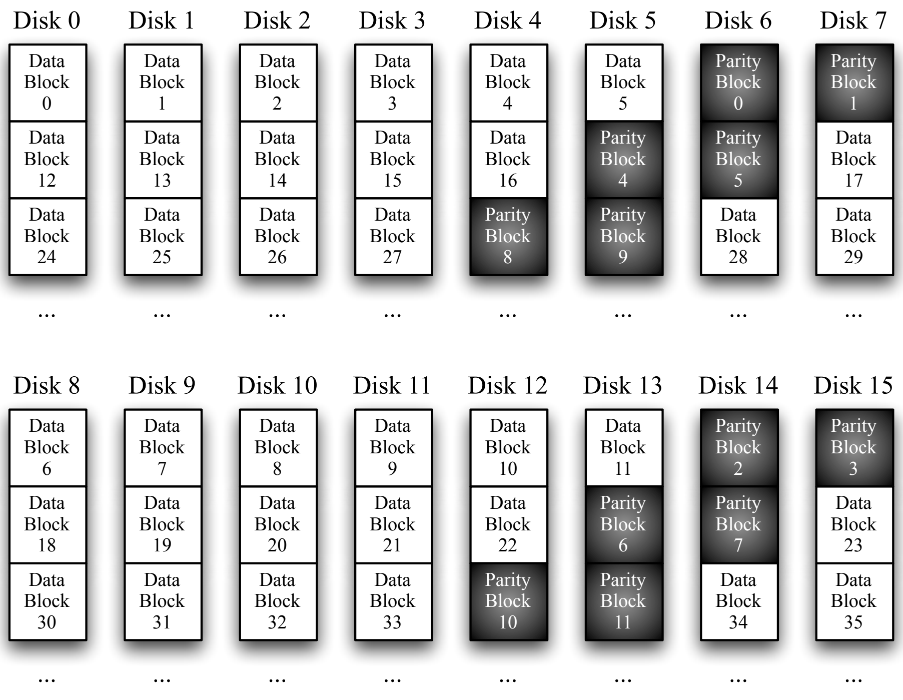
- RAID 3 (Figure 3b), another bit-striping level, computes a parity bit for the data bits in the stripe. This is a reduction in capability from RAID 2, as this provides for erasure correction without error correction, but the storage



(A) RAID 1+0



(B) RAID 5+0



(C) RAID 6+0

FIGURE 4. Sample Configurations of Hierarchical RAID Levels in Common Use Today [**6**]

overhead is much lower. Like RAID levels 4 and 5, the parity can be computed by performing a bit-wise XOR on all of the data.

- RAID 4 (Figure 3c) allows for independent parallel read accesses by storing contiguous blocks of data on each disk instead of using bit-striping. When data are organized in this way, the array can service a small read operation by reading a single disk, so k small reads can potentially be serviced in parallel.
- RAID 5 (Figure 2c) enables independent parallel read and write accesses by distributing parity between all disks. For a small write, the stripe's new parity can be computed from old parity, old data blocks, and new data blocks, so a small write requires only part of a stripe. Rotating parity blocks allows multiple parity updates to be accomplished simultaneously. The dedicated parity disks in RAID levels 2-4 cause writes to be serialized by the process of updating parity.
- RAID 6 (Figure 2d) uses rotating parity like RAID 5, but increases the fault tolerance of the array by adding another block of parity per stripe. This RAID level is the most reliable of the standard parity-based levels.

Figure 2 shows the canonical RAID levels that are widely used today. Figure 3 shows the canonical RAID levels that have fallen out of general use. These levels are no longer favored because of a lack of parallelism for small independent operations, with RAID 2 having the additional detriment of requiring much more hardware than other levels to perform erasure correction. It is notable that, while RAID 4 does lack small random write parallelism, NetApp has created a proprietary file system (the WAFL file system [**47**]) that allows many independent writes to occur simultaneously by locating them within the same stripe. NetApp has also created an alternative RAID level, RAID DP, with two dedicated parity disks [**110**]. RAID TP, which is not a NetApp product, a rotating parity RAID level with three parity blocks per stripe, has recently been introduced and is included with some hardware controllers.

There are methods that increase the reliability of RAID without requiring new RAID levels, but instead compose them. One widely used method is HRAID [**6**]. HRAID has two levels of RAID: An inner level that aggregates disks, and an outer level that aggregates arrays. For example, RAID 1+0 aggregates multiple RAID 1 arrays into a RAID 0 array. While HRAID can use any two RAID levels, the set of HRAID types in common use is quite restricted. Figure 4 shows the most popular HRAID types. These increase reliability by dedicating more capacity within the array to holding parity. For example, a RAID 6+0 array that has two inner RAID 6 arrays (like Figure 4c) can lose up to four disks: Two from disks 0–7, and two from disks 8–15. However, data loss can occur with as few as three failures if they all occur in the same set, regardless of the number of sets in the outer array. A similar approach that aggregates storage across multiple nodes in a similar way, MRAID, has been discussed [**104**].

As recently as 1992, there were no controllers available from industry that implemented RAID 5, while a university team had implemented a hardware RAID 5 controller [**55**]. Today, a wide variety of RAID implementations exists. I/O processors, like the Intel 81348 Processor [**23**], can be integrated onto a circuit board to carry out I/O-related operations between a host computer and an array of disk drives, including RAID.

1.1. Software RAID and RAID-Like File Systems. Several operating systems include software RAID implementations. Specific examples include Microsoft Windows Server 2003, which supports RAID levels 0, 1, and 5 [**106**]; Linux, which supports RAID levels 0, 1, 4, 5, and 6 [**105, 116**]; and Mac OS X, which supports RAID levels 0, 1, and 10 [**51**]. While hardware implementations have maintained a reputation of being the high-performance path for RAID, software RAID is beginning to gain a foothold in high-end installations. Linux software RAID is used in the Red Sky supercomputer at Sandia National Laboratories [**70**], and storage vendors are taking advantage of new CPU architectures to drive storage servers implementing RAID in software [**66**].

The Zettabyte File System (ZFS), a file system implemented for the Solaris operating system, includes RAID 5- and RAID 6-like functionality through RAID-Z and RAID-Z2, respectively [**68**]. One notable differentiation from typical RAID implementations is the lack of the RAID 5 and 6 “write hole,” which is the period of time where the parity and data on disk may not be consistent with each other. Power failure during this time frame can cause data to be corrupted. A further advancement provides RAID-Z3, a software implementation that provides for triple-parity RAID [**61**]. Leventhal describes these implementations as derivatives of Peter Anvin’s work on the Linux RAID 6 implementation [**5**].

Brinkmann and Eschweiler described a RAID 6-specific GPU erasure code implementation that is accessible from within the Linux kernel [**14**]. They contrast their work with that found in Chapter 5 by pointing out that their implementation is accessible from within the Linux kernel. However, their coding implementation also runs in user space; a micro-driver is used to communicate between the GPU and the kernel space components. Further, the implementation they describe performs coding suitable for RAID 6 applications, while this work describes a generalized $k+m$ RAID implementation. Another GPU implementation of RAID 6 was being pursued at NVIDIA, but has not seen public release [**54**].

Several FPGA-based implementations of Reed-Solomon codes exist for erasure correction applications, including RAID [**37**] and distributed storage [**102**]. Further applications in communications have benefited from high-speed implementations of Reed-Solomon coding on FPGAs [**60**]. A multiple disk hardware file system implementation has been created in an FPGA that supports RAID 0 [**67**].

Parallel and distributed file systems are typically installed on several nodes in a cluster that use RAID arrays as underlying storage. At the same time, the parallel file system will ensure that data are available in the case of one or more nodes becoming unavailable by using RAID techniques. Production file systems like Ceph [**109**], Lustre [**1**], and the Google File System [**35**] use replication (i.e., RAID 1 techniques) to ensure the availability of files. Several network file systems for experimental use have been presented that use coding algorithms to reduce storage overhead [**18, 93**]. An analysis of the trade-offs between replication and erasure coding in the context of distributed file systems has been conducted [**108**]; erasure coding was found to be superior for many metrics.

2. Coding Algorithms

An erasure correcting code is a mathematical construct that inserts redundant data into an information stream. These redundant data can be used for data recovery in the case of known data loss, or erasures [**63**]. The process of generating the data is called encoding. If some limited amount of data are lost, the remaining data in the information stream can be decoded to regenerate the missing data. In the context of RAID, where $k + m$ disks are composed into an array that can tolerate m failures, k chunks of user data are used in the coding process to generate m chunks of parity. There is a wide variety of codes that can be used in a RAID-like context, including many that are less computationally expensive but require more storage to implement.

To aid discussion, coding algorithms will be classified based on two characteristics: Generality (indicating whether k and/or m are fixed), and separability (with a maximum distance separable code requiring m extra chunks of storage to recover from m failures). RAID 6 codes, for example, may or may not be general; however, they must be maximum distance separable, as a RAID 6 array must survive two disk failures, but must require exactly two extra disks for code storage overhead. Non-general codes generally have better computational characteristics for given m than other codes, but they may be patent-encumbered. Non-general codes are also sparse in m, as only certain values of m are considered useful for RAID 6 or RAID TP, restricting the research devoted to codes with higher m. The following list of codes is not exhaustive, but is intended to demonstrate that codes exist in all dimensions, with certain codes being more well-suited for particular types of workloads.

2.1. General, MDS Codes. Reed-Solomon coding, while initially developed for noisy communication channels like radio, is one algorithm which can be used for RAID 6 (or any $k + m$ coding) [**20, 92**]. Specifically, Reed and Solomon described a code that offers optimal storage utilization for the reliability required, in that a system that must protect from m erasures for k equally sized pieces of data must store $k + m$ pieces of that size. Several open-source packages exist that implement Reed-Solomon coding in the context of erasure coding for storage, including zfec [**79**] and Jerasure [**89**]. RAID 6-specific optimizations of Reed-Solomon coding have been created for use in implementations, including that used in the Linux kernel [**5**]. Multi-core scheduling of polynomial operations for Reed-Solomon coding has been examined [**103**].

2.2. Non-General, MDS Codes. The simplest, most highly fault-tolerant, but least storage efficient scheme for fault tolerance is simple N-way mirroring, which is simply replicating data among many storage resources unchanged. In the $k + m$ terminology, N-way mirroring has a fixed k ($k = 1$), with varying m. Mirroring requires no computation, as no data are changed, but software implementations can suffer from reduced data bandwidth, and all implementations suffer from high storage overhead. RAID 1 is a straightforward implementation of N-way mirroring [**84**]. Creating a hierarchical RAID 1+0 system, where multiple RAID 1 arrays are treated as individual storage resources within a large RAID 0 array, is a means of increasing the usable space of a mirrored array beyond that of a single disk.

Blaum et al. developed a RAID 6-specific algorithm called EVENODD, which is provably optimal (asymptotically) in the amount of storage and number of operations required [**10**]. They describe EVENODD as being the second erasure code (after Reed-Solomon coding) that is capable of implementing RAID 6, with the benefit that it uses only XOR operations. At the time of its introduction, using XOR as the only operation was an advantage of the algorithm, as hardware RAID controllers that provide RAID 5 already included hardware XOR capabilities [**55**], allowing them to be repurposed for RAID 6.

Corbet et al. developed the Row-Diagonal Parity (RDP) algorithm [**22**], another RAID 6-specific code. They describe the algorithm as more computationally efficient than EVENODD in practice while maintaining the same asymptotic characteristics. RDP also uses only XOR operations.

2.3. General, Non-MDS Codes. Tornado Codes are a family of erasure codes that are encodable and decodable in linear time [**17**]. Tornado Codes are defined to be probablistic codes defined by a sparse system, unlike Reed-Solomon codes. Tornado Codes are considered to be inappropriate for online, block-based storage systems because of their large storage overhead, as they use much more parity than Reed-Solomon coding, and the cost of propagating changes of data chunks to affected parity chunks is significant [**111**].

2.4. Non-General, Non-MDS Codes. Weaver codes are several families of XOR-based codes that provide constrained parity in-degree [**41**]. These codes described are not MDS (having a storage efficiency of at most 50%, identical to 2-way mirroring), but have several other interesting properties that make them desirable for distributed RAID systems, including improved locality. Hafner describes several instances of Weaver codes that are up to 12 disk failure tolerant, but there is no proof that a Weaver code can be generated that can tolerate any particular number of failures.

3. General Purpose GPU Computing

Workstations with graphics output have a heavy computation load associated with 2D and 3D graphics. In order to improve overall system performance, GPUs were created to perform graphical tasks efficiently to yield increased graphics quality and increased system performance. One early observation of graphics researchers was the inherent parallelism of graphics [**90**], so parallel GPU architectures have long been in use. Many researchers who wished to perform their computations faster have attempted to apply GPUs as parallel processors, resulting in a new sub-field of computer science: General purpose computation on GPUs.

One of the first GPU applications did not target the programmable shader processors of today's GPUs, but instead targeted a texture combining mode. These functions were accessed directly via the OpenGL [**99**] or Direct3D [**40**] graphics APIs. Larson and McAllister demonstrated that a GeForce3, a GPU that used four pixel pipelines to parallelize graphics, could be used to multiply matrices by storing them as textures and using multiplicative and additive blending of these textures on a rendered polygon [**59**].

As GPU technology developed, and users were demanding more realistic real-time graphics, APIs and hardware support were created to allow developers to load their own fragment shader programs into GPUs [**11**]. This allowed programmers to create fragment shaders of arbitrary complexity (while obeying instruction count limits, which were originally quite constraining). The program still had to render to the framebuffer, but now algorithms that could not be implemented with texture units and other portions of the graphics pipelines could be created. This style of computing was first available from NVIDIA in the GeForce3 GPU [**74**]. Further capability enhancements included a full 32-bits per component, allowing 32-bit floating point precision to be obtained in computations.

Further enhancements to GPUs included efficient mechanisms for rendering directly to another texture instead of to the framebuffer memory [**12**]. This eased the creation of algorithms that required feedback, the reprocessing data that was processed earlier by the GPU. The building blocks were in place to create fast and advanced applications on GPUs for many important types of computations, including simulation of physics [**45**], protein interaction [**80**], and planetary systems [**115**]. Further applications were developed to perform numerical computing, including LU decomposition [**25, 33**] and conjugate gradient [**13**]. Advancements in programming for this style of computation include Cg, a language designed to create shader and vertex programs for GPUs with different instruction sets [**30**]; BrookGPU, an implementation of the Brook streaming language for GPUs [**16**]; and Sh (which has since become RapidMind), a metaprogramming language for GPUs [**64**].

While many applications were successfully implemented on GPUs with vertex and fragment shaders, there are significant hurdles to using this style of GPU computation. One of the most limiting is the lack of scatter capabilities [**43**]. Each running thread is assigned an output position by the rasterizer, implying that scattering functionality must be simulated through refactoring the algorithm to use gather or vertex processors to get scatter in limited contexts. Further difficulties included (before the advent of Shader Model 4) limited data type capabilities. Emulation of the unavailable types, which included double precision floating point, could prove inefficient if not done carefully [**38**]. While there are several floating point types and vectors thereof, there were no integer types or operations. Furthermore, only certain vector lengths and types are supported for some operations such as vertex texture fetch [**34**].

As the interest in programming GPUs to do non-graphics tasks increased, ATI (via Close-to-Metal, now known as AMD Stream Computing [**3**]) and NVIDIA (via CUDA [**75**]) released hardware and software to allow more general purpose tasks to be more efficiently programmed. Both include general scatter functionality and integer types. Furthermore, each allows bit operations like shifts, XOR, AND, and OR. Further contributions include NVIDIA's parallel data cache, a fast memory that can be accessed by several shader units simultaneously. These qualities taken together create computing platforms that are easier to use and more efficient for general purpose tasks than the OpenGL- or DirectX-based methods. As concern over methods of programming different types of GPUs and other multi-core devices increased, OpenCL was proposed as an open, royalty-free standard for writing programs for multi-core devices, including GPUs [**71**]. While the API and workings heavily resemble CUDA driver mode, OpenCL has an extension system similar to that of

OpenGL to facilitate vendor-specific extensions that are not part of the OpenCL API. OpenCL implementations are now available for a variety of compute devices.

Many of algorithms and applications have been implemented in CUDA, OpenCL, and AMD Stream languages. Much work has been done to implement primitives for parallel computing on GPUs, including the parallel prefix sum [**44**], an algorithm that has many practical applications. Numerical computing, while popular for OpenGL-based GPGPU applications, has received a significant performance boost because of new capabilities from explicit caching facilities [**52**]. Current NVIDIA and ATI/AMD devices offer a superset of the application possibilities of OpenGL-based methods, allowing those previous applications to be implemented to take advantage of more device features [**78, 85**]. Some storage-related algorithms that benefit from expanded data types and operations have been implemented with CUDA. AES encryption, which can be used for on-disk encryption, has been demonstrated [**113**]. SHA-1, which can be used for deduplication or content-based addressing, has also been implemented [**114**].

CHAPTER 3

The $k + m$ RAID Levels

Calculating the lifespan of a RAID array can yield falsely encouraging results. Manufacturer-estimated disk mean time to failure (MTTF) statistics are on the order of one million hours, yielding an approximate mean time between failures (MTBF) for a 32-disk RAID 6 array that exceeds 100 million years. (In comparison, mass extinction events on Earth occur on average every 62 million ± 3 million years [**95**].) While this is a display of the inadequacy of MTBF as a statistic for choosing storage infrastructure configurations (a 100,000,000 year MTBF translates to approximately 99.99999% probability of experiencing no data loss in 10 years), real-world array reliability is not reflected by this MTBF.

To mitigate challenges in disk and array reliability discussed in this chapter, a generalization of RAID called $k + m$ RAID is proposed. For this broad class of RAID, the mechanism for supporting fault tolerance is familiar. For example, in RAIDs 5 and 6, storage is organized into multiple stripes of fixed size, with each chunk of the stripe stored on a separate disk. The construct $k + m$ indicates how the chunks of each stripe are used; k is the number of chunks per stripe that store data, and m is the number of chunks per stripe that store parity. Thus, RAID 5 is identical to $k + 1$ RAID, while RAID 6 is identical to $k + 2$ RAID.

$k + m$ RAID arrays are m disk failure tolerant: Up to m disks may fail before any data have been lost. However, the benefits to using $k + m$ RAID for some applications are best realized by never having m disks fail, but instead by having some extra disks available at all times, with a minimum of one or two excess chunks per stripe depending on array requirements. This chapter also demonstrates the reasoning behind these requirements.

1. Disk Reliability

Disks are difficult to analyze from a reliability standpoint for many reasons, chief among them being the relentless pursuit of higher densities and the pressure to introduce new features quickly. Testing of new drives for their mean time to failure characteristics is necessarily accelerated, and assumptions have to be made about how the results can be applied to drives as they age. Further, the effect of UREs is under-emphasized, which increases the risk of small data losses. Finally, other unusual circumstances can lead to data corruption without outright physical failure of any component. All impact the reliability of storage, especially when using disks aggregated into arrays.

1.1. Disk Failures. Large disk population failure studies have been rare until recently, as corporations do not tend to track these statistics, and drive manufacturers do not run large disk installations for testing drives long term. Instead, drives are assumed to follow the same types of patterns that other electronics do, namely the "bathtub curve," which expresses failures caused by infant mortality, component wear, and a constant rate of failure for other reasons [**57**]. See Figure 5 for an illustration of the bathtub curve and its components.

Disk manufacturers cite impressive MTTF statistics for their disk drives that are obtained via estimates based on medium-scale studies (500 drives) at high temperatures (42° C/108° F) continuously over short periods of time (28 days), then correcting and extrapolating these findings for a drive's expected lifetime [**21**]. Unfortunately, there are problems with this type of testing:

- The tests assume that, as temperatures rise in an enclosure, failures increase. A recent study has shown that there is little correlation between drive failure and operating temperature [**86**].
- The tests assume that drive failures tend to remain constant after the first year for the drive's expected lifetime, with a high infant mortality rate [**21**].

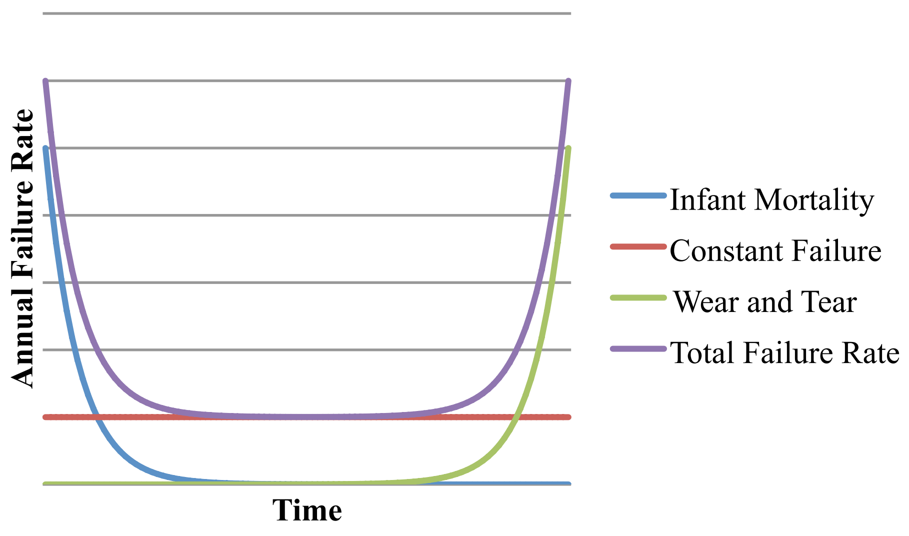


Figure 5. The Bathtub Curve as a Model for Failure [**57**]

> A recent study has shown that drive failure can begin increasing as soon as the second year of operation of a disk drive [**100**].

The authors of the above-mentioned recent studies, through the benefit of analyzing maintenance records for many large systems using millions of disks, have drawn the conclusion that drive MTTF estimates provided by drive manufacturers are too large by a factor of two to ten, or more for older disks [**86, 100**]. Such discrepancies require adjustments when calculating RAID reliability.

1.2. Unrecoverable Read Errors. A significant source of data loss that plagues hard disks is the URE, a type of error that causes data loss without outright failure of the disk [**39**]. Such errors can be caused by several factors, but the end result is the same: An entire sector (either 512 bytes or 4,096 bytes in size) of the media is unreadable, resulting in data loss.

The statistics for UREs, defined by the rate at which they occur by the Bit Error Rate (BER), often appear innocuous. Such errors are encountered for one sector per 10^{15} bits read for typical hard disks [**97**] and solid state drives [**24**], implying a BER of 10^{-15}. However, these events can make storage vulnerable to increased risk of losing

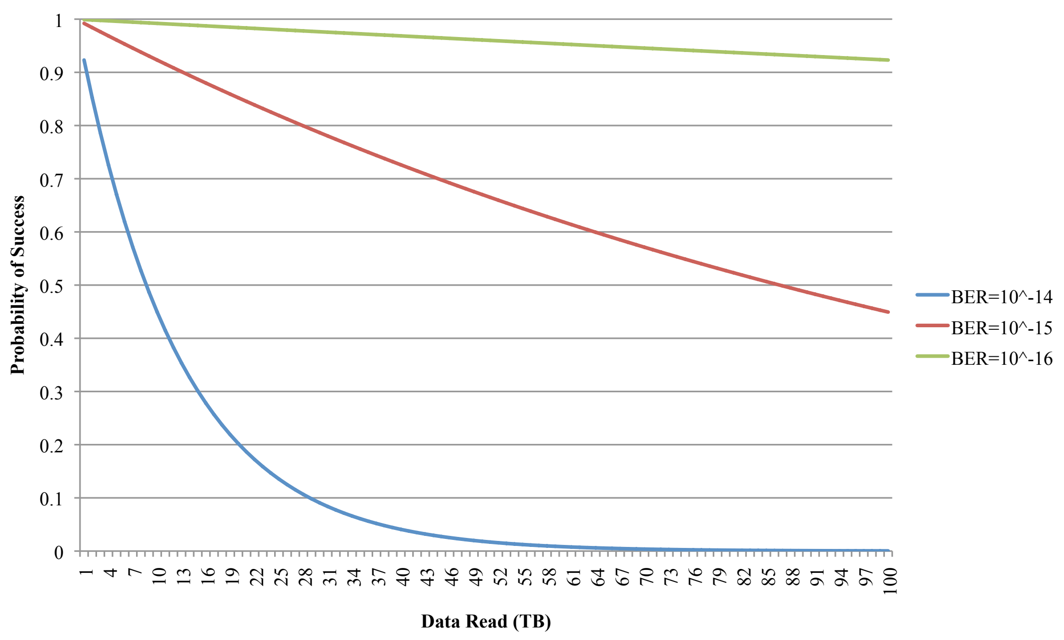


Figure 6. Probability of Avoiding a URE, Calculated with Equation 4

data as the number of bits read approaches a significant fraction of the inverse of the BER.

In a healthy RAID array, an unreadable sector is unlikely to cause significant problems, as this is a condition reported to the RAID controller. The RAID controller can then use the contents of the other disks to recover the lost data, assuming that there is excess parity in the array. However, as RAID arrays experience disk failures, an array can be left with some amount of the volume unprotected by redundancy. For RAID 6 arrays, double disk failures do happen (an intuitive reason that RAID 5 arrays are often considered inadequate), and UREs are frequent enough that a volume unprotected by redundancy is at unacceptably high risk of data loss. System administrators at Sandia National Laboratories have encountered multiple instances where production RAID 6 arrays with 10 disks have suffered double disk failures and UREs, causing extensive volume maintenance to recover data [**69**]. Figure 6 shows that, given the described configuration with two-terabyte disks and a double disk failure, the probability of surviving the rebuild process without data loss is less than 0.89, assuming a BER of 10^{-15}. The BER of disks is an example of increasing disk sizes causing significant problems when the reliability remains constant. As disks grow larger, more data must be read to reconstruct lost disks in arrays, thus increasing the probability of encountering UREs.

1.3. Further Sources of Data Loss. If an array is otherwise healthy, UREs are relatively simple to handle. There are other types of errors that are significantly more rare, but have the potential to cause user processes to receive incorrect data from storage. These errors are unreported, causing passive means of ensuring data integrity to fail. Such failures are difficult to analyze because of their infrequent (and often undetected) nature, but at least one case study has been performed in an attempt to quantify the possible impact [**7**]. Listed reasons for such errors include disk firmware bugs, operating system bugs in the I/O stack, and hardware failing in unusual ways. Some applications cannot afford to encounter such unreported errors, no matter how rare.

2. A Model for Calculating Array Reliability

Two formulas were given by Chen et al. to calculate the reliability of RAID arrays, taking only disk failures into account [**20**]:

$$RAID\ 5\ MTBF = \frac{MTTF^2}{n(n-1)MTTR} \tag{1}$$

$$RAID\ 6\ MTBF = \frac{MTTF^3}{n(n-1)(n-2)MTTR^2}, \tag{2}$$

where $MTTF$ is the MTTF of a single disk, n is the total number of disks in the array, and $MTTR$ is the time required to replace and rebuild a failed disk. Other terms were included in later derivations to incorporate other risks, such as UREs. The above formulas can be extended to $k+m$ formulations, tolerating up to m failures of $k+m=n$ disks without losing data, as follows:

$$k+m\ RAID\ MTBF = \frac{MTTF^{m+1}}{\frac{(k+m)!}{(k-1)!} \times MTTR^m}. \tag{3}$$

Unfortunately, in the paper where these formulas were originally derived, an assumption was made that the MTTR of a disk array is negligible compared to the MTTF of a disk in the array [**84**]. Even as recently as 1994, an MTTR of one hour was considered reasonable [**20**]. Such figures are no longer reasonable, as the MTTR has increased and MTTF has not substantially increased [**100**]. Even with the inclusion of hot spares, idle disks included within an array to be used as replacements upon disk failure, rebuild times can span several hours, days, or weeks for systems under significant load.

The calculation of the likelihood of encountering a URE when reading a disk is a straightforward exercise in probability. Since a hard disk operates on a sector level, read errors do not occur on a bit-by-bit basis. Instead, entire sectors are affected. As such, the sector error rate must be used to compute probability of data loss. The relationship between the probability of encountering sector errors and the amount of data read is perilous given the volume of data that is typically processed during array rebuilds, as shown in Figure 6.

In the following calculations for an array which can tolerate m failures without data loss, the Poisson distribution (denoted by $POIS$) is used to calculate several related probabilities:

The probability of encountering a URE, with the sector size expressed in bytes, is:

$$P_{ure}(bytes\ read) = \ 1 - (1 - sector\ size \times BER \times 8\frac{bits}{byte})^{\frac{bytes\ read}{sector\ size}}. \tag{4}$$

The probability of the first disk failing, where n is the number of disks in the array, and array life is expressed in years, is:

$$P(df_1) = \ 1 - POIS(0, n \times AFR \times array\ life). \tag{5}$$

The probability of the i^{th} disk failing, where $i = 2 \ldots m + 1$, within the MTTR of the previous failure, where MTTR is expressed in hours, is:

$$P(df_i) = \ 1 - POIS(0, (n - i + 1) \times AFR \times MTTR). \tag{6}$$

The probability of encountering m failed disks and a URE is:

$$P_{sector} = P_{ure}(disksize \times (n - m)) \prod_{n=1}^{m} P(df_i). \tag{7}$$

The probability of data loss caused by encountering $m+1$ failed disks or an unmitigated URE is:

$$P_{fail} = P_{sector} + \prod_{n=1}^{m+1} P(df_i) - P_{sector} \times \prod_{n=1}^{m+1} P(df_i). \tag{8}$$

The probability of data loss caused by losing all hot spares between service period, where h is the number of hot spares, and s is the service interval (in hours, where s is small for attendant technicians), is:

$$P_{hot} = POIS(h + m + 1, n \times AFR \times s/(24 \times 365.25)). \tag{9}$$

The total probability of data loss is as follows:

$$P_{loss} = P_{fail} + P_{hot} - P_{fail} \times P_{hot}. \tag{10}$$

3. Current High-Reliability Solutions

RAID 6 is a commonly supported RAID level that offers high reliability, but other variations exist that are designed to provide increased reliability. These are commonly termed hierarchical RAIDs, which configure RAID volumes containing disks (termed "inner arrays") to act as devices in a large RAID volume (termed "outer array") [**6**]. In this document, the naming scheme used for such RAIDs is RAID $a + b$, where a describes the RAID type for the inner RAIDs and b describes the RAID type for the outer RAID. The rationale behind hierarchical RAID levels is that each additional sub-array introduces more parity into the system, increasing the fault tolerance overall, even if the outer RAID does not contain any additional parity. Outer RAID 0 organizations are much more common than any other, with controllers

Inner RAID	Outer RAID Level			
Level	0	1	5	6
0	0%	$\geq$ 100%	12.5%	25%
1	$\geq$ 100%	$\geq$ 300%	$\geq$ 106.25%	$\geq$ 112.5%
5	25%	$\geq$ 106.25%	40.63%	56.25%
6	25%	$\geq$ 112.5%	56.25%	87.5%

TABLE 1. Hierarchical RAID Storage Overhead for Sample Configuration

often supporting RAID 1+0 (striping over mirrored disks), RAID 5+0 (striping over RAID 5 arrays), and/or RAID 6+0 (striping over RAID 6 arrays).

There are no theoretical restrictions on which RAID levels nest together, nor is there a limit to the depth of nesting. However, when ignoring the additional computational complexity of providing two levels of parity generation, nesting RAID levels when the outer level provides reliability requires a large investment in storage resources. Table 1 shows that, when using $4 + 1$ or $8 + 2$ configurations for inner RAIDs when possible, hierarchical RAID involves at least a 40% overhead in storage requirements while potentially doubling processing requirements.

These concerns indicate two classes of reliability within the hierarchical RAID levels. Some can be considered somewhat more reliable than non-hierarchical RAID levels, as they simply provide more inner parity without adding any outer parity (levels [1-6]+0). Others drastically increase the reliability by adding additional parity to the outer array that can be applied to recover any failure encountered by an inner array (levels [1-6]+[1-6]). From Table 1, it is clear that storage overhead for RAID [1-6]+[1-6] is high. RAID 5+5 is most storage efficient, but still requires more than 40% storage overhead. Levels [1-6]+[1-6] are not commonly implemented because of both this storage overhead and the additional level of computation. Instead, the simpler levels (RAID [1-6]+0) are most commonly used. These are straightforward to analyze from a reliability standpoint:

$$P_{loss}(nsets) = P_{loss}(nsets - 1) + P_{loss}(1) - P_{loss}(nsets - 1) \times P_{loss}(1) \qquad (11)$$

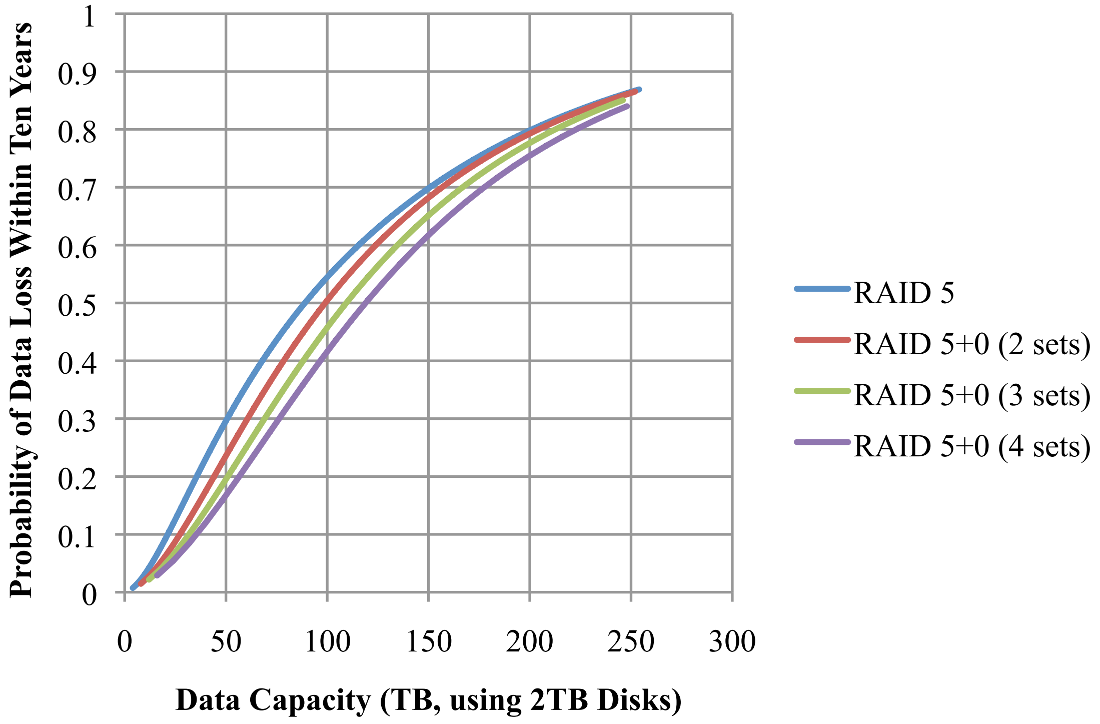


Figure 7. Comparison of Reliability: RAID 5 and RAID 5+0 with Varying Set Sizes, BER of 10^{-15}, 12-Hour MTTR, and 1,000,000-Hour MTTF

The base case is $P_{loss}(1)$, which is simply P_{loss} for the inner RAID level.

Figures 7 and 8 demonstrate the differences between RAID 5 and RAID 5+0, and RAID 6 and RAID 6+0, respectively. It is worth noting that RAID 5+0, even when split to four sets, does not appreciably increase the reliability over RAID 5 with the same capacity. The additional parity does not help because RAID 5 is not capable of correcting UREs during rebuild operations. RAID 6 does benefit more appreciably, with more than an order of magnitude difference between RAID 6 and RAID 6+0 over four sets. This increased reliability comes at the cost of quadrupling the storage overhead.

RAID 1+0, while extreme in the amount of overhead required, is computationally simple and has a high reputation for reliability. Figure 9 shows the reliability for three RAID 1+0 configurations. While more replication has higher reliability, two-way replication suffers from the same problems encountered with RAID 5+0. While

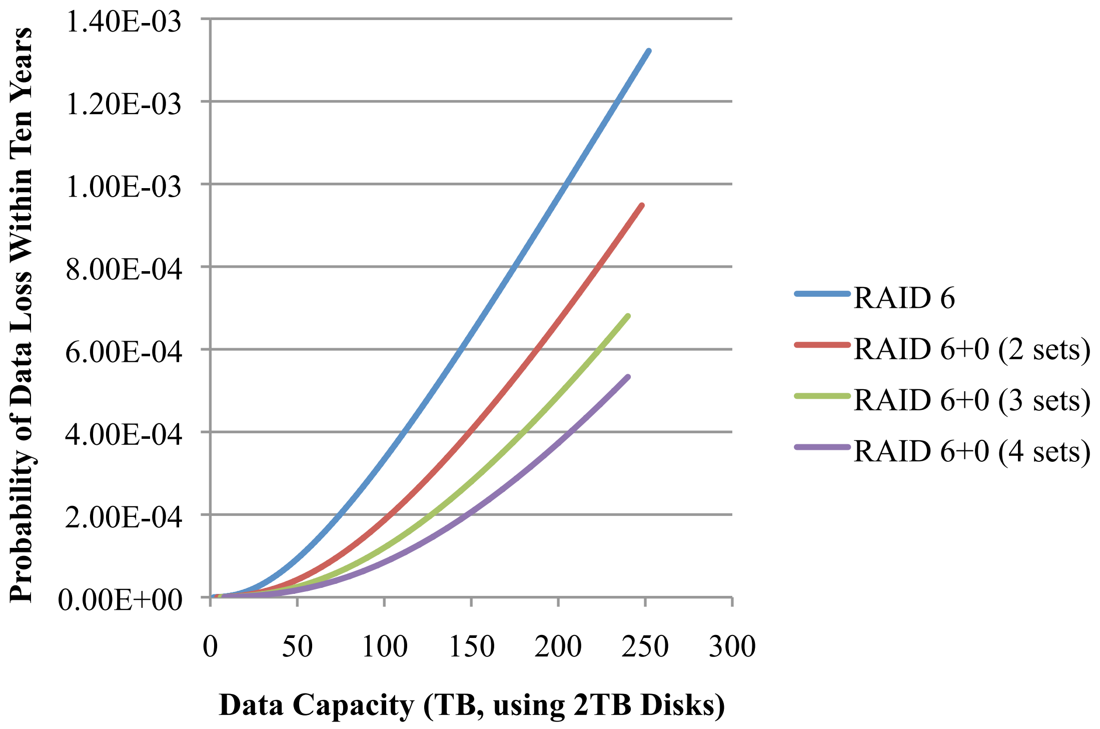


FIGURE 8. Comparison of Reliability: RAID 6 and RAID 6+0 with Varying Set Sizes, BER of 10^{-15}, 12-Hour MTTR, and 1,000,000-Hour MTTF

three-way and four-way replication do improve reliability significantly, the storage overhead is 200% and 300%, respectively.

4. $k + m$ RAID for Increased Reliability

One contribution of this work is the demonstration of a capability to run RAID arrays containing arbitrary amounts of parity with commodity hardware. Typically, today's controllers implement RAID levels 1, 5, 6, 5+0, 6+0, 1+0, and rarely RAID TP (a recently introduced triple-parity RAID level that is equivalent to $k + 3$ RAID). This work implements RAID that can dedicate any number of disks to parity, enabling any $k + m$ variant, subject to restrictions of Reed-Solomon codes pertaining to word size used.

Figure 10 shows a comparison between variants of each commonly used level: RAID 5+0, with four sets; RAID 6+0, with four sets; and RAID 1+0, with three-

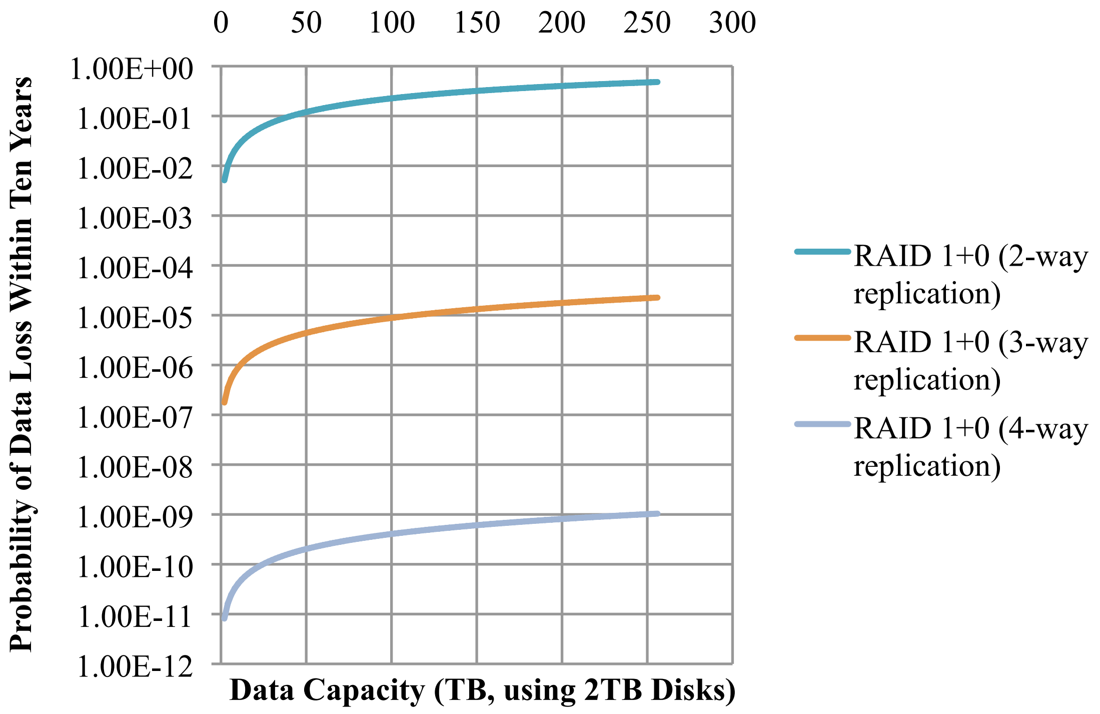


FIGURE 9. Comparison of Reliability: RAID 1+0 with Varying Replication, BER of 10^{-15}, 12-Hour MTTR, and 1,000,000-Hour MTTF

and four-way replication. It is clear that RAID 5+0 should not be used when data integrity is important. The RAID 1+0 variants show that, if one can tolerate the 200-300% storage overhead, RAID 1+0 offers excellent protection from data loss (disregarding the possibility of not knowing which data is correct in the case of data corruption). The curves for RAID 1+0 at all points have a smaller derivative than the parity-based RAIDs; this is because RAID 1+0 is the only RAID level shown that increases redundant data as capacity grows.

It is clear that, by increasing the parity, the array's expectation of survival for a time period increases by a significant amount while requiring a small investment of additional storage resources. Further, each additional parity disk increases the number of disks that may be managed within a single array substantially while keeping reliability fixed. For example, a system administrator may decide that a reliability of 99.9999% over 10 years is justified based on availability requirements. According to

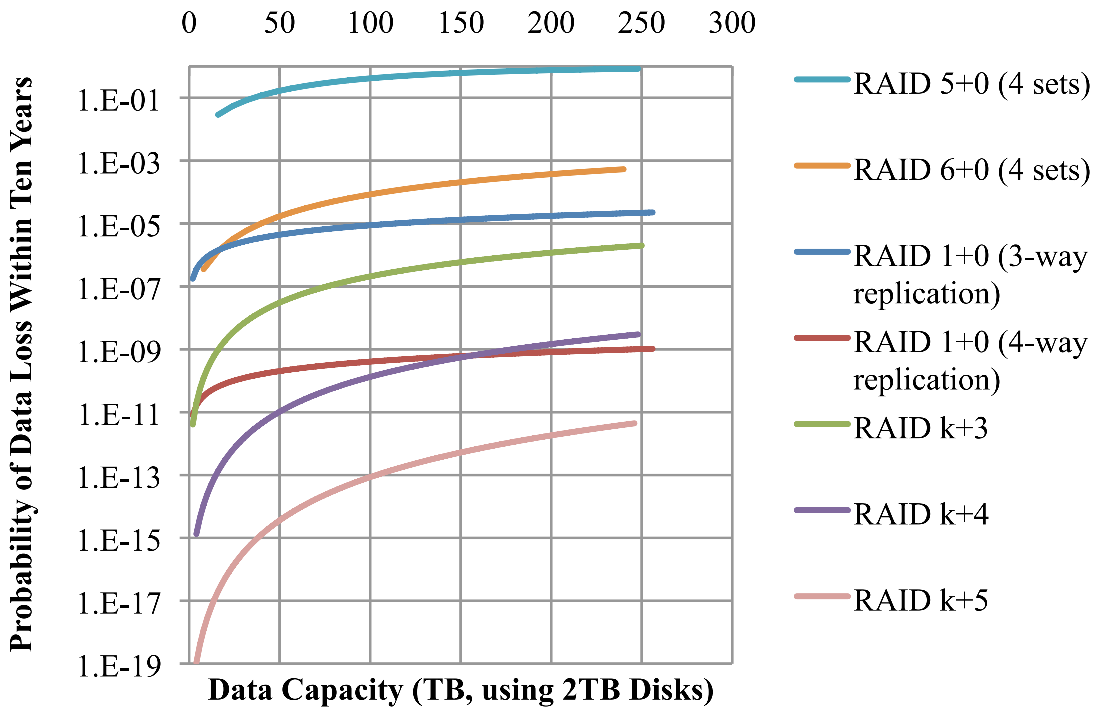


FIGURE 10. Comparison of Reliability: Several RAID Levels with BER of 10^{-15}, 12-Hour MTTR, and 1,000,000-Hour MTTF

the data behind Figure 10, found in Appendix B, this can be done with RAID 1+0, but only with three disks of data in the array, with 66% overhead. Upgrading to a RAID 6+0 array with two sets increases the data capacity supported to eight disks of data, with 50% overhead. Instead, by only adding a single parity disk to the RAID 6 array to upgrade to $k + 3$ RAID, 93 disks may be included within the array, with approximately 3.2% overhead.

4.1. Guarding Against Reduced Disk Reliability and High Load. As discussed in Section 1.1 of this chapter, studies have shown that disks are up to 10 times more likely to fail than manufacturers describe before accounting for advanced age [**86, 100**]. Further, a MTTR of 12 hours was assumed in discussions thus far, but such repair rates may not be realistic for systems servicing client requests during the rebuild. A disk drive can reasonably sustain approximately 100 MB/s of transfer, implying two terabytes will be written at that rate to complete a rebuild. If the rest

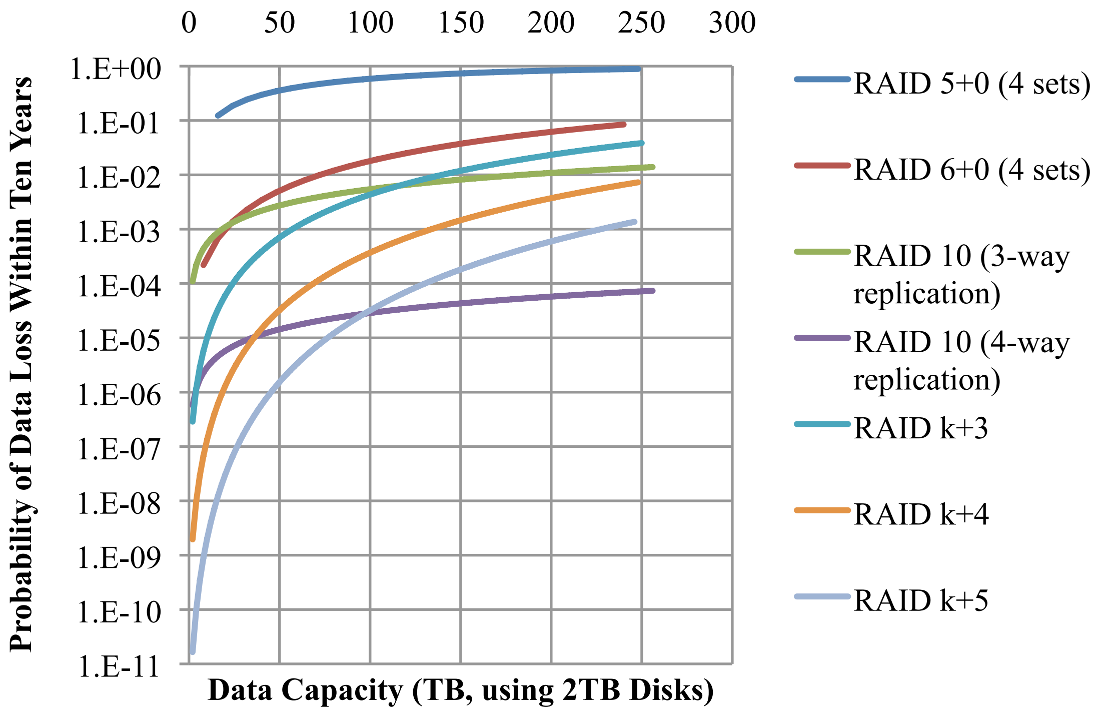


FIGURE 11. Comparison of Reliability: Several RAID Levels with BER of 10^{-15}, One-Week MTTR, and 100,000-Hour MTTF

of the array can keep pace, this operation will require a minimum of 5.6 hours to complete. For a 16-disk RAID 6 array, rebuilding a failed disk at 100 MB/s requires at least 1400 MB/s of bandwidth from other disks that can no longer be used for servicing client requests. RAID systems can reasonably experience rebuild times that are on the order of a week based on high client load and low bandwidths per disk. While declustered RAID can increase rebuild speeds, it lowers array capacity and can cause other problems based on the layouts used [**48**].

Figure 11 shows the dramatic effects of increased MTTR and decreased MTTF: Large arrays become less likely to survive for long periods without data loss. RAID 1+0 with four-way replication, RAID $n+4$, and RAID $n+5$ are the only RAIDs shown that can provide 99% reliability for arrays approaching 250 TB in capacity. Meanwhile, for 128 TB arrays, the outlook is better: RAID 1+0 with four-way replication and RAID $n+5$ can offer more than 99.99% reliability, RAID $n+4$ can offer 99.9% reliability.

While the parameters are more extreme than many will expect, this is a reasonable lower bound on reliability based on high load and worst-case MTTF (or MTTF in harsh environments). In such cases, high-parity RAID may be the only economical solution to ensure data integrity.

4.2. Read Verification for Unreported Errors and UREs. One reason UREs are so perilous is that they have a high probability of being encountered during the rebuild phase of a RAID array, when redundancy can be completely absent. Such an error can be found on a disk sector that has not been accessed for a long period, and is only found when it cannot be repaired without restoring lost data from backups. Further, corrupted data on disk from an unreported error cannot be detected when redundancy is eliminated.

A partial solution to these situations is a process known as scrubbing [**81**]. Scrubbing is a feature where a controller starts reading from the beginning of the RAID device to the end, attempting to detect UREs and parity inconsistencies as a background process during normal operation of a RAID array. Such errors are usually fixed by either rewriting the data to the same sector or by remapping the sector, but the lost data must be recovered from the parity available. Scrubbing allows for such errors to be caught before redundancy is lost in an array.

Using scrubbing in a large array that is always in use is not ideal. Such activities rely on idle time in the array, or must use some of the available bandwidth, reducing client performance. Also, scrubbing will not prevent a client from using bad data resulting from an unreported error before the scrubbing process could detect and resolve the discrepancy, demonstrating the need to verify all data on read.

While read verification alone does improve reliability significantly, disk failures can decrease or eliminate the ability to verify data integrity. To correct an unreported error, more than a single extra parity block is necessary; a single parity chunk will merely serve to detect that an error has occurred. While error detection without correction is still useful, as a long-running compute job can be immediately terminated instead of wasting further compute resources to produce a likely incorrect result, the ability to correct errors and continue work provides more utility. To correct errors, it is necessary to maintain at least two chunks of parity within the system at all times, even while operating in degraded mode.

It should be noted that RAID 1+0 does not have strong error correction (as opposed to erasure correction) capabilities when using less than four-way replication. Unreported errors or data corruption can be propagated into the data stored on a single disk, causing a problem when two supposedly identical disks contain different data. When there are only two replicas, which is the case under 2-way replication or 3-way replication with a single failure, there is no way of resolving such discrepancies without another data source. Furthermore, since all disks are executing the exact same workload by definition, unreported errors caused by firmware bugs may be encountered in several disks simultaneously, provided that they are identical models.

Similar lack of error correction can also be noticed in the parity-based RAIDs. For $k+1$, error detection is possible but error correction is completely absent. $k+2$ suffers the same problem when operating in degraded mode. This lack of error correction for unreported errors further motivates the need for $k+m$ RAID in conjunction with read verification.

4.3. Performance Impact of k+m RAID. Higher levels of parity have obvious performance impacts on small writes, and on small reads with read verification with parity, because of the necessity of accessing more disks than are involved in data operations. For example, a small write without read verification involving data stored on only one disk requires $2m+2$ I/O operations: A read of the previous contents of the data chunk, a read of all parity chunks, and a write of the updated data and parity chunks. With read verification, $k+m+2$ I/O operations must occur:

(1) **Read** $k+1$: Because at least some data must be read (minimally, the parity to update), $k+1$ chunks must be read to have any error detection capability.

If $k \geq m$, the affected data block, the parity chunks, and $k - m$ unaffected data chunks can be read.

(2) **Modify**: An update can be completed with the affected data chunk and parity chunks, or simple generation of parity if less than m chunks of parity are read.

(3) **Write** $m + 1$: The updated data and parity chunks must be written.

However, for large and/or streaming operations, there is little I/O overhead compared to the amount of data written or read, even for verified operations. Each read incurs $k + 1$ chunks per stripe read, with a single chunk of overhead, and each write incurs $k + m$ chunks written, where k is generally significantly larger than m. I/O workloads that are streaming in nature can take advantage of high-speed high-parity RAIDs without much penalty, excepting available disk bandwidth and usable capacity. Log structured file systems can create I/O workloads that are similar to a streaming pattern without a user's workload conforming to this pattern, particularly for write-heavy workloads [**96**].

5. Conclusions

Disks are difficult to analyze for reliability for a number of reasons, but the end result is the same: When aggregating several disks into an array with inadequate parity, data loss is highly probable. However, the amount of parity necessary and the optimal organization is debated widely. Current highly reliable RAID still maintains the same structure of RAID levels 1, 5, and 6, with hierarchical organizations being used to increase fault tolerance. Unfortunately, based on analysis of these levels, it has been shown that RAID 5+0 is not a significant improvement on RAID 5, and RAID 6+0 is not a large improvement on RAID 6. Further, reasonable ranges of values for array MTTR and disk MTTF can cause the reliability of an array to vary widely.

RAID levels 5, 6, 5+0, and 6+0 also do not support high levels of error correction capability. With RAID 5 and RAID 5+0, it is impossible to determine where errors have occurred within the array beyond which set is affected. RAID 6 exhibits the same behavior when operating in degraded mode. RAID 1+0 can reliably detect errors when there are at least three replicas active simultaneously within a RAID 1 set, so that voting can occur, but incurs a large storage overhead. For applications that require absolute data integrity, such levels cannot provide reassurance against storage stack bugs, firmware bugs, or certain types of hardware failure.

This chapter shows that $k + m$ RAID, a storage organization that improves on current parity-based RAID by allowing arbitrary choice of disk failure tolerance, is a potential solution to all of the identified problems. Read verification, along with error correction, can be implemented with high reliability, while providing orders of magnitude reliability improvement over hierarchical RAID levels that use more storage resources and provide fewer protections.

CHAPTER 4

A GPU-BASED RAID ARCHITECTURE FOR STREAMING WORKLOADS

Currently available GPU technologies, when incorporated into a RAID controller for high-performance computing, impose some constraints on the software architecture. Gibraltar RAID, a prototype GPU-based RAID system, targets a high-performance computing environment that primarily hosts streaming file I/O. This specialization allows for opportunities to simplify and optimize the architecture while addressing specific needs of potential users.

This chapter details the overall system that includes Gibraltar RAID as a component. It further details the major design decisions of Gibraltar RAID given the challenges and opportunities of the workload and system. It also details the architecture of the RAID layer while discussing design decisions.

1. The Software Ecosystem for Gibraltar RAID

While this research effort involves creation of a new piece of software, little software exists in isolation. Gibraltar RAID depends on two main software packages to provide necessary functionality: A GPU computation package, and a storage target. This section describes the use of each and the related consequences.

1.1. The NVIDIA CUDA Toolkit. The Gibraltar Library (detailed in Chapter 5) depends on the capabilities of the NVIDIA CUDA [**75**] toolkit for GPU computations, and must deal its limitations. CUDA is intended to be used by user space applications for accelerating many types of user computations. Computations that are performed within an operating system are not often targeted for GPU acceleration, resulting in a lack of kernel space APIs to access GPU computing resources. There are at least three possible ways to use CUDA within a RAID system. They are, in order from most difficult to least:

(1) Reverse-engineer the CUDA runtime (or eavesdrop on traffic between user space applications and the NVIDIA driver) and provide a kernel space CUDA API;
(2) create a kernel driver that passes data between the block layer and a user space CUDA-enabled daemon; or
(3) create a RAID system within a user space network storage software.

Based on the relative benefit for each development path, the third option was the strategy chosen for Gibraltar RAID. The reasoning relies not only on the relative difficulty of creating a high-quality prototype, but also its future utility. As future accelerators applicable to RAID coding become available, a user space infrastructure will likely prove most beneficial. Any conceivable accelerator intended for mainstream use can be most easily integrated and tested with this strategy. Further, as this prototype is designed to be used with high performance streaming workloads that are observed in large compute clusters, the use of network storage server software on the storage server is likely. If accessing network storage with client software on a loopback interface is efficient, this strategy can also provide DAS for a single workstation.

1.2. The Linux SCSI Target Framework. A target is an entity on a network providing storage. To use the storage offered by a target, client software (the initiator) must interact with the target. In order to follow the third design strategy, target software that includes a user space processing component is necessary. Fortunately, the standard iSCSI [**98**]/ISER [**58**] target for Linux, the Linux SCSI Target Framework [**32**] (stgt), is largely implemented in user space. While stgt does include a Linux kernel module to interact efficiently with network transports, almost all of stgt's operations are performed within a user space daemon (tgtd).

In the standard stgt distribution, the default mode of operation includes opening the backing store, which can be a device or flat file, within tgtd. The iSCSI commands are applied to the backing store with standard system calls like open, pread, pwrite, and so on. To provide the RAID functionality, a software package can provide similar calls to minimize the necessary modifications to tgtd. This is useful, as significant updates to stgt are currently released approximately once per month.

2. Design Characteristics and Implications

There are three main design characteristics that are important to the applicability and performance of Gibraltar RAID. These have important, interrelated implications that must be addressed at the outset. This section details these characteristics and their effects on Gibraltar RAID's design.

2.1. Read Verification. Chapter 3 describes in great detail the unreliable nature of disk systems and hardware. However, many segments of the high performance computing population require the utmost confidence in computations. Ensuring that the data are stored and retrieved reliably is going to become a significant factor in the correctness of results. In order to demonstrate the feasibility of read verification, it is a feature in the Gibraltar RAID system.

There are at least two ways to provide read verification: Block checksums, and parity-based verification. The T10 Data Integrity Field (DIF) [**49**], which is available on SAS and Fibre Channel hard disks, includes a field of eight bytes with every 512-byte block. This field includes many types of information, including a cyclic redundancy check (CRC) to detect bit errors. Performing this type of check in software would require an inordinate number of small memory copies, even with knowledge of the RAID stripe depth. The second, stripe verification, requires no new data to be generated beyond the RAID-required parity chunks. However, in order to verify any information in a RAID stripe, the entire stripe (data and parity) needs to be read from disk. This implies that small reads have a lesser, but analogous, penalty as writes in a parity-based system. For streaming workloads, there is a reduced penalty, as both reads and writes tend to be large and contiguous.

2.2. Asynchronous/Overlapping Operation. Asynchronous I/O, which allows the Linux kernel to manage read and write requests in the background, is sensible for storage-intensive applications. Initially, Gibraltar RAID used threads to perform synchronous reads and writes with one thread assigned per disk. Switching to asynchronous reads and writes allowed for more efficient use of resources than CPU-intensive pthread condition variables with a high thread-to-core ratio allow. While asynchronous I/O has been implemented in the Linux kernel and C libraries for some time, the methods for performing asynchronous vector I/O are not well-documented.

The benefits of using asynchronous I/O are compelling: Using a single thread to perform I/O is easier to manage (and debug) than using a team of threads. Further, fewer system calls must be made to file an I/O operation for a stripe. There is a significant disadvantage: Linux asynchronous I/O only works with devices opened with the O_DIRECT flag.

2.3. O_DIRECT and the Gibraltar Library Throughput. When specified as a flag for the Linux open system call, O_DIRECT instructs the I/O subsystem to bypass the Linux buffer cache and perform I/O directly on user space buffers. This is desirable for an application that manages large caches itself. Specifically, RAID software that verifies reads must be aware of the location of data at all times. If data received from a read call comes from the buffer cache, and has already been read by the application, the RAID software does not need to reverify the contents returned. Unfortunately, this information is not available when requesting data from the buffer cache.

O_DIRECT's implementation conflicts with the implementation of CUDA's memory allocation functions, which can permanently pin buffers for PCI-Express transfers without data copies. When attempting to perform direct I/O on a CUDA-allocated memory region, the system call will return an error and not complete. The most efficient usage of Gibraltar involves memory mapping host memory into the GPU's address space, which requires using the CUDA memory allocator. Since this mapping can not be done, that coding is much less efficient within Gibraltar RAID than it could be, resulting in a throughput of approximately 900 MB/s in the test system outlined in Appendix C. The resulting requirement is that the architecture must be pipelined to ensure good performance when the disk controller is capable of a similar throughput as the GPU. While not attempted, straightforward modifications to the Linux kernel would rectify this shortcoming.

3. Gibraltar RAID Software Architecture

The interface between Gibraltar RAID and the target is general, allowing Gibraltar RAID to be used with other network storage packages that have user space operation. Further, software that manages its own raw storage may use Gibraltar RAID directly. However, Gibraltar RAID cannot take advantage of the facilities for I/O within the operating system kernel, so many portions of the underlying secondary storage stack had to be remade and integrated within Gibraltar RAID in user space. This section details the mechanics of the components that have been provided, and how the components communicate.

Figure 12 has been provided to aid with the description following. Different data paths are outlined as follows: Writes are denoted with a “w,” reads with an “r,” and victimization with a “v.” Each interaction is noted with a letter indicating the data path and a number indicating the order in which the interactions occur.

3.1. Interface. One goal of integration into the Linux SCSI Target Framework was to keep the user interaction with the running target the same because the target was deployed in a production environment. This necessitates completely changing the operation of the target to remove support for using regular devices as storage resources, as a new command would be required to manage virtual devices. Instead, this target is intended to be used solely as a GPU-based RAID server.

The only new command creates a new Gibraltar RAID device with an arbitrary number of logical units, storing the settings in several files, one per logical unit. Each

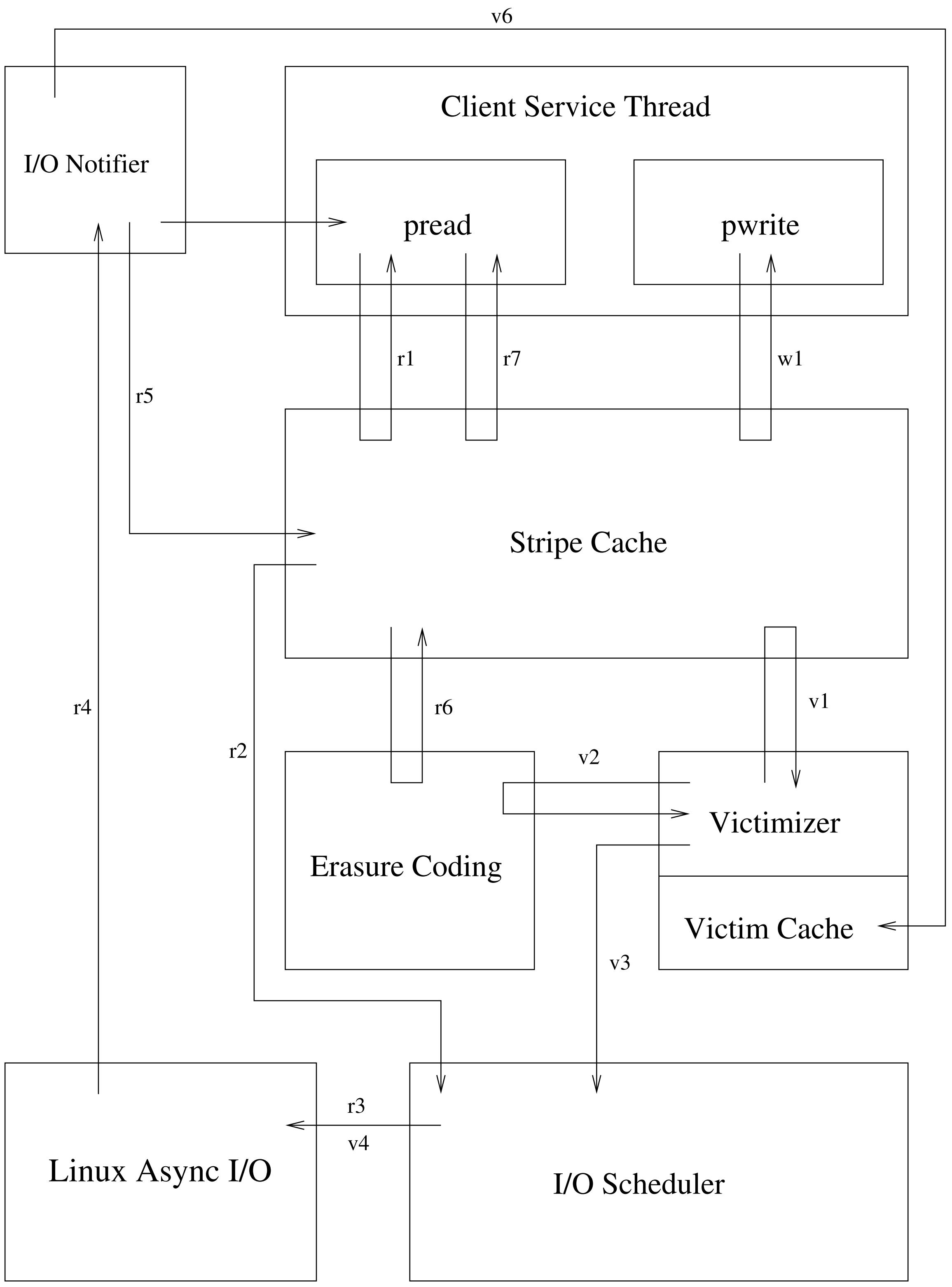


FIGURE 12. Gibraltar RAID Architecture and Data Flow Diagram

file can be used to add and remove storage resources from the target independently in the same manner used to add and remove storage devices. Each logical unit file contains configuration information like the SCSI identifier of the member drives, array status, ranges of bytes included in the logical unit, and so on. One benefit of this arrangement, in contrast to Linux md, is that Gibraltar RAID can host a large number of file systems within a single RAID device, exposed as multiple logical units.

Other possible contents of logical unit files include bitmaps that contain information on initialization status of stripes and rebuild progress. The initialization information is useful at array creation, eliminating the requirement of generating parity for unused portions of the disk. When creating an array with Linux md, for example, a lengthy syncing process runs in the background until all parity has been generated, even though the array generally contains no data that requires fault tolerance. If initialization data are stored, parity can be generated for data as they are stored. Similarly, rebuild progress can be stored to allow array rebuilding to be interrupted without losing progress.

All interaction between the target and Gibraltar RAID occurs through functions following the interfaces for the standard C library calls `pread()` and `pwrite()`. Each function takes a pointer to a user buffer, an offset into the RAID device, and the length of the desired data to be copied into the user buffer. Each request is mapped by these functions to the stripes affected, and asynchronous requests for those stripes are submitted to the Gibraltar RAID cache. These requests are filled without knowledge of whether the stripes are present in the cache, so the stripe can be requested in one of two ways:

- Request stripe with its full, verified contents; or
- Request a "clean" stripe, which may return an uninitialized stripe and does not incur disk I/O.

The second option is useful if the stripe is to be completely overwritten by a write request, or if it is expected to be overwritten. Client service threads, threads created by the target to satisfy read and write requests from network clients, call `pread` and `pwrite`.

For reads, additional stripes may also be requested in order to provide read ahead capability that takes advantage of spatial locality of disk accesses. In the case of streaming reads, there is a high probability that a read for a segment of data will be quickly followed by a request for the next. The Linux kernel buffer cache does this; read ahead is necessary to get the best performance out of a storage system under streaming workloads.

After all read or write requests have been registered, the client service thread waits for the requests to be fulfilled. In the case of a read, the routine passes each stripe to the erasure coding component to be verified or recovered. Writes can be recorded as incomplete updates to a stripe, anticipating that the whole stripe will eventually be overwritten. If this does not happen before a stripe is chosen for victimization, the stripe must be read, verified, modified with the contents of the incomplete stripe, updated with regenerated parity, and written.

3.2. Stripe Cache. Gibraltar RAID includes a stripe cache that operates asynchronously with the I/O thread and the user's requests for reads and writes. In the event of a read request to the cache, the cache submits requests to the I/O thread to read the relevant stripes from disk in their entirety, including parity chunks for read verification. This full-stripe operation can be viewed as a slight mandatory read ahead.

Writes have a simple implementation in the stripe cache, as they do not require immediate disk operations. The cache is optimistic; if the write request does not fill an entire stripe, the cache assumes that the stripe will be completely populated before being flushed to disk. Therefore, no reads are required before a partial update to a stripe is made. If a stripe is in the process of being read, the cache will force the client thread to wait until the read has been completed. If the stripe has already been previously read, it will be simply given to the client thread to be updated with the write contents. Otherwise, a blank stripe is returned, and the client is responsible for maintaining the clean/dirty statistics related to the writes requested.

When the cache is sufficiently full, the cache will start deleting clean stripes and filing operations to flush dirty stripes. The interface to accomplish this is flexible, allowing many different types of caching algorithm to be used. The default mode of operation is the Least Recently Used (LRU) caching algorithm, but higher quality algorithms may be used. Furthermore, the victim selection routine can have an internal timer used to facilitate write-behind caching.

3.3. I/O Scheduler. The scheduler receives requests from the cache for reading stripes, and receives requests from the victimization routine for writing stripes. The scheduler accumulates these requests in a queue until it is ready to service them. All of the requests are received as a batch to facilitate combining of requests that are adjacent on disk. Only write requests are combined for the following reasons:

- Clients are affected by latency if they have to wait longer for a read request to be serviced. Reads are necessarily synchronous, so a large combined read request will force all clients to wait until the entire combined request is serviced.
- Our experiments show that performing short contiguous reads easily obtains good performance from a disk. Writes, however, can be significantly slower (depending on the qualities of the system, such as hardware and configuration) unless a comparatively large amount of data are accumulated for writing at once. Figure 13 demonstrates this property. Notice that contiguous writes of 16 megabytes are slightly slower than contiguous reads of 64 kilobytes. This performance degradation is only apparent for files or devices opened with the O_DIRECT flag, which bypasses the Linux kernel buffer cache. For normal disk operation, write requests are combined in the buffer cache, hiding this potential performance issue.

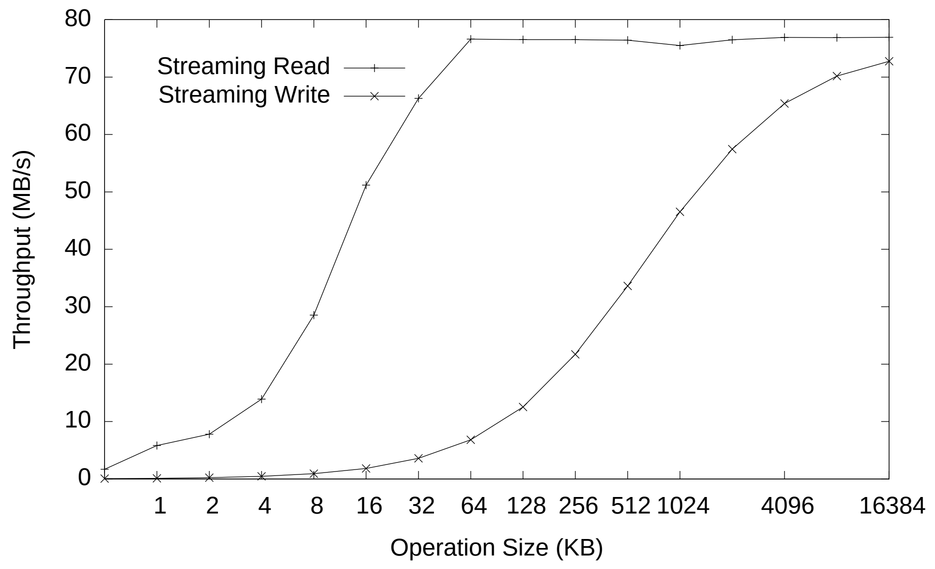


Figure 13. Performance of a Single Disk in a RS-1600-F4-SBD Switched Enclosure over 4Gbps Fibre Channel

The scheduler receives all waiting requests as a batch, ordering and combining them as necessary to achieve the highest possible disk bandwidth. The ordering algorithm currently used is the circular elevator algorithm (C-SCAN).

Combining requests is conceptually simple. If there are two writes that are adjacent on disk, it is possible to use a vector write to service the requests in a single write call. One implementation of this type of call available to applications is `pwritev()`, a vector version of `pwrite()`. Vectored operations allow a contiguous area of a disk to be the target for a combined write operation from non-contiguous areas of memory. However, using `pwritev()` is not the best strategy for a RAID system.

Asynchronous I/O, which allows the Linux kernel to manage read and write requests in the background, is more sensible for storage-intensive applications than syncronous I/O. Initially, Gibraltar RAID used the pthreads library to perform synchronous reads and writes with one thread assigned per disk. Switching to asynchronous reads and writes allowed for more efficient use of resources than CPU-intensive pthread condition variables with a high thread-to-core ratio allow. While asynchronous I/O has been implemented in the Linux kernel and C libraries for some time, the methods for performing asynchronous vector I/O are not well-documented.

There is a method of using `io_submit()` to submit `iovec` structures in an unconventional and difficult-to-discover way. The typical usage of `io_submit()` takes an array of `iocb` structures as a parameter, which describes individual I/O operations to submit asynchronously. However, to use the relatively new vectored read and write capabilities, one passes an array of `iov` structures within the `iocb` structure instead of `iocb_common` structures. These will be used to perform a vectored I/O operation. This is not noted in system documentation.

3.4. I/O Notifier. The I/O Notifier is a thread that collects events resulting from the asynchronous I/O calls, and performs record-keeping on a per-stripe basis. Once all of the asynchronous I/O calls for a stripe have been completed, the notifier notifies other threads that depend on the stripe associated with the I/O. If the stripe is undergoing eviction from the cache, this thread will initiate destruction of the stripe at I/O completion.

3.5. Victim Cache. When considering the speed that new I/O requests can arrive at the RAID controller, there is a significant delay between the decision to victimize a dirty stripe and the completion of the write associated with it. Canceling a write in progress is an inefficient action because of the combining of writes and the asynchronous completion of the writes. To aid in victimization, a victim cache is included that allows for a client read or write request to "rescue" a stripe from being deleted before it has been written, or even while the write is still in progress.

3.6. Erasure Coding. The erasure coding component uses a the Gibraltar library, which was designed to perform Reed-Solomon encoding and decoding at high rates using GPUs [**26**]. Briefly, the Gibraltar library operates by accepting k buffers of data, such as striped data for a RAID array, and returns m buffers of parity. This GPU-based parity calculation can encode and decode at speeds of well over four gigabytes per second for RAID 6 workloads on commodity GPUs. A unique feature of Gibraltar RAID is the ability to extend far beyond RAID 6 in the number of disks that may fail without data loss. Chapter 5 details the Gibraltar library.

GPU parity calculations entail transfer of significant amounts of data across the PCI-Express bus to the GPU. This implies that using the Gibraltar library in this system also incurs significant PCI-Express traffic. This traffic can be a significant concern if other hardware, like network adapters and host bus adapters, also heavily uses the PCI-Express bus.

4. Conclusions

This chapter details a RAID system designed specifically for streaming workloads. An emphasis is placed on asynchronous operation, allowing overlap between many portions of the system. For example, a decision was made to trade raw coding throughput for a more heavily pipelined architecture, resulting in a match between storage speed and GPU speed for likely configurations. Obviously, this is not a perfect architecture for all workloads and systems.

Systems with streaming I/O workloads are assumed to require several megabytes of data from a contiguous storage area. This architecture is tuned for this type of workload in several ways. Most notably, read verification requires the contents of an entire stripe from the RAID system, regardless of the number of bytes being read from that stripe. An alternative method of read verification may require only a certain amount of data per disk, which introduces some storage overhead and memory copies for reads and writes. However, such methods are likely to perform better for I/O patterns that require small reads and writes.

The benefit of this architecture is that I/O rates can easily match or exceed the speed of many high-performance SATA, SAS, or Fibre Channel controllers while satisfying increased reliability constraints. Further, future improvements to many areas can be prototyped with this controller, including trials of new caching algorithms, coding devices, and disk systems. It is a user space application, so it requires few changes to be runnable on a wide variety of Linux systems and kernels, provided that they do not deprecate the current asynchronous I/O API. A software system that follows this architecture will be useful for many significant research activities.

CHAPTER 5

GIBRALTAR

1. Introduction

Each byte that is read from or written to a RAID system must often be subjected to processing by an erasure correcting code to ensure reliability of the data. Poor performance of the erasure correcting code can limit the scale of a RAID system. Since this work is intended to create a high-speed, scalable RAID system, the requirements of the erasure correcting code are two-fold:

(1) The code must be able to provide arbitrary dimensions of erasure correction; namely, a wide range of $k + m$ coding configurations must be supported.
(2) The code's performance should be capable of supporting the full throughput of the storage resources for many configurations

In an extended RAID system, by introducing m appropriately distributed parity chunks (created with a maximum-distance separable code [**91**]) per k data chunks in a disk array stripe, an array can withstand the total failure of up to m disk drives. RAID levels 5 and 6 are two common RAID levels that are capable of $m = 1$ and $m = 2$, respectively. While $m = 1$ can be implemented with $k - 1$ exclusive-or operations, $m \geq 2$ involves a more complex calculation [**22**].

One standard method of generating more than one parity chunk is Reed-Solomon coding [**92**]. Many RAID 6-specific codes exist [**10, 22, 88**], but Reed-Solomon is often used today in RAID 6 systems, including the Linux kernel's software RAID [**5**]. While RAID 6 only requires two coded chunks, Reed-Solomon coding can be used to generate an arbitrary number of independent coded chunks, allowing redundancy of a system to scale with its size. This scaling capability is not present in many RAID 6 algorithms. Unfortunately, most conventional commodity processors are not well-suited architecturally to Reed-Solomon coding for more than two parity chunks because of required table look-up operations [**8**]. This quality results in slow software RAID when $m > 2$, necessitating alternative hardware solutions.

Reed-Solomon coding for parity computation is generally performed by hardware RAID controllers that are positioned in the data path between the computer and its storage resources. The controller can present the multiple attached storage resources as a single block device to the operating system, resulting in transparency and performance. However, a RAID controller is limited in the tasks it can accomplish. Relocating the Reed-Solomon coding to a more general purpose piece of hardware results in new flexibility. Any application that can benefit from Reed-Solomon coding would be able to offload this computation, increasing the performance of that portion of the application, either by increasing the speed of the task or overlapping it with a different computation.

Graphics Processing Units, also known as GPUs, are highly parallel devices designed to exploit the embarrassingly parallel nature of graphics rendering tasks [**31, 77**]. As conventional CPUs have transitioned through single-core, multi-thread, and multi-core devices, the anticipation is high that CPUs will become many-core devices in order to keep up with the demands of Moore's Law while mitigating increasing power consumption [**46**]. Application developers have been considering the GPU as a currently mature and highly developed computational platform with hundreds of cores. GPUs have been successfully applied to applications that are either embarrassingly parallel or have embarrassingly parallel sub-steps [**19, 33**].

Until recently, however, GPU platforms were restricted in terms of usable data types. Generally, only certain floating-point operations and data types were well supported [**83**]. Some other types could be emulated through the provided types; however, unless used judiciously, emulation often proved to be an inefficient use of the GPU's resources [**38**]. With the native types available, only applications that heavily use floating-point data and calculations could be meaningfully accelerated via GPU computation.

In order to provide acceleration for more general-purpose GPU (GPGPU) applications, NVIDIA has released its CUDA API and architecture [**75**], and ATI/AMD has released a stream computing software stack that potentially allows many high-level languages to be used for programming ATI GPUs [**2, 3**]. Both ATI and NVIDIA technology now allow operations to be performed on arbitrary binary data using their GPUs and software. This presents the opportunity for applications to perform more general data processing tasks that are well-suited to the GPU's overall architecture, but need different kinds of computation than floating point units can provide.

This chapter identifies Reed-Solomon coding as an application that both suits the general architecture of GPUs and requires the added primitive types and operations available through CUDA. In particular, results presented previously demonstrated that GPUs could show superior performance as part of a software RAID system that includes more than two parity disks by achieving an overall ten-fold speedup over a CPU implementation where $k = 3$ and $m = 3$ and beyond [**27, 28**].

This chapter describes generalized Reed-Solomon coding on programmable GPU hardware, specifically for use with applications requiring data resiliency in a manner similar to RAID. This chapter begins by describing Reed-Solomon coding in more detail within the context of RAID and distributed data applications. It goes on to detail the mapping of Reed-Solomon coding to NVIDIA GPUs. Results for performance are given, including comparison with a well-known CPU library which also implements the Reed-Solomon coding as used in this work [**89**]. Future potential trends for GPU and other highly multi-core devices are described, along with conclusions. A publicly available prototype library, Gibraltar, demonstrates the findings of this work. In order to demonstrate its practicality, a usage example and design rationale are provided.

2. Reed-Solomon Coding for RAID

The primary operation in Reed-Solomon coding is the multiplication of F, the lower m rows of an information dispersal matrix $A = \begin{bmatrix} I \\ F \end{bmatrix}$, with a vector of data elements d [**87**].

$$\begin{bmatrix} I \\ F \end{bmatrix} d = \begin{bmatrix} d \\ c \end{bmatrix} \tag{12}$$

This yields another vector of redundant elements (the coding vector, c). The redundancy of the operation comes from the over-qualification of the system: Any k elements of $e = \begin{bmatrix} d \\ c \end{bmatrix}$ may be used to recover d, even if some (or all) elements of d are not available. A more in-depth discussion is provided in the next section.

Rather than relying on integer or floating point arithmetic, the operations are performed on members of a finite field [**62**]. Addition of two numbers is implemented through an exclusive-or operation, while multiplication by two is implemented through a linear feedback shift register (LFSR) [**5**]. Multiplying two arbitrary numbers involves decomposing the problem into addition of products involving powers of two, which potentially requires a large number of operations. One useful identity that holds true in finite fields of size 2^w (where w is the number of bits per symbol) is as follows:

$$x \times y = \exp(\log(x) + \log(y)) \tag{13}$$

where the addition operator denotes normal integer addition modulo $2^w - 1$, while exp() and log() are, respectively, exponentiation and logarithm operations in the finite field using a common base [**87**]. Since $w = 8$ for RAID systems, an implementation can contain pre-calculated tables for the exp and log operations, which are each 256 bytes. Multiplication can be implemented using these tables with three table look-ups and addition modulo $2^w - 1$ instead of potentially many more logical operations. Being that decoding (recovering missing data elements from k remaining data and/or coding elements) is a similar operation, all of the above holds true for decoding as well.

Unfortunately, the type of table look-up operations used in Reed-Solomon coding does not exploit the internal vector-based parallelism of CPUs. While fast vector instructions have been included in modern CPUs, few CPU models include a parallel table look-up instruction. In the case of IBM's Power architecture, whose AltiVec instruction set includes a parallel table look-up instruction, multiplication in finite fields has been accelerated over typical CPU implementations [**8**]. Unfortunately, this capability is not common, and CPU implementations tend to be unusable for high-speed applications because of this lack of capability.

A more in-depth treatment of implementation details of Reed-Solomon coding is available [**87**].

3. Mapping Reed-Solomon Coding to GPUs

GPUs are architecturally quite different from CPUs. The emphasis of GPU architecture is to accomplish hundreds of millions of small, independent, and memory intensive computations per second in order to provide interactive graphics to its user. As such, GPUs have several interesting qualities that are directly applicable to the task of Reed-Solomon coding.

In this discussion, a buffer is a particular slice of data of s bytes. A $k + m$ coding would require k data buffers, and m coding buffers. These buffers are typically stored together, one after another, within a continuous memory region that is referred to as the buffer space. One may refer to the i^{th} buffer in the buffer space, which specifically refers to the bytes between $i \times s \ldots (i+1) \times s - 1$ within the buffer space. However, when not explicitly qualified, the i^{th} buffer indicates its contents rather than its position. For the first k buffers, buffer 0 contains the first s bytes of data, buffer 1 contains the next s bytes of data, and so on. The following m buffers contain the coding data.

3.1. GPU Architecture. One of the more well-known features of a GPU is its vast number of multi-threaded processing cores. The NVIDIA GeForce 8800 GTX, for example, features 128 cores, while the GeForce 285 GTX contains 240 [**76**]. According to NVIDIA, these cores are designed to be effective for many threads of execution as small as a few instructions. In the context of parity generation, each set of bytes in the data stream (i.e., byte b of each buffer) can be modeled accurately as an independent computation. The threading implementation allows for hiding of memory access latency if the ratio of instructions to memory accesses proves high, so it is beneficial to pack as many bytes per memory request as possible. In the approach presented here, each thread is responsible for four bytes per buffer; this takes advantage of the relatively wide 384-bit memory bus, which allows multiple threads to transfer all 32 bits each in parallel. It is unnecessary to load each thread with a significant portion of the workload to achieve good performance, even though this may result in many thousands of threads. Therefore, each thread can remain relatively small, and the GPU's scheduling of the threads can efficiently accomplish the work of creating parity for many megabytes of data at a time.

The cores are split into units called multiprocessors that contain a shared memory space along with eight cores. This shared memory space is banked, allowing up to 16 parallel accesses to occur at once, and is as fast as registers for each core. Because of its speed, bank conflicts can adversely affect performance. Given the previously mentioned 30 multiprocessors of the GeForce GTX 285, each core within a warp (a group of co-scheduled threads, 32 threads per 8-core multiprocessor) can access a separate bank of memory, allowing up to 240 simultaneous table look-up operations. Unfortunately, it is difficult to manage the tables in each shared memory block to eliminate conflicts, and the birthday paradox [**29**] dictates that the probability of conflicts are high. On average, however, simulations of this application's workload has shown that there are about 4.4 accesses per bank in order to satisfy the table look-up operations for 32 simultaneous threads. On the NVIDIA GTX 285, each access consumes four clock cycles. This implies that 17.6 clock cycles are required to satisfy 32 requests, yielding 1.82 requests per cycle per shared memory unit, or 55 requests per cycle across the chip[1].

Another interesting hardware feature benefiting Reed-Solomon coding is the support for constant memory values. As an architecture that deals primarily in accessing read-only textures for mapping onto polygons, GPUs require fast constant accesses in order to provide high graphics performance [2]. Unlike other types of memory within the CUDA architecture, constant memory is immutable. Therefore, in order to increase performance of constant memory accesses, each multiprocessor's constant accesses are cached. Once data is loaded into this cache, accesses to this data are as fast as register accesses. With this in mind, the constant memory is a prime location for the information dispersal matrix. The matrices are small, and each element is accessed in lock-step across all cores within a multiprocessor. The constant memory allows all requests to each element of the matrix to be cached and simultaneously fulfilled for all cores as if the values were in registers.

3.2. Reed-Solomon Decoding. One major contribution of this work is the improvement over previous GPU-based decoding performance [**28**]. The coding logic presented in this work is already highly optimized. It includes the minimum number of memory accesses, and has high performance. From the perspective of RAID, recovery should be as fast (or faster, depending on the number of failed disks attached to the array) so that performance of the server is not degraded upon failure of a disk.

In order to ensure this performance requirement, one of the design goals was to make the logic of coding and decoding on the GPU as similar as possible. Similarities already exist within the coding and decoding algorithms. In coding tasks, all coded buffers are generated via vector-matrix multiplication. The data vector of size k is multiplied with the information dispersal matrix, of size $m \times k$, yielding a parity vector

[1]This is an imprecise figure because the interactions of memory requests between threads running at different times are complex, but this is a reasonable estimate.

[2]This fact is derived from the emphasis on fill rates in marketing materials for consumer GPUs. The fill rate is a the theoretical limit of how fast a GPU can apply a texture to a polygon, measured in pixels per second.

c of size m. However, the problem initially is viewed as a larger problem of multiplying d with A to yield e. The redundancy comes from the overdetermined quality of the system, allowing any m entries to be removed from e, along with corresponding rows of F.

In implementing decoding, there is a design decision to be made about whether buffers must remain ordered, the user of the routine may be allowed to choose arbitrary orderings of buffers, or some combination of the two. By imposing a partial ordering on the buffers, the library can statically determine how calculations should be performed without indirect indexing into arrays, as long as the generation matrix is adjusted to reflect buffer orderings. In Gibraltar, a buffer ordering where all present buffers are listed at the beginning of the buffer space is required. Decoded buffer contents are placed after the original buffers within the buffer space. There are two reasons that justify imposing this alternative partial order:

- Contents of the buffers for decoding are not typically resident in the memory of the host performing the decoding. Instead, they are spread over devices (or another host) that become unavailable because of failure or fault. Therefore, a transfer must be incurred between the host and device, and the host is free to place the contents of the transfer anywhere in its memory.
- If k is significantly greater than m, there are simple manners of changing the order of the buffers to satisfy the layout requirement quickly in CPU memory. If $1 \leq q \leq m$ buffers have failed, only q buffers must be moved within the buffer space.

The following example demonstrates the desired reordering requirements, along with a description of the quick reordering method. Once again, this background information can be found in several sources, including [**87**].

In an example $k = 4$, $m = 4$ coding system, codes are generated in a straightforward way. An overdetermined information dispersal matrix A is generated. In order to generate the parity vector c, one multiplies the lower m rows of A (usually denoted as

F) by the data vector d. The memory layout for such an operation is simple, as each buffer is arranged linearly in memory. (See the API discussion for more information.) If each buffer is of length s, it is known at compile time where the input bytes and output bytes are for a given step, so the generation routine can be highly efficient. Memory accesses are minimal, as indices are simply computed. This simplicity allows many optimizations such as unrolling of loops within the kernel by the compiler.

However, there is some variability in how a recovery can be accomplished. Suppose that devices 0, 2, 3, and 6 fail. The state is now represented by the following equation, with unavailable elements denoted with a question mark.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ F_{0,0} & F_{0,1} & F_{0,2} & F_{0,3} \\ F_{1,0} & F_{1,1} & F_{1,2} & F_{1,3} \\ F_{2,0} & F_{2,1} & F_{2,2} & F_{2,3} \\ F_{3,0} & F_{3,1} & F_{3,2} & F_{3,3} \end{pmatrix} \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \\ c_0 \\ c_1? \\ c_2 \\ c_3 \end{pmatrix} \tag{14}$$

The goal of the recovery operation is to solve for missing portions of d with the present portions of e, which includes elements of d and c. By the means of construction, A has full rank, so rows may be eliminated without losing information until only k rows remain. Doing so requires removing the corresponding elements of e in order to maintain the correctness of the equation. The vector e and A are modified (creating e' and A', respectively) to preserve the equality, yet reflect in e' only the elements that have not been lost.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ F_{0,0} & F_{0,1} & F_{0,2} & F_{0,3} \\ F_{2,0} & F_{2,1} & F_{2,2} & F_{2,3} \\ F_{3,0} & F_{3,1} & F_{3,2} & F_{3,3} \end{pmatrix} \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = \begin{pmatrix} d_1 \\ c_0 \\ c_2 \\ c_3 \end{pmatrix} \tag{15}$$

Pre-multiplying each side by A'^{-1} will yield the desired configuration that will yield the missing data elements with only the known data elements:

$$\begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ F_{0,0} & F_{0,1} & F_{0,2} & F_{0,3} \\ F_{2,0} & F_{2,1} & F_{2,2} & F_{2,3} \\ F_{3,0} & F_{3,1} & F_{3,2} & F_{3,3} \end{pmatrix}^{-1} \begin{pmatrix} d_1 \\ c_0 \\ c_2 \\ c_3 \end{pmatrix} \tag{16}$$

As specified in [**87**], the properties of Galois field arithmetic and the methods of generation for A will ensure that these equalities hold, A' is invertible, and all elements generated are within the bounds specified by the domain of the code (e.g., eight-bit values).

One obvious optimization would be to not explicitly perform any of the row multiplications for elements that are already present in d. Modifying the algorithm to just skip these elements requires a vector that indicates the elements that need recomputing. In particular, in order to accomplish the decoding task on a GPU, this vector must be copied into every thread block, and referenced heavily via indirect indexing.

Instead, a new algorithm is used to eliminate state vectors or indirect indexing. One rearranges rows so that data elements still present in e are in the lower portion of the matrix, then ignore the lower rows when performing the multiplication. This change results in a slight performance impact on the CPU by introducing work that was not required previously. However, the problem now more closely resembles a generation problem, with no need for indirect indexing on the GPU. The previous example is now recast into the following equation.

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{17}$$

$$B \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = BA'^{-1} \begin{pmatrix} d_1 \\ c_0 \\ c_2 \\ c_3 \end{pmatrix} \tag{18}$$

In this formulation, all the information that is required in order to work optimally is the quantity of data buffers that need recovery. There will always be a need for k data or parity elements to recover the missing data elements, so buffers can be statically arranged such that k intact buffers are at the beginning of the memory region, and the remaining buffers requiring recovery can be output in the next portion of the buffer region.

4. Operational Example and Description

While Gibraltar does make use of GPUs to accomplish the vast majority of the computations required for Reed-Solomon coding, it is not necessary to use a GPU to implement all operations. Many of the operations which are performed require insignificant amounts of time to execute on a CPU compared to the coding task. With the algorithmic enhancements to minimize memory accesses, the GPU is only required to perform large numbers of small matrix-vector multiplications. All other matrix algebra, including the necessary factorizations and inversions, occur within the CPU.

An Example Program. In order to illustrate the way which Gibraltar can be used, this section will present an example program. Its operation is simple: Perform a coding operation to yield parity, erase some buffers, and decode the remaining buffers to yield the original data.

Initializing Gibraltar. A context is used to manage Gibraltar's operation. Different kernels are required to perform different coding tasks, and one application can reasonably use multiple $k + m$ coding schemes simultaneously. To satisfy this requirement, a context is referenced in each call which identifies which compiled kernels to execute when performing tasks.

While Gibraltar does make use of compile-time knowledge in order to allow static optimization of the kernels, the user executable is not required to manage the configuration of Gibraltar. Initialization of a context will compile kernels on demand for the appropriate matrix-vector operations. Such kernels are cached for later initializations with the same parameters. This compilation step causes the first use of a particular encoding to incur a delay.

```
int k = 16, m = 3;
gib_context gc;
gib_init(k, m, &gc);
```

After executing the above listing, the handle `gc` can now be used for coding operations.

In order to code and decode buffers, a buffer space must be allocated. To perform a $k + m$ encoding on buffers of size s, a region of size $(k + m) \times s$ must be allocated. Gibraltar includes an allocation function which can improve the performance of the GPU functions by introducing an optional stride between buffers and using specialized memory allocation routines. Such allocation routines can prepare buffers for high-speed transfers over the PCI-Express bus.

Since Gibraltar, via its context, has already obtained the value of $k + m$, it is only necessary to provide s. In this example, Gibraltar will be used to manage an array of integers.

```
int s = 1024 * sizeof(int);
int *buf;
// Allocate an array of (k+m)*s integers for coding
gib_alloc((void **)(&buf), s, &s, gc);
```

At this point in the program, the user can fill the buffer with the data which requires coding.

Coding. Once again, this call requires no explicit mention of k or m. The coding function call is called `gib_generate`, as it is simplest to view the coding task as generating parity data from original data. Conceptually, this is a simple call. The initialization routine had already calculated F, so the only task of this function is to manage the GPU.

```
gib_generate(buf, s, gc);
```

Losing Data. Any interesting use of Gibraltar involves unavailability of data. This example will assume that an integer array, called `fail_config` of size $k + m$, contains a zero for a buffer that is available or a one for a buffer that is not available. At most m entries in `fail_config` may be set to one.

Decoding. Technically, the only requirements for buffer positioning for the decoding routine are:

(1) The buffers indicated in the list of buffer assignments within the first k entries must be available and located in the same order at the beginning of the buffer space, and
(2) the only buffers indicated for recovery should be data buffers.

However, in order to attain the highest performance when the application requires buffers to be in order, some previously mentioned layout requirements must be followed. If buffer $0 \leq i < k$, which is a data buffer, is available, it should be positioned in the i^{th} buffer position within the buffer space. The remaining spaces can be filled with available coding buffers. Upon completion of the routine, the user can move the contents of the recovered buffers, which are located after initial k available buffers within the buffer space, into proper positions.

Note: The function is called `gib_recover`, as this routine recovers lost data from available data.

```
int good_buffers[256]; /* k of these are needed. */
int bad_buffers[256]; /* Up to m of these can be used */
int ngood = 0;
int nbad = 0;
for (int i = 0; i < gc->k + gc->m; i++) {
  if (fail_config[i] == 0)
    good_buffers[ngood++] = i;
  else if (i < gc->k) {
    bad_buffers[nbad++] = i;
  }
}

/* Reshuffle to prevent extraneous memory copies later */
for (int i = 0; i < ngood; i++) {
  if (good_buffers[i] != i && good_buffers[i] < gc->k) {
    int j = i+1;
    while(good_buffers[j] < gc->k)
      j++;
    int tmp = good_buffers[j];
    memmove(good_buffers+i+1, good_buffers+i,
            sizeof(int)*(j-i));
    good_buffers[i] = tmp;
  }
}

/* Perform memory copies */
for (int i = 0; i < gc->k; i++)
  if (good_buffers[i] != i)
    memcpy(buf + s*i, buf + s*good_buffers[i], s);

int buf_ids[256];
memcpy(buf_ids, good_buffers, gc->k*sizeof(int));
memcpy(buf_ids+gc->k, bad_buffers, nbad*sizeof(int));
gib_recover(buf, s, buf_ids, nbad, gc);

/* Replace buffers */
for (int i = 0; i < gc->k; i++) {
  if (buf_ids[i] != i) {
    int j = i+1;
    while (buf_ids[j] != i) j++;
    memcpy(buf + s*i, buf + s*j, s);
    buf_ids[i] = i;
  }
}
```

At the termination of this portion of the program, the first k buffers contain their original contents. Appendix A contains a complete description of the Gibraltar API.

5. Performance Results

An experimental program was constructed in order to measure the performance of Gibraltar and Jerasure. It encodes a set of data at varying values of $k+m$ with each buffer occupying one megabyte of memory. The program erases $\min(k, m)$ random data buffers, then recovers their original contents. The same operations are performed using Jerasure [**88**], a well-known library implementing many error correcting codes including Vandermonde-based Reed-Solomon. The results are reported from the perspective of throughput, as the results represent the idea that this library implements a filter for ensuring reliability or recovery of data intended for other purposes. An example case would be that coding is being performed on ten one-megabyte buffers in a $k = 6, m = 4$ configuration. A user does not interact with parity, so throughput is calculated by dividing the size of the data buffers (six megabytes) by the time required to encode them. Similarly, the throughput for recovery is calculated to be the size of the data buffers (six megabytes) divided by the time required to return recovered data.

It is important to note that recovery operations in Gibraltar are only intended to recover data buffers, and that coding buffers should be recovered with a subsequent call to a generation routine. This is different from Jerasure, as the coding buffers are recovered automatically if missing, even when the coding buffers are not needed or devices to store the coding buffers are unavailable. In order to ensure fair benchmarking, only data buffers are erased. Jerasure does check to ensure that no coding buffers require recovery, but no recovery must be done. The CPU operations are performed using an unaltered version of the latest available Jerasure, version 1.2.

The machine used in this test is the storage server that is fully detailed in Appendix C.

Given that the focus of Gibraltar is integration into a software RAID system, it is illustrative to compare the bandwidth achieved by Gibraltar to that of a modern disk. Given recent benchmarks of solid state drives and more conventional disks [**72**], 100 megabytes per second is a representative average bandwidth. With the methodology of

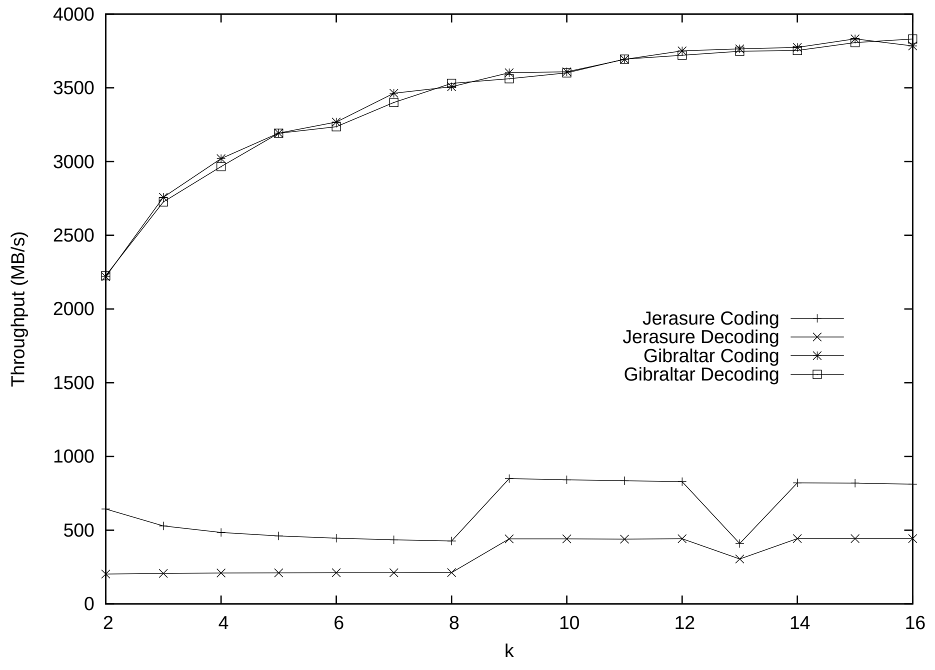


FIGURE 14. Performance for $m = 2$ Encoding and Decoding

measuring the system throughput from a user application point of view (i.e., rate that actual data is transmitted through the system, which does not include the overhead of transferring coded data), the performance provided by Gibraltar can be directly compared to the performance of data disks in a RAID set. From this perspective, it is clear from Figure 16 that Gibraltar can provide enough performance to support more than two dozen disks at extremely high levels of reliability, with enough fault tolerance to withstand failure of any four disks in the array. This is quite different from hierarchical RAID levels such as RAID 6+0, which can combine two or more RAID 6 sets into an outer RAID 0 volume. RAID 6+0 can allow up to four failures, but not allow any four arbitrary disks to fail. Such failures must be limited to two per inner RAID 6 volume. Figure 14 shows that Gibraltar can support RAID 6 arrays at full streaming bandwidth for at least three dozen data disks.

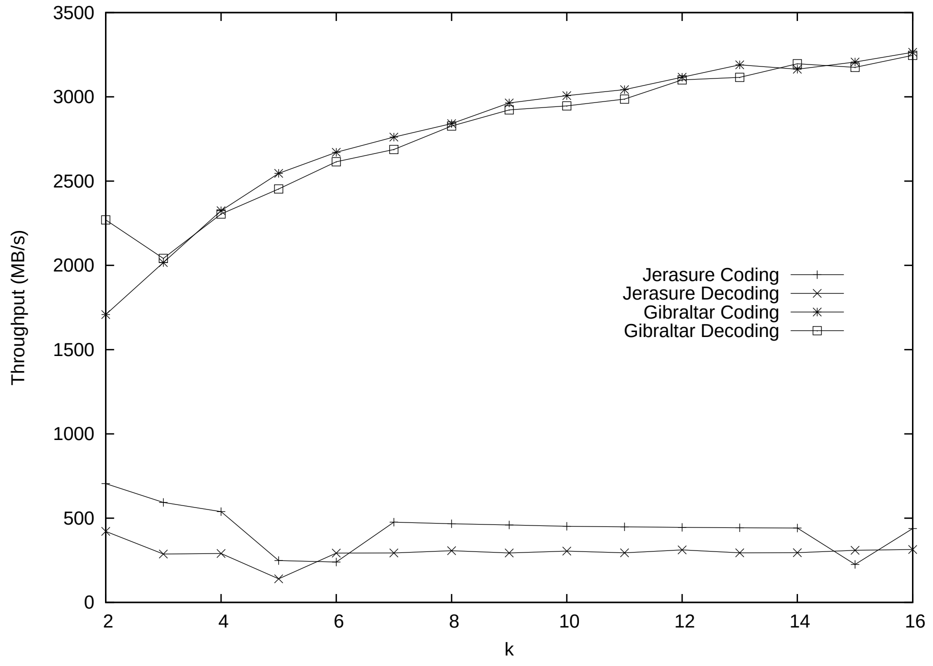


FIGURE 15. Performance for $m = 3$ Encoding and Decoding

An interesting performance characteristic is that Gibraltar has extremely similar performance for coding and decoding, while Jerasure tends to experience reduced performance for recovery. Gibraltar was designed so that the generation and recovery are nearly identical operations. Jerasure, does not, however, reconfigure the matrix in memory as needed, but instead indexes within it. Several things can cause Jerasure to be slower for decoding, including having less prefetch-friendly access patterns and creating the need for additional memory reads for indirect array accesses.

A further note: When $m > k$, the tests performed show an increase in performance because of the restrictions on which buffers may be erased during this experiment.

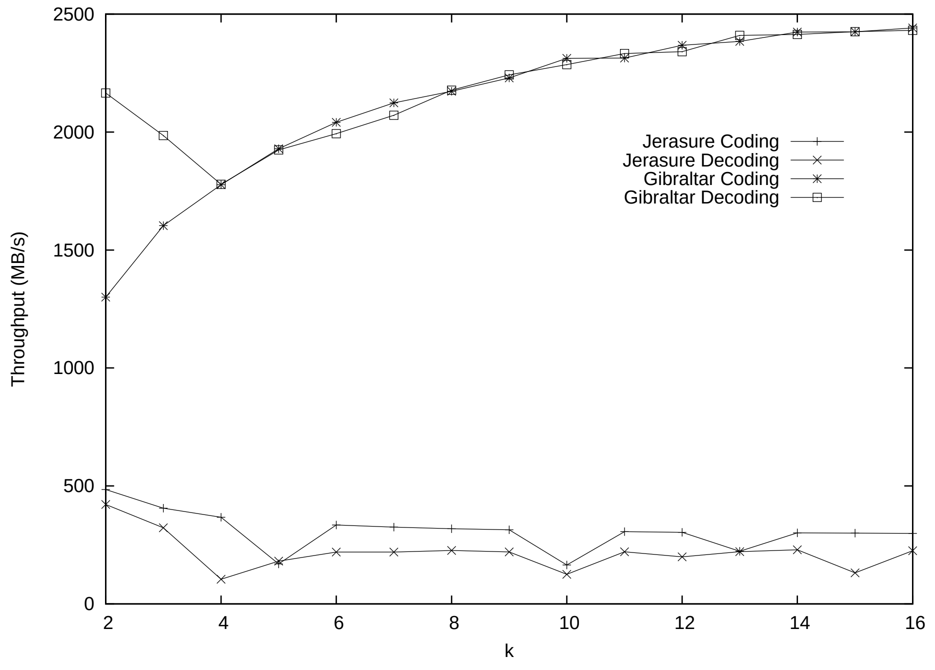


FIGURE 16. Performance for $m = 4$ Encoding and Decoding

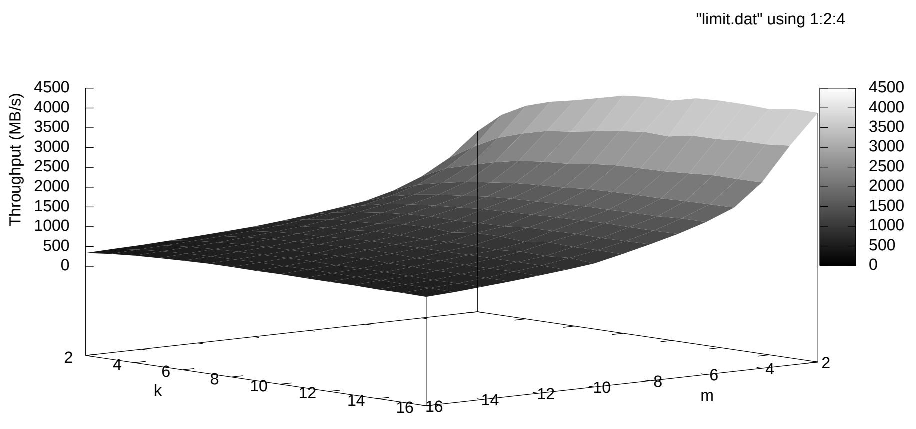


FIGURE 17. Encoding Performance for $m = 2 : 16$, $k = 2 : 16$

6. Future Trends

The current performance is highly balanced when viewed from the perspective of current NVIDIA graphics processors. In order to demonstrate this, three modes of operation have been tested:

- Operations performed on host memory. This is the typical mode of operation.
- Operations performed on GPU memory. Given the high bandwidth of GPU memory compared to PCI-Express bandwidth, this is an approximation of the maximum performance of the GPU kernel without the impacts of PCI-Express transfers.
- No operations performed on host memory, while performing transfers as if the kernel were performing computations. This is an approximation of the maximum performance of the PCI-Express bus without the impacts of GPU computation.

These experiments were also run on an NVIDIA GeForce GTX 285 with each buffer occupying one megabyte. Results show that, for a RAID 6 workload (Figure 18), the system is extremely well-balanced. For a RAID TP workload (Figure 19), the balance is also good. Current hardware starts to show more degraded balance with $m = 4$ (Figure 20) and beyond (Figure 21.)

One can see the effects of the overlapping computation and communication via the non-parallelizable overheads incurred, as represented by the near, but not precise, approximation of the minimum performance. This is interesting because of the tendency for performance of the total system to trend the PCI-Express performance with some overhead (approximately one gigabyte per second of throughput) when bandwidth-bound. This overhead is much smaller when coding becomes computation-bound. This is due to the large amount of data that must be transported to and from the GPU before work can start and after work has ended, respectively. The volume of data is caused by the massive number of processing elements, and the CUDA runtime's attempts to apply the entire GPU's computational capacity.

Such performance characteristics at this stage imply that, in order to have appreciably large performance improvements for $m = 2$ and $m = 3$, both the PCI-Express implementation and the computation power within the GPU must increase. However, if GPU power increases disproportionately, it becomes possible and reasonable to

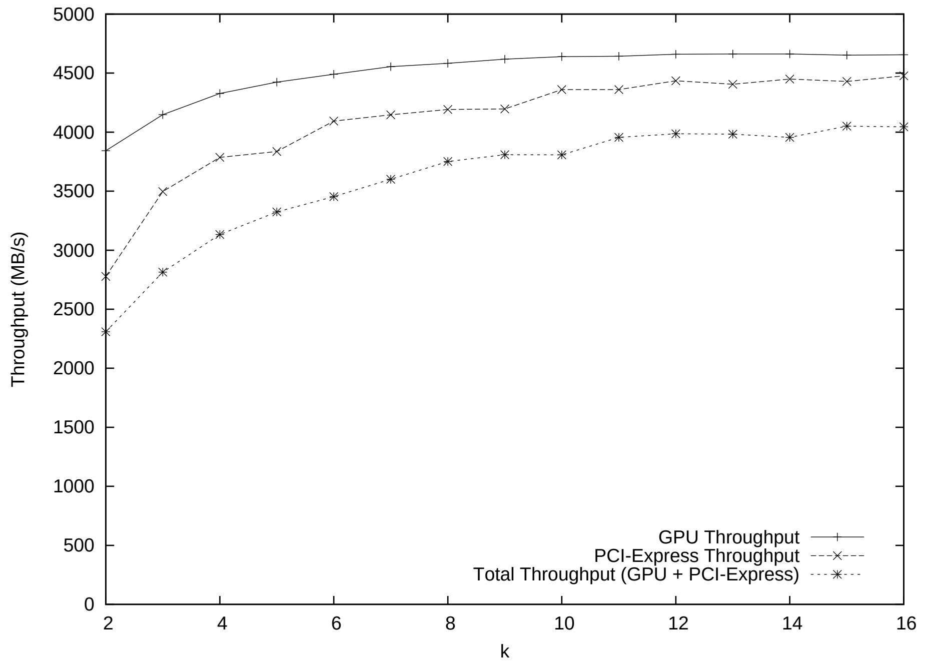


FIGURE 18. Excess PCI-Express Performance over GPU Performance for $m = 2$

perform encodings and decodings with increasing values of m without impact on speed, excepting the bandwidth requirements to transfer the extra coded buffers from the GPU after computation.

One major GPU architectural feature that most limits the speed of this computation is the same feature that makes the GPU attractive: The multi-banked memories. While such memories are useful in accelerating this computation, there are architectural changes that can improve performance of random table look-ups in shared memory. Through simulation, the authors have quantified the effects of many potential design changes.

- Increase the number of banks per shared memory unit. The upcoming Fermi architecture [**77**] from NVIDIA is designed with a larger shared memory, allowing for the possibility of a larger number of four-byte banks. Current

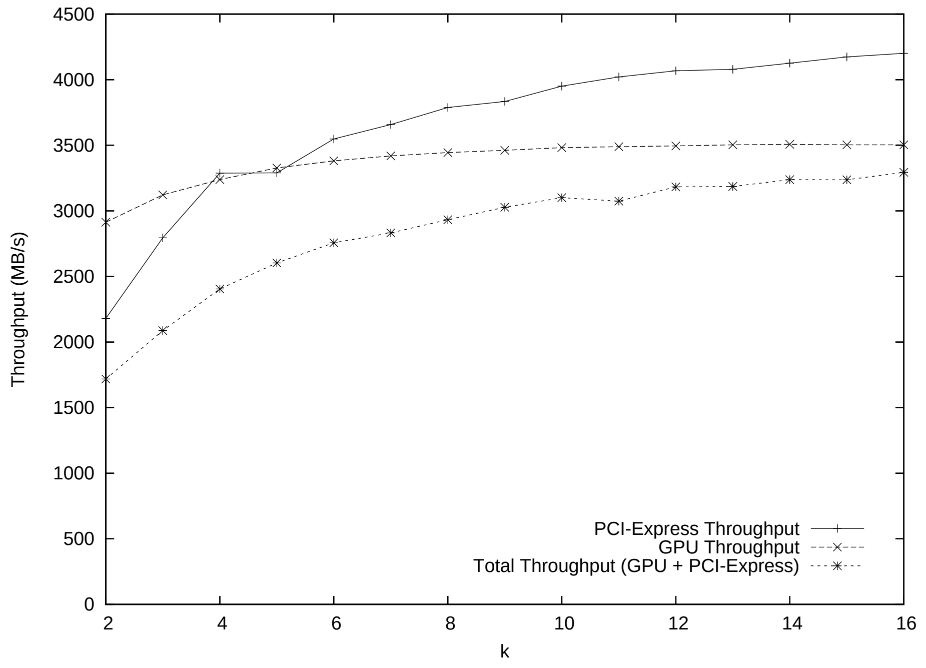


FIGURE 19. Excess PCI-Express Performance over GPU Performance for $m = 3$

shared memory sizes are 16 kilobytes, to be increased to up to 48 kilobytes (with an additional 16 kilobytes dedicated to an L1 cache.) If the number of banks were tripled along with this memory size, conflicting accesses would reduce to 2.84 per warp. Similar benefits could be achieved by shrinking the size of the banks and increasing their number, resulting in the same amount of shared memory. For example, 64 one-byte banks would yield 2.32 conflicting accesses per warp, while 32 two-byte banks would yield 3.15.

- Decrease the number of cores per multiprocessor, and increase the number of multiprocessors. This course of action would reduce the effect of the birthday paradox by reducing the number of competing cores. This would likely be an expensive option, as the number of shared memory units on the chip would grow quickly, but it would also be highly effective. Halving the number of cores to four, which run 16 threads together, would reduce conflicting accesses

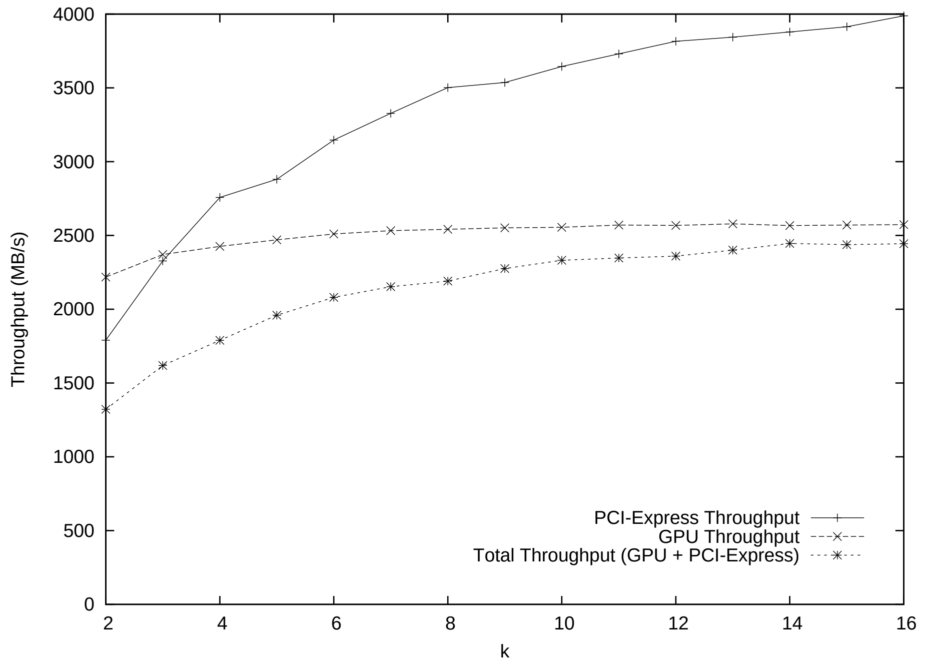


FIGURE 20. Excess PCI-Express Performance over GPU Performance for $m = 4$

to 2.91 per warp, while two cores (with eight threads) would only have 2.00 conflicting accesses per warp. A single core for four threads and the same shared memory would average 1.32 conflicting accesses per warp.

- Increase the speed of the shared memory. The G80 architecture requires four clock cycles to satisfy a shared memory request, which causes conflicting accesses between threads that are not necessarily running simultaneously. Allowing the memory to satisfy single-byte requests in a single memory cycle would decrease the number of conflicting accesses per warp to 2.91. This would have the same effect as halving the number of cores per shared memory unit and doubling the number of shared memory units.

An interesting feature that could improve the speed of random look-ups in small tables is the ability to load a small array into a particular bank, with the ability to

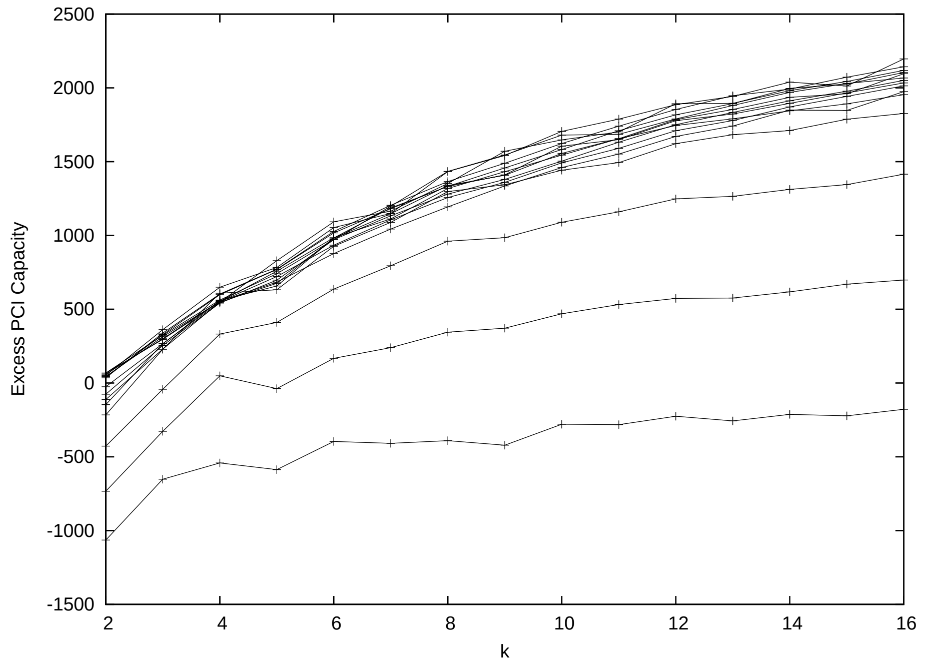


FIGURE 21. Excess PCI-Express Performance over GPU Performance for $m = 2..16$

perform normal indexing into the array. The programmer could then load multiple copies of the table simultaneously, querying each through different threads without conflict, or with controlled conflict. This approach was tested, but the indexing overhead was somewhat worse than the delay incurred by bank conflicts.

One interesting view into this data is the comparison of PCI-Express bandwidth and computational capacity of the GPU, as illustrated by Figure 21. Each line on the graph corresponds to a particular value for m, where higher lines correspond to larger values of m. Currently, the only value of m that, for all values of k, has excess computational capacity compared to PCI-Express bandwidth is $m = 2$, which corresponds to RAID 6. $m = 3$ comes much closer for many useful values of k to being balanced, but extra compute capacity in future GPUs will be useful in bringing the other values of m into reach of fully consuming PCI-Express bandwidth.

Furthermore, solutions that trade PCI-Express bandwidth for extra computation power remain effective. For example, it would be possible to split this computation across multiple GPUs by copying all data buffers to both GPUs, then have each GPU compute a fraction of the parity buffers.

7. Conclusions and Future Work

This chapter describes a general-purpose Reed-Solomon coding library, Gibraltar, which is suitable for use in applications requiring efficient data resiliency in a manner similar to RAID. Its immediate value stems from using CUDA-enabled GPUs to perform coding and decoding tasks, which has proven to be between 5- and 10-fold faster than a well-known CPU code running on a processor that costs three times as much as the GPU used by Gibraltar.

Several other non-storage applications can benefit from high-speed Reed-Solomon coding. For example, high-speed encryption and pseudorandom number generation become possible [**56**]. Similarly, secret sharing can be performed with large secrets at high speeds [**101**].

This work demonstrates the applicability of certain multicore architectures to Reed-Solomon coding, as well as provides analysis into the effects of the design parameters for these architectures. Guidance has been provided as to the most beneficial architectural decisions related to Reed-Solomon coding, which apply equally well to other applications with the same data access patterns.

As part of this work, Gibraltar, a practical library for Reed-Solomon coding and decoding on GPUs, has been developed. A generic, rational API has been developed that allows computation to be performed on many types of devices that do not have direct access to host memory, allowing back ends to be produced for many types of accelerators.

CHAPTER 6

PERFORMANCE EVALUATION OF A GPU-BASED RAID IMPLEMENTATION

The Gibraltar library is demonstrably fast enough to be used within the context of a RAID controller. However, there are other factors that must be experimentally measured to ensure that a GPU-based controller is feasible. This chapter will establish the effectiveness of Gibraltar RAID as part of a storage server intended for streaming I/O workloads. Comparisons will be made with Linux md, the software RAID implementation that is included with the Linux kernel. Tests will focus on DAS operation and NAS cases over Infiniband.

1. DAS Testing

DAS is an important use case for many RAID applications. In this arrangement, the RAID software or hardware delivers data to applications directly through the system call interface. For most RAID infrastructure, this is normal operation. However, for the architecture presented for this work, DAS is considered an unusual mode of operation, as the software is integrated directly into the target software. In order to have the benefit of interaction with the storage through a file system, the application will have to access the storage through the target.

For this use case, the target has been started on the test machine, and the Open-iSCSI initiator is used to connect to the target software through the loopback interface. This presents some potential advantage to other RAID infrastructure, as this is an extra layer of software through which storage is accessed.

To evaluate DAS performance, the following tests are performed.

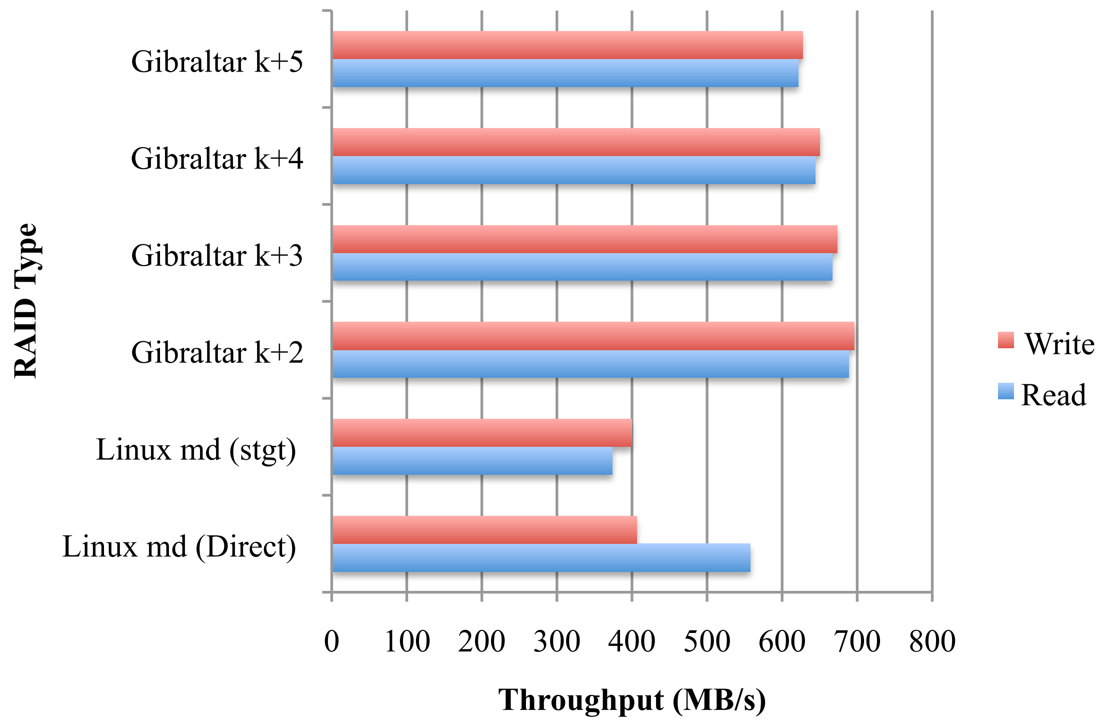


FIGURE 22. Streaming I/O Performance for DAS in Normal Mode

- Read: A single client runs dd on the raw device with a block size of one megabyte until one hundred gigabytes have been read.
- Write: A single client runs dd on the raw device with a block size of one megabyte until one hundred gigabytes have been written.

There are two configurations of Linux md that are tested. The first, which is the typical md use case, has the storage accessed directly through its device entry. This is the device named `/dev/md`*x* after creation of the RAID array. The second configuration is to use an unaltered version of stgt to offer the md device over the loopback interface. This is mostly to show the extra latency involved in using the stgt software. Four $k + m$ RAID levels supported by Gibraltar RAID are used. They are distinguished by their value of m, which is number of disks that may fail without data loss. Figure 22 shows the results for normal mode, where the array has zero failed disks.

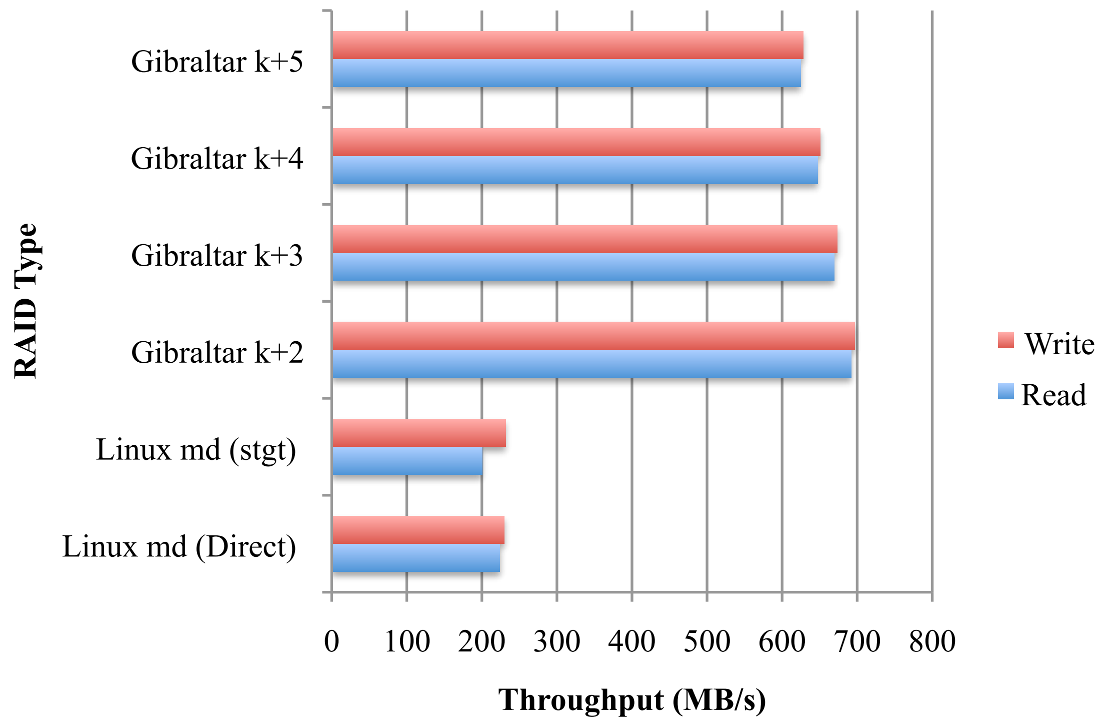


FIGURE 23. Streaming I/O Performance for DAS in Degraded Mode.

There are several interesting characteristics demonstrated by the results of this test. For instance, the difference between md direct, when reading and writing, is significant. Linux md, when writing, performs the coding generation required for RAID 6 recovery when a drive fails. Reading, however, does not incur any coding computations. This is in contrast to Gibraltar RAID, which performs coding to verify reads. Another interesting feature is that accessing the md RAID through stgt incurs a significant read penalty, while write throughput is unaffected.

Gibraltar RAID shows good performance for many RAID levels. The $m = 2$ configuration, which is equivalent to the Linux md in disk failure tolerance, is significantly faster for both reads and writes. While this indicates extra efficiency in integrating the storage management into the target, Gibraltar RAID verifies all reads with the same routines used to generate parity. Hence, Gibraltar RAID provides more data protection than md while providing increased throughput as well.

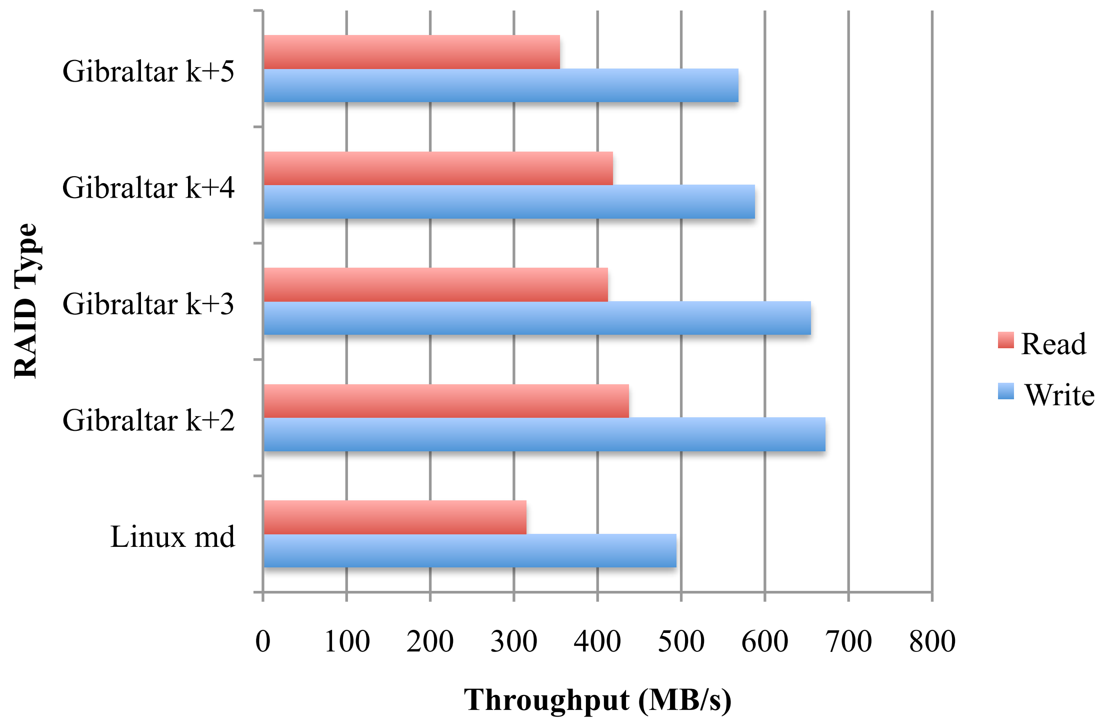


FIGURE 24. Streaming I/O Performance for NAS in Normal Mode for a Single Client

Figure 23 shows the results for degraded mode, where the array has one disk marked as faulty. It shows that Gibraltar RAID, because of the symmetric nature of the algorithms for coding and decoding, maintains the same levels of performance regardless of RAID type. Linux md experiences a loss of the majority of its performance. Interestingly, stgt does not affect md at all in degraded mode. This was not true for normal mode reads, indicating a loss of efficiency between tgt and md that is not experienced by Gibraltar RAID.

2. Single Client NAS Testing

In order to quantify the effects of introducing latency, testing with a single client on a high-speed network was also performed. This serves as a characterization of network effects by themselves in order to provide a more informed analysis of multi-client network testing. The testing parameters are the same as the previous tests, with

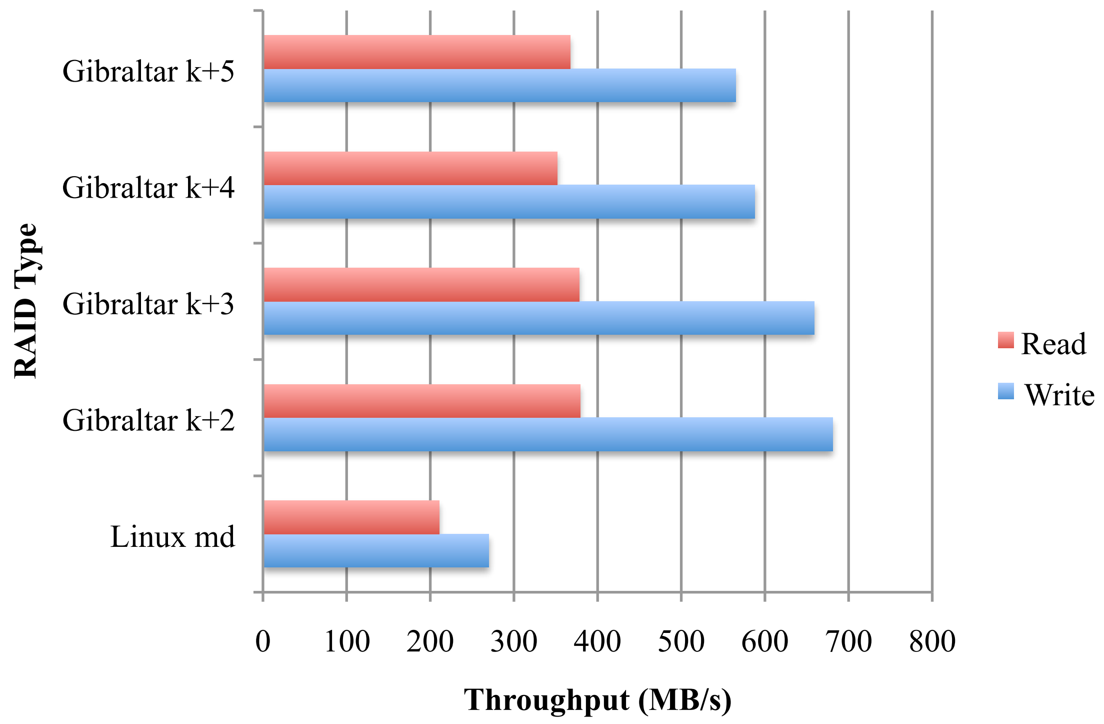


Figure 25. Streaming I/O Performance for NAS in Degraded Mode for a Single Client

a single client performing either a read or write in one megabyte chunks until one hundred gigabytes have been read or written. The network used is 4x DDR Infiniband.

Figure 24 shows the results of testing with all RAID configurations in normal mode. While the testing on DAS showed nearly equal read and write performance for Linux md through stgt, introduction of the network changed performance significantly. Reads were slower and writes were faster, introducing a 200 MB/s difference between the two. The same patterns can be seen in Gibraltar RAID, except write rates were consistent between tests. Gibraltar RAID maintains its performance superiority over Linux md.

Figure 25 shows the results of testing with all RAID configurations in degraded mode. Linux md has the expected amount of performance degradation, while Gibraltar RAID does seem to have a slight decrease in performance as well. This is not expected, and may be attributable to a performance bug. However, even with the slight decrease

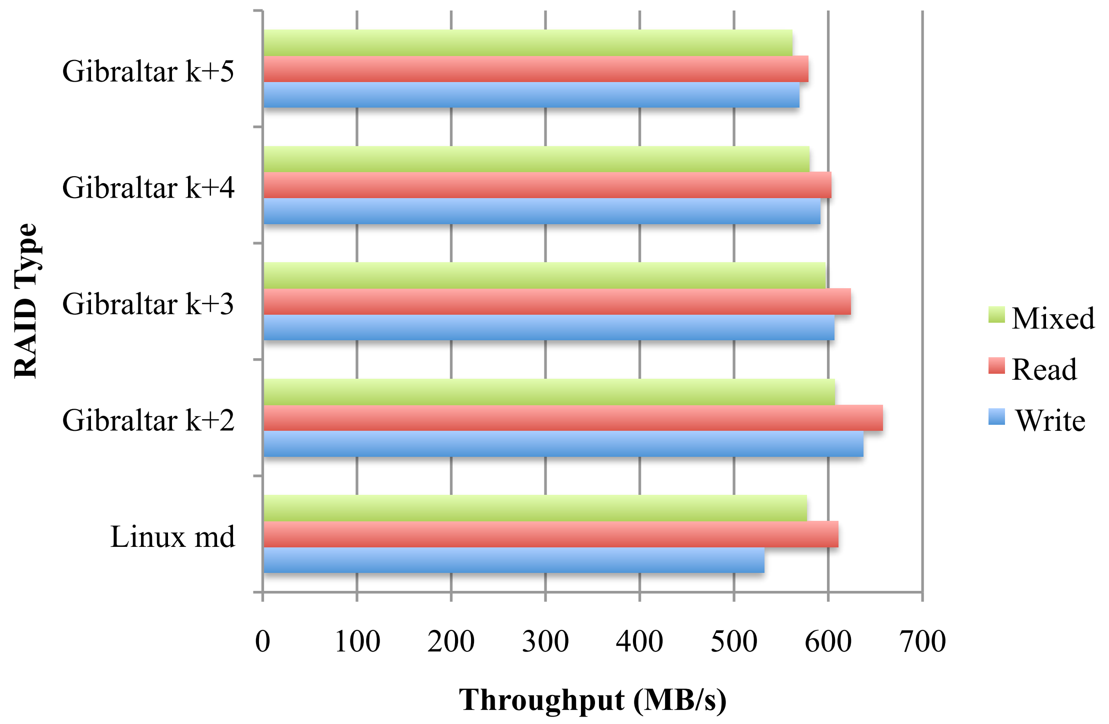


Figure 26. Streaming I/O Performance for NAS in Normal Mode for Four Clients

in degraded performance, Gibraltar RAID is still significantly faster than Linux md. Figures 24 and 25 show that there is a significant overhead related to network reads using stgt. This is possibly related to latency, as a read requires a network round-trip before the read call can be passed. Writes, on the other hand, do not display this characteristic. This is because writes can be completed in parallel with the application, in the background.

3. Multiple Client NAS Testing

This prototype is intended to be a storage server in a networked cluster of computers, necessitating a networked storage test. Four clients are used in the tests described in this chapter, with all connected to the same 4x DDR Infiniband network. To provide the storage, each target was configured to export four 128-gigabyte volumes, with one mounted per server. Three tests were performed: Each client reads one hundred

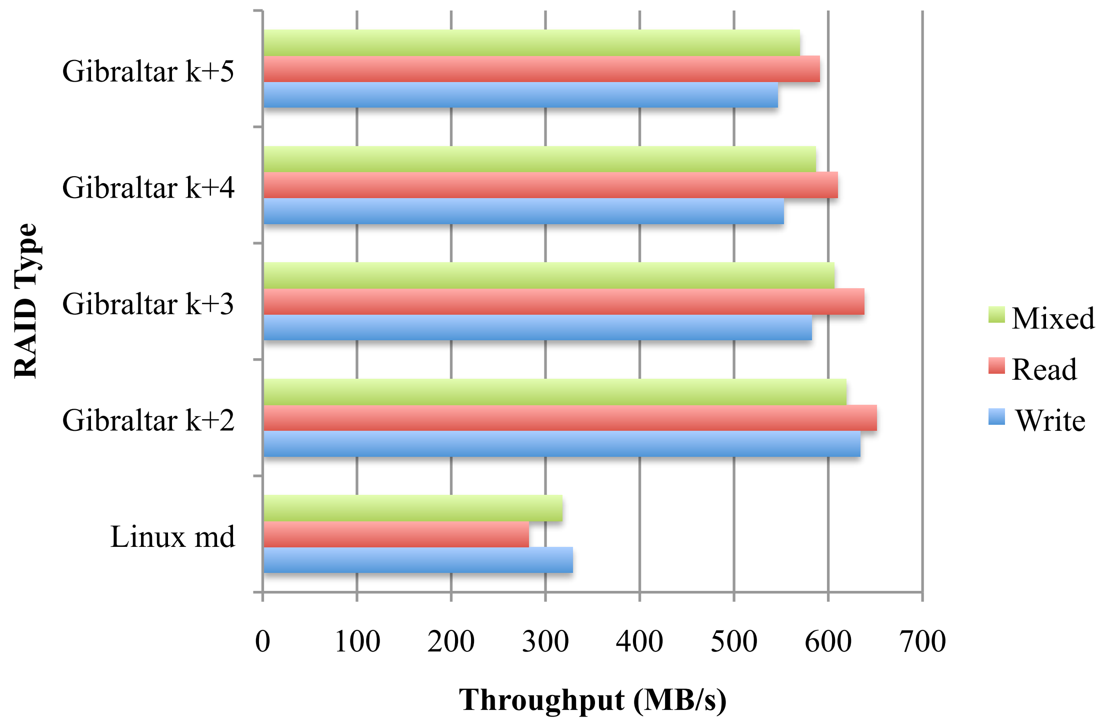


FIGURE 27. Streaming I/O Performance for NAS in Degraded Mode for Four Clients

gigabytes from its volume, each client writes one hundred gigabytes to its volume, and half perform each action on their respective volumes. This is a test of two main characteristics of the Gibraltar RAID controller:

- For all tests and modes, the software should handle all requests fairly without starving any client.
- In degraded mode for a mixed workload, the compute kernel for each volume must be changed several times. While the kernels are similar in their functionality, the degraded mode kernel has distinct operational requirements when compared to the normal mode kernel. The mixed-mode test for degraded mode will indicate whether there are problems with the current software and/or hardware configuration.

Figure 26 shows the results for normal mode. Interestingly, Linux md shows higher performance than any previous configuration, including direct access in Figure 22.

This is most likely caused by host-side cache effects. In any case, Gibraltar RAID performance remains at least as good as Linux md's performance for many test cases, with the $k + 5$ configuration being slightly slower but still competitive.

Figure 27 shows the results for degraded mode. The same situation is observed here: Performance is somewhat better for Linux md than in the DAS case. However, the slower overall operation of Linux md in degraded mode is apparent, resulting in slow operation in comparison with the Gibraltar RAID configurations. Gibraltar RAID's performance remains nearly identical to the normal mode cases, demonstrating the efficiency of degraded mode operations.

4. Conclusions

Gibraltar RAID, while implemented entirely in user space, presents a formidable competitor to the standard Linux software RAID system, md, for streaming workloads. Building Gibraltar RAID into the target was likely to be a good choice for the network storage case, provided an efficient architecture for the Gibraltar RAID layer. However, even directly interfacing with the target by mounting over the loopback interface is more efficient for Gibraltar RAID in both reading and writing. This is a surprising result, as md does significantly less work during reads than Gibraltar RAID does, given the requirement that Gibraltar RAID verify reads with parity.

Gibraltar RAID's use of Reed-Solomon coding for erasure correction provides demonstrable performance benefits over Linux md while providing efficient data protection. While md suffers significant performance degradation for writes, Gibraltar RAID demonstrates better read and write performance for five disk-failure-tolerant configurations and beyond. Further, degraded mode is no longer a concern with Gibraltar RAID and streaming workloads, while md loses up to 50% of its performance.

This chapter has provided several test scenarios for an important application area, streaming workloads. By comparing against Linux md, the efficiency of Gibraltar RAID can be directly evaluated. Gibraltar RAID's RAID 6 performance always surpasses that of md. In fact, the $k+5$ configuration is competitive with md's RAID 6 configuration. Degraded mode performance is vastly superior for Gibraltar RAID across the wide range of levels tested. In short, for these workloads, Gibraltar RAID is technologically sound and a significant advance on software RAID in use today.

CHAPTER 7

FUTURE WORK AND EXTENSIONS

Extended RAID that is supported by GPU computation is an important result on its own. However, there are further fruitful research paths that can build on the knowledge of GPU-based RAID. These include exploration of alternative implementation platforms, reapplication of underlying technology to create new capabilities, and alternative storage organization enabled by decoupling RAID computation from disk drives. This section provides an overview of some future work that follows from this work.

1. Failing in Place for Low-Serviceability Storage Infrastructure

When describing RAID reliability, the MTTDL calculations will often assume that the time to replace a failed disk is relatively low, or even negligible compared to rebuild time. For platforms that are in remote areas collecting and processing data, or for installations designed for density rather than serviceability, it may be impossible to access the system to perform maintenance for long periods. A disk may fail early in a platform's life, but not be replaceable for much longer than typical for RAID arrays. Service periods may be scheduled at intervals on the order of months, or the platform may not be serviced until the end of a mission in progress.

An alternative application of high-parity RAID is for a storage brick can be designed to have a MTTDL that extends well beyond the useful lifetime of the hardware used. Such a system would require less service throughout its lifetime. This approach is similar to the Xiotech ISE [**112**], but through high-parity RAIDs can increase reliability and performance. Such a design can be deliberately placed remotely (without frequent support) in order to provide colocation, or meet installation cost goals.

While it is possible to include hot spares within a storage brick, they do not protect the system from UREs during the rebuild process. Furthermore, when a hot spare is being used in reconstruction, there is a window of vulnerability during which disks are more likely to fail because of the stress of the rebuilding process. This can potentially leave the array without excess parity to protect against UREs or to verify reads.

In a fail-in-place infrastructure without hot spares, degraded mode performance is important, as failure is considered to be the normal operating case. It has been shown that x86-64 CPUs do have ways of optimizing parity *generation* with vector instruction sets, there are not any similar means of increasing the speed of degraded mode operation of an array [**5, 8**]. An important design goal of a high-parity RAID system is to have adequate performance for data recovery and generation of parity. The GPU-based parity computation library, presented in Chapter 5, satisfies this requirement.

1.1. Extra Parity or Hot Spares? One solution is to continue using RAID 6, but to include enough hot spares in an array to prevent triple-disk failures between service periods. Unfortunately, hot spares do not provide improved mitigation for UREs. An administrator managing an inaccessible system does not have the luxury of restoring a lost sector from backups, if they exist. Even locally-installed arrays that encounter UREs cause significant workload for administrators to repair the volume. As disks become larger, the incidence of UREs will be increasingly incident with a disk failure, creating a need for increased parity for improved data recovery.

Further, using hot spares creates a window of vulnerability during which another disk may fail, with the potential of data loss. Rebuild times are increasing as disks grow in size, making this window of vulnerability larger over time. However, if all disks are kept on line, with extra disks providing extra capacity for parity, there is no window of vulnerability except when no fault tolerance is left in the array.

2. Multi-Level RAID for Data Center Reliability

Protection from data loss is a high priority for those designing data center storage systems. However, another important metric to data center operators is availability. These concerns extend far beyond the traditional high performance computing community into e-commerce and financial institutions where downtime is generally equated to a loss in earning potential. While traditional RAID systems can protect against data loss, RAID does not protect against failure of other components within a server, such as the power supplies or interconnect. It has been found that, while disks are possibly the most failure-prone component, other components within the system cannot be ignored when designing a reliable storage system [**53**]. An alternative storage organization using RAID concepts may be able to mitigate many types of failure in storage infrastructure.

Currently, data center storage is most commonly implemented by creating several independent storage servers and either installing a parallel file system (*e.g.*, Lustre [**1**] or PVFS [**50**]) across all nodes, or dealing with several independent non-parallel file systems on each storage server, as is the common case with storage area networks. Some extra reliability can be provided with controller failover, with active-passive configurations supplying a stand-by controller that can respond to requests for a failed primary controller, or active/active configurations that pair two in-service controllers. In an active/active configuration, if one controller becomes unavailable, the second may service requests for the other while continuing to service its own clients.

Figure 28 shows an active-passive configuration or active/active where controller load is a bottleneck. Figure 29 shows an active/active configuration with multiple storage resources. Each figure includes a network connection between controllers with a coherence protocol, as each must maintain the incoherent cache contents of the other. This way, when a controller fails, no data that has only been written to cache will be lost. This serves to increase the availability of a storage cluster at the cost of

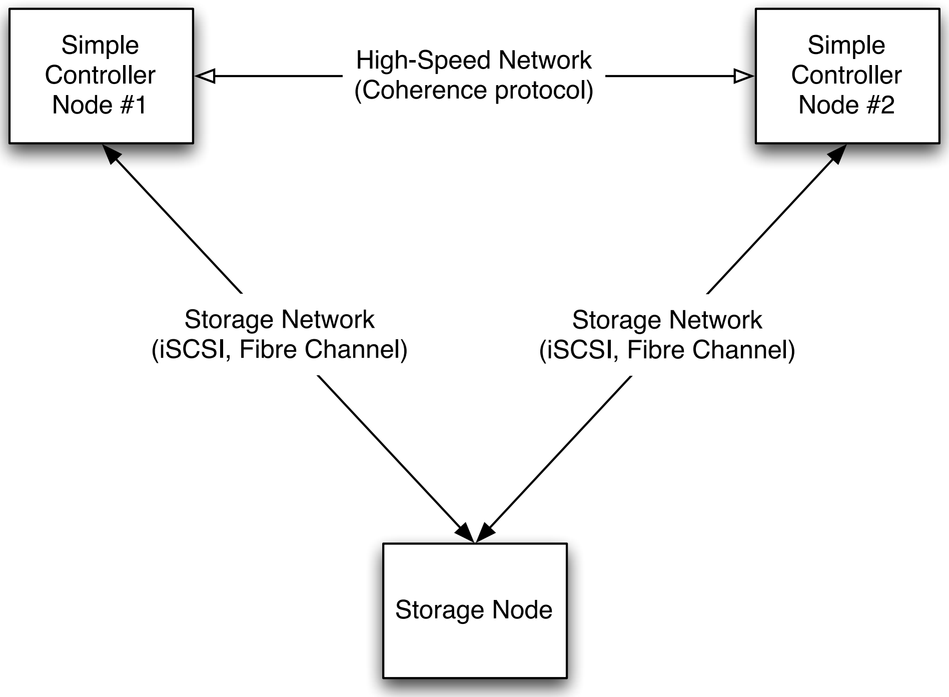

Simple Controller Node #1
High-Speed Network (Coherence protocol)
Simple Controller Node #2
Storage Network (iSCSI, Fibre Channel)
Storage Network (iSCSI, Fibre Channel)
Storage Node

FIGURE 28. Network Diagram for Typical Active/Passive Configuration, or Active/Active with High Controller Load

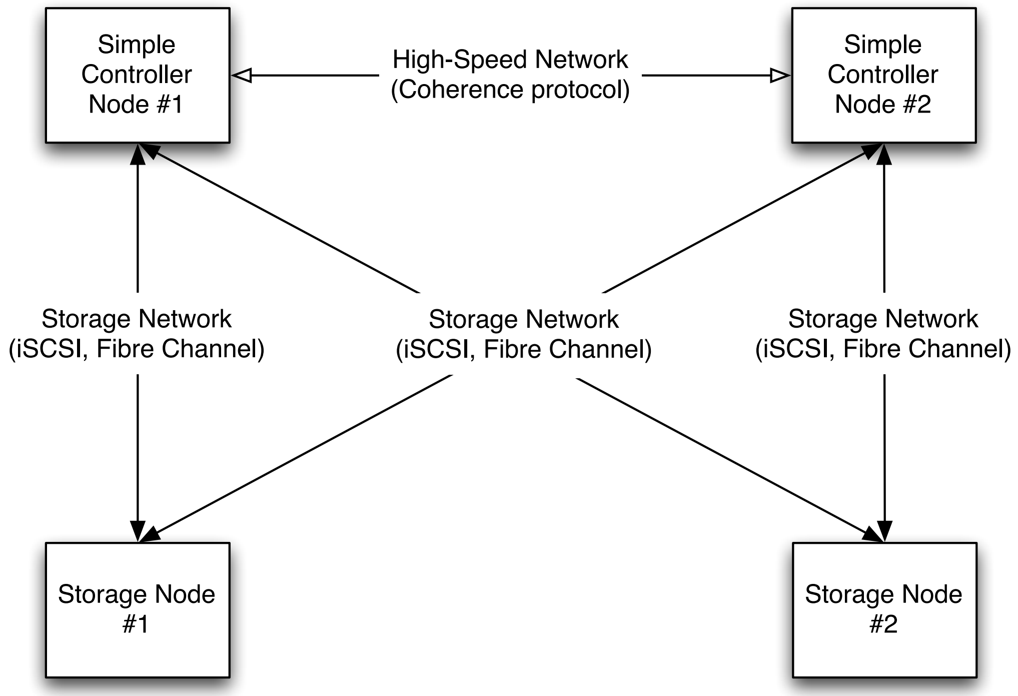

Simple Controller Node #1
High-Speed Network (Coherence protocol)
Simple Controller Node #2
Storage Network (iSCSI, Fibre Channel)
Storage Network (iSCSI, Fibre Channel)
Storage Network (iSCSI, Fibre Channel)
Storage Node #1
Storage Node #2

FIGURE 29. Network Diagram for a Typical Active/Active Configuration

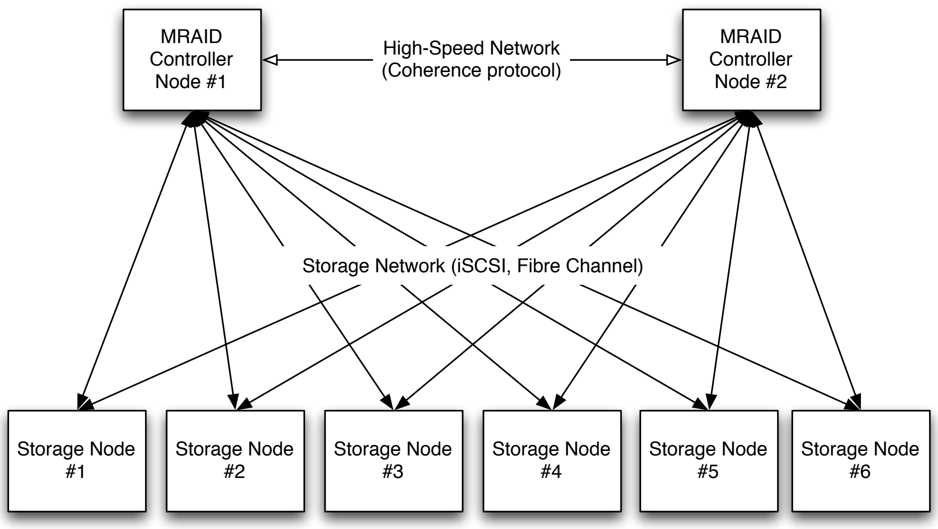


Figure 30. Network Diagram for an Active MRAID Configuration

decreased speed when a controller fails. However, failure of storage nodes is still a catastrophic event that leads to data unavailability.

Figure 30 shows an improved organization, here termed Active MRAID, that can be used with a hardware-decoupled high-speed RAID implementation to tolerate outright failure of storage nodes without a parallel filesystem. This is accomplished by treating individual servers as storage devices, as multilevel RAID does. The differences are apparent in controller organization: There is typically only one top-level MRAID controller, leading to a reduction in availability. Active MRAID can include multiple top-level controllers, with each controller striping data across all storage nodes. Data resiliency can be implemented by using a GPU with Gibraltar to generate RAID stripes to split among the underlying storage servers.

With the inclusion of the active/active controller pair, which is not part of the MRAID concept, several nodes within the network can become unavailable simultaneously. An additional financial benefit is the possible reduction of cost for each individual storage node; each node can be less reliable and host fewer disks. Instead of relying on each storage node to be a massive collection of disks that can serve storage at high rates with high reliability, larger numbers of slower, less-reliable nodes can be deployed. In fact, each storage node may not require hardware or GPU-based RAIDs, but can use RAID 5 locally with software-only implementations, reducing costs further.

Several key questions include the ultimate scalability of this concept. Active/active controller pairs are common today, but it is reasonable to question the inclusion of more than two active controllers per storage pool. As storage clusters tend to be much smaller than compute clusters, different network topologies can also be explored which can increase the efficiency of interactions between more than two controllers, especially when write-through caching is used and storage nodes are fully appointed servers.

3. Combining RAID with Other Storage Computations

The bandwidth within a graphics card is on the order of 100 GB/s, while PCI-Express bandwidth remains less than 10 GB/s. When dealing with large amounts of data, performance is best when all necessary operations can be applied to data within a GPU. The data's presence in GPU memory can be applied to active storage and three traditional storage services: Encryption, deduplication, and compression.

Active storage is a storage paradigm where excess processing power can be applied to storage applications, allowing certain tasks to be accomplished by the storage system on behalf of the user [**94**]. Given the application of GPU computation to all reads and writes for a RAID array as outlined in this work, all data will have been transferred to a GPU before reading and writing. Active storage concepts could be applied while the data is passing through the GPU, given an appropriate GPU-based active storage infrastructure.

Further, GPU-based encryption may be suitable for storage applications. Work has already been done to investigate AES encryption using CUDA-based GPUs [**9**]. While full-disk encryption is available for storage servers based on per-server keys, this type of solution could provide a more flexible means of encryption on a per-user basis.

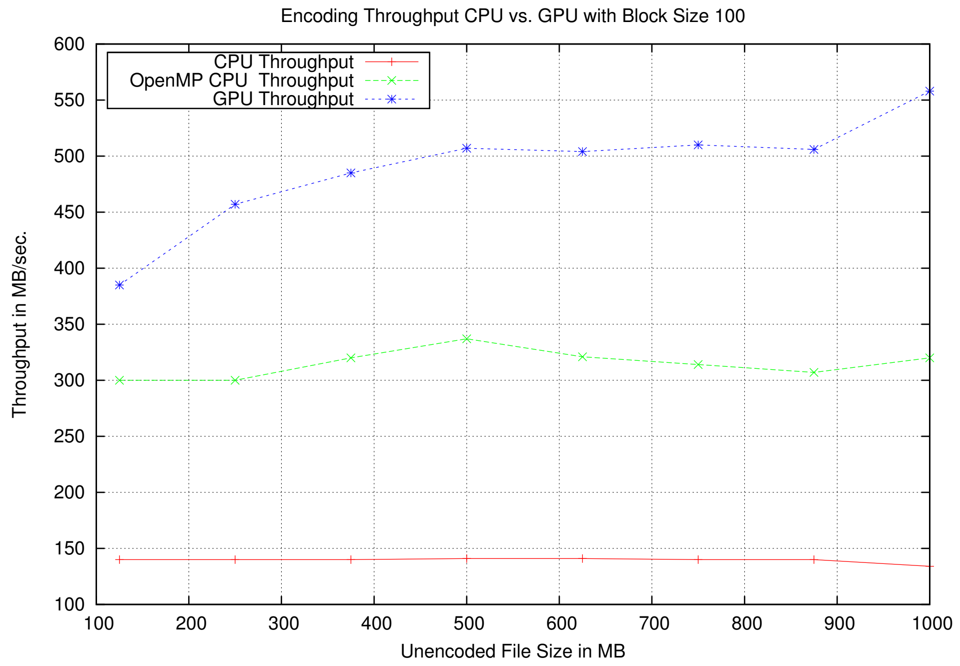


FIGURE 31. Huffman Encoding Rates for an Intel i7 Extreme Edition 975 and an NVIDIA Geforce GTX 285.

Deduplication can be implemented using cryptographically secure hashing, which has also been implemented on GPUs [**114**].

Finally, block-level compression can be applied to user data. If efficient enough, it could be applied to increase the user-visible bandwidth of the disk subsystem. Research to find good trade-offs for GPU computation speed and storage efficiency is ongoing, but Robert Cloud has preliminary results showing that Huffman coding and decoding can be accelerated with a CUDA-based GPU. Figure 31 shows the encoding rates for a single core of an Intel i7 Extreme Edition 975, all cores of an Intel Extreme Edition 975, and a single NVIDIA GeForce GTX 285. Figure 32 shows the decoding rates for the same hardware.

4. Checkpoint-to-Neighbor

The Scalable Checkpoint Restart (SCR) library, which bypasses the global parallel filesystem for checkpointing purposes and instead uses node-local storage across the machine, has experimentally been used to provide hundreds gigabytes per second of

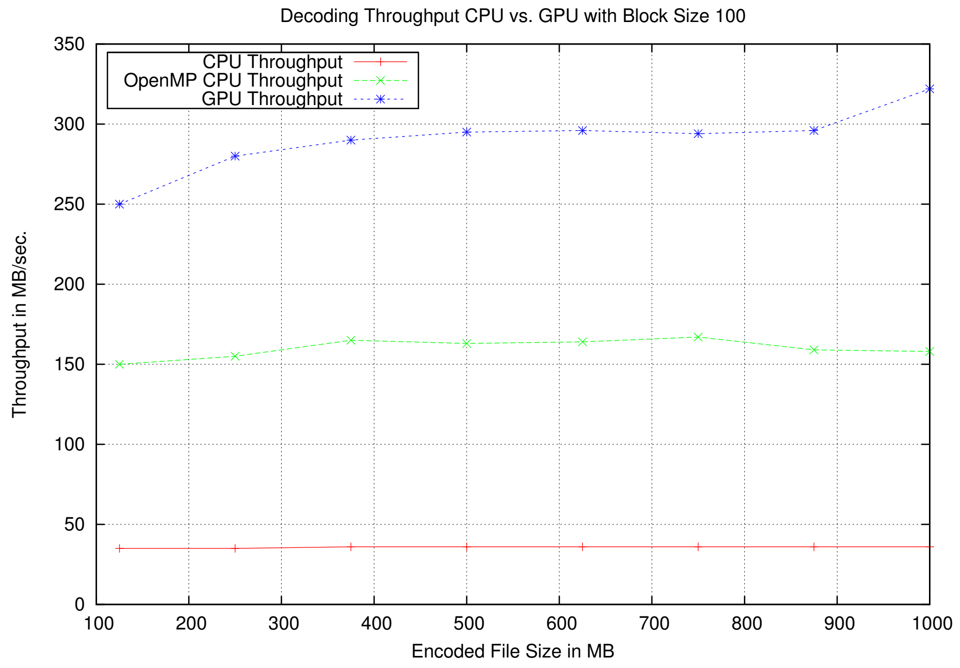


FIGURE 32. Huffman Decoding Rates for an Intel i7 Extreme Edition 975 and an NVIDIA Geforce GTX 285.

bandwidth to checkpointing activities within an application[**15**]. The library functions by creating a replica of a node's important job data and storing the replica on the storage of another node. Other operating modes are described, including an XOR mode that mimics a RAID 5 volume that is distributed across a team of nodes. However, if a GPU is available within the node, it becomes possible to more widely disburse portions of a checkpoint, allowing for higher reliability with comparatively smaller amounts of node-local storage.

5. Alternative Platforms

While GPUs have been shown to be effective at RAID computations, the user space RAID infrastructure developed for this work lowers the barrier to entry for testing other devices RAID devices. So long as an implementation of the Gibraltar API can be developed for the device, a RAID implementation is trivial to test with a real I/O workload. There are several known platforms that may make worthwhile Reed-Solomon coding back ends. These include FPGAs, CPUs implementing new vector instructions, and projected platforms with CPUs and GPUs integrated on the same die. Furthermore, as future CPU platforms evolve, the architectures used may lend themselves to the massively multicore processing strategies used in NVIDIA GPUs for this work.

CHAPTER 8

CONCLUSIONS

Many sites desire two main features from a RAID controller: Assurance that the data read are correct, and high-speed operation [**107**]. If software RAID could meet this need, the impact would be significant. While current RAID technology has served the community well for the last two decades, two main concerns have crept to the forefront of storage system practitioners' attention. First, quality hardware RAID is expensive (as shown in Figure 1 on page 3), making it inaccessible to a broad community. Second, current RAID levels may not be reliable enough to accommodate changing storage technology (as demonstrated in Figure 11 on page 33). This work offers a significant advancement in RAID, GPU-based high-parity RAID, which provides solutions to both problems.

Many system administrators look to HRAID as a method of prolonging the usefulness of traditional RAID levels. This work has shown that HRAID levels do not significantly reduce the probability of losing data to disk failure or UREs. For manufacturer-supplied failure statistics, Figure 7 (on page 29) shows that there is little difference in reliability between a RAID 5 array and a four-set RAID 5+0 array of the same capacity. Further, Figure 8 (on page 30) shows a single order of magnitude improvement between a RAID 6 array and a four-set RAID 6+0 array with the same capacity. While RAID 1+0 can offer extreme levels of reliability, it requires a large investment in disks compared to parity-based RAID methods. Many within the storage industry have largely accepted that current storage hardware cannot guarantee the safety of data. Garth Gibson, one of the pioneers of RAID, has recently proposed that the metric of choice for reporting RAID reliability be the average number of bytes lost per year [**36**].

This work has demonstrated that $k + m$ RAID, a RAID level with administrator-selectable fault tolerance within an array, can reasonably solve this reliability problem. The convenience of the feature is one of its most valuable traits. Using the equations given in Chapter 3, with parameters that are known or can be reasonably estimated, a storage system administrator can choose the amount of parity required to satisfy reliability requirements. Further uses include testing initial deployments of storage infrastructure in a production setting. If the hard disks used are affected by a batch-correlated defect, the probability of survival is significantly increased by using more parity in the array [**82**]. Other strategies include, upon noticing increased failure rates, having an administrator add more storage for increased parity to the live system. This kind of capability is not available with current RAID 6 systems, which only provides the recourse of initiating a full system backup.

This work has also demonstrated that replication-based schemes, like RAID 1+0 and the Google File System (GFS) [**35**], are not as reliable as relatively low-parity $k+m$ RAID levels. When keeping capacity constant, RAID 1+0 has different characteristics than parity-based RAID levels that make RAID 1+0 asymptotically more reliable. However, as shown in Figure 10, $k + 3$ RAID with 128 two-terabyte disks maintains better reliability while offering the same amount of user storage when compared to three-set RAID 1+0 with 375 two-terabyte disks. This can be directly compared to the reliability features of the GFS because it creates three replicas of each file by default. For Figure 11, where failure rates are determined by empirical data and rebuild times are increased, $k + 3$ RAID is again the more reliable configuration.

While $k + m$ RAID can create more reliable storage with less hardware than any other parity-based RAID, it is computationally expensive as software RAID on x86 and x86-64 CPUs. The parity is computed with Reed-Solomon coding, which involves heavy use of table look-up operations. This work describes the Gibraltar library, a prototype library that provides high-speed Reed-Solomon coding for RAID on NVIDIA

CUDA GPUs. Current GPUs have been thoroughly analyzed for their fit to the task, with further evaluation of potential future evolutionary changes.

For $k + 3$ workloads, the Gibraltar library is capable of sustaining six-fold the performance of a Jerasure, a well-known CPU-based Reed-Solomon code library. Algorithm modifications have improved data reconstruction rates on the GPU, yielding symmetric performance for encoding and decoding. This symmetry is an important quality to software RAID applications, as degraded mode operation causes decreased performance for reads. Without symmetry, user-level performance decreases and reconstruction times increase. Improving the performance of degraded mode allows for fast recovery or fail-in-place functionality. As failures become more prevalent, one can estimate that an array of disks will usually have at least one disk offline. The Gibraltar library can maintain good performance under these circumstances.

While the Gibraltar library demonstrates new concepts for RAID encoding and decoding, and provides a valuable resource for user space storage applications, it is not immediately applicable within a RAID controller. This work describes Gibraltar RAID, a software RAID proof of concept that provides a traditional block device for general use, with parity computations provided by the Gibraltar library. This was a significant hurdle, as CUDA GPUs are not accessible from kernel space. Potential performance problems could have inhibited the ability of a RAID system to make full use of GPU-based coding. Gibraltar RAID is integrated with the Linux SCSI Target Framework (stgt), the standard network storage protocol target package for Linux. This design leverages many capabilities of the stgt code base, including the ability to serve network storage with iSCSI, Fibre Channel over Ethernet, iSER, and other storage protocols.

To provide the best performance, much of the traditional storage stack was designed around the Gibraltar library in user space. Gibraltar RAID is about 25% faster than Linux md for direct-attached normal RAID 6 streaming reads. Gibraltar RAID is 75% faster for direct-attached normal RAID 6 streaming writes. Direct-attached normal streaming read and write performance remains superior through $k + 5$ RAID, which improves reliability over RAID 6 by up to eight orders of magnitude. Given that Gibraltar RAID verifies the results of read operations against parity, its RAID 6 mode improves assurance against corrupted data and bugs better than Linux md does, permitting its use in HPC applications that require high confidence in results.

The most extreme performance differences can be seen when comparing Gibraltar RAID and Linux md in degraded mode. Disk failures cause no performance degradation in Gibraltar RAID. Linux md, in comparison, suffers severe performance loss: Performance decreases by more than 50% for reads. Under these conditions, md may require significantly longer to repair a failed disk than Gibraltar RAID, especially when under load. Further, it is unreasonable to leave the array in this degraded state because of increased risk of overall failure and unacceptable performance. Higher parity Gibraltar RAID, however, can maintain superior performance under perpetual failure, allowing significantly larger RAID arrays than are possible with RAID 6.

Gibraltar RAID is a demonstration of a software RAID infrastructure without software RAID's main drawback: Low performance. This leaves the benefit that software RAID has long enjoyed over hardware RAID: Flexibility. While MRAID with hardware acceleration has been discussed in research circles, practitioners of software RAID have been doing similar organizations without difficulty. For example, with Linux md, any block device listed in the `/dev` directory can be used as part of an array. For example, iSCSI/iSER shares (which may be backed by RAID 6 arrays on remote machines) can be assembled into an MRAID on one machine.

The high-speed nature of Gibraltar RAID allows for its use as an MRAID infrastructure in environments where performance is a priority. MRAID's purpose is to allow storage infrastructure to withstand more types of failures than can be handled by RAID alone. While active/active and active/passive controller pairs can withstand controller failures, and MRAID can withstand multiple storage node or enclosure failures, no solution currently provides both types of functionality in a single package because of the coding bottleneck. With the use of GPU-based RAID, this infrastructure is now possible.

Further advances enabled by the Gibraltar library include user-managed distributed data storage. Most distributed data storage uses replicas for reliability, typically requiring at least 200% overhead in bandwidth and storage space. However, a significant bandwidth savings and reliability increase can be obtained by storage of coded pieces of a file. These pieces can be stored with different services to decrease chances of administrative mistakes causing data loss.

In conclusion, this work provides a firm foundation for RAID to evolve to offer more reliability in a changing technological landscape. The family of $k + m$ RAID levels, which embody the generalization of parity-based RAID concepts, has been defined and proposed. It is shown to offer significantly better protections than fundamental and hierarchical RAID levels today. A proof of concept for an advanced, inexpensive (as demonstrated in Appendix D), high-performance software RAID controller that uses GPUs for coding has been created and benchmarked, demonstrating its performance advantages over current software RAID technology under streaming workloads. Furthermore, GPU-based RAID provides a means of assembling large amounts of storage in a cluster environment for a significantly reduced cost. Additional reliable storage applications are enabled by a practical library for storage coding and decoding, allowing data storage in a broader context than RAID, while enabling previously impractical RAID variations.

References

[1] Lustre file system: High-performance storage architecture and scalable cluster file system. `http://www.raidinc.com/pdf/whitepapers/lustrefilesystem_wp.pdf`, December 2009.

[2] AMD. ATI CTM guide. `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`, 2006.

[3] AMD. ATI Stream technology: Technical overview. `http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf`, 2008.

[4] AMD. The future is fusion. The industry-changing impact of accelerated computing. `http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf`, 2008.

[5] H. Peter Anvin. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2009.

[6] Sung Hoon Baek, Bong Wan Kim, Eui Joung Joung, and Chong Won Park. Reliability and performance of hierarchical RAID with multiple controllers. In *PODC '01: Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, pages 246–254, New York, NY, USA, 2001. ACM.

[7] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[8] Raghav Bhaskar, Pradeep K. Dubey, Vijay Kumar, and Atri Rudra. Efficient Galois field arithmetic on SIMD architectures. In *SPAA '03: Proceedings of the*

Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pages 256–257, New York, NY, USA, 2003. ACM Press.

[9] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi. Design of a parallel AES for graphics hardware using the CUDA framework. *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 0:1–8, 2009.

[10] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 245–254, 1994.

[11] OpenGL Architecture Review Board. ARB_fragment_shader. `http://www.opengl.org/registry/specs/ARB/fragment_shader.txt`, June 2003.

[12] OpenGL Architecture Review Board. EXT_framebuffer_object. `http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt`, January 2005.

[13] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[14] A. Brinkmann and D. Eschweiler. A microdriver architecture for error correcting codes inside the Linux kernel. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10, New York, NY, USA, 2009. ACM.

[15] Greg Bronevetsky and Adam Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report LLNL-TR-415791, Lawrence Livermore National Laboratory, 2009.

[16] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[17] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 56–67, New York, NY, USA, 1998. ACM.

[18] J.W. Byers, M. Luby, and M. Mitzenmacher. Accessing multiple mirror sites in parallel: Using Tornado Codes to speed up downloads. volume 1, pages 275–283, March 1999.

[19] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006*, pages 203–209, Toronto, Ont., Canada, 2006. Canadian Information Processing Society.

[20] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[21] Gerry Cole. Estimating drive reliability in desktop computers and consumer electronics systems. Technical Report TP-338.1, Seagate, November 2000.

[22] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST 04)*, pages 1–14, 2004.

[23] Intel Corporation. Intel 81348 I/O processor datasheet. `http://download.intel.com/design/iio/datashts/31503803.pdf`, December 2007.

[24] Intel Corporation. *Intel X25-E SATA Solid State Drive Product Manual*, May 2009.

[25] Matthew L. Curry and Anthony Skjellum. Improved LU decomposition on graphics hardware. Poster at Supercomputing 2006 conference.

[26] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Gibraltar: A library for RAID-like Reed-Solomon coding on programmable graphics processors. Technical report. Submitted to Concurrency and Computation: Practice and Experience.

[27] Matthew L. Curry, Anthony Skjellum, H.Lee Ward, and Ron Brightwell. Accelerating Reed-Solomon coding in RAID systems with GPUs. In *IEEE International Symposium on Parallel and Distributed Processing, 2008*, pages 1–6, April 2008.

[28] Matthew L. Curry, H. Lee Ward, Anthony Skjellum, and Ron Brightwell. Arbitrary dimenision Reed-Solomon coding and decoding for extended RAID on GPUs. In *3rd Petascale Data Storage Workshop held in conjunction with SC08*, November 2008.

[29] Anirban DasGupta. The matching, birthday and the strong birthday problem: A contemporary review. *Journal of Statistical Planning and Inference*, 130(1-2):377 – 389, 2005. Herman Chernoff: Eightieth Birthday Felicitation Volume.

[30] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[31] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C*. Addison-Wesley Professional, second edition, 1995.

[32] Tomonori Fujita and Mike Christie. tgt: Framework for storage target drivers. In *Proceedings of the Linux Symposium*, July 2006.

[33] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[34] Philipp Gerasimov, Randima Fernando, and Simon Green. Shader model 3.0: Using vertex textures. Whitepaper, June 2004. `ftp://download.nvidia.com/developer/Papers/2004/Vertex_Textures/Vertex_Textures.pdf`.

[35] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[36] Garth Gibson and Lin Xiao. Reliability/resilience panel. `http://institute.lanl.gov/hec-fsio/conferences/2010/presentations/day2/Gibson-HECFSIO-2010-Reliability.pdf`, 2010.

[37] M. Gilroy and J. Irvine. Raid 6 hardware acceleration. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1 –6, August 2006.

[38] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Accelerating double precision FEM simulations with GPUs. In Frank Hülsemann, Markus Kowarschik, and Ulrich Rüde, editors, *Proceedings of the 18th Symposium on Simulation Technique (ASIM 2005)*, pages 139–144. SCS Publishing House e.V, September 2005.

[39] Jim Gray and Catherine van Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft, Dec 2005.

[40] Kris Gray. *Microsoft DirectX 9 Programmable Graphics Pipeline.* Microsoft Press, 2003.

[41] James Lee Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 16–16, Berkeley, CA, USA, 2005. USENIX Association.

[42] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2), April 1950.

[43] Mark Harris. *GPU Gems 2*, chapter Mapping Computational Concepts to GPUs. Addison-Wesley Professional, 2005.

[44] Mark Harris, Shubhabrata Sengupta, and John D. Owens. *GPU Gems 3*, chapter Parallel Prefix Sum (Scan) with CUDA. Addison-Wesley Professional, 2007.

[45] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[46] From a few cores to many: A tera-scale computing research overview. `http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`, 2006.

[47] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. Technical Report 3002, NetApp, Inc., January 1995.

[48] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. pages 23–35, 1992.

[49] Keith Holt. End-to-end data protection justification. `http://www.t10.org/ftp/t10/document.03/03-224r0.pdf`, July 2003.

[50] Walter B. Ligon III and Robert B. Ross. An overview of the parallel virtual file system. In *Proceedings of the 1999 Extreme Linux Workshop*, 1999.

[51] Apple Inc. Mac OS X, Mac OS X server: How to use Apple-supplied RAID software. `http://support.apple.com/kb/HT2559`, 2008.

[52] EM Phonetics Inc. Cula. `http://www.culatools.com/`, September 2010.

[53] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage*, 4(3):1–25, 2008.

[54] Cyndi Jung. Personal communication, 2007.

[55] Randy H. Katz, Peter M Chen, Ann L. C Drapeau, Edward K Lee, K. Lutz, Ethan L. Miller, S. Seshan, and David A. Patterson. RAID-II: Design and implementation of a large scale disk array. Technical report, Berkeley, CA, USA, 1992.

[56] A. Kiayias and Moti Yung. Cryptography and decoding Reed-Solomon codes as a hard problem. In *Theory and Practice in Information-Theoretic Security, 2005. IEEE Information Theory Workshop on*, pages 48–48, Oct. 2005.

[57] G.A. Klutke, P.C. Kiessler, and M.A. Wortman. A critical look at the bathtub curve. *Reliability, IEEE Transactions on*, 52(1):125 – 129, mar. 2003.

[58] M. Ko, M. Chadalapaka, J. Hufferd, U. Elzur, H. Shah, and P. Thaler. Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA). RFC 5046 (Proposed Standard), October 2007.

[59] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM.

[60] Seungbeom Lee, Hanho Lee, Chang-Seok Choi, Jongyoon Shin, and Je-Soo Ko. 40-Gb/s two-parallel Reed-Solomon based forward error correction architecture for optical communications. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 882 –885, nov. 2008.

[61] Adam Leventhal. Triple-parity RAID-Z. `http://blogs.sun.com/ahl/entry/triple_parity_raid_z`, 2009.

[62] Rudolf Lidl and Harald Niedrreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, 1994.

[63] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, and Daniel A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47:569–584, 2001.

[64] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters, July 2004.

[65] Prince McLean. NVIDIA pioneering OpenCL support on top of CUDA. `http://www.appleinsider.com/articles/08/12/10/nvidia_pioneering_opencl_support_on_top_of_cuda.html`, 2008.

[66] Chris Mellor. Rorke's drift towards RAID ASIC replacement. `http://www.channelregister.co.uk/2009/04/16/rorke_nehalem_raid/`, April 2009.

[67] Ashwin A. Mendon, Andrew G. Schmidt, and Ron Sass. A hardware filesystem implementation with multidisk support. *Int. J. Reconfig. Comput.*, 2009:1–1, 2009.

[68] Sun Microsystems. *Solaris ZFS Administration Guide*. October 2009.

[69] Steve Monk. Personal communication, 2010.

[70] Steve Monk and Joe Mervini. Lustre on Red Sky, April 2010.

[71] Aaftab Munshi, editor. *The OpenCL Specification: Version 1.1*. Khronos OpenCL Working Group, June 2010.

[72] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*, pages 145–158, New York, NY, USA, 2009. ACM.

[73] Tim Nufire. Petabytes on a budget: How to build cheap cloud storage. `http://blog.backblaze.com/2009/09/01/petabytes-on-a-budget-how-to-build-cheap-cloud-storage/`, 2009.

[74] NVIDIA. The infinite effects GPU. `http://www.nvidia.com/object/LO_20010612_4376.html`, 2001.

[75] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. Santa Clara, CA, 2007.

[76] NVIDIA Corporation. *NVIDIA CUDA: Compute Unified Device Architecture, Programming Guide (Version 2.1 Beta)*. Santa Clara, CA, 2008.

[77] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Fermi, 2009.

[78] Lars Nyland, Mark Harris, and Jan Prins. *GPU Gems 3*, chapter Fast N-Body Simulation with CUDA. Addison-Wesley Professional, 2007.

[79] Zooko O'Whielacronx. zfec 1.4.6. `http://pypi.python.org/pypi/zfec`.

[80] Vijay Pande. Folding@Home on ATI GPU's: A major step forward. `http://www.stanford.edu/group/pandegroup/folding/FAQ-ATI.html`.

[81] J.-F. Pâris, A. Amer, D.D.E. Long, and T.J.E. Schwarz. Evaluating the impact of irrecoverable read errors on disk array reliability. pages 379 –384, nov. 2009.

[82] Jehan-François Pâris and Darrell D. E. Long. Using device diversity to protect data against batch-correlated disk failures. In *StorageSS '06: Proceedings of the Second ACM Workshop on Storage Security and Survivability*, pages 47–52, New York, NY, USA, 2006. ACM Press.

[83] Suryakant Patidar, Shiben Bhattacharjee, Jag Mohan Singh, and P. J. Narayanan. Exploiting the shader model 4.0 architecture. Technical Report 145, International Institute of Information Technology, Hyderabad, 2007.

[84] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, New York, NY, USA, 1988. ACM.

[85] James C. Phillips and John E. Stone. Probing biomolecular machines with graphics processors. *Communications of the ACM*, (10):34–41, October 2009.

[86] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 17–28, Berkeley, CA, USA, 2007. USENIX Association.

[87] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[88] J. S. Plank. The RAID-6 Liberation codes. In *FAST-2008: 6th USENIX Conference on File and Storage Technologies*, February 2008.

[89] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[90] Michael Potmesil and Eric M. Hoffert. The pixel machine: A parallel image computer. In *SIGGRAPH '89: Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, pages 69–78, New York, NY, USA, 1989. ACM.

[91] Irving S. Reed and Xuemin Chen. *Error-control Coding for Data Networks.* Kluwer Academic Publishers, 1999.

[92] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[93] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The Oceanstore prototype. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2003. USENIX Association.

[94] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[95] Robert A. Rohde and Richard A. Muller. Cycles in fossil diversity. *Nature*, 434:208–210, March 2005.

[96] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[97] Samsung. Consumer class Spinpoint F3EG. `http://www.samsung.com/global/system/business/hdd/prdmodel/2010/2/11/888084f3eg_spec.pdf`.

[98] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Proposed Standard), April 2004. Updated by RFCs 3980, 4850, 5048.

[99] Dave Schreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, seventh edition, 2007.

[100] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 1–1, Berkeley, CA, USA, 2007. USENIX Association.

[101] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[102] P. Sobe and V. Hampel. FPGA-accelerated deletion-tolerant coding for reliable distributed storage. In *ARCS 2007 Proceedings*, pages 14–27, Berlin, 2007. Springer.

[103] Peter Sobe. Parallel Reed/Solomon coding on multicore processors. *Storage Network Architecture and Parallel I/Os, IEEE International Workshop on*, 0:71–80, 2010.

[104] Alexander Thomasian. Multi-level RAID for very large disk arrays. *SIGMETRICS Perform. Eval. Rev.*, 33(4):17–22, 2006.

[105] Linus Torvalds. Summary of changes from v2.6.1 to v2.6.2. `http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.2`.

[106] Robert Walker. Redundancy on a budget: Implementing a RAID file server. `http://technet.microsoft.com/en-us/magazine/2005.05.raid.aspx`, 2008.

[107] H. Lee Ward. Personal communication, 2010.

[108] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 328–337. Springer Berlin / Heidelberg, 2002.

[109] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI*, pages 307–320, 2006.

[110] Jay White and Chris Lueth. RAID-DP: NetApp implementation of double-parity RAID for data protection. Technical Report 3298, NetApp, Inc., May 2010.

[111] M. Woitaszek and H.M. Tufo. Fault tolerance of Tornado Codes for archival storage. In *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 83 –92, 0-0 2006.

[112] Xiotech. ISE - the new foundation of data storage. `http://www.xiotech.com/ise-technology.php`.

[113] Takeshi Yamanouchi. *GPU Gems 3*, chapter AES Encryption and Decryption on the GPU. Addison-Wesley Professional, 2007.

[114] Lin Zhou and Wenbao Han. A brief implementation analysis of SHA-1 on FPGAs, GPUs and Cell processors. In *The Proceedings of the International Conference on Engineering Computation*, pages 101–104, May 2009.

[115] Simon F. Portegies Zwart, Robert G. Belleman, and Peter M. Geldof. High performance direct gravitational n-body simulations on graphics processing units. *New Astronomy*, 13(2):103–112, March 2008.

[116] Jakob Østergaard and Emilio Bueso. The software-RAID HOWTO, v.1.1.1. http://tldp.org/HOWTO/Software-RAID-HOWTO.html, March 2010.

APPENDIX A

APPLICATION PROGRAMMING INTERFACES

This work represents a significant software effort. It required engineering interfaces to provide access to resources in an efficient and easily used way. This appendix documents APIs for software developed in connection with this work.

1. The Gibraltar Library

To address the need for improved RAID implementations, as well as provide this functionality to other software, the Gibraltar library, a C library using NVIDIA's CUDA technology, was created. The Gibraltar library provides data parity and recovery calculations through a generic, flexible API that hides the details of where the computations are being performed. While it does support GPU computation, it can allow for CPU failover, as well as the ability to use alternate computational devices with the same API. It is designed to provide the higher performance of a GPU while also being easy to use. This section provides an overview of the API, along with comments about design decisions.

1.1. gib_init. `int gib_init(int k, int m, gib_context *gc);`

This function initializes the library's runtime to perform $k + m$ codings. The library is capable of performing many different types of codings simultaneously, so this function may be called several times with varying values for k and m. The `gib_context` object is used in future calls in order to identify the type of coding to be performed. As will be verified by examining the further function definitions, the return values are error codes, while the products of the functions are returned by reference.

When this function is called, the GPU is initialized (if necessary), the routines to perform coding and decoding are compiled for the specific values of k and m (if necessary), and the programs are loaded into the GPU for later use.

1.2. gib_destroy. `int gib_destroy(gib_context gc);`

When the user no longer anticipates performing the coding represented by a context, its resources on the GPU can be released for other uses. Once destroyed, `gc` can no longer be used by the program unless it is reinitialized.

1.3. gib_alloc. `int gib_alloc(void **buffers, int buf_size, int *ld, gib_context gc);`

The Gibraltar library works under the assumption that all buffers are allocated end-to-end in main memory, with the start of each buffer separated by `ld`, which is an abbreviation of "leading dimension", bytes, while the first `buf_size` bytes are used. When buffers are placed directly next to each other, `ld` = `buf_size`. In general, `ld` ≥ `buf_size`, and `ld` is used to satisfy some memory alignment constraint. Notice that `ld` is a return value instead of a parameter, as `ld` is determined by the library based on the target device. The amount of total memory allocated is `ld` $\times (k + m)$.

It is not necessary to allocate buffers with this function. However, there are often ways to increase performance depending on the underlying device (e.g. by using specialized memory allocation functions, such as CUDA's `cuMemAllocHost` [**75**]), or by altering the stride of memory accesses during the coding process (as governed by `ld`). For example, with certain methods of CPU coding, performance doubles if `ld` is not divisible by two. However, it is also not necessary to use `ld` as given.

1.4. gib_free. `int gib_free(void *buffers);`

The allocation function is able to use non-standard memory allocation functions, so the user may not know how to appropriately free the memory associated. Furthermore, the use of functions can change over time because of limited resources or the size of allocation. `gib_free` is used to free memory allocated with `gib_alloc`.

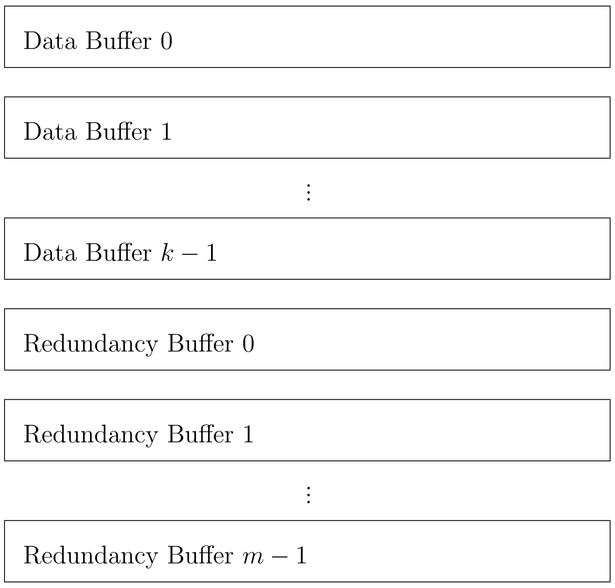


FIGURE 33. Buffer Layout for `gib_generate`

1.5. gib_generate. `int gib_generate(void *buffers, int buf_size, gib_context gc);`

This function performs the encoding of the data buffers into parity buffers according to the parameters set by the `gib_context`. The `buf_size` is the size of one coding buffer, or ld as returned by `gib_alloc`.

The situations where data structures are coded for the sake of coding are rare. Coding operations are usually part of another data moving task, such as writing to files, transfer over networks, etc. Furthermore, in order to improve performance, the library encourages that applications load data into buffers preallocated with `gib_alloc`. To take advantage of these conditions, the library imposes a particular style of layout on the data that it operates upon, which is in the form of a multi-dimensional array embedded within a single array.

Figure 33 illustrates how the layout is interpreted. In short, the buffers are situated in memory such that byte i of data buffer j (i.e., $D_{j,i}$) is at the location `buf[`$j \times ld + i$`]`. Once the routine has been run, the output is situated at the end of the allocated space, such that byte i of parity buffer j (i.e., $C_{j,i}$) is at location `buf[`$(k+j) \times ld + i$`]`.

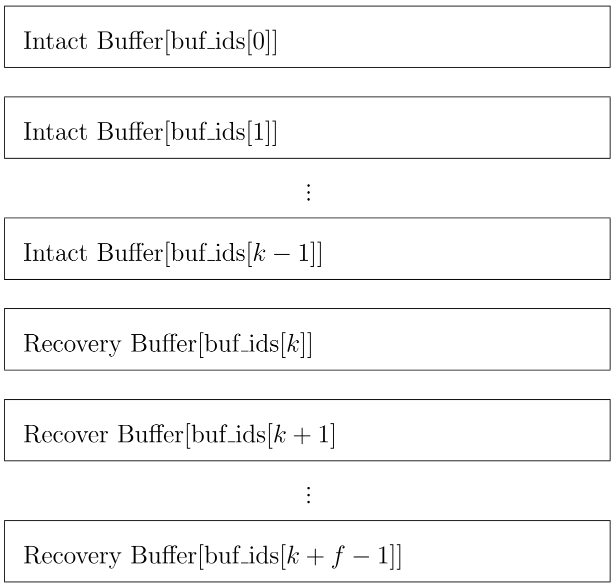

Intact Buffer[buf_ids[0]]
Intact Buffer[buf_ids[1]]
Intact Buffer[buf_ids[$k-1$]]
Recovery Buffer[buf_ids[k]]
Recover Buffer[buf_ids[$k+1$]
Recovery Buffer[buf_ids[$k+f-1$]]

FIGURE 34. Buffer Layout for `gib_recover`

The output can be considered to have been computed by a function RS such that for byte i in all input buffers as follows:

$$RS(\{D_{0,i}, D_{1,i}, \ldots, D_{k,i}\}) = \{C_{0,i}, C_{1,i}, \ldots, C_{m,i}\}$$

1.6. gib_recover. `int gib_recover(void *buffers, int buf_size, int *buf_ids, int recover_last, gib_context gc);`

This function, given k intact buffers, will regenerate the contents up to $f \leq m$ of the remaining buffers, which have presumably lost their data. The `buf_ids` variable contains a list of integers identifying the order of the buffers found in the variable `buffers`.

While `gib_generate` took some liberties in dictating the layout of the buffers in memory, this function is more strict in some ways, less strict in others. Figure 34 indicates the layout required to use `gib_recover`.

In order to use the function, all of the k intact buffers should be positioned in the first $k \times ld$ bytes of `buffers`. The order is not imposed by the library, and they do not have to appear in sorted order. However, `buf_ids` should be populated with the order of the buffers as given.

The last entries in `buf_ids` should be set to the identities of the buffers the application wishes to have recovered. The variable `recover_last` indicates the value f, which is the number of buffers that should be recalculated. When the function returns, the last f buffers will contain the original data.

If it is not necessary to recover m buffers, it is acceptable to pass only $k+f$ entries. However, all buffers should be situated adjacent to each other with no gaps (except constant stride between buffers as provided for by the `ld` parameter.)

2. Gibraltar RAID

A simple API has been provided which allows the RAID infrastructure to interface with a higher-level piece of storage software. The RAID infrastructure interfaces with the Linux SCSI Target Framework. However, one may choose to exercise many other options to provide access to this storage, including writing applications directly to the RAID infrastructure without use of a kernel-level filesystem. To reduce the implementation effort required to use this infrastructure in a new target or driver, a simple API has been provided.

2.1. gr_fopen. `int gr_fopen(char *path, uint64_t *size);`

This function opens a RAID logical unit specification indicated by `path`. This file contains information required to start the array, including component device names, the offset into the array at which the logical unit begins, and the $k+m$ coding used. It initializes all data structures and libraries required, including the Gibraltar library. It returns a logical unit number (LUN) to use for operations on the volume. The size of the volume is stored in the variable `size`, which is sent to the client via the target software.

2.2. gr_pread. `int gr_pread(int fd, void *buf, size_t nbytes, __off64_t offset);`

This function reads from a currently open logical unit. Its interface is identical to the standard C system call `pread`.

2.3. gr_pwrite. `int gr_pwrite(int fd, const void *buf, size_t nbytes, __off64_t offset);`

This function writes to a currently open logical unit. Its interface is identical to the standard C system call `pread`.

2.4. gr_clean. `void gr_clean(int fd);`

This function flushes all data associated with the file descriptor `fd` to the disks in the RAID array.

APPENDIX B

ADDITIONAL DATA

This chapter includes data that was used to gerenate figures in this document. This includes difficult-to-generate data as well as experimental performance data.

Table 2. This table gives the data for Figures 7 and 8 on pages 29 and 30. This data describes the probability of data loss within ten years given varying RAID levels and capacities, with disk size of two terabytes, disk MTTF of one million hours, MTTR of 12 hours, and BER of 10^{-15}.

Cap. (TB)	RAID 6	RAID 6+0 (2 sets)	RAID 6+0 (3 sets)	RAID 6+0 (4 sets)	RAID 5	RAID 5+0 (2 sets)	RAID 5+0 (3 sets)	RAID 5+0 (4 sets)
2	8.82E-08							
4	3.36E-07	1.76E-07			7.29E-03			
6	7.99E-07		2.64E-07		1.39E-02			
8	1.52E-06	6.71E-07		3.53E-07	2.20E-02	1.45E-02		
10	2.54E-06				3.15E-02			
12	3.88E-06	1.60E-06	1.01E-06		4.20E-02	2.75E-02	2.17E-02	
14	5.55E-06				5.34E-02			
16	7.58E-06	3.05E-06		1.34E-06	6.56E-02	4.35E-02		2.88E-02
18	9.96E-06		2.40E-06		7.84E-02		4.10E-02	
20	1.27E-05	5.08E-06			9.16E-02	6.19E-02		
22	1.58E-05				1.05E-01			
24	1.93E-05	7.76E-06	4.57E-06	3.20E-06	1.19E-01	8.23E-02	6.46E-02	5.43E-02
26	2.31E-05				1.33E-01			
28	2.73E-05	1.11E-05			1.47E-01	1.04E-01		
30	3.18E-05		7.62E-06		1.61E-01		9.15E-02	
32	3.66E-05	1.52E-05		6.09E-06	1.75E-01	1.27E-01		8.52E-02
34	4.17E-05				1.89E-01			
36	4.72E-05	1.99E-05	1.16E-05		2.03E-01	1.51E-01	1.21E-01	
38	5.30E-05				2.17E-01			
40	5.90E-05	2.54E-05		1.02E-05	2.31E-01	1.75E-01		1.20E-01
42	6.54E-05		1.67E-05		2.44E-01		1.52E-01	
44	7.20E-05	3.16E-05			2.57E-01	1.99E-01		
46	7.88E-05				2.71E-01			
48	8.59E-05	3.86E-05	2.27E-05	1.55E-05	2.83E-01	2.24E-01	1.84E-01	1.58E-01

Table 2. Continued on next page.

Cap. (TB)	RAID 6	RAID 6+0 (2 sets)	RAID 6+0 (3 sets)	RAID 6+0 (4 sets)	RAID 5	RAID 5+0 (2 sets)	RAID 5+0 (3 sets)	RAID 5+0 (4 sets)
50	9.33E-05				2.96E-01			
52	1.01E-04	4.62E-05			3.09E-01	2.48E-01		
54	1.09E-04		2.99E-05		3.21E-01		2.17E-01	
56	1.17E-04	5.45E-05		2.22E-05	3.33E-01	2.72E-01		1.97E-01
58	1.25E-04				3.45E-01			
60	1.33E-04	6.35E-05	3.81E-05		3.56E-01	2.96E-01	2.50E-01	
62	1.42E-04				3.68E-01			
64	1.51E-04	7.32E-05		3.03E-05	3.79E-01	3.20E-01		2.38E-01
66	1.60E-04		4.74E-05		3.90E-01		2.83E-01	
68	1.69E-04	8.35E-05			4.00E-01	3.42E-01		
70	1.78E-04				4.11E-01			
72	1.88E-04	9.44E-05	5.78E-05	3.99E-05	4.21E-01	3.65E-01	3.16E-01	2.78E-01
74	1.97E-04				4.31E-01			
76	2.07E-04	1.06E-04			4.41E-01	3.87E-01		
78	2.17E-04		6.93E-05		4.51E-01		3.48E-01	
80	2.27E-04	1.18E-04		5.08E-05	4.60E-01	4.08E-01		3.19E-01
82	2.37E-04				4.69E-01			
84	2.47E-04	1.31E-04	8.18E-05		4.78E-01	4.29E-01	3.79E-01	
86	2.58E-04				4.87E-01			
88	2.68E-04	1.44E-04		6.33E-05	4.96E-01	4.49E-01		3.59E-01
90	2.79E-04		9.53E-05		5.04E-01		4.09E-01	
92	2.90E-04	1.58E-04			5.13E-01	4.68E-01		
94	3.01E-04				5.21E-01			
96	3.12E-04	1.72E-04	1.10E-04	7.71E-05	5.29E-01	4.87E-01	4.39E-01	3.97E-01
98	3.23E-04				5.37E-01			
100	3.34E-04	1.87E-04			5.45E-01	5.05E-01		
102	3.45E-04		1.25E-04		5.52E-01		4.67E-01	
104	3.56E-04	2.02E-04		9.24E-05	5.60E-01	5.22E-01		4.35E-01

TABLE 2. Continued on next page.

Cap. (TB)	RAID 6	RAID 6+0 (2 sets)	RAID 6+0 (3 sets)	RAID 6+0 (4 sets)	RAID 5	RAID 5+0 (2 sets)	RAID 5+0 (3 sets)	RAID 5+0 (4 sets)
106	3.68E-04				5.67E-01			
108	3.79E-04	2.17E-04	1.42E-04		5.74E-01	5.39E-01	4.94E-01	
110	3.91E-04				5.81E-01			
112	4.03E-04	2.33E-04		1.09E-04	5.88E-01	5.55E-01		4.70E-01
114	4.14E-04		1.59E-04		5.95E-01		5.20E-01	
116	4.26E-04	2.50E-04			6.02E-01	5.71E-01		
118	4.38E-04				6.08E-01			
120	4.50E-04	2.67E-04	1.77E-04	1.27E-04	6.14E-01	5.86E-01	5.44E-01	5.04E-01
122	4.62E-04				6.21E-01			
124	4.74E-04	2.84E-04			6.27E-01	6.00E-01		
126	4.86E-04		1.96E-04		6.33E-01		5.68E-01	
128	4.99E-04	3.01E-04		1.46E-04	6.39E-01	6.14E-01		5.37E-01
130	5.11E-04				6.45E-01			
132	5.23E-04	3.19E-0438						

End of TABLE 2.

Table 3. This table gives the data for Figure 9. This data describes probability of data loss within ten years with varying capacity and replication (number of sets) within a RAID 1+0 array. The disks size is two terabytes, the MTTF is one million hours, the MTTR is 12 hours, and the BER is 10^{-15}.

Cap. (TB)	2 sets	3 sets	4 sets	Cap. (TB)	2 sets	3 sets	4 sets
2	5.10E-03	1.76E-07	8.12E-12	130	2.83E-01	1.15E-05	5.28E-10
4	1.02E-02	3.53E-07	1.62E-11	132	2.87E-01	1.16E-05	5.36E-10
6	1.52E-02	5.29E-07	2.44E-11	134	2.90E-01	1.18E-05	5.44E-10
8	2.03E-02	7.05E-07	3.25E-11	136	2.94E-01	1.20E-05	5.52E-10
10	2.53E-02	8.82E-07	4.06E-11	138	2.97E-01	1.22E-05	5.60E-10
12	3.02E-02	1.06E-06	4.87E-11	140	3.01E-01	1.23E-05	5.68E-10
14	3.52E-02	1.23E-06	5.68E-11	142	3.05E-01	1.25E-05	5.76E-10
16	4.01E-02	1.41E-06	6.49E-11	144	3.08E-01	1.27E-05	5.84E-10
18	4.50E-02	1.59E-06	7.31E-11	146	3.12E-01	1.29E-05	5.93E-10
20	4.99E-02	1.76E-06	8.12E-11	148	3.15E-01	1.30E-05	6.01E-10
22	5.47E-02	1.94E-06	8.93E-11	150	3.19E-01	1.32E-05	6.09E-10
24	5.95E-02	2.12E-06	9.74E-11	152	3.22E-01	1.34E-05	6.17E-10
26	6.43E-02	2.29E-06	1.06E-10	154	3.26E-01	1.36E-05	6.25E-10
28	6.91E-02	2.47E-06	1.14E-10	156	3.29E-01	1.38E-05	6.33E-10
30	7.39E-02	2.64E-06	1.22E-10	158	3.32E-01	1.39E-05	6.41E-10
32	7.86E-02	2.82E-06	1.30E-10	160	3.36E-01	1.41E-05	6.49E-10
34	8.33E-02	3.00E-06	1.38E-10	162	3.39E-01	1.43E-05	6.58E-10
36	8.80E-02	3.17E-06	1.46E-10	164	3.43E-01	1.45E-05	6.66E-10
38	9.26E-02	3.35E-06	1.54E-10	166	3.46E-01	1.46E-05	6.74E-10
40	9.72E-02	3.53E-06	1.62E-10	168	3.49E-01	1.48E-05	6.82E-10
42	1.02E-01	3.70E-06	1.70E-10	170	3.53E-01	1.50E-05	6.90E-10
44	1.06E-01	3.88E-06	1.79E-10	172	3.56E-01	1.52E-05	6.98E-10
46	1.11E-01	4.05E-06	1.87E-10	174	3.59E-01	1.53E-05	7.06E-10
48	1.16E-01	4.23E-06	1.95E-10	176	3.62E-01	1.55E-05	7.14E-10
50	1.20E-01	4.41E-06	2.03E-10	178	3.66E-01	1.57E-05	7.22E-10
52	1.25E-01	4.58E-06	2.11E-10	180	3.69E-01	1.59E-05	7.31E-10
54	1.29E-01	4.76E-06	2.19E-10	182	3.72E-01	1.60E-05	7.39E-10
56	1.33E-01	4.94E-06	2.27E-10	184	3.75E-01	1.62E-05	7.47E-10
58	1.38E-01	5.11E-06	2.35E-10	186	3.79E-01	1.64E-05	7.55E-10
60	1.42E-01	5.29E-06	2.44E-10	188	3.82E-01	1.66E-05	7.63E-10
62	1.47E-01	5.47E-06	2.52E-10	190	3.85E-01	1.67E-05	7.71E-10
64	1.51E-01	5.64E-06	2.60E-10	192	3.88E-01	1.69E-05	7.79E-10
66	1.55E-01	5.82E-06	2.68E-10	194	3.91E-01	1.71E-05	7.87E-10
68	1.60E-01	5.99E-06	2.76E-10	196	3.94E-01	1.73E-05	7.96E-10
70	1.64E-01	6.17E-06	2.84E-10	198	3.97E-01	1.75E-05	8.04E-10
72	1.68E-01	6.35E-06	2.92E-10	200	4.00E-01	1.76E-05	8.12E-10
74	1.72E-01	6.52E-06	3.00E-10	202	4.03E-01	1.78E-05	8.20E-10

Table 3. Continued on next page.

Cap. (TB)	2 sets	3 sets	4 sets	Cap. (TB)	2 sets	3 sets	4 sets
76	1.77E-01	6.70E-06	3.08E-10	204	4.07E-01	1.80E-05	8.28E-10
78	1.81E-01	6.88E-06	3.17E-10	206	4.10E-01	1.82E-05	8.36E-10
80	1.85E-01	7.05E-06	3.25E-10	208	4.13E-01	1.83E-05	8.44E-10
82	1.89E-01	7.23E-06	3.33E-10	210	4.16E-01	1.85E-05	8.52E-10
84	1.93E-01	7.40E-06	3.41E-10	212	4.19E-01	1.87E-05	8.60E-10
86	1.97E-01	7.58E-06	3.49E-10	214	4.21E-01	1.89E-05	8.69E-10
88	2.02E-01	7.76E-06	3.57E-10	216	4.24E-01	1.90E-05	8.77E-10
90	2.06E-01	7.93E-06	3.65E-10	218	4.27E-01	1.92E-05	8.85E-10
92	2.10E-01	8.11E-06	3.73E-10	220	4.30E-01	1.94E-05	8.93E-10
94	2.14E-01	8.29E-06	3.82E-10	222	4.33E-01	1.96E-05	9.01E-10
96	2.18E-01	8.46E-06	3.90E-10	224	4.36E-01	1.97E-05	9.09E-10
98	2.22E-01	8.64E-06	3.98E-10	226	4.39E-01	1.99E-05	9.17E-10
100	2.26E-01	8.82E-06	4.06E-10	228	4.42E-01	2.01E-05	9.25E-10
102	2.30E-01	8.99E-06	4.14E-10	230	4.45E-01	2.03E-05	9.34E-10
104	2.34E-01	9.17E-06	4.22E-10	232	4.48E-01	2.05E-05	9.42E-10
106	2.37E-01	9.34E-06	4.30E-10	234	4.50E-01	2.06E-05	9.50E-10
108	2.41E-01	9.52E-06	4.38E-10	236	4.53E-01	2.08E-05	9.58E-10
110	2.45E-01	9.70E-06	4.46E-10	238	4.56E-01	2.10E-05	9.66E-10
112	2.49E-01	9.87E-06	4.55E-10	240	4.59E-01	2.12E-05	9.74E-10
114	2.53E-01	1.00E-05	4.63E-10	242	4.61E-01	2.13E-05	9.82E-10
116	2.57E-01	1.02E-05	4.71E-10	244	4.64E-01	2.15E-05	9.90E-10
118	2.61E-01	1.04E-05	4.79E-10	246	4.67E-01	2.17E-05	9.98E-10
120	2.64E-01	1.06E-05	4.87E-10	248	4.70E-01	2.19E-05	1.01E-09
122	2.68E-01	1.08E-05	4.95E-10	250	4.72E-01	2.20E-05	1.01E-09
124	2.72E-01	1.09E-05	5.03E-10	252	4.75E-01	2.22E-05	1.02E-09
126	2.75E-01	1.11E-05	5.11E-10	254	4.78E-01	2.24E-05	1.03E-09
128	2.79E-01	1.13E-05	5.20E-10	256	4.80E-01	2.26E-05	1.04E-09

End of Table 3.

Table 4. This table gives the data for Figure 10 on page 32. It describes the probability that data will be lost within 10 years if the disk size is two terabytes, the MTTF is one million hours, the BER is 10^{-15}, and the MTTR of the array is 12 hours.

Cap. (TB)	RAID 5+0	RAID 6+0	RAID 1+0	RAID 1+0	RAID 1+0	RAID $k+3$	RAID $k+4$	RAID $k+5$
	(4 sets)	(4 sets)	(2 sets)	(3 sets)	(4 sets)			
2			5.10E-03	1.76E-07	5.10E-03	4.06E-12		
4			1.02E-02	3.53E-07	1.02E-02	1.93E-11	1.34E-15	1.08E-19
6			1.52E-02	5.29E-07	1.52E-02	5.52E-11	4.46E-15	4.12E-19
8		3.53E-07	2.03E-02	7.05E-07	2.03E-02	1.23E-10	1.13E-14	1.18E-18
10			2.53E-02	8.82E-07	2.53E-02	2.35E-10	2.44E-14	2.82E-18
12			3.02E-02	1.06E-06	3.02E-02	4.03E-10	4.66E-14	5.92E-18
14			3.52E-02	1.23E-06	3.52E-02	6.42E-10	8.16E-14	1.13E-17
16	2.88E-02	1.34E-06	4.01E-02	1.41E-06	4.01E-02	9.64E-10	1.34E-13	2.01E-17
18			4.50E-02	1.59E-06	4.50E-02	1.38E-09	2.08E-13	3.38E-17
20			4.99E-02	1.76E-06	4.99E-02	1.91E-09	3.10E-13	5.39E-17
22			5.47E-02	1.94E-06	5.47E-02	2.56E-09	4.46E-13	8.27E-17
24	5.43E-02	3.20E-06	5.95E-02	2.12E-06	5.95E-02	3.35E-09	6.22E-13	1.23E-16
26			6.43E-02	2.29E-06	6.43E-02	4.28E-09	8.45E-13	1.77E-16
28			6.91E-02	2.47E-06	6.91E-02	5.38E-09	1.12E-12	2.48E-16
30			7.39E-02	2.64E-06	7.39E-02	6.64E-09	1.46E-12	3.40E-16
32	8.52E-02	6.09E-06	7.86E-02	2.82E-06	7.86E-02	8.08E-09	1.88E-12	4.58E-16
34			8.33E-02	3.00E-06	8.33E-02	9.70E-09	2.37E-12	6.06E-16
36			8.80E-02	3.17E-06	8.80E-02	1.15E-08	2.95E-12	7.90E-16
38			9.26E-02	3.35E-06	9.26E-02	1.36E-08	3.63E-12	1.01E-15
40	1.20E-01	1.02E-05	9.72E-02	3.53E-06	9.72E-02	1.58E-08	4.42E-12	1.29E-15
42			1.02E-01	3.70E-06	1.02E-01	1.83E-08	5.32E-12	1.61E-15
44			1.06E-01	3.88E-06	1.06E-01	2.10E-08	6.35E-12	2.00E-15
46			1.11E-01	4.05E-06	1.11E-01	2.39E-08	7.52E-12	2.46E-15
48	1.58E-01	1.55E-05	1.16E-01	4.23E-06	1.16E-01	2.71E-08	8.84E-12	2.99E-15

Table 4. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
50			1.20E-01	4.41E-06	1.20E-01	3.05E-08	1.03E-11	3.62E-15
52			1.25E-01	4.58E-06	1.25E-01	3.42E-08	1.20E-11	4.34E-15
54			1.29E-01	4.76E-06	1.29E-01	3.81E-08	1.38E-11	5.16E-15
56	1.97E-01	2.22E-05	1.33E-01	4.94E-06	1.33E-01	4.23E-08	1.58E-11	6.11E-15
58			1.38E-01	5.11E-06	1.38E-01	4.67E-08	1.80E-11	7.18E-15
60			1.42E-01	5.29E-06	1.42E-01	5.15E-08	2.05E-11	8.39E-15
62			1.47E-01	5.47E-06	1.47E-01	5.65E-08	2.31E-11	9.76E-15
64	2.38E-01	3.03E-05	1.51E-01	5.64E-06	1.51E-01	6.18E-08	2.60E-11	1.13E-14
66			1.55E-01	5.82E-06	1.55E-01	6.73E-08	2.92E-11	1.30E-14
68			1.60E-01	5.99E-06	1.60E-01	7.32E-08	3.26E-11	1.49E-14
70			1.64E-01	6.17E-06	1.64E-01	7.93E-08	3.63E-11	1.70E-14
72	2.78E-01	3.99E-05	1.68E-01	6.35E-06	1.68E-01	8.58E-08	4.03E-11	1.94E-14
74			1.72E-01	6.52E-06	1.72E-01	9.25E-08	4.45E-11	2.20E-14
76			1.77E-01	6.70E-06	1.77E-01	9.96E-08	4.91E-11	2.48E-14
78			1.81E-01	6.88E-06	1.81E-01	1.07E-07	5.40E-11	2.79E-14
80	3.19E-01	5.08E-05	1.85E-01	7.05E-06	1.85E-01	1.15E-07	5.92E-11	3.13E-14
82			1.89E-01	7.23E-06	1.89E-01	1.23E-07	6.48E-11	3.50E-14
84			1.93E-01	7.40E-06	1.93E-01	1.31E-07	7.07E-11	3.91E-14
86			1.97E-01	7.58E-06	1.97E-01	1.39E-07	7.70E-11	4.35E-14
88	3.59E-01	6.33E-05	2.02E-01	7.76E-06	2.02E-01	1.48E-07	8.37E-11	4.83E-14
90			2.06E-01	7.93E-06	2.06E-01	1.57E-07	9.08E-11	5.34E-14
92			2.10E-01	8.11E-06	2.10E-01	1.67E-07	9.83E-11	5.90E-14
94			2.14E-01	8.29E-06	2.14E-01	1.77E-07	1.06E-10	6.50E-14
96	3.97E-01	7.71E-05	2.18E-01	8.46E-06	2.18E-01	1.87E-07	1.15E-10	7.15E-14
98			2.22E-01	8.64E-06	2.22E-01	1.98E-07	1.23E-10	7.85E-14
100			2.26E-01	8.82E-06	2.26E-01	2.08E-07	1.33E-10	8.60E-14
102			2.30E-01	8.99E-06	2.30E-01	2.20E-07	1.42E-10	9.40E-14
104	4.35E-01	9.24E-05	2.34E-01	9.17E-06	2.34E-01	2.31E-07	1.53E-10	1.03E-13

TABLE 4. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
106			2.37E-01	9.34E-06	2.37E-01	2.43E-07	1.63E-10	1.12E-13
108			2.41E-01	9.52E-06	2.41E-01	2.55E-07	1.74E-10	1.21E-13
110			2.45E-01	9.70E-06	2.45E-01	2.67E-07	1.86E-10	1.32E-13
112	4.70E-01	1.09E-04	2.49E-01	9.87E-06	2.49E-01	2.80E-07	1.98E-10	1.43E-13
114			2.53E-01	1.00E-05	2.53E-01	2.93E-07	2.11E-10	1.55E-13
116			2.57E-01	1.02E-05	2.57E-01	3.07E-07	2.25E-10	1.67E-13
118			2.61E-01	1.04E-05	2.61E-01	3.21E-07	2.39E-10	1.80E-13
120	5.04E-01	1.27E-04	2.64E-01	1.06E-05	2.64E-01	3.35E-07	2.53E-10	1.94E-13
122			2.68E-01	1.08E-05	2.68E-01	3.49E-07	2.68E-10	2.09E-13
124			2.72E-01	1.09E-05	2.72E-01	3.64E-07	2.84E-10	2.25E-13
126			2.75E-01	1.11E-05	2.75E-01	3.79E-07	3.00E-10	2.42E-13
128	5.37E-01	1.46E-04	2.79E-01	1.13E-05	2.79E-01	3.95E-07	3.17E-10	2.59E-13
130			2.83E-01	1.15E-05	2.83E-01	4.11E-07	3.35E-10	2.77E-13
132			2.87E-01	1.16E-05	2.87E-01	4.27E-07	3.53E-10	2.97E-13
134			2.90E-01	1.18E-05	2.90E-01	4.44E-07	3.73E-10	3.17E-13
136	5.68E-01	1.67E-04	2.94E-01	1.20E-05	2.94E-01	4.60E-07	3.92E-10	3.39E-13
138			2.97E-01	1.22E-05	2.97E-01	4.78E-07	4.13E-10	3.61E-13
140			3.01E-01	1.23E-05	3.01E-01	4.95E-07	4.34E-10	3.85E-13
142			3.05E-01	1.25E-05	3.05E-01	5.13E-07	4.56E-10	4.10E-13
144	5.97E-01	1.89E-04	3.08E-01	1.27E-05	3.08E-01	5.32E-07	4.78E-10	4.36E-13
146			3.12E-01	1.29E-05	3.12E-01	5.50E-07	5.02E-10	4.63E-13
148			3.15E-01	1.30E-05	3.15E-01	5.69E-07	5.26E-10	4.92E-13
150			3.19E-01	1.32E-05	3.19E-01	5.88E-07	5.51E-10	5.22E-13
152	6.24E-01	2.12E-04	3.22E-01	1.34E-05	3.22E-01	6.08E-07	5.76E-10	5.53E-13
154			3.26E-01	1.36E-05	3.26E-01	6.28E-07	6.03E-10	5.86E-13
156			3.29E-01	1.38E-05	3.29E-01	6.49E-07	6.30E-10	6.20E-13
158			3.32E-01	1.39E-05	3.32E-01	6.69E-07	6.58E-10	6.55E-13
160	6.49E-01	2.36E-04	3.36E-01	1.41E-05	3.36E-01	6.90E-07	6.87E-10	6.92E-13

TABLE 4. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
162			3.39E-01	1.43E-05	3.39E-01	7.12E-07	7.17E-10	7.31E-13
164			3.43E-01	1.45E-05	3.43E-01	7.33E-07	7.48E-10	7.71E-13
166			3.46E-01	1.46E-05	3.46E-01	7.56E-07	7.79E-10	8.13E-13
168	6.73E-01	2.61E-04	3.49E-01	1.48E-05	3.49E-01	7.78E-07	8.12E-10	8.57E-13
170			3.53E-01	1.50E-05	3.53E-01	8.01E-07	8.45E-10	9.02E-13
172			3.56E-01	1.52E-05	3.56E-01	8.24E-07	8.80E-10	9.49E-13
174			3.59E-01	1.53E-05	3.59E-01	8.47E-07	9.15E-10	9.98E-13
176	6.96E-01	2.88E-04	3.62E-01	1.55E-05	3.62E-01	8.71E-07	9.51E-10	1.05E-12
178			3.66E-01	1.57E-05	3.66E-01	8.95E-07	9.88E-10	1.10E-12
180			3.69E-01	1.59E-05	3.69E-01	9.20E-07	1.03E-09	1.16E-12
182			3.72E-01	1.60E-05	3.72E-01	9.45E-07	1.07E-09	1.21E-12
184	7.17E-01	3.15E-04	3.75E-01	1.62E-05	3.75E-01	9.70E-07	1.11E-09	1.27E-12
186			3.79E-01	1.64E-05	3.79E-01	9.96E-07	1.15E-09	1.33E-12
188			3.82E-01	1.66E-05	3.82E-01	1.02E-06	1.19E-09	1.40E-12
190			3.85E-01	1.67E-05	3.85E-01	1.05E-06	1.23E-09	1.46E-12
192	7.36E-01	3.44E-04	3.88E-01	1.69E-05	3.88E-01	1.07E-06	1.28E-09	1.53E-12
194			3.91E-01	1.71E-05	3.91E-01	1.10E-06	1.32E-09	1.60E-12
196			3.94E-01	1.73E-05	3.94E-01	1.13E-06	1.37E-09	1.67E-12
198			3.97E-01	1.75E-05	3.97E-01	1.16E-06	1.41E-09	1.75E-12
200	7.55E-01	3.73E-04	4.00E-01	1.76E-05	4.00E-01	1.18E-06	1.46E-09	1.82E-12
202			4.03E-01	1.78E-05	4.03E-01	1.21E-06	1.51E-09	1.90E-12
204			4.07E-01	1.80E-05	4.07E-01	1.24E-06	1.56E-09	1.99E-12
206			4.10E-01	1.82E-05	4.10E-01	1.27E-06	1.61E-09	2.07E-12
208	7.71E-01	4.03E-04	4.13E-01	1.83E-05	4.13E-01	1.30E-06	1.67E-09	2.16E-12
210			4.16E-01	1.85E-05	4.16E-01	1.33E-06	1.72E-09	2.25E-12
212			4.19E-01	1.87E-05	4.19E-01	1.36E-06	1.78E-09	2.34E-12
214			4.21E-01	1.89E-05	4.21E-01	1.39E-06	1.83E-09	2.44E-12
216	7.87E-01	4.35E-04	4.24E-01	1.90E-05	4.24E-01	1.42E-06	1.89E-09	2.54E-12

TABLE 4. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
218			4.27E-01	1.92E-05	4.27E-01	1.45E-06	1.95E-09	2.64E-12
220			4.30E-01	1.94E-05	4.30E-01	1.48E-06	2.01E-09	2.75E-12
222			4.33E-01	1.96E-05	4.33E-01	1.51E-06	2.07E-09	2.85E-12
224	8.02E-01	4.67E-04	4.36E-01	1.97E-05	4.36E-01	1.55E-06	2.13E-09	2.97E-12
226			4.39E-01	1.99E-05	4.39E-01	1.58E-06	2.20E-09	3.08E-12
228			4.42E-01	2.01E-05	4.42E-01	1.61E-06	2.26E-09	3.20E-12
230			4.45E-01	2.03E-05	4.45E-01	1.64E-06	2.33E-09	3.32E-12
232	8.16E-01	5.00E-04	4.48E-01	2.05E-05	4.48E-01	1.68E-06	2.39E-09	3.44E-12
234			4.50E-01	2.06E-05	4.50E-01	1.71E-06	2.46E-09	3.57E-12
236			4.53E-01	2.08E-05	4.53E-01	1.75E-06	2.53E-09	3.70E-12
238			4.56E-01	2.10E-05	4.56E-01	1.78E-06	2.60E-09	3.84E-12
240	8.28E-01	5.33E-04	4.59E-01	2.12E-05	4.59E-01	1.81E-06	2.68E-09	3.98E-12
242			4.61E-01	2.13E-05	4.61E-01	1.85E-06	2.75E-09	4.12E-12
244			4.64E-01	2.15E-05	4.64E-01	1.88E-06	2.83E-09	4.27E-12
246			4.67E-01	2.17E-05	4.67E-01	1.92E-06	2.90E-09	4.42E-12
248	8.40E-01		4.70E-01	2.19E-05	4.70E-01	1.96E-06	2.98E-09	
250			4.72E-01	2.20E-05	4.72E-01	1.99E-06		
252			4.75E-01	2.22E-05	4.75E-01			
254			4.78E-01	2.24E-05	4.78E-01			
256			4.80E-01	2.26E-05	4.80E-01			

End of TABLE 4.

TABLE 5. This table gives the data for Figure 11 on page 33. It describes the probability that data will be lost within 10 years if the disk size is two terabytes, the MTTF is 100,000 hours, the BER is 10^{-15}, and the MTTR of the array is one week.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID k+3	RAID k+4	RAID k+5	
2			0.000109248	5.74E-07	2.87E-07	1.96E-09	1.65E-11	
4			0.000218484	1.15E-06	1.16E-06	9.76E-09	9.82E-11	
6			0.000327709	1.72E-06	2.89E-06	2.91E-08	3.41E-10	
8		0.000218484	0.000436921	2.30E-06	5.75E-06	6.73E-08	8.99E-10	
10			0.000546121	2.87E-06	9.99E-06	1.33E-07	2.00E-09	
12			0.00065531	3.44E-06	1.58E-05	2.38E-07	3.96E-09	
14			0.000764486	4.02E-06	2.36E-05	3.93E-07	7.19E-09	
16	0.122866	0.000679516	0.000873651	4.59E-06	3.34E-05	6.11E-07	1.22E-08	
18			0.000982804	5.17E-06	4.55E-05	9.08E-07	1.96E-08	
20			0.00109194	5.74E-06	6.01E-05	1.30E-06	3.02E-08	
22			0.00120107	6.32E-06	7.75E-05	1.80E-06	4.48E-08	
24	0.185878	0.00137204	0.00131019	6.89E-06	9.77E-05	2.43E-06	6.45E-08	
26			0.0014193	7.46E-06	0.000121127	3.21E-06	9.05E-08	
28			0.00152839	8.04E-06	0.000147791	4.16E-06	1.24E-07	
30			0.00163747	8.61E-06	0.000177914	5.30E-06	1.66E-07	
32	0.243644	0.00228438	0.00174654	9.19E-06	0.000211665	6.65E-06	2.20E-07	
34			0.0018556	9.76E-06	0.000249198	8.23E-06	2.85E-07	
36			0.00196464	1.03E-05	0.000290668	1.01E-05	3.66E-07	
38			0.00207368	1.09E-05	0.000336222	1.22E-05	4.62E-07	
40	0.296501	0.0034079	0.0021827	1.15E-05	0.000386004	1.46E-05	5.78E-07	
42			0.00229171	1.21E-05	0.000440141	1.74E-05	7.15E-07	
44			0.0024007	1.26E-05	0.000498765	2.05E-05	8.77E-07	

TABLE 5. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
46			0.00250969	1.32E-05	0.000562	2.40E-05	1.07E-06	
48	0.345152	0.00473635	0.00261866	1.38E-05	0.000629967	2.79E-05	1.28E-06	
50			0.00272763	1.44E-05	0.00070277	3.23E-05	1.54E-06	
52			0.00283658	1.49E-05	0.000780521	3.71E-05	1.82E-06	
54			0.00294551	1.55E-05	0.00086332	4.24E-05	2.15E-06	
56	0.39016	0.00626423	0.00305444	1.61E-05	0.000951273	4.83E-05	2.53E-06	
58			0.00316336	1.67E-05	0.00104446	5.47E-05	2.95E-06	
60			0.00327226	1.72E-05	0.00114298	6.16E-05	3.42E-06	
62			0.00338115	1.78E-05	0.0012469	6.92E-05	3.95E-06	
64	0.431944	0.00798642	0.00349003	1.84E-05	0.00135633	7.75E-05	4.55E-06	
66			0.00359889	1.89E-05	0.00147131	8.64E-05	5.20E-06	
68			0.00370775	1.95E-05	0.00159194	9.60E-05	5.93E-06	
70			0.00381659	2.01E-05	0.00171827	0.000106284	6.74E-06	
72	0.47079	0.00989718	0.00392542	2.07E-05	0.00185038	0.000117372	7.63E-06	
74			0.00403424	2.12E-05	0.00198832	0.00012925	8.60E-06	
76			0.00414305	2.18E-05	0.00213214	0.000141948	9.67E-06	
78			0.00425185	2.24E-05	0.00228191	0.000155497	1.08E-05	
80	0.506949	0.0119913	0.00436063	2.30E-05	0.00243767	0.000169928	1.21E-05	
82			0.0044694	2.35E-05	0.00259947	0.000185271	1.35E-05	
84			0.00457816	2.41E-05	0.00276735	0.000201555	1.50E-05	
86			0.00468691	2.47E-05	0.00294136	0.000218811	1.66E-05	
88	0.540623	0.0142633	0.00479564	2.53E-05	0.00312154	0.000237071	1.84E-05	
90			0.00490437	2.58E-05	0.00330791	0.000256363	2.03E-05	
92			0.00501308	2.64E-05	0.00350051	0.000276718	2.23E-05	
94			0.00512178	2.70E-05	0.00369938	0.000298166	2.45E-05	
96	0.571999	0.0167081	0.00523047	2.76E-05	0.00390455	0.000320737	2.68E-05	
98			0.00533915	2.81E-05	0.00411602	0.000344461	2.93E-05	
100			0.00544781	2.87E-05	0.00433383	0.000369367	3.20E-05	

TABLE 5. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
102			0.00555646	2.93E-05	0.004558	0.000395484	3.49E-05	
104	0.601222	0.0193197	0.00566511	2.99E-05	0.00478857	0.000422845	3.80E-05	
106			0.00577374	3.04E-05	0.00502551	0.000451474	4.12E-05	
108			0.00588235	3.10E-05	0.00526887	0.000481404	4.47E-05	
110			0.00599096	3.16E-05	0.00551865	0.000512662	4.84E-05	
112	0.628449	0.022093	0.00609955	3.22E-05	0.00577488	0.000545279	5.23E-05	
114			0.00620813	3.27E-05	0.00603754	0.00057928	5.64E-05	
116			0.0063167	3.33E-05	0.00630665	0.000614695	6.08E-05	
118			0.00642526	3.39E-05	0.00658221	0.000651553	6.54E-05	
120	0.653817	0.0250227	0.00653381	3.44E-05	0.00686425	0.000689883	7.03E-05	
122			0.00664234	3.50E-05	0.00715274	0.000729709	7.55E-05	
124			0.00675086	3.56E-05	0.00744769	0.00077106	8.09E-05	
126			0.00685938	3.62E-05	0.0077491	0.000813965	8.67E-05	
128	0.677457	0.0281039	0.00696787	3.67E-05	0.008057	0.00085845	9.27E-05	
130			0.00707636	3.73E-05	0.00837133	0.000904541	9.90E-05	
132			0.00718484	3.79E-05	0.00869211	0.000952264	0.000105663	
134			0.0072933	3.85E-05	0.00901935	0.00100165	0.000112638	
136	0.699478	0.0313308	0.00740175	3.90E-05	0.00935304	0.00105272	0.00011995	
138			0.00751019	3.96E-05	0.00969315	0.0011055	0.000127609	
140			0.00761862	4.02E-05	0.0100397	0.00116001	0.000135625	
142			0.00772703	4.08E-05	0.0103926	0.00121629	0.000144009	
144	0.719996	0.0346986	0.00783544	4.13E-05	0.010752	0.00127436	0.000152772	
146			0.00794383	4.19E-05	0.0111177	0.00133423	0.000161922	
148			0.00805221	4.25E-05	0.0114898	0.00139594	0.000171472	
150			0.00816058	4.31E-05	0.0118683	0.00145952	0.000181432	
152	0.739113	0.0382023	0.00826893	4.36E-05	0.0122531	0.00152498	0.000191812	
154			0.00837728	4.42E-05	0.0126443	0.00159234	0.000202624	
156			0.00848561	4.48E-05	0.0130417	0.00166164	0.000213879	

TABLE 5. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
158			0.00859393	4.54E-05	0.0134455	0.00173289	0.000225587	
160	0.756929	0.0418373	0.00870224	4.59E-05	0.0138555	0.00180612	0.00023776	
162			0.00881054	4.65E-05	0.0142718	0.00188135	0.000250408	
164			0.00891883	4.71E-05	0.0146944	0.0019586	0.000263543	
166			0.0090271	4.77E-05	0.0151231	0.0020379	0.000277177	
168	0.773524	0.045598	0.00913536	4.82E-05	0.0155581	0.00211926	0.000291321	
170			0.00924361	4.88E-05	0.0159992	0.00220271	0.000305985	
172			0.00935185	4.94E-05	0.0164465	0.00228827	0.000321182	
174			0.00946008	5.00E-05	0.0168999	0.00237596	0.000336923	
176	0.788986	0.04948	0.00956829	5.05E-05	0.0173595	0.00246581	0.00035322	
178			0.00967649	5.11E-05	0.0178251	0.00255782	0.000370084	
180			0.00978469	5.17E-05	0.0182968	0.00265202	0.000387527	
182			0.00989286	5.22E-05	0.0187745	0.00274844	0.000405561	
184	0.803393	0.0534786	0.010001	5.28E-05	0.0192583	0.00284708	0.000424198	
186			0.0101092	5.34E-05	0.019748	0.00294798	0.000443449	
188			0.0102173	5.40E-05	0.0202437	0.00305115	0.000463327	
190			0.0103255	5.45E-05	0.0207453	0.0031566	0.000483842	
192	0.816819	0.0575896	0.0104336	5.51E-05	0.0212528	0.00326436	0.000505008	
194			0.0105417	5.57E-05	0.0217663	0.00337445	0.000526836	
196			0.0106498	5.63E-05	0.0222855	0.00348688	0.000549338	
198			0.0107579	5.68E-05	0.0228106	0.00360167	0.000572527	
200	0.829325	0.0618076	0.0108659	5.74E-05	0.0233416	0.00371885	0.000596414	
202			0.010974	5.80E-05	0.0238783	0.00383841	0.000621011	
204			0.0110821	5.86E-05	0.0244207	0.00396039	0.000646331	
206			0.0111901	5.91E-05	0.0249689	0.00408481	0.000672386	
208	0.840978	0.0661287	0.0112981	5.97E-05	0.0255227	0.00421167	0.000699188	
210			0.0114061	6.03E-05	0.0260822	0.00434099	0.000726748	
212			0.0115141	6.09E-05	0.0266474	0.00447279	0.000755081	

TABLE 5. Continued on next page.

Cap. (TB)	RAID 5+0 (4 sets)	RAID 6+0 (4 sets)	RAID 1+0 (2 sets)	RAID 1+0 (3 sets)	RAID 1+0 (4 sets)	RAID $k+3$	RAID $k+4$	RAID $k+5$
214			0.0116221	6.14E-05	0.0272181	0.00460708	0.000784196	
216	0.851835	0.0705485	0.0117301	6.20E-05	0.0277945	0.00474389	0.000814109	
218			0.0118381	6.26E-05	0.0283764	0.00488322	0.000844829	
220			0.011946	6.32E-05	0.0289638	0.00502509	0.000876369	
222			0.012054	6.37E-05	0.0295566	0.00516952	0.000908743	
224	0.861953	0.0750632	0.0121619	6.43E-05	0.030155	0.00531652	0.000941962	
226			0.0122698	6.49E-05	0.0307588	0.0054661	0.000976038	
228			0.0123777	6.55E-05	0.031368	0.00561829	0.00101098	
230			0.0124856	6.60E-05	0.0319825	0.00577308	0.00104681	
232	0.871378	0.0796676	0.0125935	6.66E-05	0.0326024	0.00593051	0.00108354	
234			0.0127014	6.72E-05	0.0332276	0.00609057	0.00112116	
236			0.0128092	6.77E-05	0.0338581	0.00625329	0.00115971	
238			0.0129171	6.83E-05	0.0344939	0.00641867	0.00119919	
240	0.880159	0.0843583	0.0130249	6.89E-05	0.0351349	0.00658674	0.00123962	
242			0.0131328	6.95E-05	0.035781	0.00675749	0.001281	
244			0.0132406	7.00E-05	0.0364323	0.00693095	0.00132335	
246			0.0133484	7.06E-05	0.0370888	0.00710713	0.00136668	
248	0.888341		0.0134562	7.12E-05	0.0377504	0.00728604		
250			0.0135639	7.18E-05	0.038417			
252			0.0136717	7.23E-05				
254			0.0137795	7.29E-05				
256			0.0138872	7.35E-05				

End of TABLE 5.

	m				
k	2	3	4	5	6
2	2309.38	1717.56	1322.04	1047.34	864.98
3	2813.71	2086.78	1618.79	1285.67	1071.28
4	3132.76	2405.06	1789.37	1423.89	1204.17
5	3323.53	2603.08	1959.55	1539.48	1295.86
6	3454.28	2756.08	2080.85	1616.03	1356.79
7	3600.79	2832.18	2152.40	1689.67	1428.91
8	3751.27	2932.56	2190.92	1730.41	1464.99
9	3808.38	3026.82	2276.07	1756.99	1489.61
10	3807.74	3100.18	2331.64	1815.15	1514.00
11	3955.39	3074.02	2347.52	1830.46	1546.59
12	3986.72	3182.68	2360.06	1843.88	1549.90
13	3982.61	3185.82	2400.29	1848.01	1561.14
14	3955.93	3237.88	2445.41	1871.31	1567.36
15	4050.53	3237.07	2437.51	1881.39	1603.31
16	4044.45	3293.68	2443.71	1887.14	1601.56

TABLE 6. The Gibraltar Library Performance on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 2 \ldots 6$, with Buffer Size of 1 MB, as Reflected in Figure 17

	m				
k	7	8	9	10	11
2	763.18	676.77	600.70	533.73	492.73
3	916.36	822.09	727.62	648.59	589.94
4	1051.19	919.75	818.29	723.04	660.76
5	1128.05	997.76	850.59	776.51	706.37
6	1188.53	1050.74	891.65	806.54	720.01
7	1230.74	1089.18	965.73	870.01	765.37
8	1281.34	1072.32	984.87	858.44	790.13
9	1299.83	1096.65	942.76	848.38	796.64
10	1323.76	1107.57	1018.89	868.71	798.12
11	1350.08	1165.60	1023.42	867.48	799.99
12	1358.84	1172.33	973.39	851.83	802.57
13	1381.24	1187.53	1055.48	897.45	805.88
14	1370.69	1174.83	1029.63	879.48	805.57
15	1370.56	1187.10	1037.68	921.28	810.10
16	1382.91	1198.96	1029.38	872.06	790.13

TABLE 7. The Gibraltar Library Performance on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 7 \ldots 11$, with Buffer Size of 1 MB, as Reflected in Figure 17

	m				
k	12	13	14	15	16
2	452.93	415.02	385.83	365.71	341.61
3	539.97	499.69	466.14	435.59	408.57
4	605.65	553.94	516.47	486.03	453.26
5	645.89	591.13	552.13	515.45	483.81
6	672.24	618.13	574.43	538.70	509.30
7	696.90	631.09	586.96	551.70	527.37
8	723.35	636.21	598.62	563.19	530.44
9	737.70	661.07	607.60	564.73	524.69
10	736.32	682.86	597.50	576.24	531.00
11	711.15	660.93	604.96	569.09	523.51
12	732.54	660.62	609.60	561.21	519.13
13	718.64	655.20	596.73	571.55	524.49
14	724.17	661.73	589.17	551.52	509.89
15	732.71	652.51	595.41	556.15	512.24
16	720.72	655.08	597.42	543.40	503.67

Table 8. The Gibraltar Library Performance on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 12 \ldots 16$, with Buffer Size of 1 MB, as Reflected in Figure 17

	m				
k	2	3	4	5	6
2	2778.42	2179.54	1790.53	1518.63	1320.13
3	3496.42	2794.34	2327.76	2053.95	1806.59
4	3785.82	3287.33	2757.87	2463.69	2174.12
5	3836.02	3289.96	2880.86	2541.69	2296.14
6	4093.34	3547.78	3146.59	2858.78	2622.15
7	4146.44	3658.04	3327.30	3020.34	2813.03
8	4191.48	3788.29	3501.97	3249.89	2968.68
9	4196.54	3833.37	3535.78	3289.25	3061.04
10	4360.35	3951.00	3644.88	3423.47	3194.88
11	4360.23	4020.75	3730.80	3487.01	3273.59
12	4434.59	4067.80	3815.37	3599.95	3382.19
13	4405.00	4079.33	3843.65	3657.64	3454.51
14	4449.25	4125.41	3878.54	3675.30	3484.18
15	4429.48	4173.88	3914.40	3735.62	3543.29
16	4477.27	4201.24	3988.26	3780.19	3609.63

Table 9. PCI-Express Throughput for Coding Workloads on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 2 \ldots 6$, with Buffer Size of 1 MB, as Reflected in Figures 18, 19, 20, and 21

	m				
k	7	8	9	10	11
2	1176.89	1047.45	954.66	873.14	800.00
3	1602.92	1437.32	1311.79	1219.42	1131.39
4	1968.88	1764.91	1647.31	1533.51	1415.23
5	2112.91	1949.03	1810.54	1693.41	1585.60
6	2415.47	2244.82	2081.75	1941.75	1834.97
7	2612.34	2447.86	2266.25	2141.73	2011.49
8	2777.78	2617.27	2440.22	2312.82	2192.27
9	2862.40	2708.10	2553.77	2414.68	2304.28
10	3001.55	2829.97	2701.54	2556.47	2420.73
11	3103.67	2930.21	2785.80	2653.04	2528.96
12	3206.65	3034.15	2894.39	2755.18	2639.92
13	3264.17	3103.82	2959.51	2833.84	2721.70
14	3330.93	3176.75	3044.93	2919.23	2802.69
15	3385.71	3242.38	3103.40	2972.55	2862.69
16	3452.88	3301.14	3175.61	3056.49	2943.55

TABLE 10. PCI-Express Throughput for Coding Workloads on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 7 \ldots 11$, with Buffer Size of 1 MB, as Reflected in Figures 18, 19, 20, and 21

	m				
k	12	13	14	15	16
2	743.99	693.72	647.38	610.50	578.06
3	1052.05	986.85	929.95	869.61	831.62
4	1327.50	1244.03	1160.49	1108.89	1048.77
5	1498.27	1418.13	1347.14	1274.40	1215.54
6	1748.36	1642.04	1565.35	1478.27	1404.50
7	1911.74	1814.89	1738.01	1647.76	1570.28
8	2072.21	1969.76	1876.79	1798.80	1719.91
9	2183.30	2082.37	1975.51	1908.57	1834.57
10	2321.07	2206.83	2120.80	2040.41	1963.33
11	2425.37	2326.27	2229.98	2139.65	2059.94
12	2536.03	2429.25	2348.81	2249.80	2179.69
13	2594.50	2499.13	2427.00	2332.10	2260.39
14	2703.86	2581.59	2511.30	2414.21	2356.35
15	2751.49	2598.12	2567.61	2484.92	2358.14
16	2836.79	2743.19	2645.41	2561.88	2480.08

TABLE 11. PCI-Express Throughput for Coding Workloads on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 12 \ldots 16$, with Buffer Size of 1 MB, as Reflected in Figures 18, 19, 20, and 21

	m				
k	2	3	4	5	6
2	2778.42	2179.54	1790.53	1518.63	1320.13
3	3496.42	2794.34	2327.76	2053.95	1806.59
4	3785.82	3287.33	2757.87	2463.69	2174.12
5	3836.02	3289.96	2880.86	2541.69	2296.14
6	4093.34	3547.78	3146.59	2858.78	2622.15
7	4146.44	3658.04	3327.30	3020.34	2813.03
8	4191.48	3788.29	3501.97	3249.89	2968.68
9	4196.54	3833.37	3535.78	3289.25	3061.04
10	4360.35	3951.00	3644.88	3423.47	3194.88
11	4360.23	4020.75	3730.80	3487.01	3273.59
12	4434.59	4067.80	3815.37	3599.95	3382.19
13	4405.00	4079.33	3843.65	3657.64	3454.51
14	4449.25	4125.41	3878.54	3675.30	3484.18
15	4429.48	4173.88	3914.40	3735.62	3543.29
16	4477.27	4201.24	3988.26	3780.19	3609.63

Table 12. GPU Throughput for Coding Workloads on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 2 \ldots 6$, with Buffer Size of 1 MB, as Reflected in Figures 18, 19 and 20

	m				
k	7	8	9	10	11
2	1176.89	1047.45	954.66	873.14	800.00
3	1602.92	1437.32	1311.79	1219.42	1131.39
4	1968.88	1764.91	1647.31	1533.51	1415.23
5	2112.91	1949.03	1810.54	1693.41	1585.60
6	2415.47	2244.82	2081.75	1941.75	1834.97
7	2612.34	2447.86	2266.25	2141.73	2011.49
8	2777.78	2617.27	2440.22	2312.82	2192.27
9	2862.40	2708.10	2553.77	2414.68	2304.28
10	3001.55	2829.97	2701.54	2556.47	2420.73
11	3103.67	2930.21	2785.80	2653.04	2528.96
12	3206.65	3034.15	2894.39	2755.18	2639.92
13	3264.17	3103.82	2959.51	2833.84	2721.70
14	3330.93	3176.75	3044.93	2919.23	2802.69
15	3385.71	3242.38	3103.40	2972.55	2862.69
16	3452.88	3301.14	3175.61	3056.49	2943.55

Table 13. GPU Throughput for Coding Workloads on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 7 \ldots 11$, with Buffer Size of 1 MB, as Reflected in Figures 18, 19 and 20

	m				
k	12	13	14	15	16
2	743.99	693.72	647.38	610.50	578.06
3	1052.05	986.85	929.95	869.61	831.62
4	1327.50	1244.03	1160.49	1108.89	1048.77
5	1498.27	1418.13	1347.14	1274.40	1215.54
6	1748.36	1642.04	1565.35	1478.27	1404.50
7	1911.74	1814.89	1738.01	1647.76	1570.28
8	2072.21	1969.76	1876.79	1798.80	1719.91
9	2183.30	2082.37	1975.51	1908.57	1834.57
10	2321.07	2206.83	2120.80	2040.41	1963.33
11	2425.37	2326.27	2229.98	2139.65	2059.94
12	2536.03	2429.25	2348.81	2249.80	2179.69
13	2594.50	2499.13	2427.00	2332.10	2260.39
14	2703.86	2581.59	2511.30	2414.21	2356.35
15	2751.49	2598.12	2567.61	2484.92	2358.14
16	2836.79	2743.19	2645.41	2561.88	2480.08

TABLE 14. GPU Throughput for Coding Workloads on a GeForce GTX 285, $k = 2 \ldots 16$, $m = 12 \ldots 16$, with Buffer Size of 1 MB, as Reflected in Figures 18, 19 and 20

Configuration	Read	Write
Linux md (Direct)	558	406
Linux md (stgt)	374	400
Gibraltar k+2	689	696
Gibraltar k+3	667	673
Gibraltar k+4	644	650
Gibraltar k+5	621	627

TABLE 15. Streaming I/O Performance for DAS in Normal Mode, as Reflected in Figure 22

Configuration	Read	Write
Linux md (Direct)	224	230
Linux md (stgt)	200	232
Gibraltar k+2	692	697
Gibraltar k+3	670	674
Gibraltar k+4	647	651
Gibraltar k+5	625	628

TABLE 16. Streaming I/O Performance for DAS in Degraded Mode, as Reflected in Figure 23

Configuration	Write	Read
Linux md	494	315
Gibraltar k+2	672	437
Gibraltar k+3	655	412
Gibraltar k+4	588	418
Gibraltar k+5	568	355

Table 17. Streaming I/O Performance for NAS in Normal Mode for a Single Client, as Reflected in Figure 24

Configuration	Write	Read
Linux md	270	211
Gibraltar k+2	681	379
Gibraltar k+3	659	378
Gibraltar k+4	588	352
Gibraltar k+5	565	367

Table 18. Streaming I/O Performance for NAS in Degraded Mode for a Single Client, as Reflected in Figure 25

Configuration	Write	Read	Mixed
Linux md	531.9148936	610.6870229	577.2005772
Gibraltar k+2	636.9426752	657.8947368	606.9802731
Gibraltar k+3	606.0606061	624.024961	597.0149254
Gibraltar k+4	591.7159763	603.3182504	579.7101449
Gibraltar k+5	568.9900427	578.8712012	561.7977528

Table 19. Streaming I/O Performance for NAS in Normal Mode for Four Clients, as Reflected in Figure 26

Configuration	Write	Read	Mixed
Linux md	328.6770748	282.0874471	317.9650238
Gibraltar k+2	633.9144216	651.465798	619.1950464
Gibraltar k+3	582.2416303	637.9585327	606.0606061
Gibraltar k+4	552.4861878	609.7560976	586.5102639
Gibraltar k+5	546.4480874	590.8419498	569.8005698

Table 20. Streaming I/O Performance for NAS in Degraded Mode for Four Clients, as Reflected in Figure 27

APPENDIX C

PLATFORMS AND TESTING ENVIRONMENTS

The ability to reproduce experiments is tantamount to their success. In an effort to make the results as transparent as possible, this appendix details all relevant configuration details of the computers used.

Component	Version/Type
CPU	Intel Core i7 Extreme 975
RAM	6 GB 1333 Mhz DDR3 tri-channel
Operating System	Debian 5.0
OS Volume Storage	6 × 32 GB Imation Mobi 3000 flash disks
Raw Storage for RAID	32 × 750 GB Seagate Barracuda ES hard disks
Raw Storage Interface	2 × Fibre channel 4Gbps
Raw Storage Enclosure	2 × RS-1600-F4-SBD
GPU	NVIDIA GeForce GTX 285, 2GB
CUDA	2.2
Network	4x DDR infiniband
OFED	1.4
Linux SCSI Target Framework	1.0.6 with iSER support

Table 21. The testing platform server specifications. All data contained in this work was from experiments using this machine.

Component	Version/Type
CPU	2 × Intel Xeon E5504
RAM	24 GB 800 Mhz DDR3 tri-channel
Operating System	Redhat Enterprise Linux 5
Network	4x DDR infiniband
Open-iSCSI	6.2.0.869.2
OFED	1.5

Table 22. The testing platform client specifications. For the tests outlined in Sections 2 and 3 of Chapter 6, these machines were accessing storage served by the machine in Table 21.

APPENDIX D

A Sample One-Petabyte GPU-Based Storage System

In connection with grant CNS-0821497 from the National Science Foundation, UAB is at the time of this writing procuring a GPU-based storage cluster designed to host computations and reliable storage simultaneously. While this cluster is significantly more capable than a plain storage system (as surveyed in Figure 1), the cost per gigabyte can still be compared directly to understand the benefits of using GPU-based RAID in a cluster deployment. A single node's specifications can be found in Table 23. The networking costs for a cluster of up to 24 nodes with QDR Infiniband is $11,746.

Qty.	Component
1	Supermicro 4U CSE-846E26-R1200B Rackmount Chassis
1	Supermicro X8DAH+-F Dual Xeon Server Board
2	Xeon X5650 CPUs (Westmere, 6 cores)
32	WD2003FYYS 2 TB 7200 RPM Hard Disks
6	4 GB DDR3-1333 Registered ECC Memory
1	Mellanox/Supermicro Dual Port QDR Infiniband NIC
1	NVC2050 TESLA
2	LSI 9212-4i4e HBAs

Table 23. A Single Node Configuration in a GPU-Based Storage Cluster, Total Cost $14,906

Figure 1 showed that prices for reliable storage ranged from $0.80-$2.80 per gigabyte for a petabyte purchase from many well-known hardware RAID vendors. Table 24 shows that, when varying the number of nodes to provide at least one petabyte of storage, GPU-based RAID can be used to build systems with widely varying reliability for $0.26-$0.32 per gigabyte while providing extensive computation resources. For an even larger cost savings, the system could be built more economically by using less expensive GPUs, slower processors, and less RAM per node.

m	Data Capacity (GB, Usable)	# Nodes Required	$/GB
2	1020000	17	0.260
3	1044000	18	0.268
4	1008000	18	0.278
5	1026000	19	0.287
6	1040000	20	0.298
7	1000000	20	0.310
8	1008000	21	0.322

TABLE 24. Storage Cluster Design Parameters and Cost Per Gigabyte