UNIVERSITY OF CALIFORNIA

SANTA CRUZ

INCORPORATING SOLID STATE DRIVES INTO DISTRIBUTED STORAGE SYSTEMS

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Rosie Wacha

December 2012

The Dissertation of Rosie Wacha
is approved:

Professor Scott A. Brandt, Chair

Professor Carlos Maltzahn

Professor Charlie McDowell

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Incorporating Solid State Drives into Distributed Storage Systems

Rosie Wacha

Big data stores are becoming increasingly important in a variety of domains including scientific computing, internet applications, and business applications. For price and performance reasons, such storage is comprised of magnetic hard drives. To achieve the necessary degree of performance and reliability, the drives are configured into storage subsystems based on RAID (Redundant Array of Independent Disks). Because of their mechanical nature, hard drives are relatively power-hungry and slow compared to most other computer components. Big data centers spend tremendous amounts on power, including cooling, adding significantly to their overall costs. Additionally, drives are orders of magnitude slower than electrical computer components, resulting in significant performance challenges any time disk I/O is required. Recently, SSDs (solid state drives) have emerged based on flash memory technology. Although too expensive to replace magnetic disks altogether, SSDs use less power and are significantly faster for certain operations.

This dissertation examines several new architectures that use a limited amount of faster hardware to decrease the power consumption and increase the performance or data redundancy of RAID storage subsystems. We present RAID4S, which uses SSDs for parity storage in a disk-SSD hybrid RAID system. Because of its better

performance characteristics, SSD parity storage reduces the disk overhead associated with parity storage and thereby significantly reduces the disk overheads caused by RAID. This decreases the power consumption and can be used to increase the performance of simple RAID schemes or increase redundancy by enabling the use of higher order RAID schemes with less performance penalty. Storing parity on SSDs can reduce the average number of I/Os serviced by the remaining disks by up to 25-50%. By replacing some hard drives with SSDs, we reduce power and improve performance. Our RAID4S-modthresh optimization improves performance in certain workloads.

The other two architectures expose RAID's inability to handle heterogeneity in workloads and hardware. RAIDE is motivated by a workload imbalance we detected in stripe-unaligned workloads. By placing faster hardware to handle the higher workload at the edges of the array, we observed higher throughput. RAIDH places parity on faster devices based on device weights. Higher weighted devices are faster and thus store additional parity. The slowest devices may store no parity at all. This technique can be used on any heterogeneous array to enable faster random write throughput.

Acknowledgements

Scott Brandt is the best advisor I have ever had. His endless and unconditional support has enabled me to achieve more than I dreamed I could.

I owe much to the Systems Research Lab, especially Michael Sevilla for his work on RAID4S-modthresh. I appreciate all the thoughtful feedback and ideas I received for this project from the rest of my labmates, especially from Sasha Ames, David Bigelow, Joe Buck, Adam Crume and Noah Watkins. Linda Preston and Genine Scelfo helped me negotiate the administrative side of UCSC.

The financial support I received from UC Regents, GAANN and ISSDM fellowships gave me great freedom to explore. My ISSDM advisors provided me with continuing guidance and direction. Thank you to John Bent, Gary Grider, Meghan McClelland, and James Nunez.

I am fortunate to have such great friends who have helped me along on this journey. Bo Adler taught me to believe in myself. Josh Hodas has given me all the best advice that I didn't want to hear. Ian Pye, Caitlin Sadowski and Jaeheon Yi provided me with encouragement and sustenance; I am grateful for our friendship.

Most of all, thank you to my family. Julya, you listened whenever I needed someone to hear me. Mom, you helped me move on and always knew what was best for me. Noah, your love keeps me going every day.

Chapter 1

Introduction

The amount of digital data stored for long periods is growing while increasingly inexpensive new hardware becomes available. It was estimated that in 2003, five exabytes of new information were generated and 92% of that information is stored on magnetic media [50, 91]. Big data store technology has enabled many advances in scientific computing. For example, biological applications including sequencing the human genome and protein modeling require fast access to large amounts of data. The growth and increased popularity of the internet has required the development of advanced search engines such as Google that store continuously updating copies of the data on the internet. Also, financial applications require fast access to data with guaranteed correctness. In all of these applications, data must be continuously available at high transfer rates while minimizing costs. Meeting these requirements uses a tremendous amount of power.

Data centers consume such large amounts of power that power management is a

challenging problem [63]. Hard drives are estimated to consume 27% of total power in data centers [59, 84]. Power consumption costs for storage arrays include not only running the hard disks and controllers but also cooling the system since much of the energy used by computing components is converted to heat [9]. Approaches to solving the power problem include housing data centers near power plants and Google's more novel approach of floating data centers at sea, taking advantage of readily available wind, waves and water for power and cooling [24, 73]. We examine reducing the power consumption requirements of data storage.

Magnetic disk drives have been the mainstay of data storage in information technology for the last 50 years. During this time, disks have gained in performance and reliability, but disk drives still remain complicated electro-mechanical systems made from many different components consuming significant amounts of power and producing heat. While disk reliability is usually sufficient to equip personal computers and servers with only a single disk drive, large data centers need to protect their data against disk failure, whether this is a device failure, a failure of a component that renders many disks inaccessible or destroys the data on them (e.g. power supply) or the failure of a few blocks. Adding redundancy also increases power consumption by increasing the total number of disks consuming power.

New electronic technologies are emerging based on non-volatile memory, the most popular being flash memory. Flash memory entered the consumer market partly due to its low cost and durability for mobile devices. Larger flash devices are now being

developed into SSDs (solid state drives), which have interfaces matching those of hard drives. This makes it particularly convenient to replace hard drives with SSDs. SSDs have performance characteristics that are closer to DRAM (Dynamic Random Access Memory), but provide persistent data storage when powered off. Power consumption of SSDs is about 10 times lower than high performance hard drives.

Storage arrays are continuously growing to accommodate the wealth of new data that is produced each year. The total data capacity of a storage array can be increased by replacing low capacity disks with higher capacity disks or by adding new disks. Larger arrays exacerbate several challenges that storage arrays face. One of these challenges is that a group of disks that are individually reliable together form a system that is less reliable. Suppose one disk has a mean time to failure (MTTF) of 10 years and we have an array containing 100 disks. On its own, the reliability for one disk is acceptable for most applications because the disk only has a 10% chance of failing each year. Assuming that failures are independent, we expect the 100 disk array to experience 10 disk failures each year. On top of full disk failures, disks suffer from block errors and these can be hidden by the operating system and disk intelligence and may remain undetected until reconstruction [32]. Disks with larger capacities experience more inter-disk errors so replacing small disks with larger disks does not remove these problems. The cost of power to run a large array of disks is also high.

Big data generated by scientific applications, web archive companies such as the Internet Archive, and other organizations often requires greater throughput than

individual magnetic disks provide. To speed up access to these larger pieces of data, systems can use striping, where the same data unit (a file or a database table) is split over several devices that can be accessed in parallel. Striping provides parallel access because more actuators serve data concurrently. Data longevity is also a concern and redundant data is typically stored to protect data in spite of one or more hardware failures. To reduce storage space overhead, the mathematical parity of several data blocks is calculated by taking the exclusive-or of all data bits and storing the parity instead of an actual copy of the data.

This dissertation focuses on reducing power while maintaining good performance and reliability in a distributed storage system. The key change from traditional storage systems is to incorporate small numbers of SSDs into disk-based storage systems. Specifically, we are investigating adding SSD to RAID (Reduntant Array of Independent Disks). By replacing parity disks with SSD, write performance is improved while power is reduced. We demonstrate an improvement in RAID performance at low cost by using small amounts of expensive SSD. Traditional RAID schemes are designed to work well based on the performance of hard drives. For example, RAID striping is designed so that each block of a stripe is in the same LBN (Logical Block Number) of the drive, in order to minimize seeking.

The completed work consists of four main ideas. Chapter 3 presents RAID4S, which replaces the parity drive in RAID4 with an SSD. Our analysis shows that using SSD for the parity drive in a RAID4 configuration reduces load on the remaining disks

by up to 50% for random workloads. The hardware experimentation shows up to 3.3X small write throughput improvement over RAID4.

Chapter 4 introduces RAID4S-modthresh, an optimization to the RAID4S workload that better utilizes the SSD. In particular, small writes that span 50% of the array are kept as read-modify-writes instead of converted into reconstruct-writes. This introduces an extra parity read but isolates the workload to only the devices that are written to, leaving others free to work on other requests.

RAIDE is a RAID setup that benefits from small write workloads by placing faster devices at the edges of the array. Chapter 5 provides an overview of results demonstrating that some small write requests produce uneven read workloads across the array. The devices at the edges are required to complete more reads than the devices in the middle. RAIDE improves array performance by placing SSDs at the edges of RAID5 arrays.

Lastly, Chapter 6 presents RAIDH, a layout strategy that provides a way to use RAID for arrays with fully heterogeneous devices. We introduce an algorithm to convert arbitrary device weights into an array of parity locations. This array maps data to parity locations such that the amount of parity on each device matches its specified weight.

Contributions of this work:

1. Faster, lower power, and simpler RAID solutions using SSDs
2. Cost comparison of disk- and SSD-based storage systems
3. Analysis of small write workloads to show workload unevenness
4. RAID parity distribution technique for heterogeneous arrays

Chapter 2

Background and Related Work

2.1 Data Layouts for Redundancy and Performance

Magnetic disk drives have been the mainstay of data storage in information technology for the last 50 years. During this time, disks have gained in performance and reliability, but disk drives still remain complicated electro-mechanical systems made from many different components and consuming significant amounts of power and producing heat. While disk reliability is usually sufficient to equip personal computers and servers with only a single disk drive, large data centers need to protect their data against disk failure, whether this is a device failure, a failure of a component that renders many disks inaccessible or destroys the data on them (e.g. power supply) or the failure of a few blocks. Adding redundancy also increases power consumption by increasing the number of devices storing data.

Storage system designers choose between several possibilities to generate redundancy. A classic method is back-up of data on archival media, usually a tape library but increasingly now a secondary array of disks. Backup inherently suffers from the problem of not protecting recently generated data (that has not yet been scheduled to be backed-up). Thus backup needs to do a combination of incremental backups and full backups to not overburden the I/O capability of the backed up storage system. Another common method is to store redundant data within the storage system. The simplest technique for storing data redundantly within a storage system is mirroring or higher levels of replication. Tandem in the 1980s offered reliable computer systems that used a pair of mirrored disks [12]. These types of fault tolerant systems are popular with the banking industry. Currently, the Google File System (GFS) [30] needs to provide high availability without a noticeable degraded mode and by default stores three copies of data chunks.

Replication allows reads from any of the devices that store replicated data and therefore offers good read performance. For instance, a read from a pair of mirrored disks can be directed to the one with the least queueing, or if queueing is absent, can be directed to the one whose actuator is closest to the target track. This would lead to approximately 30% reduction in seek time. The drawback is that writes need to be directed to several devices.

Since the storage overhead of replication is high and since many data centers are not limited by performance, other types of redundancy generation are important.

Mathematically, all types of redundancy can be described as an erasure correcting code that stores m data blocks on n disks such that all data is guaranteed to be accessible despite up to k disk failures. For example, a system of m triplicated disks uses $n = 3m$ disks in total and can survive up to $k = 2$ failures, and often many more. The following subsections describe several redundancy techniques and their impacts on reliability, storage overhead, and performance.

RAID

Big data generated by scientific applications, web archive companies such as the Internet Archive and Google, and other organizations often requires greater throughput than individual magnetic disks provide. To speed up access to these larger pieces of data, systems can use striping, where the same data unit (a file or a database table) is split over several devices that can be accessed in parallel. Striping provides parallel access because more actuators serve data concurrently. In the late eighties, disk drives differed vastly in capacity and speed, and several groups at IBM, Stanford, and UC Berkeley, considered the possibility of using smaller, inexpensive disks to replace a single, larger disk. However, with disk mean time to failure around 50,000 hours (nearly six years), the longevity of data on these arrays was in doubt.

Patterson et al. incorporated existing redundancy techniques and striping into storage systems by adding a parity device to m data devices, called the result a RAID (Redundant Array of Independent Disks) [58], and distinguished several RAID levels or

layouts. RAID 0 has no parity but stripes data across drives to increase throughput by providing parallel access to data. Figure 2.1 shows the RAID 1 layout which stores identical copies of data on multiple disks and is also referred to as "mirroring". Figure 2.2 shows two layouts using the idea of parity to reduce the storage overhead of storing redundant data. Parity of several data blocks is calculated by taking the exclusive-or of all data bits. The specific layouts here are referred to as 3+1 layouts, because there are three data disks corresponding to one parity disk. RAID 4 places parity blocks on a dedicated disk. This presents a bottleneck on the parity disk because the parity disk must be accessed for every write. In order to spread out the parity disk accesses, parity is distributed across all disks in the RAID 5 layout.

Small write problem

RAID 5 greatly improves the storage overhead for redundant data over mirroring, but at a performance cost for small modifications to existing data. As an example, consider a small write that fits in one block. In order to complete the write with a RAID 5 configuration, the old data and corresponding parity block are read in order to compute the new parity. Then the new data and new parity are written. This results in four disk accesses when a mirrored layout would only require two. Our proposed technique of replacing the parity drive with an SSD mitigates this problem. See Chapter 3 for details.

RAID 1

disk 1	disk 2	disk 3	disk 4
D_{A1}	D_{A1}	D_{A3}	D_{A3}
D_{B1}	D_{B1}	D_{B3}	D_{B3}
D_{C1}	D_{C1}	D_{C3}	D_{C3}
D_{D1}	D_{D1}	D_{D3}	D_{D3}

Figure 2.1: RAID1 mirrored data layout requires high 2x storage overhead but has good read performance. Any single failure is tolerated, as well as some multiple failures such as disks 1 and 3 in this example.

RAID 4

stripe	disk 1	disk 2	disk 3	disk 4
A	D_{A1}	D_{A2}	D_{A3}	P_A
B	D_{B1}	D_{B2}	D_{B3}	P_B
C	D_{C1}	D_{C2}	D_{C3}	P_C
D	D_{D1}	D_{D2}	D_{D3}	P_D

RAID 5

stripe	disk 1	disk 2	disk 3	disk 4
A	D_{A1}	D_{A2}	D_{A3}	P_A
B	D_{B1}	D_{B2}	P_B	D_{B3}
C	D_{C1}	P_C	D_{C2}	D_{C3}
D	P_D	D_{D1}	D_{D2}	D_{D3}

Figure 2.2: RAID4 and RAID5 data and parity layouts have a lower storage overhead than RAID1, but still tolerate any single failure. The parity distribution of RAID5 eliminates the parity disk bottleneck in RAID4.

Parity striping

The parity striping optimization relies on a basic understanding of physical disk geometry [66]. Disks are mechanical devices composed of one or more circular disk platters which are made up of circular tracks which are broken up into sectors. Data is read by a head or actuator that moves back and forth across the disk to switch tracks, which are read or written while the disk spins. A disk access requires moving the head to the new

location (called a seek) and waiting for the disk platter to spin until the head sits on top of the start of the track (called a latency).

Gray et al. [31] found that transaction-based systems such as databases do not need the data bandwidth RAID 5 offers but instead need better small write throughput. Mirroring provides the best throughput of small accesses because many disks service transactions at the same time, however the storage space overhead is too high. Parity striping is designed to provide small write performance comparable to mirroring but with the lower storage overhead of a RAID 5 array. An N+2 parity striped array is N+1 logical disks, each one storing data that is not striped, and a spare disk. Parity is stored on a subset of each disk in a striped layout similar to RAID 5. Most small reads will be serviced by a single disk and writes by two disks. The main benefit of parity striping compared to RAID 5 is that the request service rate is higher. This is particularly important for reducing the typical number of queued requests in database applications. The main downside is that overall throughput is lower because data access isn't parallelized as well as with RAID 5 where data for a single application is stored in relatively small chunks and those pieces are stored on many individual disks that can be accessed in parallel. Note that another way to achieve this effect is to make the striping unit in RAID 5 quite large, so that small accesses fit on a single disk. This preserves the benefits of greater bandwidth for large sequential accesses while preserving the small access throughput.

Parity declustering

When an array experiences a hardware failure and before it is repaired or rearranged, the array operates in degraded mode while reconstructing data in real time. Performance in the degraded mode of a RAID 5 depends on the size of the disks in the RAID because a read access to the lost disk involves all other disks. In practice, this limits to the number of disks in a RAID 5 organization. When creating the layout for a larger disk array, it is common to concatenate several RAID 5 arrays. In this case, the effect of a failed disk will be concentrated on the remaining disks in that sub-array. Parity declustering spreads the additional load due to a disk failure over all surviving disks in the array.

Alvarez et al. [6] outlined six desirable properties for disk layouts tolerating one failure:

1. Single failure correcting
2. Distributed parity
3. Distributed reconstruction
4. Large write optimization
5. Maximal parallelism
6. Efficient mapping

Parity declustering [38,39,56] is a technique that optimizes each of these properties through the use of smaller logical arrays is large physical arrays. Parity declustering

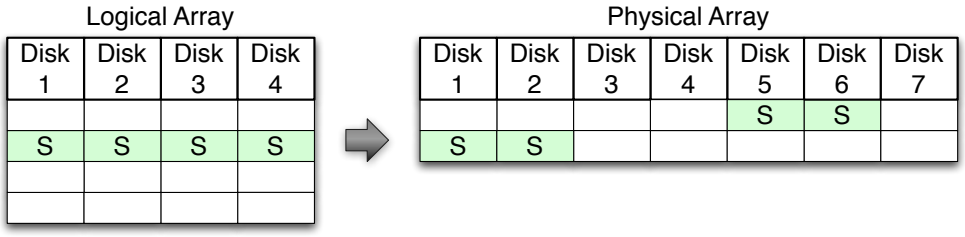

Figure 2.3: Parity declustering assigns parity stripes to a smaller subset of physical disks and breaks up parity groups so that they aren't all stored on the same device. Without parity declustering, the parity stripe would span all seven physical disks.

spreads out blocks belonging to parity groups on one disk so that the set of disks sharing those parity groups is larger than the number of parity groups on that one disk. This improves reconstruction performance because the remaining disks do not have double their normal workload for as long a time. The intuitive idea is that a large number of physical disks are organized into several smaller logical arrays, thus allowing the parity stripe to be stored on a subset of the physical disks. That subset is the set of disks required to reconstruct that data if a disk in that logical array fails. Any disks not in that subset are not affected by the reconstruction. Figure 2.3 shows an example of how parity declustering spreads parity stripes onto an array. By uniformly distributing parity groups across the disks, the workload of recreating data after a failure is also uniformly distributed across all disks. Since rebuilding proceeds in parallel across a larger number of disks, it completes more quickly than without declustering. Parity declustering is used in the Panasas Parallel File System [87].

Reconstruction performance improvements

Devices fail often, especially in large arrays, and reconstruction is time consuming and throttles foreground (i.e. non-reconstruction) workload I/O for its duration. PRO [80] is a popularity-based reconstruction optimization that prioritizes rebuild of frequently accessed data. The evaluation replays a read-only web trace and shows 1.3-44.7% reductions in reconstruction time, dependent on the RAID configuration. User response time is also reduced in many cases.

WorkOut [89] is another technique that uses popularity in its design. It implements a surrogate RAID to handle popular reads and all writes. WorkOut is activated at reconstruction by initializing some space in a secondary array for the reconstructing array. Writes are logged and popular reads are cached in the surrogate RAID array. WorkOut reduces reconstruction time, increases user response times, and increases MTTDL by more than 5X over the default software RAID reconstruction algorithm. Our RAIDH technique (described in Chapter 6) would help improve the throughputs on the secondary array. A further optimization could place small, popular read data on the SSD, thus further utilizing the heterogeneous hardware.

Disk arrays with higher fault tolerance

RAID and similar techniques provide good reliability for small arrays of small disks, but as arrays and individual disk sizes have grown, the probability of a double failure has become too great to ignore. There are many different erasure codes that tolerate multiple

failures, serving many different application requirements, such as storage overhead efficiency, performance, and reliability [14, 15, 26, 35, 36, 47, 61, 62, 93]. Maximum distance separable (MDS) codes are those that provide optimal storage efficiency. Specifically, an MDS code that stores m redundant symbols tolerates up to m erasures.

Bit and block errors have become a real problem as individual disk capacities have grown. Disk capacities at the time that RAID was developed were under 10 GB [58] and today 1 TB disks are common. Block errors are an even greater problem because they can be hidden by the operating system and disk intelligence even though they are relatively rare [32]. As a result, block errors are often only discovered when rebuilding after a failed disk and thus are an even greater problem. If more than one block error occurs in a RAID5 stripe, that data is lost. For this reason, erasure codes tolerating multiple failures were developed. The primary performance measures are the speeds at which parity can be generated and data reconstructed.

Early work extended RAID5 to use parity to tolerate multiple disk failures. One of the clear benefits to XOR-based codes is that they are computationally efficient and easy to understand. Hellerstein et al. [36] present binary (XOR-based) linear codes in a combinatorial framework. Their codes tolerate up to three failures in the general case and suggested that codes tolerating more than three failures would not be useful in the future. EVENODD [13] is an MDS code that has optimal performance, no recursive computations, and tolerates up to two disk failures. MDS codes are optimally space efficient. Row-Diagonal Parity (RDP) [26] is an XOR-based code similar to EVENODD

but proven computationally optimal. Each data segment belongs to one row parity segment and one diagonal parity segment. The diagonal parity includes data segments and row parity segments if they are included in the diagonal. RDP protects data against up to two disk failures. The downside is that small writes are an even greater problem because the diagonal "stripe" is larger than a row "stripe", which is the same as a traditional RAID5 stripe. WEAVER codes [35] are generally not MDS but are optimal in storage efficiency among all codes with the same parity in-degree and fault tolerance level. Parity in-degree is the number of data values associated with a parity value. The data out-degree is the number of parity values associated with a data value. Benefits include high fault tolerance, data is localized on a small number of devices, symmetry and simplicity. The main drawback is that space efficiency is not better than 50%, so that tolerating N failures requires at least 2N disks in the storage system. SSPiRAL [7] (Survivable Storage using Parity in Redundant Array Layouts) is a parity scheme using the same number of disks that a mirrored scheme would use, but tolerating additional failures and thus having a higher mean time to data loss (MTTDL). Because SSPiRAL uses parity, its performance is better than schemes using erasure coding. One of the goals of SSPiRAL is to describe codes in terms of available resources rather than fault tolerance. This property makes it easier to configure SSPiRAL in a real system.

Reed-Solomon codes [61] are a class of codes that use matrix multiplication and Galois Field arithmetic to provide arbitrary fault tolerance in storage systems. There is usually a higher performance overhead when using Reed-Solomon, but recent work

improves the software implementations [33].

2.2 Very Large Storage Arrays

Data placement

Several layouts for data on very large storage arrays have been proposed [18, 19, 85]. Large arrays are highly dynamic: new disks must be added both to facilitate data growth as well as to replace failed disks. Typically, large storage arrays use some form of sparing so that disks do not need to be physically replaced as often (see Section 2.3 for more information about sparing). Placement algorithms must adapt to these changes while still maintaining good performance to access and update data. Another concern for very large storage systems is management and distributing potential bottlenecks to allow the system to scale.

Early work looked at placement for storage area networks (SANs). The basic SAN architecture is composed of just a bunch of disks (JBODs) connected to dedicated network and servers. Brinkmann et al. [18] investigated several algorithms that use hash functions to place data onto an array of storage devices. They did not find any technique that guaranteed a unique address for a given piece of data when heterogeneous sizes of devices enter and leave the SAN.

Enterprise storage solutions have great benefits in terms of centralized, easy to use, management but pose challenges when a customer wants to upgrade their storage

capacity or policies if the provider doesn't offer specifically what the customer needs. Storage virtualization has created viable solutions to these problems [19]. Storage virtualization combined with SAN management environments provide flexible centralized management for a large array of heterogeneous disks and hardware. They also offer management features such as mirroring and snapshots. The main drawback mentioned is that there isn't a way of managing reliability in the event of failures of entire storage units and their performance in dynamic environments. The first of these is solved by another storage virtualization environment [17] and the second is introduced by allowing dynamic environments, which enterprise solutions do not typically offer.

Data placement algorithms need to handle replicated data to optimize reliability. The Replication Under Scalable Hashing (RUSH) [41] algorithms provide adjustable replication at the fine granularity of the object level. RUSH also guarantees that the replicas of an individual object will be placed on different disks. The Controlled Replication Under Scalable Hashing (CRUSH) [86] algorithm is an extension of RUSH that also handles load distribution and guarantees that replicas are stored on different failure domains. Brinkmann et al. [16] also look at placement of redundant data blocks. They specifically address the problem of efficiently placing redundant blocks on different devices while spreading data out across devices so that each device stores its fair share of the total capacity of the system.

Ensuring reliability of data

The time between when a disk failure (including block errors) occurs and when it is detected is a vulnerable period for the data stored on that disk and in the rest of its reliability group. Any subsequent failures in the reliability group are more likely to lead to data loss and with a single redundancy scheme such as RAID5 one additional failure will cause data loss. Thus it is clearly very important to detect errors as quickly as possible to reduce the time before a recovery or reorganization begins.

Disk scrubbing [71] is the act of reading blocks periodically to ensure that the data stored is still accessible. Storing signatures with data and checking the signatures while scrubbing ensures that the stored data is valid. Even if each disk in an array is only scrubbed once per year, reliability is greatly improved.

The vulnerability period after an error is detected and recovery has been initiated also has a significant influence on reliability. There are several techniques that aim to speed up the recovery process to reduce the time that the array is more vulnerable to additional failures.

Xin et al. [92] investigate reliability mechanisms for very large scale (2 PB) storage systems in order to handle non-recoverable read errors and disk failures. Fast Mirroring Copy (FMC) improves reliability by speeding up recovery and spreading the rebuilt data across available storage nodes. Lazy Parity Backup (LBP) creates parity blocks of cold data in the background. They store signatures of data objects and verify the signatures when the data is read back. This provides a mechanism for detecting block errors that

would normally lead to data corruption. Power conservation is not considered in this work.

Xin et al. [91] developed another technique to improve reliability in very large storage systems. FAst Recovery Mechanism (FARM) is a distributed recovery algorithm that reduces recovery time by using declustering (also known as distributed sparing). They distributed mirrored copies or redundancy groups so recovery is faster. Declustering provides improved performance in degraded mode, but also improves rebuild time and an improvement in reliability is shown in this work. They also analyzed replacement batch sizes and determined that at their scale, it didn't significantly affect reliability to delay replacements until up to 8% of the disks must be replaced in the batch. Vaidya looked at improving performance during recovery in distributed systems with a two-level recovery scheme [82]. By distinguishing between more and less common failure events, they optimize for the common case and improve performance. Fast recovery time is very important and we will investigate the performance impact of our RAID scheme during degraded and recovery modes.

2.3 Self-Configuring Disk Arrays

Storage array configuration is a challenging task and system administrators need increasing expertise to sufficiently manage growing storage systems. Automating some of the optimizations is an attractive feature to reduce the human effort required and adapt

to changing workloads and hardware. We anticipate some automatic reconfiguration of RAID codes based on the amount of SSD space available and the specific disks currently active (powered on) and available (free space).

HP AutoRAID

The AutoRAID [88] system migrates data between mirrored and RAID 5 automatically. Frequently accessed data is stored in a mirrored arrangement providing high redundancy and performance. Less frequently accessed data is migrated to a RAID 5 array to save on storage overhead cost at lower performance. This migration is done automatically and transparently through workload monitoring. The storage hierarchy is implemented in a smart array controller. Initially all data is mirrored, but when the disk is full data is migrated to RAID 5.

The HP AutoRAID provides two classes of storage to data, namely mirroring and a virtual RAID Level 5 layout. AutoRAID uses mirroring for recently written data and RAID5 for older data. Mirroring takes up extra space but improves small I/O performance. RAID5 uses only one extra disk (wastes only 1/N of total disks) but is slower for writes. New disks can be any size and can be used for mirroring until space is needed.

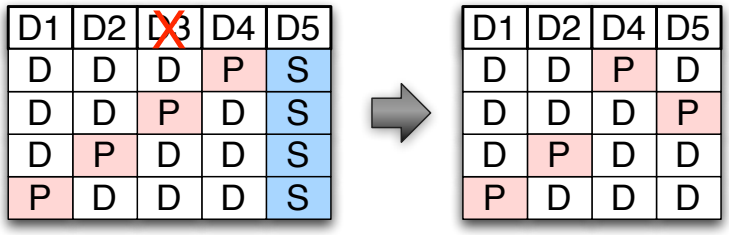

Figure 2.4: RAID5 rebuild of disk 3 with a dedicated spare disk.

Sparing

The performance of storage arrays that use erasure codes can be improved by adding additional disks, known as "spares".

Lowering the time to repair failed disks is important for reliability, so disk arrays commonly use one or more hot spares [52, 64]. Adding several spare disks when installing as well as servicing a storage array also has the benefit of reducing the timeliness requirement of replacing failed disks in addition to starting the reconstruction process immediately after a failure occurs.

Dedicated sparing

Dedicated sparing reserves one or more idle disks to serve as replacements for failed disks. Figure 2.4 is Thomasian's example [79] that shows a RAID5 array with a dedicated hot spare disk, disk 5. If any of disks 1-4 fail, the data and parity is immediately rebuilt onto the spare. One of the downsides to having one or more dedicated spares is that if the spare happens to fail, the failure is not discovered until the disk is needed. This can

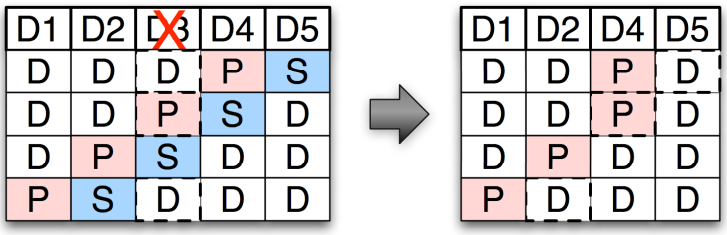

Figure 2.5: RAID5 rebuild of disk 3 with distributed sparing.

be fixed by performing automatic periodic disk scrubbing. Another downside is that the bandwidth of the spare disk is wasted since it is not storing any data until it replaces a failed disk. Distributed sparing addresses this concern by spreading the spare blocks across all disks in the array.

Distributed sparing

Distributed sparing puts data onto the spare disks while retaining free space that can be used to replace data from a failed disk. Distributed sparing [52, 79] uses the hot spares even before disk failure(s) in order to improve performance. Just as RAID5 derives from the RAID4 layout by distributing the parity disk, distributed sparing distributes the spare blocks. In the resulting configuration, each disk contains parity data, user data, and spare space. In the event of a failure, the disk array reorganizes itself. Reconstructed data is written to the spare spaces on the remaining disks. The layout becomes closer to a RAID5 layout, with less spare space.

Figure 2.5 is another of Thomasian's examples [79] and shows RAID5 with distributed sparing. Before disk 3 fails, five disks are sharing the load that originally was

serviced by four disks, resulting in a load at each disk that is 80% of the load with a dedicated spare disk. Reconstruction after the failure on disk 3 involves reading from three of the remaining disks and writing to the fourth to reconstruct the lost data and place it in the spare blocks.

Parity sparing

Parity sparing [64] is similar to distributed sparing but goes one step further in utilizing the spare space. Instead of keeping the spare space empty, an additional parity is stored in its place. A disk failure essentially causes a reorganization of the array into a larger array. For example, supposed there are two disk arrays, A and B. One disk in array A fails. The remaining disks in array A and all of array B are combined into a single parity group. Distributed sparing is demonstrated to be the best sparing technique because rebuilds are faster than parity sparing rebuilds and performance is comparable between them.

2.4 Solid-State Drives (SSDs)

New electronic technologies are emerging based on non-volatile memory, the most popular being flash memory. Flash memory entered the consumer market partly due to its low cost and durability for mobile devices. Larger flash devices are now being developed into SSDs, which have interfaces matching those of hard drives. This makes it particularly convenient to replace hard drives with SSDs.

One of the downsides to flash memory is that random writes are slower than sequential writes. Data can only be written to cells in a block that has been erased. Erasing a block takes about the same amount of time as the write, thus doubling the time to complete a small write if the block was not already erased. Researchers have designed several algorithms to improve small write performance by buffering writes [10, 45]. Samsung has announced a new flash memory SSD that has average write speeds close to its read speeds [2]. While the details of their implementation is not public knowledge, the performance improvements are available nonetheless.

Narayanan et al. [54] evaluated the total dollar cost (power and hardware) of configuring storage systems made of disks, flash-based SSDs, or a two-tiered design. The workloads are traces from large and small data centers responsible for Microsoft's Exchange server, MSN storage, and some smaller servers. In all 49 workloads, they found that either the disk-based or two-tiered approach is cheaper. The power savings does not translate to a significant reduction in the cost of using SSD because power is not expensive enough. The work does not consider the cost of building a power plant capable of powering a data center that consumes more power.

Burst Buffers [48] use SSDs to temporarily store bursty writes until they can be stored permanently on underlying disk-based storage. This storage layer allows data to be pushed out of memory more quickly, thus improving application throughputs by allowing the application to continue executing instead of waiting for slower permanent storage. This type of caching layer also helps ameliorate the small write problem and

could be used in combination with our hybrid RAID techniques at the currently disk-only storage layer.

Another recent non-volatile technology is MEMS (MicroElectroMechanical Systems). Like flash, MEMS-based storage is expected to have price and performance measurements falling somewhere between hard drives and RAM. Uysal et al. proposed using MEMS-based storage in several possible configurations in disk arrays [81]. They looked at several mirrored configurations and replacing disks with MEMS. They found that their best technique was to use MEMS-based storage as the primary data copy and store the mirror on disk in a log structure (to improve data sequentiality). The work does not consider storing parity on MEMS-based storage.

Another SSD technology is also emerging. Phase Change Memory (PCM) [29], leverages the physical states of materials (such as amorphous or crystalline) to store bits of information. PCM has write endurance at least 3 orders of magnitude greater than flash. Because PCM does not rely on electrical charge to store data, the longevity of bits written is higher since there is no degradation over time due to charge dissipation. Several companies are working on PCM prototypes and it is expected to hit the market in the next few years [23, 53]. Our proposed work with SSD may extend itself to newer technologies as well.

2.5 Mitigating RAID's Small Write Problem

Standard parity-based RAID arrangements suffer from a performance issue known as the "small write problem". In a non-redundant array, small updates require two disk accesses and in a mirrored array, they require three. However, a small read-modify-write operation requires four disk operations in RAID4 and RAID5. The first two are reading the data and its parity from two different disks. Then the new parity is calculated and the new data and parity are written back to those two disks. For larger writes that fill at least one stripe, the old parity does not need to be read because all new data is written and parity can be constructed directly from that data.

Parity logging [74–77] is a technique that improves small write performance in parity-based storage systems. Parity updates are stored on a dedicated log disk so that small, random writes can be batched into larger, more sequential writes. Magnetic disks inherently provide poor random access throughput due to the relatively large head seek and rotational positioning delays required for every access. For large accesses, the transfer time trumps these positioning latencies and thus throughput is much higher.

A basic parity logging system is shown in Figure 2.6. The read-modify-write loop proceeds as follows. Data is read from one of the disks. The XOR between the old data and the modified data is temporarily stored in memory in a dedicated log buffer. The modified data is immediately written back to the disk as in RAID5. When there are enough parity updates in the log buffer to warrant good throughput on the disk, they are written out to the parity log disk. When the log disk fills, the parity and log disks are

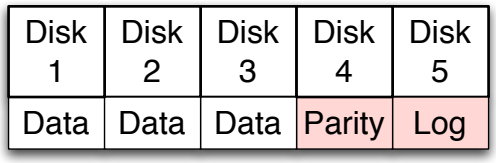

Figure 2.6: RAID4 with parity logging.

read, parity updates are calculated in memory, and all modified parity is written to the parity disk (mostly sequential). This reduces disk time on average by a factor of two and for a real on-line transaction processing or database workload the improvement is even greater.

Another approach to improving the small write problem and reducing the overhead of writing to a parity disk uses non-volatile random-access memory (NVRAM). NVRAM is similar in function to disks but provides faster access times comparable to memory. It is persistent memory like disk storage meaning that once it is stored it will remain until deleted even when the device is powered off. NVRAM is more expensive per Gigabyte than disk, but inexpensive enough to be used in small quantities to improve disk systems.

FEARLESS (Flash Enabled Active Replication for Low End Survivable Storage) [20] aims to make it more tractable to put RAID into commodity systems such as laptops or MP3 players where it isn't feasible to have multiple hard disks. The assumption is that data will be backed up regularly (daily) to a secondary storage system or backup disk. In between backups, all new data is written to a single disk and a mirror copy is written to a flash device. All recently modified data is stored on the flash device. If the hard disk fails, data can be restored using the daily backup and the flash device

containing recent modifications. This system improves reliability but not availability. The data needs to be restored before it can be accessed. This requirement makes the FEARLESS design only suitable for applications that do not require constant uptime.

Bisson redirected I/O from the disk to flash memory in laptops [11]. The main goal was to reduce power by aggressively powering down the disk while improving the reliability of the disk by reducing the number of disk power cycles. This work motivated our idea to use flash in multi-disk storage systems.

2.6 Low Power Storage Systems

Energy consumption is a significant cost of disk arrays. Recent estimates are that disks utilize 27% of the total power used by a typical data center [59, 84]. Significant heat is released by the great deal of power consumed and again more power must be expended in order to provide adequate cooling to protect fragile computer components from overheating. Disks, in particular, are sensitive to temperature changes which affect reliability [60, 70].

Massive Arrays of Idle Disks (MAIDs) [25] are used to reduce power in a commercial product from Copan Systems [34]. Their storage array consumes at least 75% less power compared to a traditional array with all disks powered continuously. Pergamum [78] is a disk-based archival system that aims to replace tape archives by keeping at least 95% of disks inactive. One of the enabling techniques is that each node has

dedicated NVRAM for data signatures and metadata so that many requests can be serviced without requiring idle disks to be powered on. We expect to keep disks powered off for longer periods by placing parity on flash-based SSDs.

RAID and other redundancy techniques are commonly used to protect data in storage arrays. Typical RAID installations do not normally provide any consideration of power costs. However, optimizations directly relying on redundancy and RAID have been shown fruitful. The Diverted Accesses technique [59] conserves energy in disk arrays by segregating redundant data and putting it on disks that can remain powered off. They also rely on buffering writes to some form of NVRAM. They are able to conserve 20-61% of the energy consumed by disks in their storage system. The power savings technique relies on significant redundancy of data in the storage system, which is an assumption we might not be able to rely on. PARAID [84] uses a gear-shifting automobile analogy to divide disks into groups based on how much performance the array requires. When lower bandwidth is necessary, the array can down-shift into a gear that uses fewer disks and powers down unused disks. A 37% power reduction was demonstrated for this scheme. The main downside is that the layout results in potentially long recoveries in the worst case due to cascaded recoveries starting at a low gear and progressively recovering at each higher gear. Additional work is needed to improve RAID layouts to reduce power but also shorten or maintain recovery time. Our work will not increase recovery time because SSDs are inherently faster than disks. The parity reads will be faster than data reads from the disks in the array stripe.

2.7 Real Systems

Data Oasis [5] is a parallel file system used by the Triton supercomputer at San Diego Supercomputing Center. The storage configuration uses RAID5 on Lustre objects and supports 500MB/s per node and 2.5GB/s aggregate across the 800TB system. High throughput is the top priority in this system, providing the ability to quickly move data where it's needed for processing. Because the storage hardware is homogeneous, incorporating weighted parity would require substituting some hard drives for SSDs or faster hard drives.

Facebook's Open Vault [94] is a modular storage design enabling incremental hardware additions to increase capacity and bandwidth. Each storage unit contains 30 hard drives, which are replaced individually after failure. These large storage units might stripe data internally while providing redundancy at a higher level. For example, each storage unit may represent a single device in traditional RAID5 or RAID6. In this RAID scheme, weighted parity could favor faster storage units such as recent additions containing newer hard drives or SSDs.

Chapter 3

RAID4S: Supercharging RAID Small Writes with SSD

3.1 Improving RAID Small Write Performance

Parity-based RAID techniques improve data reliability and availability, but at a significant performance cost, especially for small writes. Flash-based solid state drives (SSDs) provide faster random I/O and use less power than hard drives, but are too expensive to substitute for all of the drives in most large-scale storage systems. We investigate the tradeoffs of incorporating SSDs into RAID arrays. The cost of replacing all disks with SSD is too high (cost per unit size for SSDs is at least twice that of disks). When replacing some disks with SSDs, the balance of power, performance, and reliability is shifted. Power is reduced while performance is increased. On the other hand, performance could

be maintained while adding reliability in the form of additional parity stored.

RAID layouts [58] provide data reliability and availability in storage systems. However, this benefit comes at a performance cost. In particular, updating parity significantly degrades small random write performance. Figure 3.1 shows the small write performance of a striped RAID0 system and the reduced performance of parity-based RAID. A key goal of our research is to reduce this small write overhead and allow parity-based RAID schemes to approach the performance of RAID0. As caches grow and become more effective, large storage workloads are increasingly write-intensive. Some techniques (e.g., WAFL [37]) transform small writes into large writes, but some small writes are inevitable, particularly as free disk space becomes fragmented over time. RAID4S addresses this problem by parallelizing small writes without hurting large write or read performance.

There are two ways to update parity in a RAID4 or RAID5 system when a partial stripe is written. Writes that span at least half of the drives in an array—large writes—may be most efficiently accomplished by reading the data of the remaining drives in the stripe, computing the new parity, then writing the new data and parity to disk. Writes that span less than half of the drives in an array—small writes—may be most efficiently accomplished by reading the old data and parity of the update blocks (which are to be overwritten), XORing the old and new data with the old parity, and writing the new data and parity to disk. As shown in Figure 3.2, small writes require $2(m+1)$ I/Os, where m is the number of drives in the stripe that are to be written. On the other hand, large writes

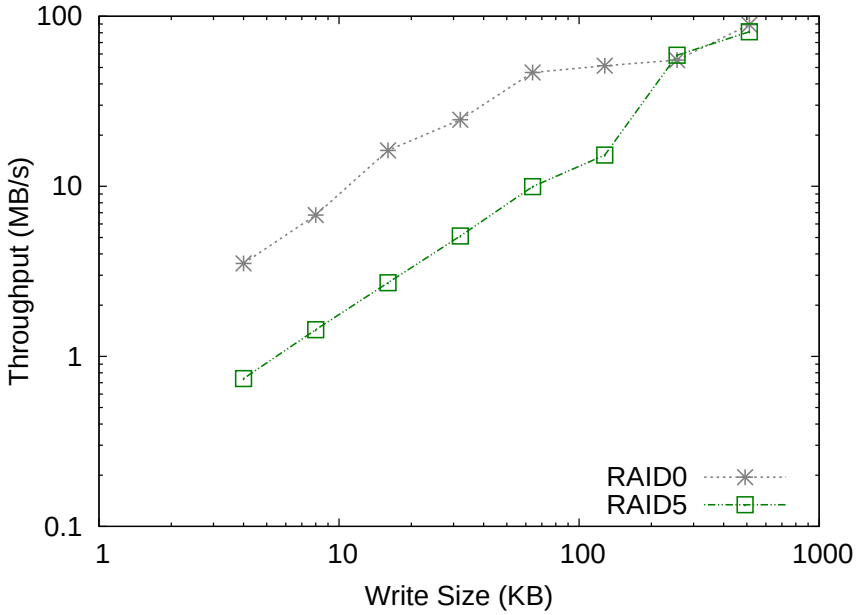

Figure 3.1: The RAID5 parity disk significantly limits small write performance.

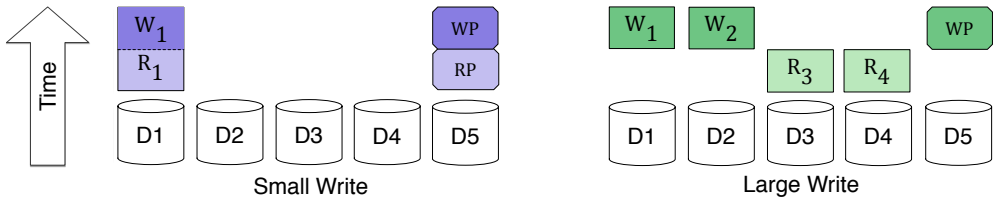

Figure 3.2: Small writes require a read-modify-write operation to update parity, whereas large writes construct new parity by reading the rest of the data in the stripe.

performed with the reconstruct write technique require $N + 1$ I/Os per write, where N is the total number of drives per stripe. The choice of which method to use is usually based on the number of I/Os needed for a given write—small writes require fewer I/Os whenever $m < N/2$, and large writes require fewer I/Os whenever $m \geq N/2$.

RAID4 arrays use a dedicated parity drive; the parity for all stripes are on a single device. With a dedicated parity drive, small write I/Os hit different combinations of data disks, but the parity update load *always* hits the parity disk. RAID5 distributes the parity update load over all of the disks in the array by rotating which disk handles the parity for different stripes. This results in slightly more complicated controllers and the disks must still collectively handle the extra parity load, but results in better performance as all disks share in the extra load imposed by parity updates and all disks serve a fraction of reads. RAID4 and RAID5 perform the same when handling full stripe writes.

The advent of new storage technologies provides an opportunity to rethink traditional algorithms and architectures. Solid state disks (SSDs) provide fast random I/O and much faster random read performance than disk drives. SSDs are a natural high-performance replacement for disks in RAID systems, but because of their higher cost per bit, they are not yet cost-effective as a replacement for all disks in data centers [54]. An alternative is to replace only some disks, but because of the uniform distribution of I/Os in RAID5, replacing a subset of the drives with SSDs provides only a marginal performance improvement. However, because of its non-uniform distribution of I/Os—usually a disadvantage—the somewhat simpler and usually lower-performance

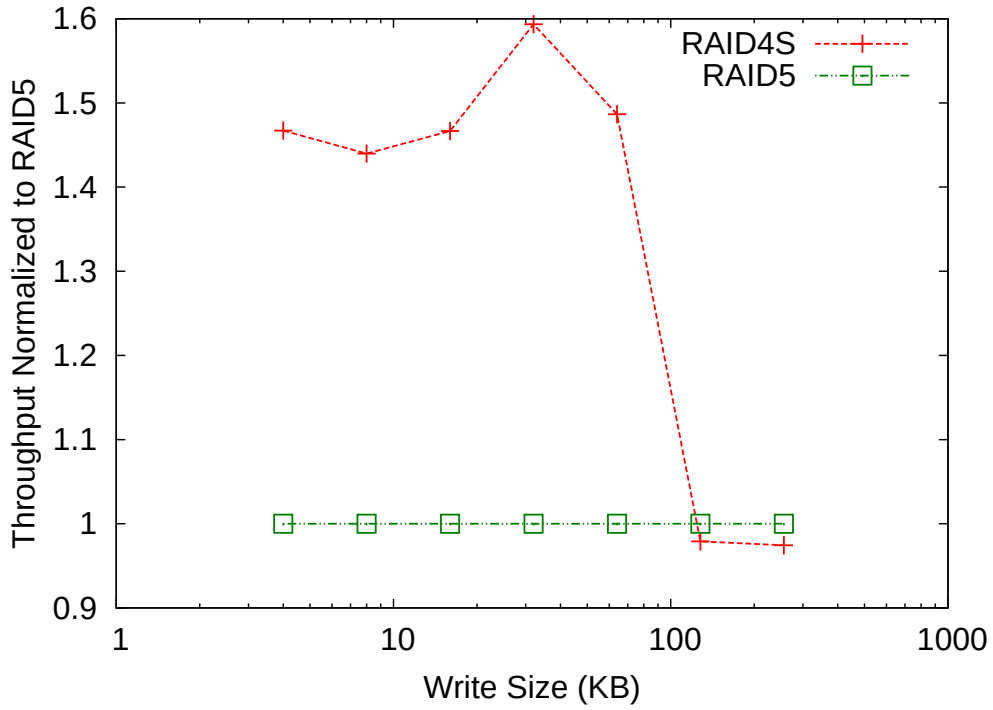

Figure 3.3: Normalized to RAID5, RAID4S performs up to 1.6X better. As expected, the RAIDs perform similarly for large writes.

RAID4 is an excellent candidate for augmentation with SSD. Our system, RAID4S, leverages the inherent non-uniformity of I/Os in RAID4 by replacing the single RAID4 parity drive with a faster SSD. The faster SSD alleviates the RAID4 parity bottleneck, offloads work from the slower devices, more fully utilizes the faster device, and as a result provides greater overall I/O performance than even RAID5.

By replacing the RAID4 parity disk with an SSD, RAID4S speeds up the parity accesses so that multiple small writes to different combinations of data disks may proceed in parallel in the time that the set of parity accesses occurs on the SSD. In the best case, where a set of small writes each span a single drive and the SSD is N times faster than disk, N separate I/Os may be performed to N data drives in the same time

that all N parity I/Os are performed on the SSD parity device. This allows complete parallelization of N small writes, which complete as a set in the time it takes to complete a single small write in RAID4 or $\lceil (N+1)/2 \rceil$ small writes in RAID5. Figure 3.3 shows a 1.6X throughput improvement over RAID5 on a random write workload.

Most workloads include small and large reads and writes. Large writes still happen at the speed of the slowest disk in RAID4S, but with one less disk, large writes may see a slight increase in performance. Sets of reads may be slightly slower on RAID4S as they are spread over N disks instead of N+1, but this effect is only seen on workloads containing $> 90\%$ reads. Overall, the performance improvement from RAID4S depends upon the speed of the SSD relative to the disks and the fraction of small writes in the workload. Although large writes see only marginal performance improvement, large writes and reads underutilize the SSD, providing additional time that may be spent performing parity operations for small writes and allowing a RAID4S system to provide additional speedups on mixed workloads with SSDs less than N times faster than disk. Degraded mode operation is also improved by RAID4S over traditional RAID4 and RAID5 by offloading the more frequent parity writes to a single faster device. Finally, the faster random-access performance of the parity device may enable more sophisticated reconstruction techniques.

This chapter incorporates flash-based SSD into traditional disk-based RAID arrays. We replace the parity disk with an SSD in order to reduce power and improve small write performance. Our initial simulation results show that power and performance are

improved. Our hardware microbenchmarks and trace replay results show significant small write performance increases over RAID5. Because we only replace a small number of disks with SSDs, the hardware replacement cost is low. We implemented RAID4S, a cost-effective, high-performance technique for improving RAID small-write performance using an SSD for parity storage in a disk-based RAID array.

Random write results show a 4HDD+1SSD RAID4S array achieves throughputs up to 1.6X higher than a similar RAID5 array. Section 3.8 shows the results of several other workload experiments. RAID4S replayed transaction workload traces up to 3.3X faster than RAID4 and up to 1.75X faster than RAID5. RAID4S has no performance penalty on disk workloads consisting of up to 90% reads and its benefits are enhanced by the effects of file systems and caches.

3.2 Related Work

The recent availability and lower cost of SSDs has encouraged new research into incorporating solid state memory into storage systems. In this section we review recent projects that use SSDs alone as well as incorporated into HDD-based systems. Lastly, we outline several related projects that used other types of solid state memory to achieve goals related to RAID4S.

All-SSD RAID arrays

Im and Shin [43] propose a RAID5-based system to improve reliability and reduce wear in an SSD array. Partial parity updates are computed based on the data being written, without reading old parity or other data to compute actual parity. These parity updates are stored in an NVRAM cache until periodically flushed. This technique could be used in our work to reduce parity updates for frequently rewritten data.

Balakrishnan et al. [8] propose Diff-RAID to address the issue of simultaneous failures due to wear out. By unevenly distributing parity across SSDs in a RAID5-based scheme, SSDs are more likely to fail independently. This work identifies a significant reliability issue with all-SSD RAID5, motivating the use of an alternate technique over the obvious SSD replacement scheme which is more likely to lead to data loss, thus improving overall data reliability. Our work does not directly address reliability and because there is a single SSD in our current work, we cannot easily use this clever idea.

Park et al. [57] recognize the SSD wear-out issue and the additional write load of storing parity. They consider an all-SSD RAID5 scheme with wear leveling to move parity to the device with lowest wear on that segment when it reaches a wear threshold. The performance evaluation compares to basic RAID5 and shows up to 28% better performance in an all-read workload. We chose to reduce the total system cost of incorporating SSDs into a storage system, so this technique cannot be used directly in our work. Section 3.9 discusses a multiple-SSD parity scheme we plan to look at for future work, where we could incorporate some of these all-SSD ideas to improve SSD

performance and reliability.

Hybrid SSD-HDD RAID arrays

Mao et al. [51] proposed HPDA, a hybrid SSD-Disk RAID array of several SSDs and two disks. The disks are used for buffering small writes and for recovery. Our work isolates parity on an SSD in order to leverage the better random read and write throughput compared to disks. Kim et al. [46] utilize workload requirements in their HybridStore storage system to place and migrate data between hard drives and SSDs. Both these works use SSDs in storage systems to address specific workload requirements that HDDs struggle to meet. We focus specifically on RAID and optimize for the small write workload.

Proximal I/O [68] uses an SSD buffer (called a staging area) for small writes, then intelligently schedules and places many small writes in a small subset of physical blocks. The disk services several writes per disk rotation, because they are located very near each other and accessing the next one doesn't require a full seek. The storage subsystem underneath the buffer is a disk-based RAID4. While proximal I/O provides a significant small write performance improvement, it requires a complicated custom RAID architecture to take advantage of optimized hard drive layout strategies. RAID4S uses existing RAID algorithms and installation is as easy as replacing the RAID4 parity drive with an SSD.

Other solid state technology

Hong et al. [40] investigate reliability in the context of arrays of MEMS devices. Larger storage capacities comparable to current disk sizes are realized by combining several MEMS devices in a RAID5 configuration. There are two main architectures of these RAID5 building blocks: 1) Use MEMS as a write buffer to improve disk write performance by amortizing many writes and writing to disk when idle and 2) MEMS cache to reduce disk reads. This work does not directly motivate our work, but it would be possible to do a similar RAID5 architecture of SSDs to get higher capacities. This is probably done internally in modern SSDs, so is outside the scope of this work.

Uysal et al. [81] introduce the logdisk architecture which uses a mirrored RAIDO, where one copy is a striped disk written as a log and the other is a striped MEMS device. This gives the best performance/cost ratio since writes to disk are fast and reads can be serviced by the faster MEMS devices. This work uses a more heterogeneous RAID architecture, by having mirrors that are made up of different devices. RAID4S doesn't particularly speed up reads compared to RAID5, aside from freeing up disks from as many writes by not storing any parity there.

3.3 Small Write Performance

Data reliability is maintained by introducing redundancy into storage systems but introduces several overheads. The most basic form of redundancy is mirroring, which

stores one or more identical copies of data. While mirroring improves reliability, the storage space overhead of mirroring is often too high. RAID 4 and RAID 5 calculate a parity across several disks (those disks are referred to as a stripe) and store the parity as redundant data. A single disk failure can be tolerated in a stripe by using the parity and remaining disks to recalculate the data that was on the failed disk. While this technique provides good reliability at lower storage space overhead, there is a performance impact for small writes.

The small write problem for RAID 4 and RAID 5 occurs when writes are small enough to not fill an entire stripe. As an example, consider a small write that fits in one block. In order to complete the write, the old data and corresponding parity block are read in order to compute the new parity. Then the new data and new parity are written. This results in four disk accesses when a mirrored layout would only require two.

Since parity is accessed every time data is written, parity blocks require greater performance than data blocks. RAID 5 spreads the parity so that each disk suffers the same additional load due to parity reads and writes. Our work is motivated by the observation that parity blocks simply require faster access times. By placing parity on SSD, the throughput of the system is improved because SSD reads and writes are generally much faster than disk reads and writes.

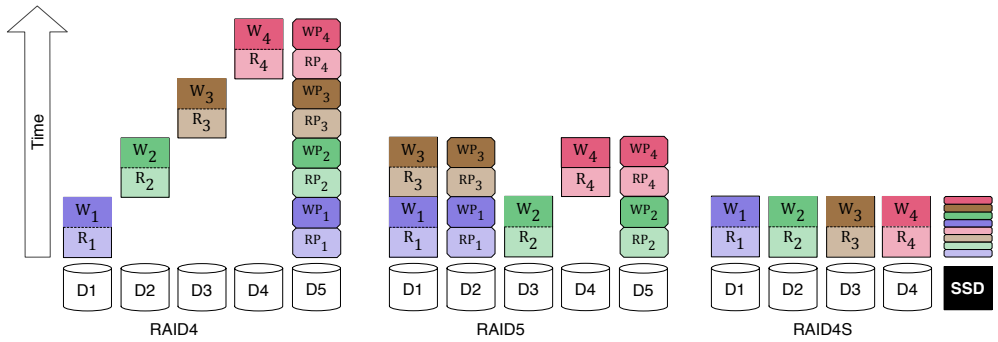

Figure 3.4: RAID4: parity drive is bottleneck; RAID5: data and parity accessed in parallel; RAID4S: parity offloaded to SSD and data accesses are parallelized.

3.4 The RAID4S System

As mentioned above, SSDs provide fast random I/O and much faster random read performance than disk drives, but are not yet cost-effective as a replacement for all disks in data centers due to their higher cost per bit [54]. RAID4S addresses the RAID small write problem of parity overhead by concentrating the parity I/Os on a single faster SSD device.

RAID4S has a significant advantage over RAID4 and RAID5. Not only are fewer total I/Os required for each hard drive, but the other hard drives are not affected by the small write. Figure 3.4 illustrates the different behavior for small writes on each RAID layout. RAID5 improves on RAID4 by distributing the parity update load over all of the disks in the array, but they must still handle the extra parity load. RAID4S replaces the parity disk in a RAID4 array with a fast SSD device, speeding up the parity accesses so that multiple small writes to different data disks may proceed in parallel while the set of parity accesses occurs on the SSD. If the SSD is N times faster than

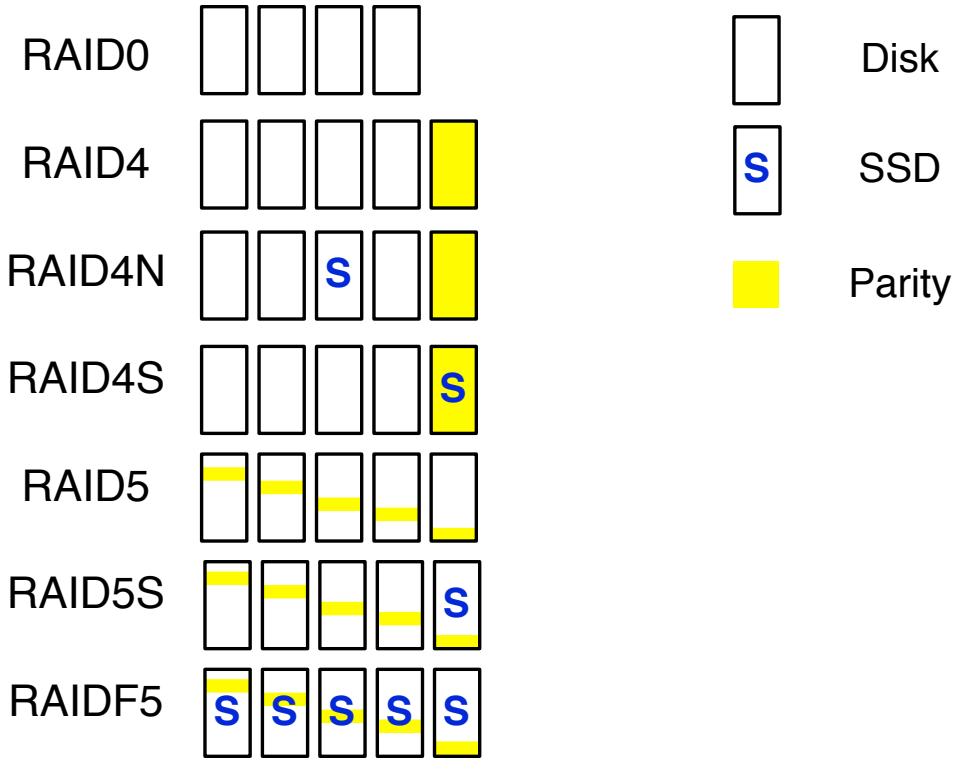

Figure 3.5: For RAID4 and RAID5, all devices consist of disks. RAID4S stores parity on an SSD and data on disks. RAID4N stores parity on a disk but replaces one data disk with the SSD and RAID5S replaces a disk with an SSD using the RAID5 parity layout. Lastly, RAIDF5 is a RAID5 layout with SSDs fully replacing all disks.

disk, N separate single-block I/Os may be performed in parallel to N data drives in the same time that all N parity I/Os are performed on the SSD parity device. This allows complete parallelization of the N small writes, which complete as a set in the time it takes to complete a single small write in RAID4 or $\lceil (N+1)/2 \rceil$ small writes in RAID5.

Figure 3.5 contextualizes RAID4S with other common RAID layouts. RAID0 is the most basic data striping strategy and does not store any parity data. RAID4 and RAID5 are the standard all-disk parity-based RAID layouts. RAID4N places an SSD in

a non-parity position whereas our RAID4S layout stores parity on the SSD. RAID5S also uses one SSD but, because each RAID5 device stores a distribution of data and parity, the SSD is not as fully utilized. RAIDF5 is an all-SSD layout that we use for cost analysis and a measure of top performance.

The choice of N is important in RAID4S. In the most extreme case, N parallel small writes may impose N I/Os on the single parity device, requiring the SSD parity device to ideally be N times faster than the disk drives. In the average case, in which large and small reads and writes are intermixed, a slower parity device is sufficient. This may be a problem as SSD costs are reduced by higher densities in technologies known as MLC (multi-level cell) and TLC (triple-level cell). Increasing flash density reduces random I/O performance and increases wear, so SLC (single-level cell) flash SSDs are used in this work.

Most workloads include a mix of small and large reads and writes. RAID4S large writes still happen at the speed of the slowest disk, but with one less disk large writes may see a slight increase in performance in RAID4S. Overall, the performance improvement from RAID4S depends upon the speed of the SSD relative to the disks and the fraction of small writes in the workload. As caches grow and become more effective, large storage workloads are increasingly write-intensive. Some techniques (e.g., WAFL [37]) transform small writes into large writes, but some small writes are inevitable, particularly as the disks fill up. RAID4S addresses this problem. Although large writes are expected to see only marginal performance improvement, large writes

Device	4KB (IOPS)	Capacity (GB)	Cost ($)	$/IOPS	$/GB
Intel X25-E	3,300	64	729.99	0.22	11.4
Intel 510	8,000	250	569.49	0.07	2.28
OCZ Vertex 3	55,000	240	434.99	0.01	1.81
OCZ IBIS	120,000	960	2,699.00	0.02	2.81
WD2503ABYX		250	69.99		0.28

Figure 3.6: Price comparison for several modern SSDs and one HDD. OCZ IBIS provides cost-effective high performance, but the large capacity comes at too high a cost. The OCZ Vertex 3 is a more viable option for replacing a small hard drive.

and reads underutilize the SSD, providing additional time that may be spent performing parity operations for small writes and allowing a RAID4S system to provide additional speedups on mixed workloads with SSDs less than N times faster than disk. Degraded mode operation is also improved by RAID4S over traditional RAID4 and RAID5 by offloading the more frequent parity writes to a single faster device. Finally, the faster random-access performance of the parity device may enable more sophisticated reconstruction techniques.

3.5 The Low Cost of RAID4S

RAID4S is particularly appealing because it provides a way to upgrade an existing HDD-based storage array without replacing all storage. An existing RAID4 system can perform up to 3.3X better by replacing the parity drive with an SSD. Several researchers analyzed the costs to replace hard drives with flash SSDs and found SSDs to be too expensive to replace disks [44, 54]. However, none have investigated replacing

a subset of hard drives in a parity-based scheme. We computed the cost of several RAID+SSD architectures and found RAID4S to be a good compromise between cost and performance. Figure 3.6 outlines the characteristics of a few currently available SSDs and a disk drive of similar size [55].

We created a cost-benefit analysis of the OCZ Vertex 3 with our hardware results using the older Intel X25-E. RAIDF5 is a RAID5 layout with all hard drives replaced by SSDs. Figure 3.7 shows that RAID4S provides 1.75X higher throughput than RAID5 at about twice the total cost. RAIDF5 provides a higher throughput increase for its cost, but the total cost is prohibitively high at about 6X over RAID5 hardware. While a better result would enable 2X performance improvement at 2X cost, RAID4S has a much lower total cost than RAIDF5 and provides an incremental improvement.

The cost used is the total cost of all hard drives and SSDs for each system. It would be even cheaper to augment an existing RAID4 system with SSDs by replacing one hard drive with an SSD. We ignored the other machine hardware that would be required, thus reducing total system cost and highlighting the difference in storage costs. If we had included the other hardware common to all RAID systems (which is likely to be more expensive than the price of five hard drives), the ratios between the systems would be reduced and RAID4S might seem unfairly price competitive. Also, because the Intel X25-E we used is slower than the SSDs we chose for price comparisons, the performance shown is lower than we'd expect with the newest and cheapest SSD. This means that at the cost shown, it's likely that performance with the OCZ Vertex 3 would

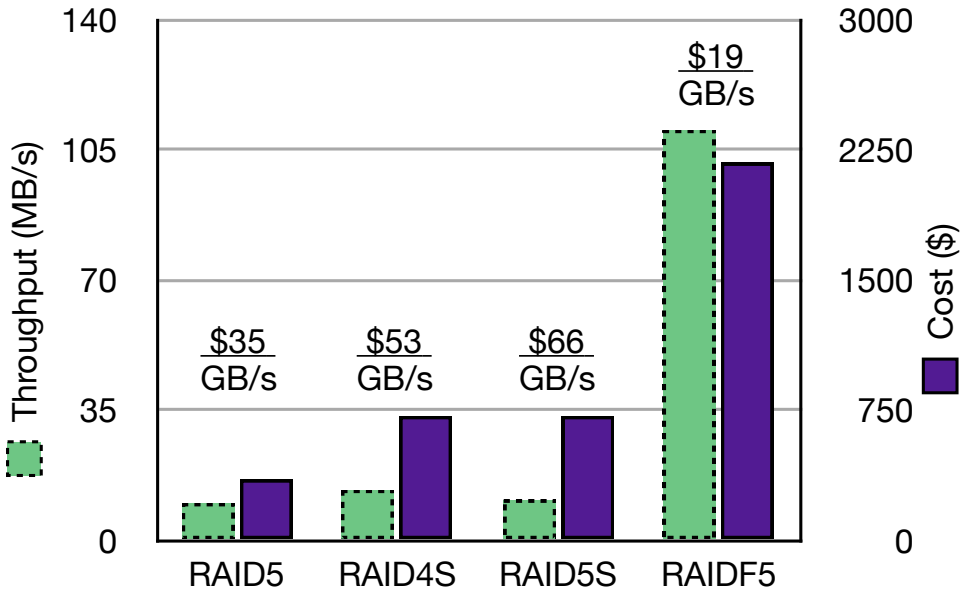

Figure 3.7: SSDs are an obvious, yet expensive, enhancement to classic RAID systems. Replacing all devices in a RAID system gives a large performance improvement but at a very high cost. RAID4S splits the difference; with a cost 2X higher than RAID5, bandwidth for small write workloads is up to 1.75X higher than RAID5. The numbers above each RAID configuration are the ratio between cost and performance for that layout.

actually be higher. In the case of RAID4S, this means the SSD would support larger array widths and even more write-intensive workloads.

3.6 Reduced Power Consumption

Large disk arrays consume a great deal of power. We propose a system where up to 50% of the disk drives can be powered down in the average normal operating case. The first step is keeping disks powered down for longer periods of time. Data updates in an array using RAID or other redundancy techniques require access to more than one disk. We

	Read (W)	Write (W)	Idle (W)	Sleep (W)
Samsung 64GB SLC SSD	0.5	0.35	0.2	0.2
Western Digital WD20EADS	6.00	6.00	3.7	0.80

Table 3.1: Power consumption in Watts of I/O operations for a Samsung solid state drive and a Western Digital hard drive.

	Throughput (MB/s)	Seek Time (ms)
Samsung 64GB SLC SSD	100.0	0.0
Western Digital WD20EADS Drive	77.9	13.2

Table 3.2: Performance of I/O operations for a Samsung solid state drive and a Western Digital hard drive.

propose adding SSD to our storage system and analyzed the expected power reduction.

Our first experiment compares the power requirements of a single disk and SSD performing synthetic workloads. The experiment uses data from vendor specification sheets and a simulated workload to run the experiment against. We chose to use the Western Digital WD20EADS [4], a low-power SATA hard drive. Its performance is not advertised in the specifications, but its power requirements are lower than high-performance hard drives. For the SSD, we chose a typical flash device that was available for purchase, the Samsung 64 GB SATA SLC SSD [3]. The power requirements of the two devices are summarized in Table 3.1. The expected performance characteristics are summarized in Table 3.2. Throughput and seek times are not provided in the specifications, so these are actual numbers measured by Tom's Hardware using the h2benchw benchmark [69].

We measure the total power to run a workload through the system while varying the

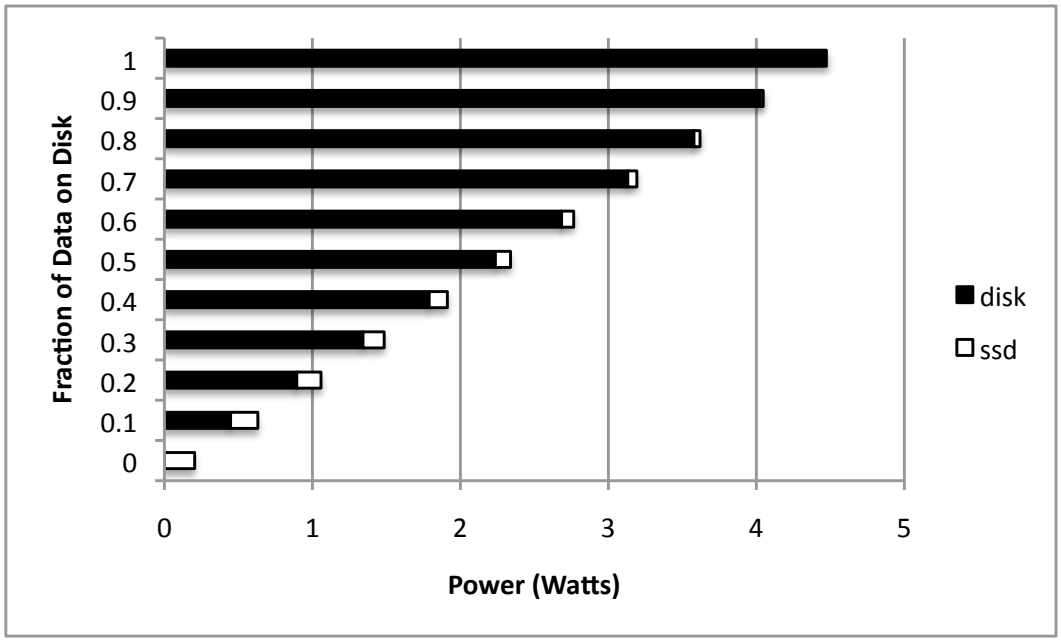

Figure 3.8: Power consumption of random read workload storing data on disk vs. flash. Reading data from SSD consumes significantly less power than disk.

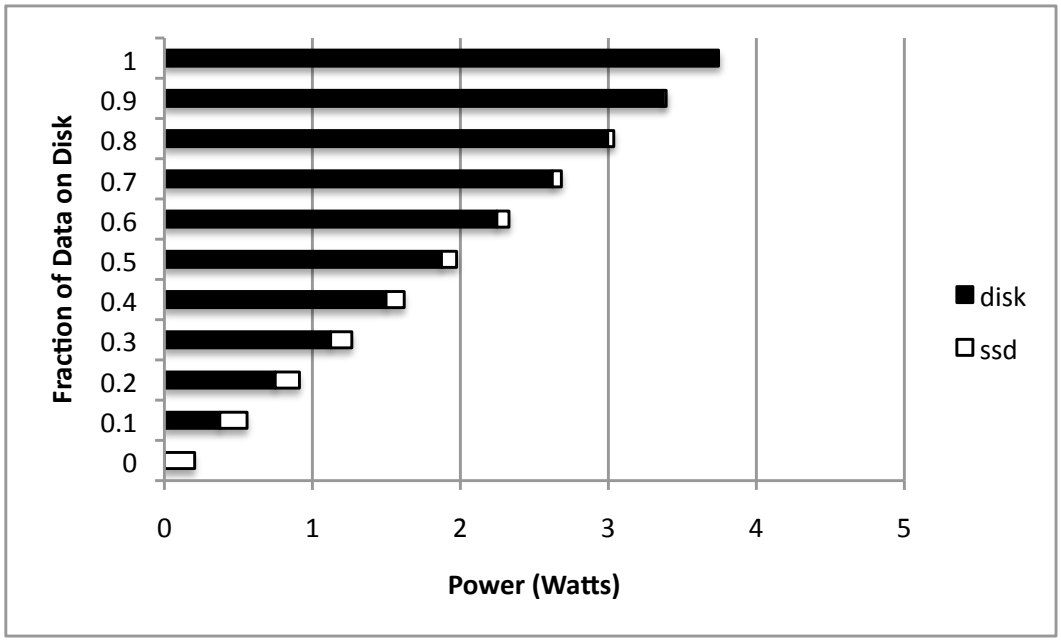

Figure 3.9: Power consumption of sequential read workload storing data on disk vs. flash. Reading data from SSD consumes significantly less power than disk.

amount of SSD storing a fraction of the data. The workload is a synthetically generated read workload and has a fixed rate of 1.5 MB/s with a request size of 64 KB. We calculate the number of I/O events per second and use that to determine the amount of time spent reading. The random workload performs a seek for every read request, thus reducing the amount of time spent idle. The hard drive specification does not mention seek power consumption so we assumed that it is the same as read power consumption since the disk is spinning while the seek takes place. We calculated the total power consumption to run the workload on disk and SSD, then varied the fraction of each to show how much SSD can improve power consumption for a disk-SSD hybrid array. Figures 3.8 and 3.9 show that SSD uses less power than disk. The results show that replacing the workload of just a fraction of the disks dramatically reduces the power used by the storage system. The random workload consumes slightly more power on disk, but the power consumption of SSD does not depend on the workload.

We started looking at RAID layouts and started with the power consumption of storing data in a mirrored layout. We have a storage array with two devices, either both disk or one disk and one SSD. The hard drive and SSD are the same as in the previous experiment [3, 4]. We simulated the power reduction caused by storing the mirror on SSD and the result is in Figure 3.10. As expected, the power consumption is reduced almost by half by replacing half of the disks with SSD. However the hardware cost to replace so many disks is quite high, so we looked at other RAID arrangements.

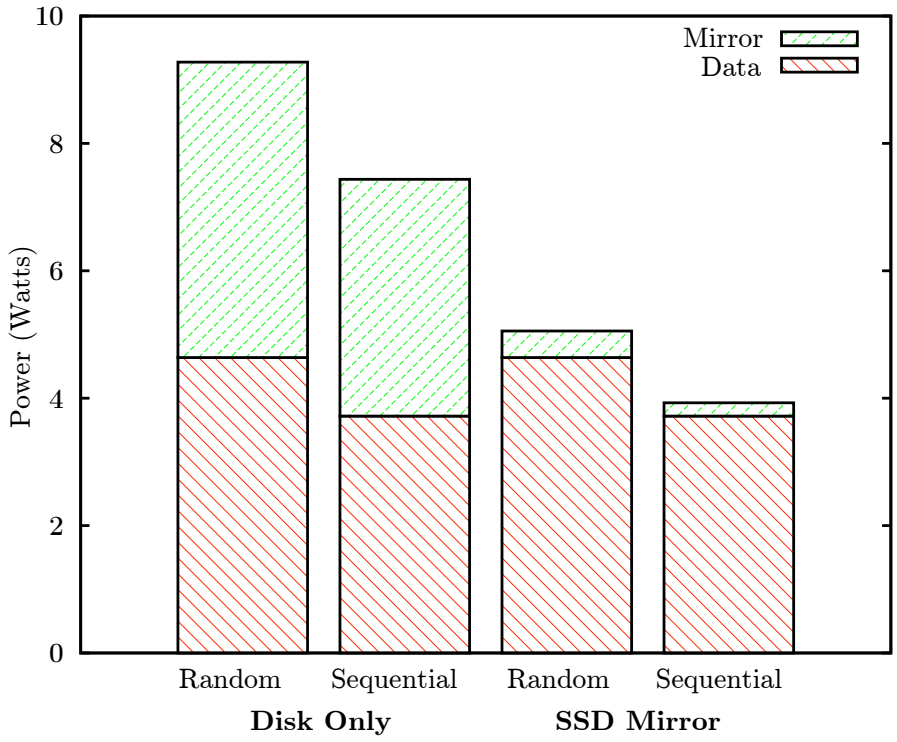

Figure 3.10: Two-disk configuration measuring power consumption of a write workload with data written to disk and mirror written to disk or SSD.

3.7 RAID4S Simulation Results

The RAID4 layout stores parity on a dedicated disk. Data is located on the remaining two or more disks in the stripe. Data is written to the disk in one of two ways, depending on the size of the write. For large writes which are at least the size of an entire stripe, data and parity can be written directly. For smaller writes, the old parity must be read from disk in order to recompute the new parity while leaving old data in the stripe. This overhead is more significant than is apparent because disks have poor random access

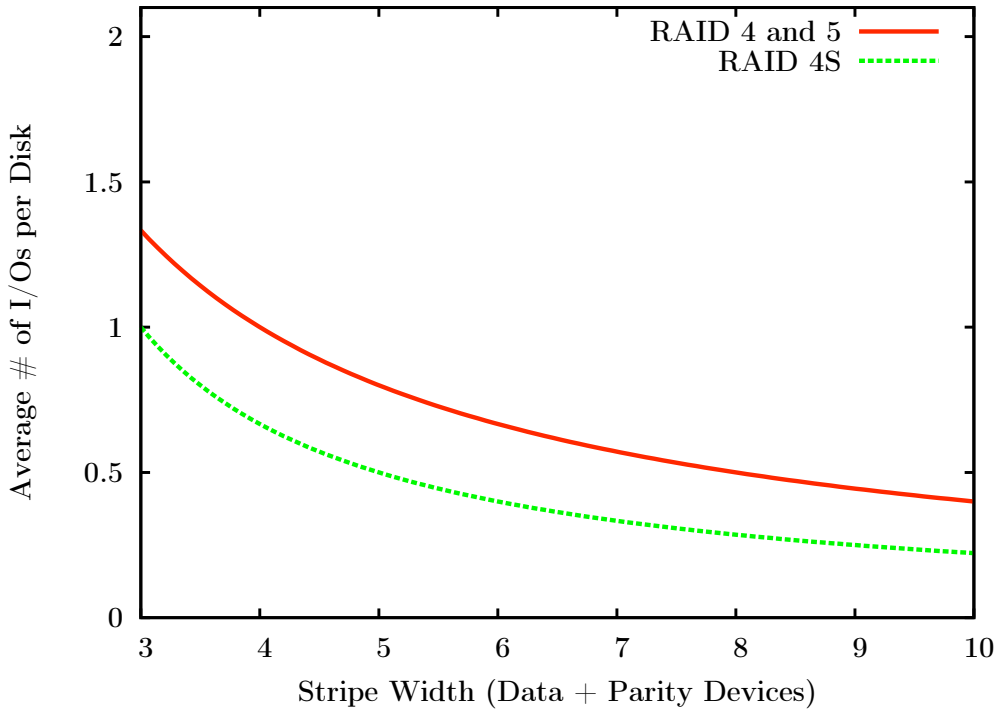

Figure 3.11: Average number of I/Os per disk for a small write workload.

performance.

RAID4 has a bottleneck on the parity disk, so its actual performance in a disk-based configuration is worse than RAID5. By replacing the parity disk with SSD, we show that performance is actually better than RAID 5. Consider a small write that is smaller than a full stripe. The old data and old parity are read, the new parity computed using the new data, and finally new data and new parity are written. Note that in order to compute the new parity, we must do an exclusive-or operation with the old parity and old data to take out the old data. Then an exclusive-or between this intermediate parity and the new data computes the new parity. This is a total of two reads and two writes.

Stripe Width	RAID4	RAID4S	% Reduction
3	1.33	1	25
4	1	0.67	33
5	0.8	0.5	38
10	0.4	0.22	44
50	0.08	0.041	49
100	0.04	0.020	49.5
200	0.02	0.010	49.7
300	0.013	0.007	49.8

Table 3.3: Reduction in the average number of I/Os per disk required for every small write in RAID4 and RAID4S arrays.

For RAID4 and RAID5, those reads and writes are amortized over the disks in the stripe. Note that the distribution of I/Os is different in RAID4 and RAID5. The RAID4 parity drive services more I/Os than the rest of the drives. However, the average number of I/Os serviced per disk is the same in both cases. To compute the average number of I/Os per disk for a disk-based RAID4 or RAID5 arrangement, we have:

$$RAID5 = \frac{4}{W} \tag{3.1}$$

where W is the number of devices in the stripe. This includes the data disks and parity disks. For the RAID4S case, we have reduced the number of I/Os going to disk as well as the number of disks in the stripe. We compute the average number of I/Os as:

$$RAID4S = \frac{2}{W-1} \tag{3.2}$$

Figure 3.11 shows this result. As expected, the average number of I/Os per disk is reduced by adding SSD. Note that disks in RAID4 and RAID5 arrays of the same

stripe width service the same average workload. In RAID4, the parity drive in fact services half the total I/Os for a small write workload. The variance in disk workloads is quite high in RAID4 and that is a reason why it is not typically used. Because RAID5 distributes parity across all drives, the average number of I/Os serviced by each drive is actually very close to the actual number serviced by each drive. In RAID4S, the average workload for the remaining disk drives also has very little variance since all parity is offloaded to the SSD parity device.

For small stripe widths, the number of I/Os serviced by the disks in the array is reduced by 25-44%. Notice that the amount that the disk workload is reduced by using a RAID4S configuration instead of a RAID4 configuration depends on the stripe size. Table 3.3 shows the percentage of reduction in workload caused by replacing parity drives with SSDs for various stripe sizes. When the stripe width is small, the parity disk that is removed to be replaced by an SSD makes a noticeable impact on the workload left to the remaining disks. In other words, offloading the parity workload of a large array results in a higher reduction in the average workload because the parity device is working harder. For example, the workload is reduced only 25% in a 3-device array because the average workload goes from 4 I/Os on 3 drives to 2 I/Os on 2 drives. Removing one drive to replace with an SSD makes an impact on the workload per drive. For larger stripe widths, removing just one disk does not affect the remaining workload as much.

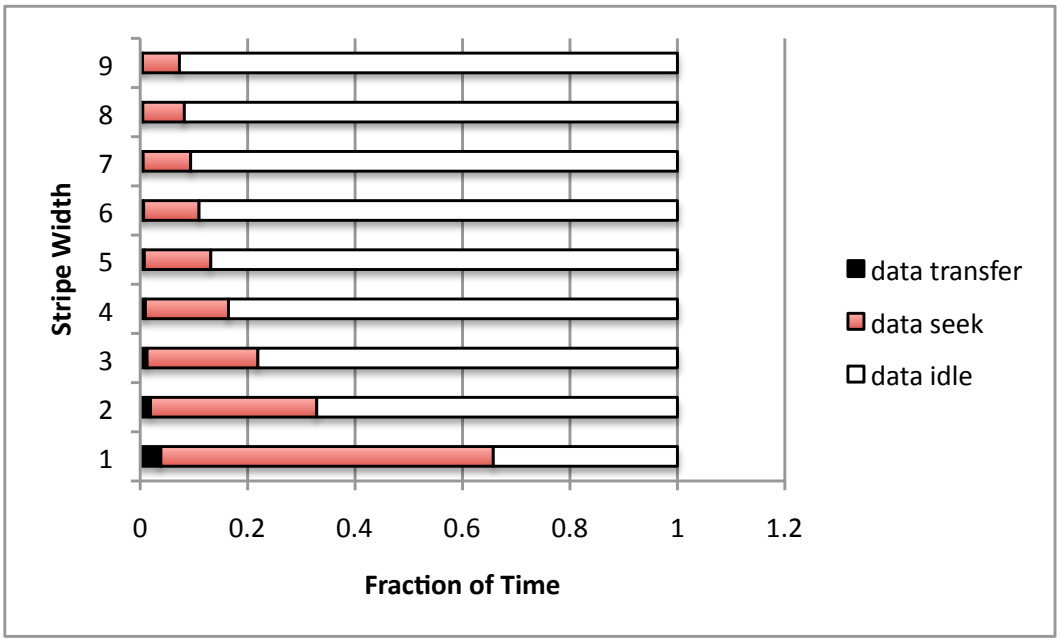

Figure 3.12: Breakdown of disk behavior when performing a 1.5 MB/s random write workload on a striped array with no redundancy. Larger stripe widths distribute the write load so that each disk spends more time idle.

Simulated array performance

The first performance simulation measures the amount of time each disk spends transferring data, seeking, and idle. We use the same disk described in Tables 3.1 and 3.2. Figures 3.12 and 3.13 show that with larger stripes, each disk spends proportionally more time idle. This is because the write workload is spread across more disks.

3.8 Experimental Methodology & Results

Our experimental results highlight the small-write performance benefits of RAID4S. We use the XDD benchmark [67], which can perform I/O to block devices without a

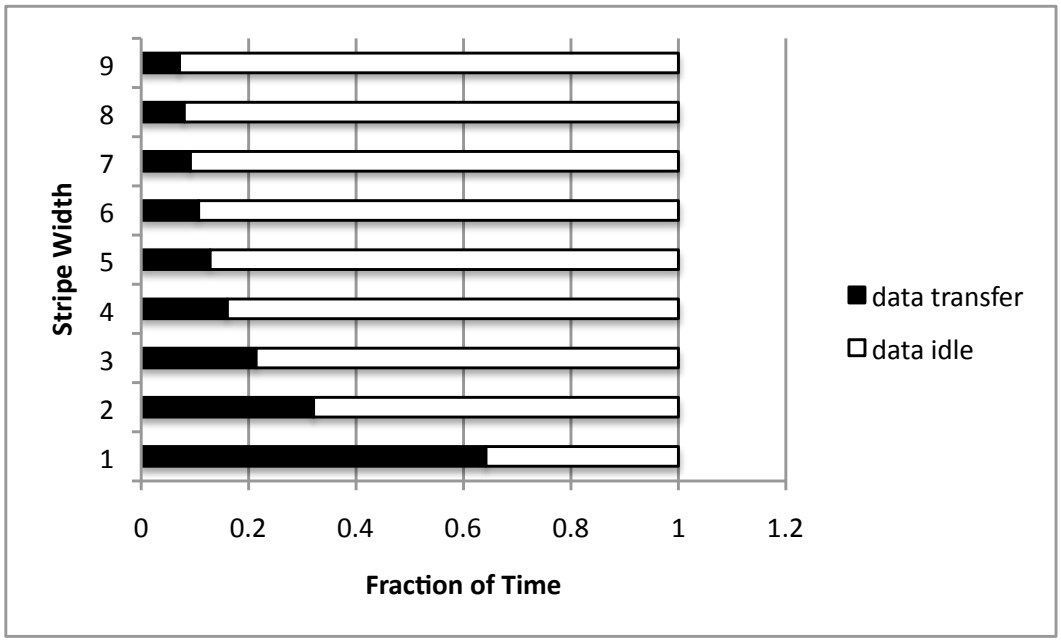

Figure 3.13: Breakdown of disk behavior when performing a 50 MB/s sequential write workload on a striped array with no redundancy. Larger stripe widths distribute the write load so that each disk spends more time idle.

file system, run a database benchmark, and replay workload traces created by an OLTP workload at a financial institution. Experiments were run on a machine with an Intel Core i7 CPU with 2GB RAM and 9GB swap space running Ubuntu Linux 10.04. The software RAID was set up with mdadm 2.6.7.1. We used 640GB 7200 RPM Western Digital Caviar Black SATA 3.0Gb/s hard drives with 32MB cache and 64GB Intel X25-E SSDs. The disk array consists of five disks, unless specified otherwise, with data striped in 64KB chunks. Each RAID configuration is implemented on disks or SSDs using 12.25GB device partitions, starting at the outer-most sectors. For the investigated configurations (see Figure 3.5) we used five different partitions, all within the outer 10% of the cylinders of each disk.

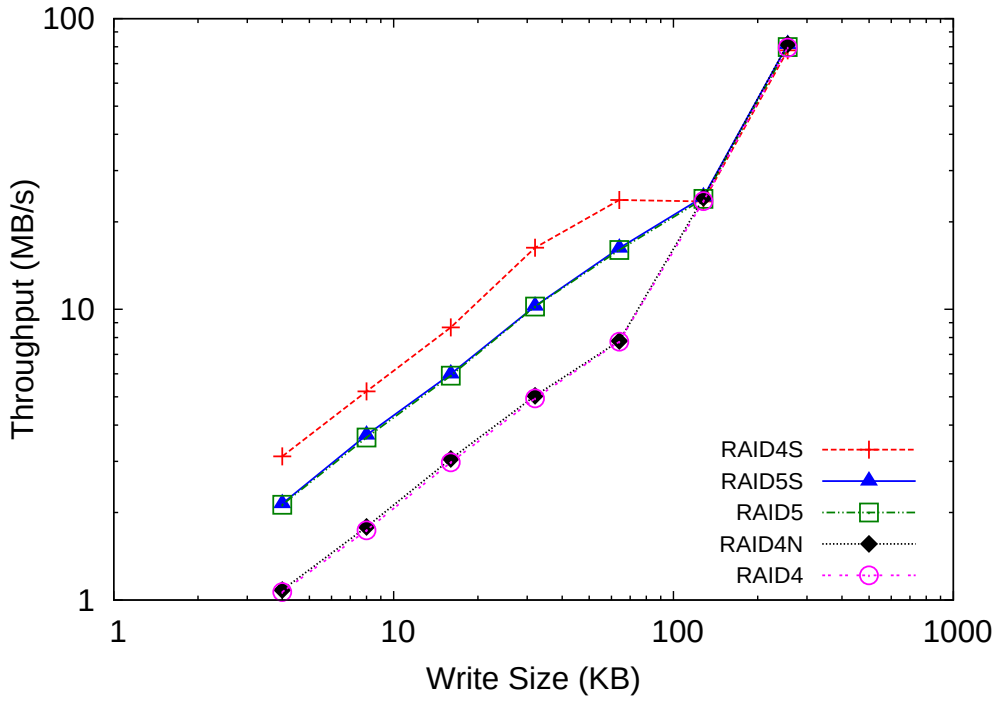

Figure 3.14: The random write throughput experiment bypassing the file system cache shows that RAID4S provides nearly 1.6X throughput over RAID5 for random I/Os under 128KB. These small write sizes highlight the throughput difference between each RAID layout. At 128KB, the parity update is completed as a reconstruct-write; thus the RAID4S SSD is only used for the final parity write and provides little benefit.

Block aligned random write

The XDD benchmark is configured to perform random writes of a fixed size for each experiment. We use direct I/O to bypass the buffer cache and send writes to the RAID devices with a queue depth of 10. The reason we avoid buffer caching is to show the performance for random I/O that is likely to result from servicing requests that don't fit in the cache. Each experiment was repeated 3 times and the average is plotted. The length of each experiment is a total of 256MB of random writes of various sizes written to each RAID device; this run length was chosen to be large enough to make the error

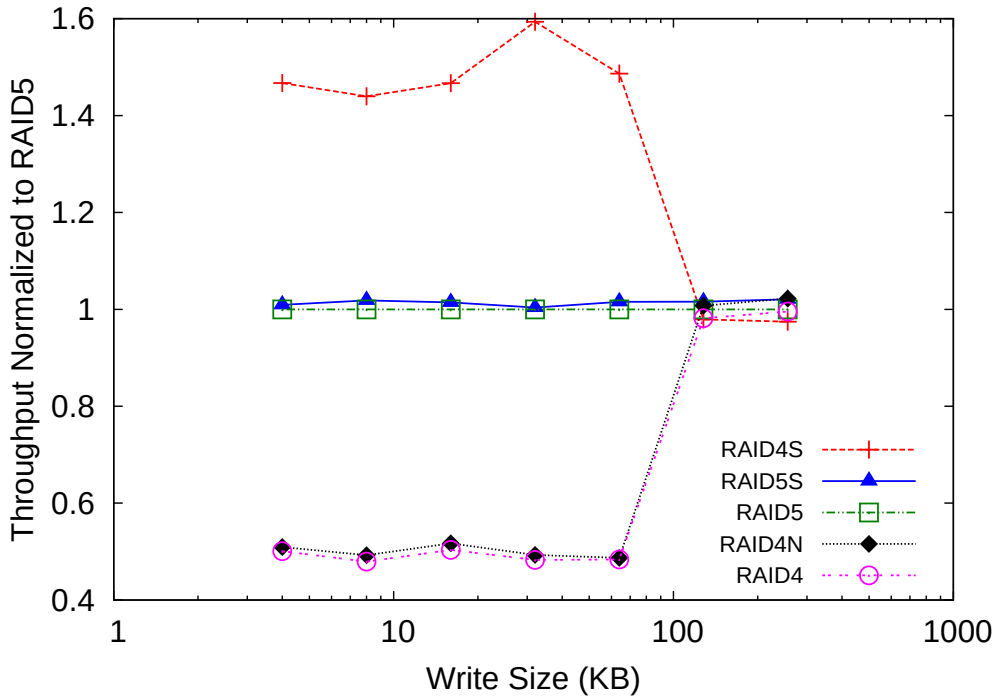

Figure 3.15: Normalized to RAID5, the highest throughput over RAID5 is nearly 1.6X at 32KB.

bars negligible.

Figures 3.14 and 3.15 show the throughput of random writes varying from size 4KB to 256KB in powers of two and writing a total of 256MB. At 128KB, the throughputs approach each other, indicating that the RAID algorithm has changed at this point and is no longer doing small writes. We verified this by running iostat to measure total reads and writes; further details are found in Section 3.8.

Block unaligned random write

Because RAID4S performs well for small random writes, we investigated additional write sizes. We looked at small non-4KB-aligned random writes varying from 1KB

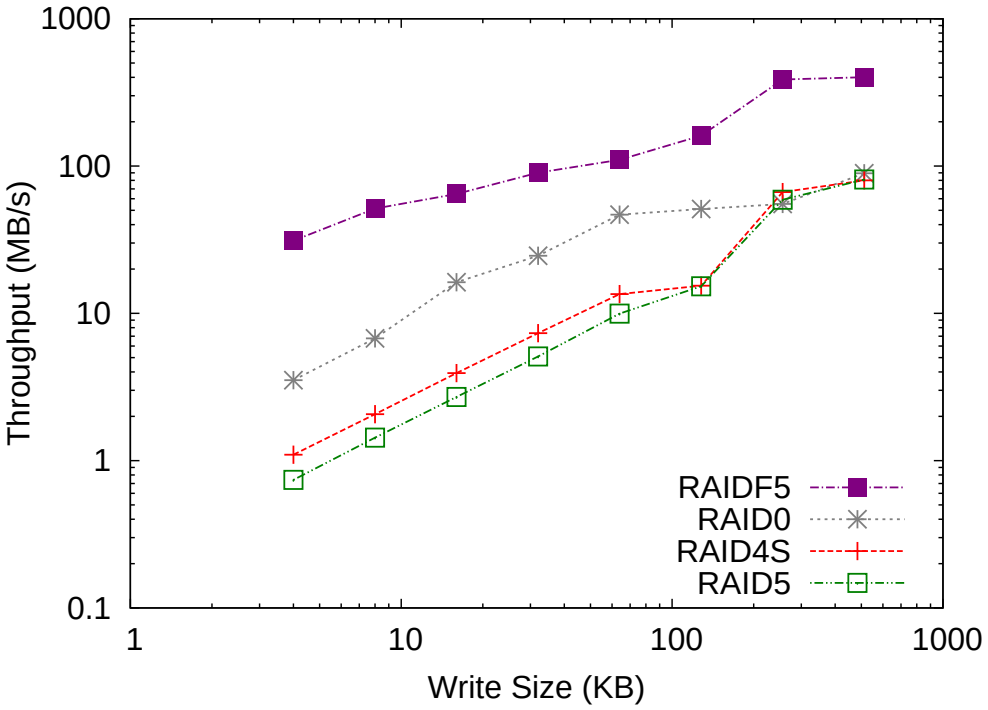

Figure 3.16: The biggest gain in throughput from SSDs is for small random writes. At larger sizes, there is a smaller difference between RAIDF5 and RAID0.

to 16KB in increments of 1KB. We ran each experiment until 256MB of I/O was completed. Figures 3.17 and 3.18 show the result of this experiment. Because the device sector size is 4KB, writing to sizes that are not multiples of 4KB results in significantly degraded performance. SSDs see an additional performance impact from unaligned writes because of how writes are performed. An SSD cannot simply overwrite data in place, but must first erase the block that is to be overwritten. Wear leveling techniques aim to maintain erased blocks ready for new writes, reducing this overhead. Older SSDs may have larger erase block sizes, so care must be taken to avoid writes that are significantly smaller than that size. While we do not expect to see these types of

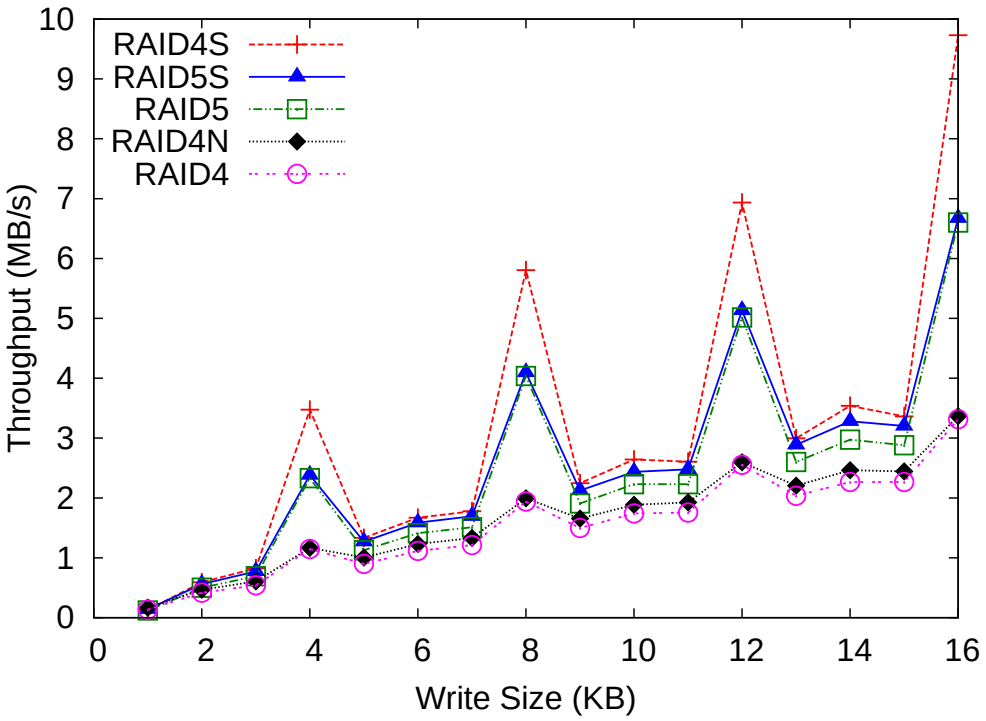

Figure 3.17: Small, irregular writes show the degraded performance where the I/O size is not a multiple of 4KB. At the 4KB points, the benefit of RAID4S is more pronounced.

unaligned workloads in practice, we show that RAID4S does not perform worse than RAID5 in this situation and in fact performs better.

Further analysis with iostat

The iostat tool in Linux provides CPU and I/O usage data. We used iostat to analyze the workload seen at the devices in our RAIDs. We compared the data from iostat to our expectations based on the input workload and the characteristics of the RAID algorithms. This provided us with better understanding and confirmation of our results.

The first iostat analysis looks at the read count at each device during a write workload. Figures 3.19, 3.20, and 3.21 show this for the same random write experiment

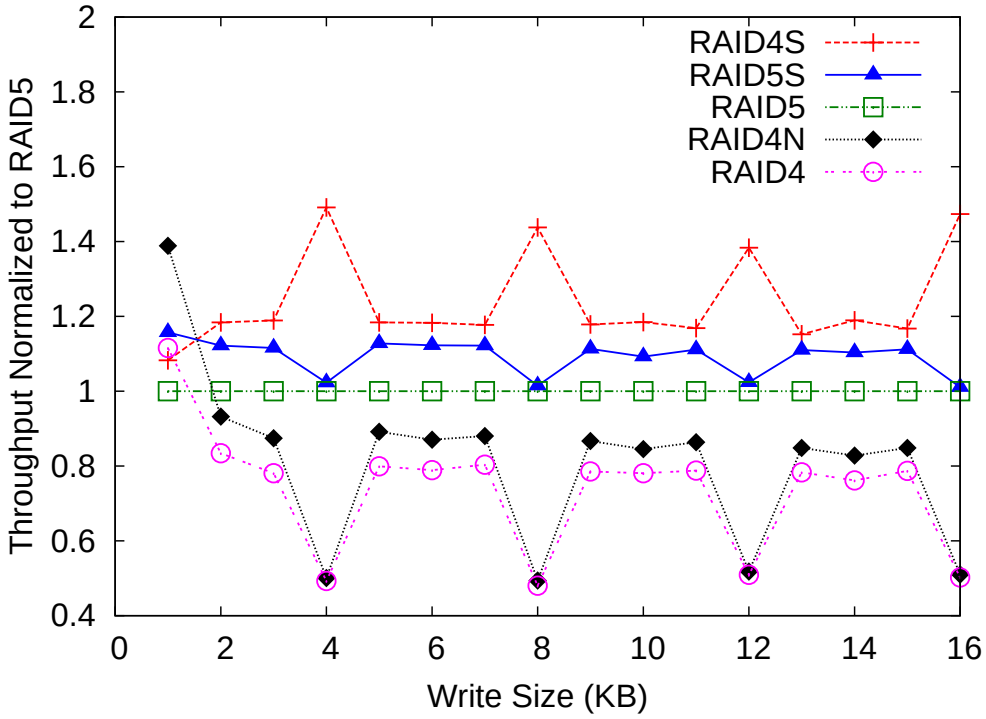

Figure 3.18: Normalized to RAID5, the benefit of RAID4S is more pronounced. At multiples of 4KB, RAID4S is about 1.4X faster than RAID5.

shown in Figure 3.14. The existence of reads for small write sizes indicates the small write problem, where data and parity are read in order to compute parity. For small writes that fit on a single device, a read-modify-write occurs and the parity is read frequently. Small writes to half the stripe perform a large write. For 128KB writes, the parity device is not read at all. Instead, the remaining data devices in the stripe are read and parity is computed without reading from the parity device. Thus for one half to one full stripe writes, RAID4S is not expected to perform better than other RAID layouts. For 256KB writes, no reads occur because these are full stripe writes and parity is computed with the 256KB of data to be written.

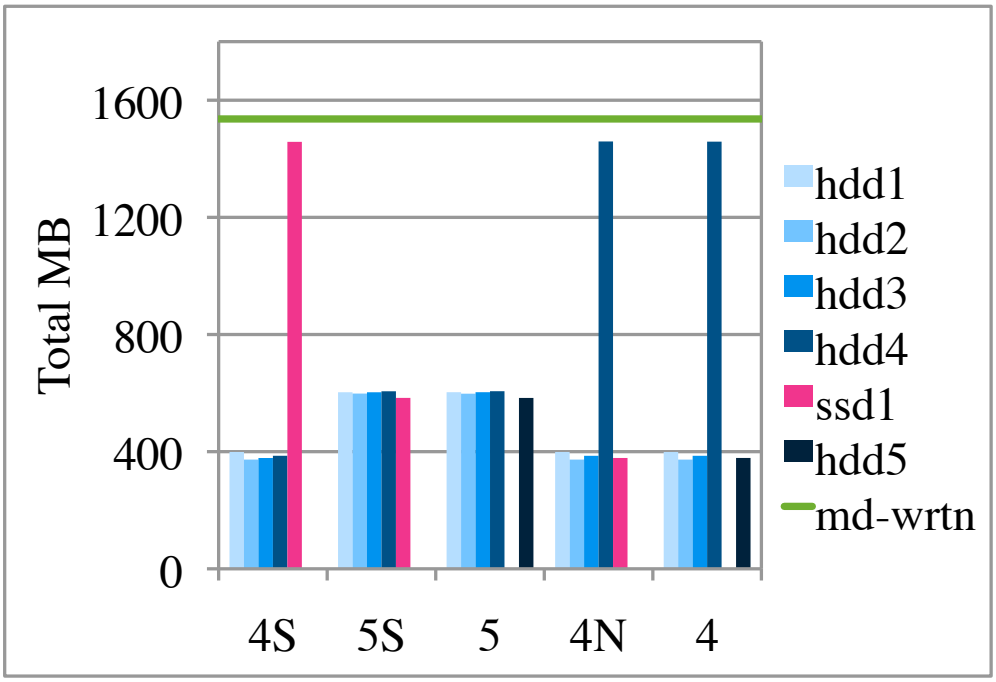

Figure 3.19: 64KB read count at each device during a random write workload. Each write fits on a single device, so each one requires a parity read. This is seen in the RAID4-variant spikes; these workloads are skewed so that the parity device works harder than the data devices. This enables the benefits of RAID4S, since the SSD is faster than the drives.

The next iostat experiment investigates the device write count when completing the same write workload. Figures 3.22, 3.23, and 3.24 show the write count of each device within each RAID while performing the random write workload. As we saw with reads, the write workload to each device changes from 64KB to 128KB. Array small writes 64KB and below fit in a single RAID chunk on one device. In RAID4 and RAID5, these are completed by doing a read-modify-write. At 128KB, writes now fit on two (out of four) data devices. Every time we update parity, two data devices have been written to. This is why there are fewer parity writes than we see in the 64KB case. At 256KB, each write exactly fills a stripe so the workload is the same for RAID4- and RAID5-based

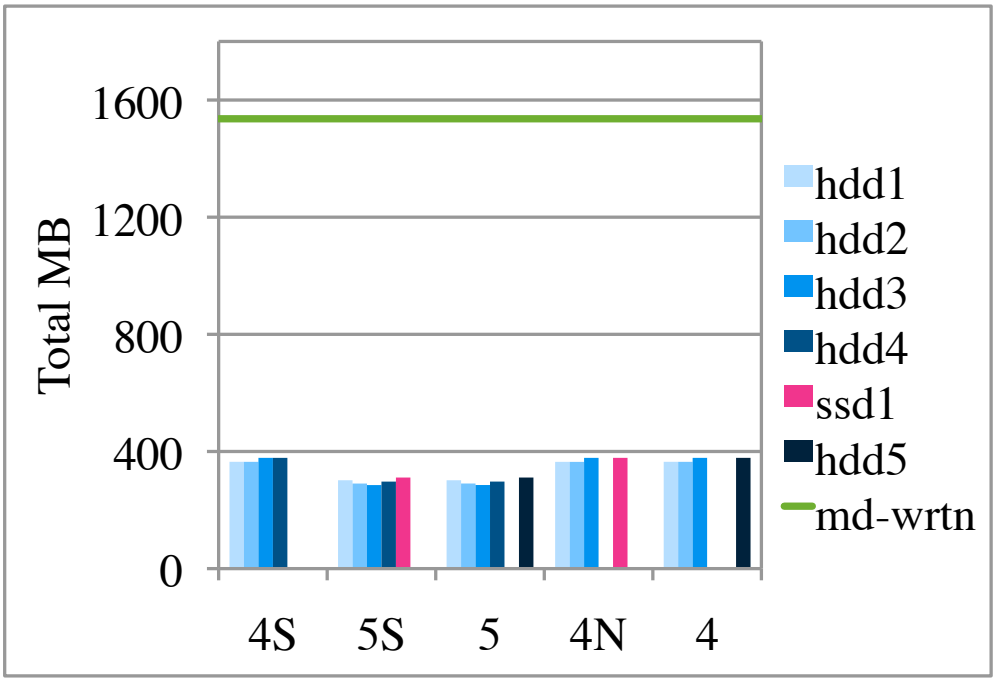

Figure 3.20: 128KB read count at each device during a random write workload. This is the half stripe case, so parity is not read.

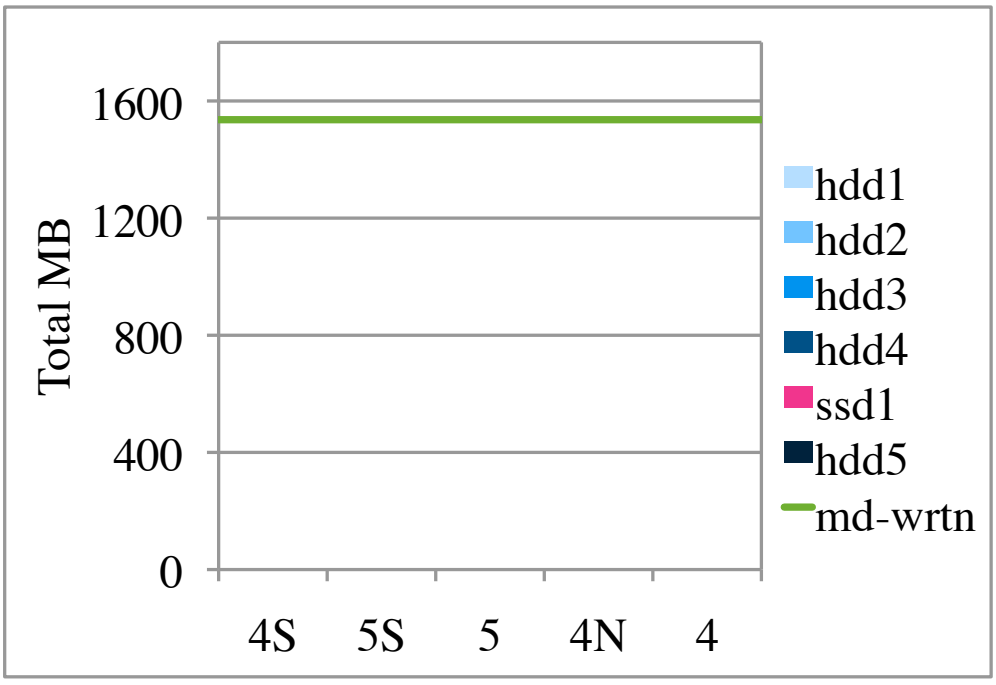

Figure 3.21: Read count at each device during a random write workload. At 256KB, there are no reads because it is a full stripe.

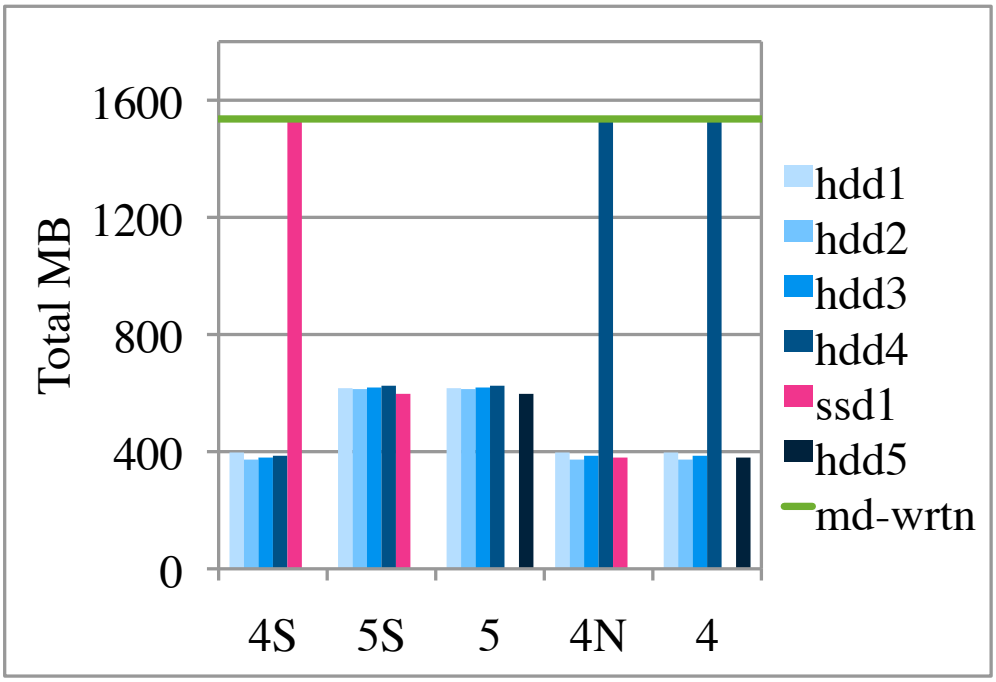

Figure 3.22: The write count to the RAID device is the same for all write sizes and is the total size written in the experiment. The small write size show that RAID4-based RAIDs write to the parity device much more frequently than any other device.

data layouts.

Random I/O varying read-write ratio

Figure 3.25 shows the result of running a random I/O workload varying the ratio of reads to writes. RAID4S beats the other RAID layouts up to 90% read workloads. This is promising because mixed workloads are more common than write-only workloads in non-archival storage. Our expected system has a large cache which services many reads, leaving mostly writes to be serviced by the storage system. This result shows that RAID4S is well suited to this type of data requirement.

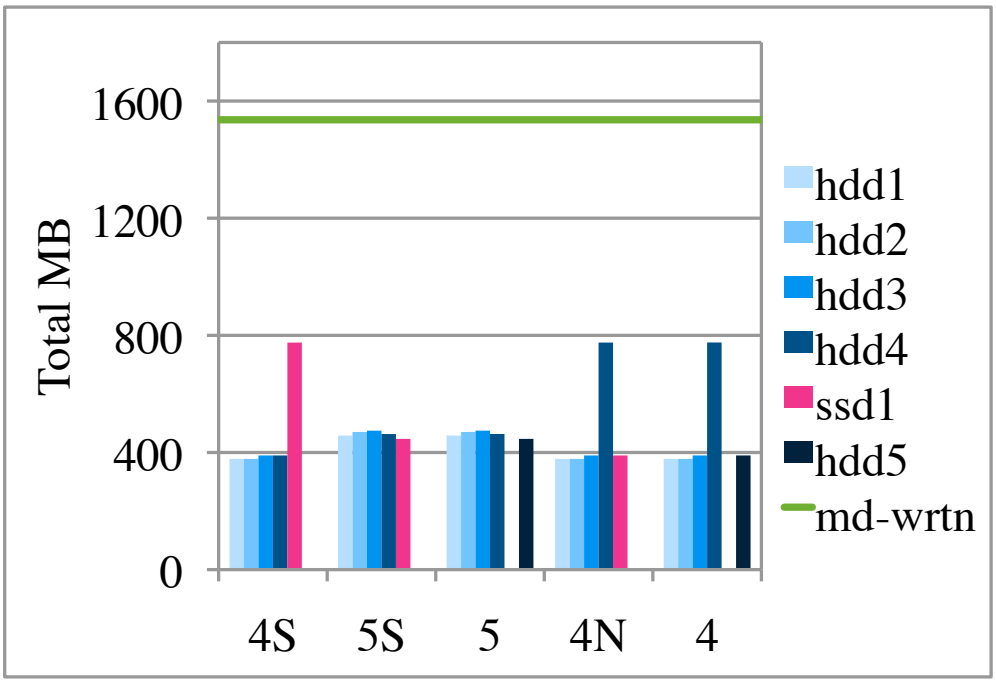

Figure 3.23: Half stripe writes show RAID4-variant parity devices doing more writes than data devices.

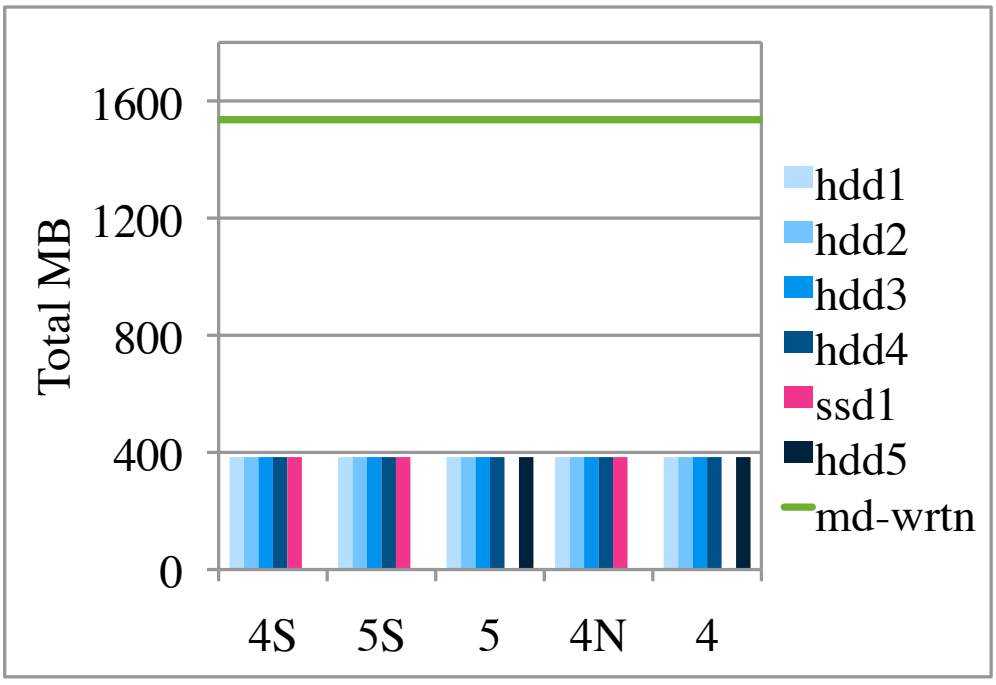

Figure 3.24: At full stripes, all RAIDs complete the same number of writes to each device.

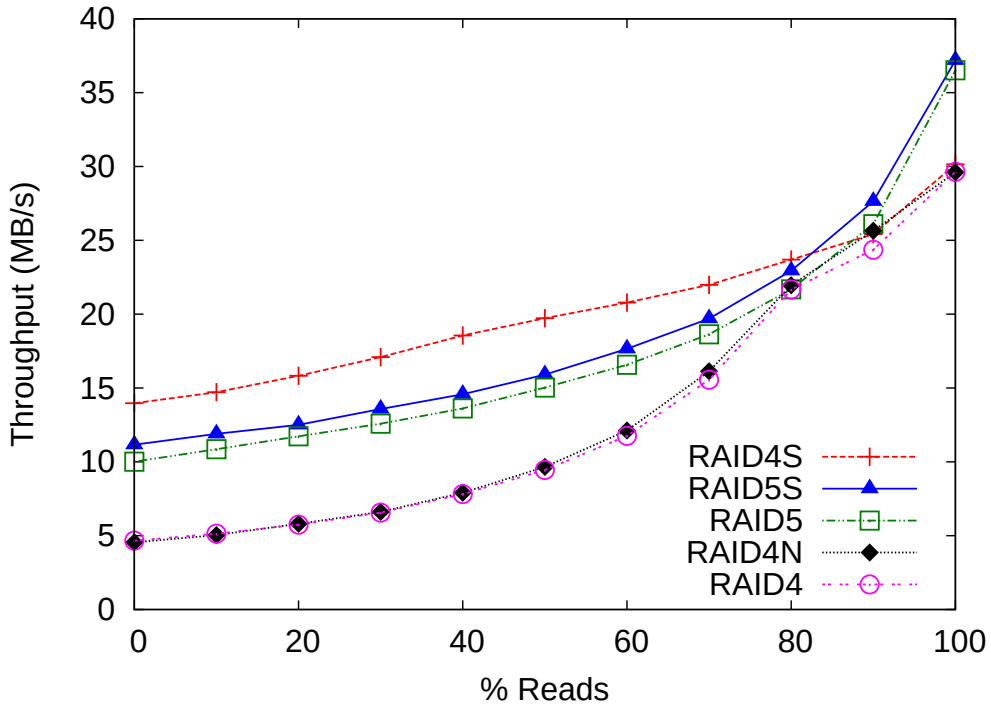

Figure 3.25: Random I/O experiment varying the ratio of reads to writes for a 64KB I/O size. RAID4S has the highest throughput for $\leq 90\%$ reads. This indicates that mostly read workloads still benefit from RAID4S. Because read-only workloads do not need to read parity, distributed parity RAIDs perform better by utilizing all devices.

Throughput on EXT3 file system

Direct I/O reduces extra caching outside the device level by disabling the buffer cache, so using RAID below a file system presents a different workload to the RAID device. In particular, small writes may be cached and aggregated into larger writes, resulting in reduced benefit for RAID4S. Our next experiment shows the performance of RAID4S on an ext3 file system. We mounted ext3 with asynchronous I/O and data journaling enabled. Figure 3.26 shows the throughput of running an XDD random write experiment similar to what we saw in Figure 3.14. While RAID4S does not perform significantly

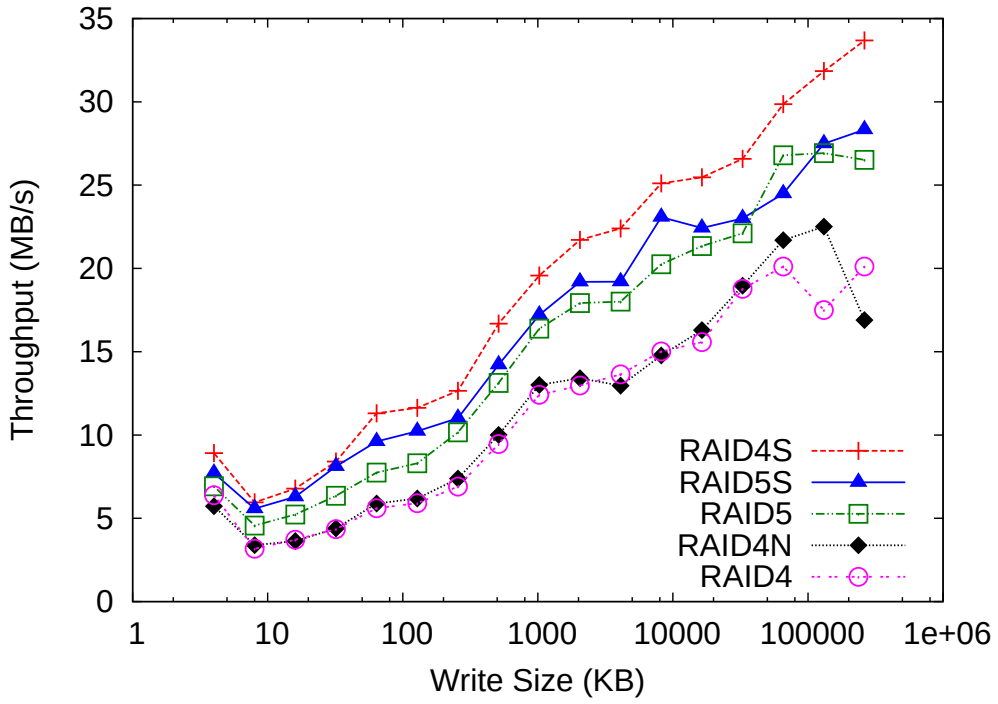

Figure 3.26: The random write throughput experiment on an ext3 file system shows that larger I/O sizes still reap the benefits of using RAID4S.

better than other layouts, it does consistently perform slightly better. Even at large write sizes, RAID4S throughput still beats RAID5S. This is likely due to ext3 journaling, which adds periodic small writes to the workload.

TPC-C benchmark

The TPC-C benchmark is an OLTP benchmark simulating users interacting with products in a warehouse. We chose this benchmark to investigate the performance of RAID4S in a more realistic environment. We used the open source implementation TPCC-UVa [49] and configured the database with one warehouse, 10 terminals per warehouse (to intro-

duce parallel I/O), 20 minute ramp-up, 480 minutes of measurement time, and vacuums every 60 minutes up to 6 total. Our results showed no difference in performance between RAID4S, RAID5, and RAID5S. All RAID layouts completed 12.6 transactions per minute. We investigated the workload that TPC-C performs, and discovered that the read-write ratio is 1.9:1 [22]. Figure 3.25 shows that RAID4S performs best on workloads with more writes. Another feature of TPC-C is that the most common I/O sizes are 8KB and 16KB. We see in Figure 3.26 that RAID4S and RAID5S perform similarly for these write sizes and RAID5 performs slightly worse. Because TPC-C is also run on an ext3 file system and the workload is not very write-intensive, these results are expected.

Degraded arrays

When a data device fails, the array continues to operate in degraded mode until the device is replaced. Read requests of data residing on the failed device require reading from all remaining devices, including the device that stores parity for that data, to reconstruct. This type of request is improved slightly by RAID4S, but because the significant workload is disk reads, the SSD is not heavily used. In the small write case, the operation depends on whether the write is to the failed device or to a working device. If the block resides on a remaining drive, the write proceeds normally with a read-modify-write. If on the failed drive, the parity needs to be updated but the old data is not available for a read-modify-write. Thus, a large write operation occurs: the data

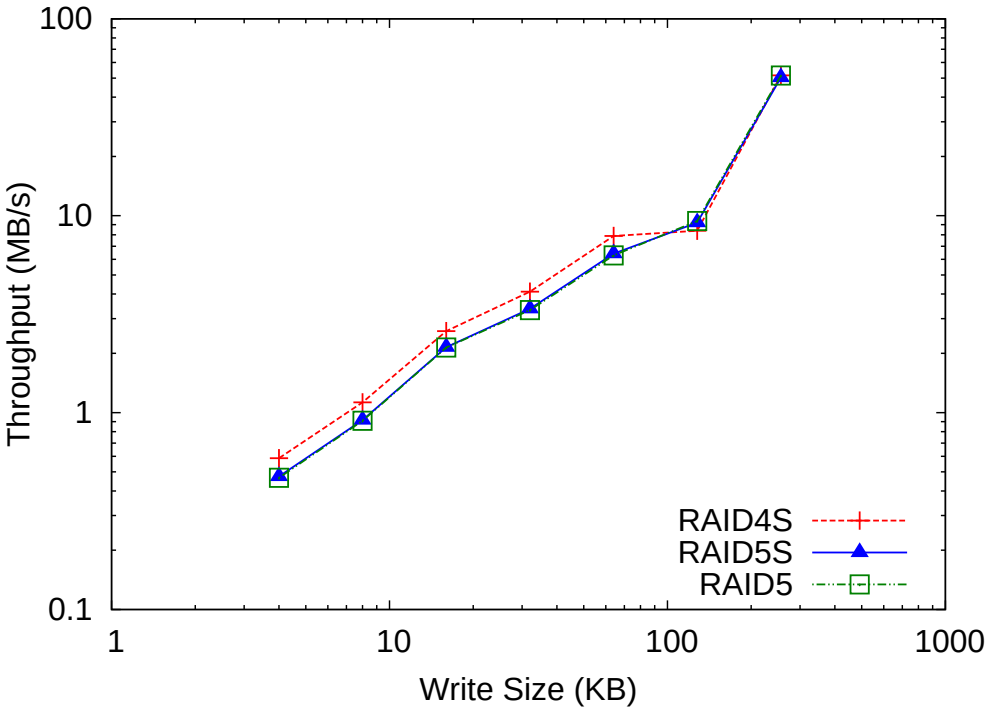

Figure 3.27: The random write throughput experiment with a failed data drive shows that RAID4S provides higher throughput for random I/Os under 128KB.

from the remaining devices are read and the parity is computed based on the existing data and the new block, and is written to the parity device. In the first case, RAID4S performs significantly better than RAID4 and RAID5. In the large write case, however, most I/O time is spent reading data off the remaining data devices. Thus, RAID4S does not benefit as much from this workload.

We tested our hypothesis by failing one data drive in each RAID and executing a random write workload. Figures 3.27 and 3.28 show that the difference in performance between the RAIDs is fairly small. RAID4S still performs best for small writes and the RAIDs all perform similarly for large writes. The additional disk read and write overhead when writing to the missing drive makes RAID4- and RAID5-based schemes

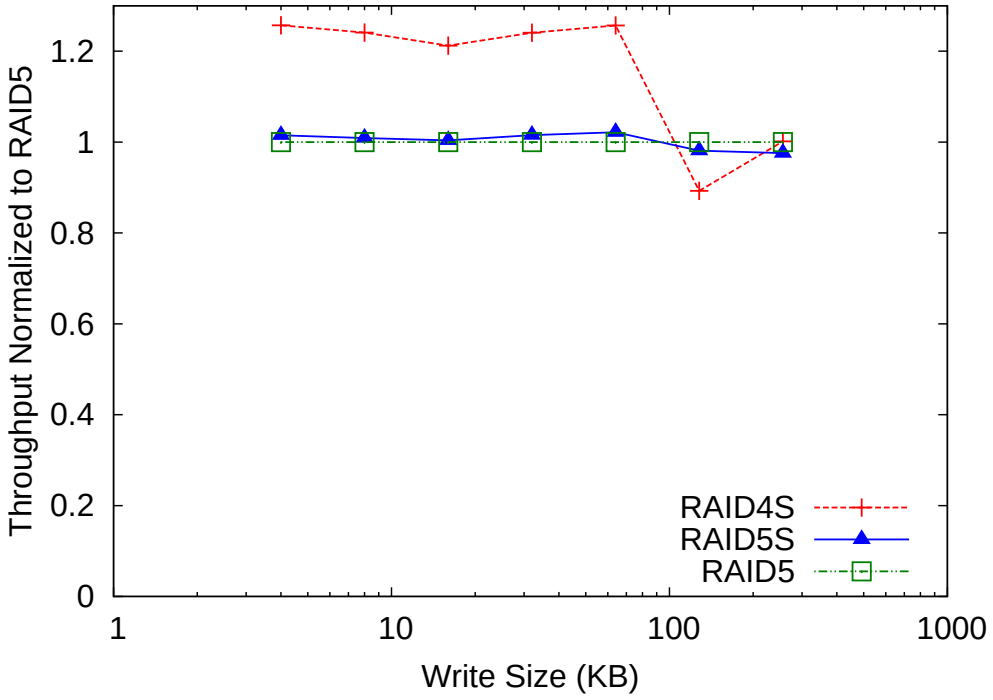

Figure 3.28: Normalized to RAID5, RAID4S performs 1.2X faster than RAID5.

perform more similarly, since they all have significant overhead which masks the other differences in performance between them.

RAID4S performs slightly worse than RAID5 at the 128KB write size. This is due to the difference in workload presented to devices within the RAIDs. In RAID4S, the failed drive contains only data, so every write must update parity. In RAID5, the failed device stores some parity. Some writes will inevitably belong to stripes with parity stored on the failed device. For writes to stripes with lost parity, there is no need to do a read-update-write and these writes proceed without the parity update overhead.

This degraded write experiment showed that RAID4S does not significantly hurt performance when a device has failed. The SSD is able to keep up with the additional

workload. RAID4S even improves performance for random writes that fit on a single device.

Financial workload

We used the SPC Financial traces [1] to evaluate RAID4S with a real world workload. The Financial traces are OLTP workloads from two large financial institutions. They include two total traces, each containing I/O to several devices, and we chose to replay the first trace. We replayed the I/O of 21 of the 24 devices, ignoring three that require a larger block range than our test system supports.

The replay without a file system in Figures 3.29 and 3.30 show that RAID4S performs better than other RAIDs in all cases where the RAIDs did not perform similarly. This is a clear case showing that RAID4S performs at least as well as RAID5. We also replayed the Financial1 trace through the ext3 file system. Figure 3.31 shows the result. The IOPS are normalized to RAID5 in Figure 3.32, which more clearly shows the improvement of RAID4S. The best result is at ASU0, where RAID4S performs 1.75X better than RAID5. This is the highest improvement we saw in any experiment, and it's on a real workload. The file system actually improves the relative performance of RAID4S.

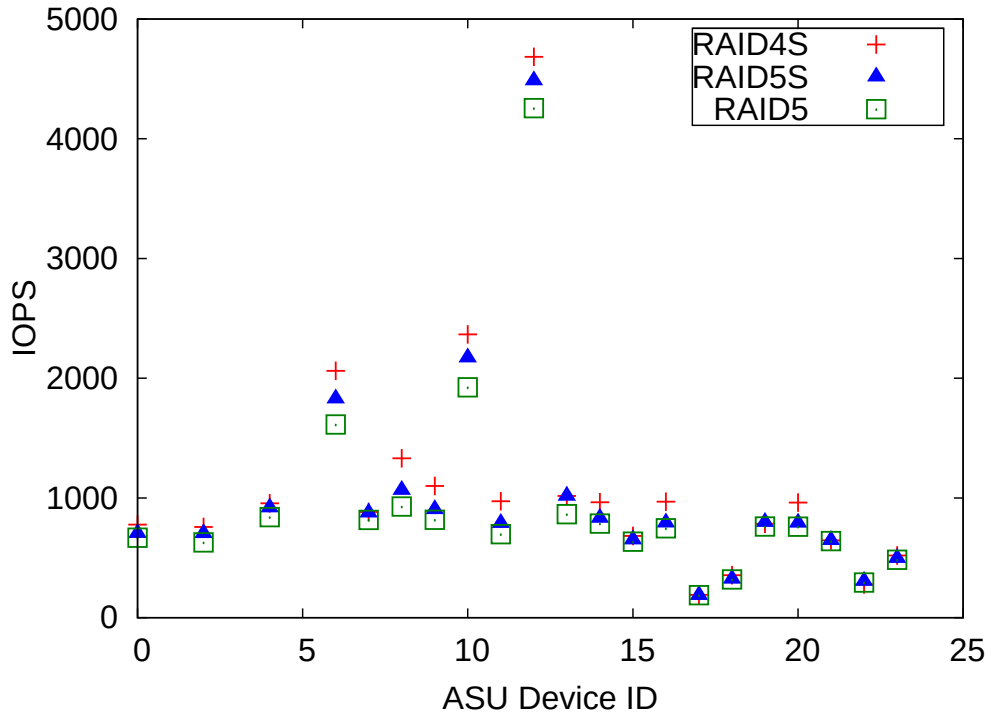

Figure 3.29: RAID4S replayed the trace directly to the device faster than RAID5S and RAID5.

Reliability analysis

One of the benefits of RAID4 in a disk-based low-power storage system is that it allows more disks to remain powered off. Essentially the parity disk remains powered on while only the disks containing data being accessed are powered on. On the other hand, RAID5 spreads parity on all disks. The disk storing parity for a particular access is less likely to be powered on and thus more power cycles will be required in a RAID5 arrangement than in a RAID4 setup. RAID4 is a better setup for reducing power at the expense of performance.

Our work replaces the parity disk in RAID4 with SSD. This mitigates the perfor-

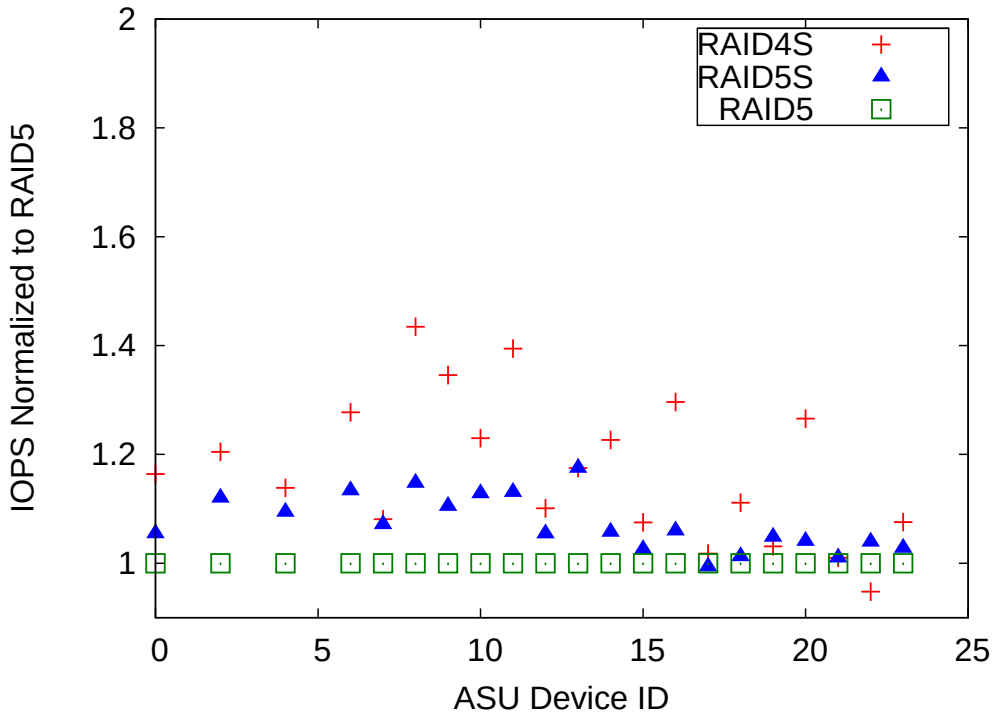

Figure 3.30: Normalized to RAID5, RAID4S usually performs better up to 1.4X.

mance problems of disk-based RAID4 while reducing power even further. Because parity is on SSD, each data access where the data is on one disk will only require powering that one disk and the SSD storing the parity. This will result in fewer disk power cycles, thereby improving reliability.

On the other hand, unlike disk drives, SSDs wear out with each erase-write cycle. Because of this, it is possible to predict failure based on number of writes to an SSD. The 64GB Intel X25-E [90] has a lifetime of 2PB of random writes. With a nonstop 100MB/s random write workload, the device would last for 7.7 months. A less aggressive workload with some reads and sequential writes would wear out the SSD slower than this worst case. It would also be possible to incorporate other techniques [43] to improve

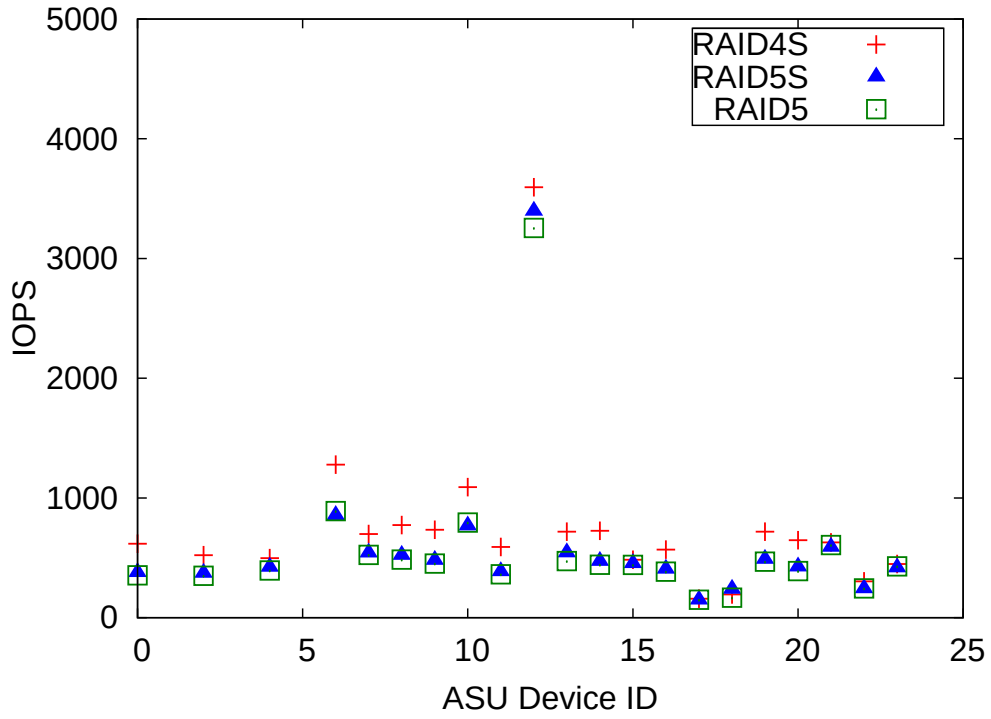

Figure 3.31: Replaying the Financial trace through the EXT3 file system results in even greater improvement from RAID4S.

reliability.

3.9 Conclusions and Future Work

RAID4S is a cost-effective hybrid SSD-HDD RAID layout. The cost/performance ratio for RAID4S is $53/GB, 1.5X higher than RAID5 at $35/GB, but the total system cost is incrementally more expensive. On the other hand, replacing all drives with SSDs results in a 10X increase in system cost.

The performance benefits were shown in several experiments. By isolating parity data on an SSD, parity accesses are faster and overall performance is improved over

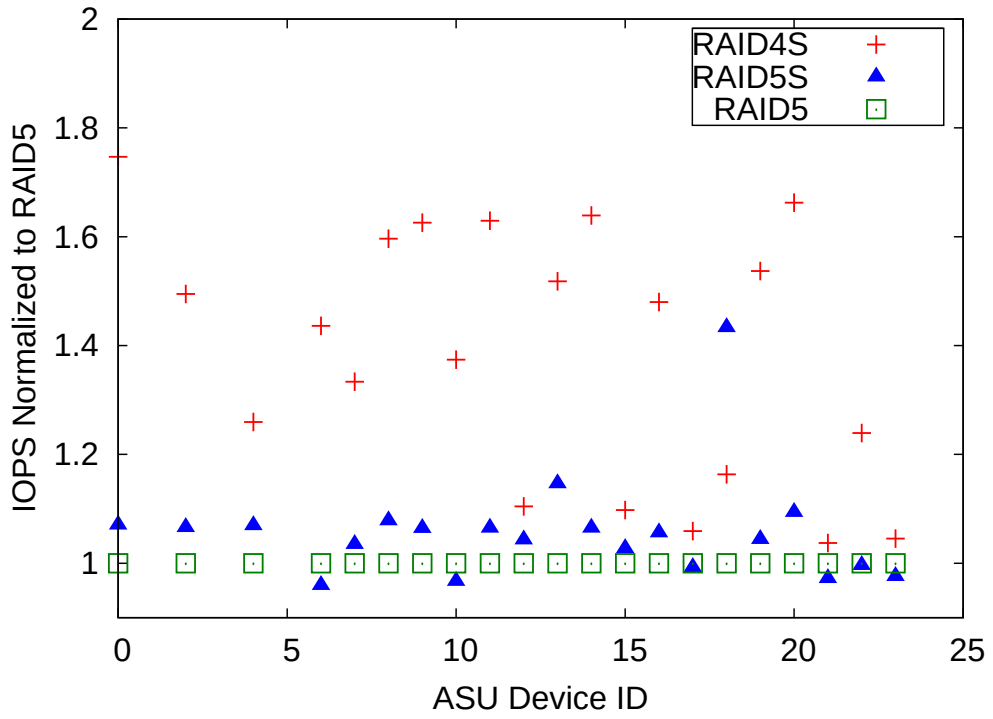

Figure 3.32: Normalized to RAID5, RAID4S performs up to 1.75X better than RAID5.

both RAID4 and RAID5, using the simpler algorithms associated with RAID4. We show that a single SSD improves RAID performance by up to 3.3X over RAID4 and 1.75X over RAID5 on workloads dominated by small writes and providing slightly lower performance on extremely read-intensive workloads. The benefits of RAID4S are increased by file systems and caches, which tend to increase randomness, the proportion of writes, and the number of small writes (such as for metadata updates).

Future work will focus on several RAID4S extensions to improve performance. The analysis presented in Section 3.8 motivates a software change to the RAID4 algorithm to improve RAID4S algorithm. Half-stripe sized writes are optimized for a homogeneous device array and avoid reading parity. Because RAID4S parity reads are fast, we

implemented a change to the md RAID kernel module to perform half-stripe writes as small writes. This optimization is presented in Chapter 4. RAID4S performance and the width of the RAID units is currently limited by the speed of the SSDs. We plan to further explore this interrelationship with the goal of finding the optimal performance ratios for a given number of disks and representative workloads. We also want to look into further enhancing RAID4S performance by striping parity over several SSDs collectively serving in the role of the parity device. This will enable smaller SSDs to be combined to match typically larger hard drive capacities. We will also explore the increased feasibility of higher-order RAID techniques involving additional parity information, such as row-diagonal parity schemes [26], with the addition of one or more parity SSD devices. Lastly, the higher performance of the parity device may change the RAID small/large write I/O tradeoffs, leading to greater realizable performance gains on medium sized writes by changing the small/large selection algorithms in RAID software.

Chapter 4

RAID4S-modthresh: Large Writes Threshold Modification

The goal of this research is to improve the performance of the RAID4S system. For RAID4S, the throughput slope of small-writes, the magnitude of medium sized writes on both sides of the small/large-write selection threshold, and the speedup of small-write performance suggest that medium-writes using the large-write parity calculation will perform better if they use the small-write parity calculation instead. RAID4S-modthresh uses a modified write selection algorithm which only affects medium-writes and keeps the parity calculation and corresponding throughput for small-writes and large-writes intact. RAID4S-modthresh is implemented into the software RAID controller module in the Linux source. Results show a maximum speedup of 1.4X over RAID4S for medium-writes without imposing performance or reliability penalties on the rest of the

system. Several other projects for modifying the software RAID controller to improve performance are described in future work. This work was completed for UCSC course CMPS221 and later published as a technical report [72].

4.1 Modifying Large Writes for RAID4S

RAID systems help bridge the gap between processor and cache technology by improving either performance or reliability. Different RAID levels utilize a variety of striping, mirroring, and parity techniques to store data. This allows the user to select the RAID level that best fits the application's cost, performance, complexity, and redundancy requirements [77]. RAID schemes based on parity enhance storage reliability by managing a recovery parity disk but parity calculations degrade performance [21]. This is especially apparent when performing small-writes. A key goal in this chapter is to improve upon the RAID4S system so that the overhead of parity based RAIDs can be minimized even further in an effort to approach RAID0 schemes [65]. Figure 4.1 shows the improved throughput measurements of the new write selection algorithm implemented in this chapter.

When calculating the parity for a parity-based RAID, the RAID controller selects a computation based on the write request size. For maximum efficiency, large-writes, which span at least half the drives, must first compute the parity by XORing the drives

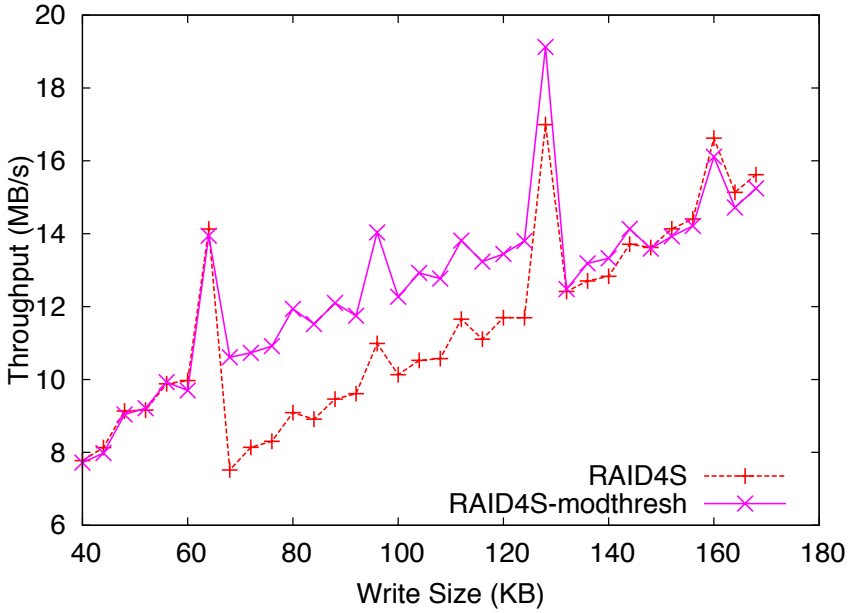

Figure 4.1: RAID4S-modthresh provides a noticeable speedup over RAID4S when performing medium-sized writes. The RAID4S-modthresh write selection algorithm classifies medium-writes as small-writes, so it uses the small-write parity calculation and enjoys the benefits of RAID4S for medium-sized writes.

to be written with the remaining drives. The large-write parity calculation is:

$$p_{\text{large-write}} = d_{\text{new}} \oplus r_{\text{old}}$$

where d is the data on the drive(s) to be written, p is the parity drive, and r is the remaining drive(s). After computing the parity, the data is written to the disks and the new parity is written to the parity disk.

Small-writes that span less than half the drives are most efficient when parity is computed by XORing the old and new data with the old parity. The small-write parity

calculation is:

$$p_{\text{small-write}} = d_{\text{old}} \oplus d_{\text{new}} \oplus p_{\text{old}}$$

where d is the data on the drive(s) to be written and p is the parity drive. After the parity is computed, the new data is written to the drives and the new parity is written to the parity disk [21, 77, 83]

Small-writes require $2(m+1)$ disk I/Os while large writes always require $N+2$ disk I/Os, where N is the number of drives in a stripe and m is the size of the write in terms of the number of drives it spans. The instance where four accesses are needed for each small-write instead of two is known as the small-write problem [74]. This is most apparent with small-writes, whose throughput, compared to the same sized write for mirrored RAIDs (i.e. RAID0), is severely penalized [83] for every small-write. This bottleneck can be seen in Figure 3.4.

Traditionally, RAID5 is favored over RAID4 because, as noted in [77], the parity disk is a throughput bottleneck when performing small-writes. RAID4S is a RAID4 configuration comprised of data HDDs and 1 dedicated parity SSD which aims to overcome the parity bottleneck in an effort to improve small-write performance . With this scheme, small-write performance increased because the SSD parity drive can process small-write reads and writes much faster. This allows small-writes on data disks (the rest of the RAID system) to be performed "in parallel". By simply isolating the parity on the SSD, RAID4S provides excellent performance while still achieving the same reliability of parity-based RAIDs. Figure 4.2 shows the increased throughput of RAID4S over

RAID5 and RAID5S. RAID4S provides nearly 1.75 times the throughput over RAID5 and 3.3 times the throughput for random writes under 128KB (small-writes) [83]. The overall performance speedup is a function of the speed of the SSD relative to the other disks and the percentage of small writes performed. RAID4S alleviates the RAID4 parity bottleneck, offloads work from the slower devices, and fully utilizes the faster device.

The configuration and utilization of the entire system's resources allow such high speedups without sacrificing reliability and cost. Implementing the RAID with 1 SSD instead of N makes RAID4S cheaper than other RAIDs with SSDs and still retains the reliability of parity-based RAIDs. This solution addresses the small-write problem of parity overhead by intelligently injecting and configuring more efficient resources into the system.

For RAID4S, the throughput slope of small-writes and the magnitude of medium sized writes on both sides of the small/large-write selection threshold suggest that medium-writes using the large-write parity calculation will perform better if they use the small-write parity calculation instead.

This research improves the performance of RAID4S by introducing a new write selection algorithm to the software RAID controller. The new write selection algorithm modifies the threshold of RAID4S to allow the parity of smaller large-writes (medium-writes) to be calculated using the small-write parity calculation. The new system that uses the new write selection algorithm has been named **RAID4S-modthresh**.

This modification to the RAID controller only affects medium-writes and keeps the parity calculation and corresponding throughput for small-writes and large-writes intact. Furthermore, changes to the write selection algorithm do not affect the reliability or cost of the system since the hardware is not being changed.

This modification is accomplished by examining the software RAID controller module in the Linux source, modifying the kernel code, and loading an external Loadable Kernel Module (LKM) into the kernel. The results of initial tests are encouraging and illustrate a speedup of 2 MB/s for all medium writes without imposing performance or reliability penalties on the rest of the system.

The rest of the chapter outlines the steps taken to implement RAID4S-modthresh and analyzes its resulting performance. Section 4.2 discusses the Background and Methodology of the implementation and focuses on the theoretical performance implications of modifying the threshold. The section also discusses the background for the code and kernel modules that is necessary for understanding the implementation details. Section 4.3 describes the implementation in detail and shows the exact changes that need to be made to implement RAID4S-modthresh. Section 4.4 shows how RAID4S-modthresh is tested and provides a brief analysis of the results. Section 4.5 acknowledges related works with similar goals but different implementations.

4.2 Methodology and Background

The goal of this work is to utilize the small-write performance enhancements of RAID4S by modifying the write selection algorithm. The following sections discuss the performance measurements of RAID4S and explain how these measurements motivated the implementation of the RAID4S optimization. A discussion of `raid5.c`, the current software RAID controller, is also included in this section.

Motivation

To baseline RAID4S write performance for 4KB block aligned random writes, XDD was configured to make various write sizes, in powers of 2, ranging from the minimum block size to the RAID stripe size (4KB, 8KB, 16KB, ..., 128KB, and 256KB). RAID4S is configured with 4 data drives and 1 parity SSD using 256KB stripes. In this setup, each drive receives 64KB chunks from the data stripe. Figure 4.2 shows the throughput measurements for the block aligned random writes.

The results show that the write threshold for small and large-writes is at 128KB because this is where all RAID systems achieve the same throughput. Throughput is the same (or very close) at large-writes for most RAID configurations because the parity calculation is reliant on the stripe data and disk bandwidth and cannot be be parallelized with other large-writes. This is consistent with the write selection algorithm of RAID4S. The write selection algorithm will choose a large-write parity calculation for 128KB sized writes because 128KB writes span at least half of the drives.

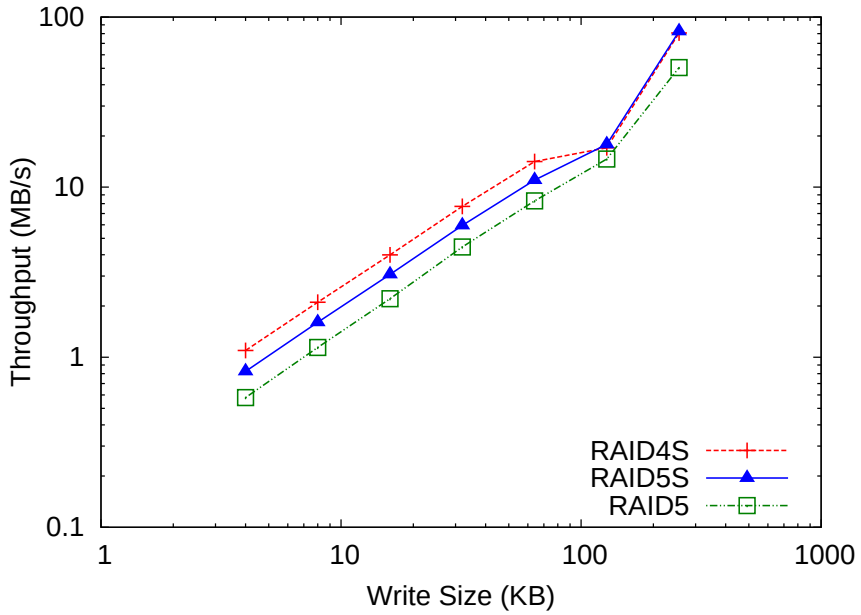

Figure 4.2: The throughput of RAID4S compared to RAID5 and RAID5S. Using a parity drive significantly improves the performance of small-writes because of the parallel capabilities that the SSD provides. The curve of the small-writes and the magnitudes of the 64KB and 128KB writes suggest that changing the 128KB large-write parity calculation into a small-write parity calculation can improve the performance even more.

Motivation for this project stems from the results of RAID4S and the throughput measurements of the block aligned writes. The results suggest that performing more small-write parity calculations will improve the throughput for medium sized writes.

Throughput vs. write size slope

The shape of the curves in Figure 4.2 suggested that for RAID4S, the parity for 128KB writes should be calculated using the small-write parity computation instead of the

large-write parity computation. The slope of RAID4S approaching 64KB is different than the slope leaving the 64KB write-size because the write selection algorithm is now using the large-write parity computation. This sudden negative change in slope implies that a medium-sized might perform better if it was calculated as a small-write.

The interpolation of the RAID4S curve in Figure 4.2 also suggests that small-writes will outperform large-writes for medium-write sizes. The large small-writes have a lower slope but higher performance than smaller large-writes (medium-writes). It should be noted that interpolating the small-write curve much further beyond the halfway point (128KB) would result in lower throughput than the large-write performance despite the higher initial performance.

Magnitude of medium writes

The magnitude of the throughput of large small-writes and small large-writes is very close. Figure 4.3 is a subset of the throughput values shown in Figure 4.2 and is set on a non-log scale. The figure shows that the last small-write of RAID4S, 64KB, has a throughput that is only about 3 MB/s slower than the first large-write, 128KB. Since small-write performance for RAID4S increased dramatically, it is logical to predict that the write after the last small-write may enjoy some speedup over its predecessor, which should improve its throughput over its current large-write parity calculation throughput.

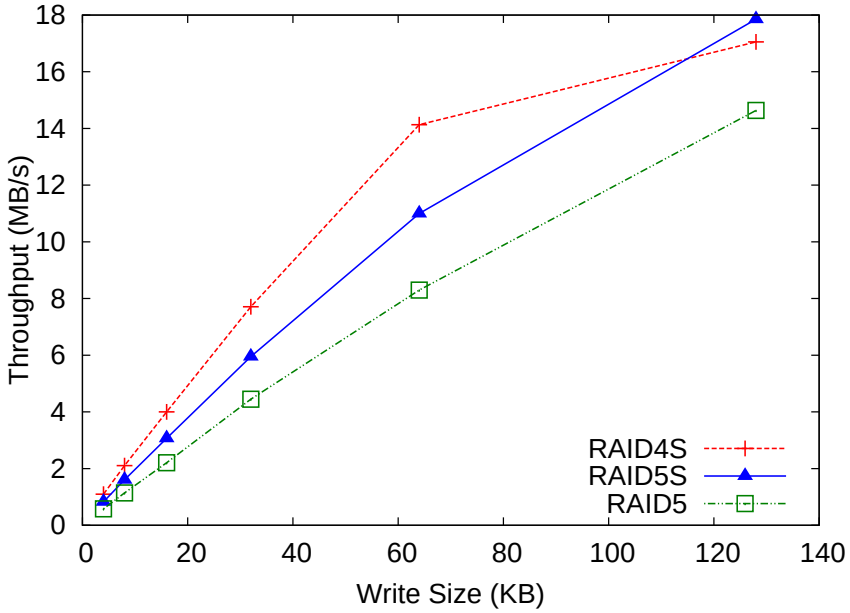

Figure 4.3: RAID4S shows similar throughput for 64KB and 128KB sized writes. This suggests that the performance improvement of RAID4S will make 128KB sized writes faster using the small-write parity calculation.

Extending parallelism to medium-writes

RAID4S was shown to improve small-write performance because of the use of a more efficient parity drive, which facilitated parallel small-writes. The same concept can be applied to medium writes in the RAID4S system but since the RAID4S write selection algorithm uses the traditional threshold for write selection, it needs to be modified so that large writes are classified as small-writes.

Figure 4.4 shows how medium-writes can be processed in parallel. RAID4S-modthresh would attempt to take advantage of the improved small-write performance

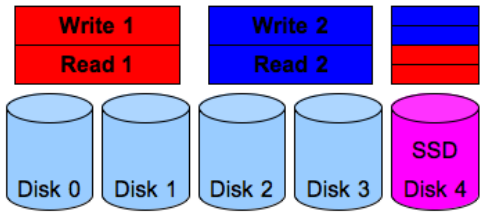

Figure 4.4: RAID4S classifies medium sized writes as large-writes because a read-modify-write will need to read 3 disks while a reconstruct-write only requires 2 disk reads. RAID4S-modthresh classifies medium sized writes as small-writes because small-writes can be processed in parallel on a small-write RAID4S system.

and not the fact that writes can be processed in parallel. Theoretically, the large-write parity calculation would perform equally as well as the situation in Figure 4.4 but since RAID4S has shown such an extreme throughput enhancement for small-writes, it seemed worth it to try changing the write selection algorithm. Since many of the small-writes can perform faster, it may open up more opportunities for medium-writes to process in parallel alongside small-writes.

The raid5.c code

The file `raid5.c` is the RAID software controller (`/linux-source/drivers/md/raid5.c`) which manages RAID4 and RAID5 configurations. In the controller, each disk is assigned a buffer stripe. This buffer stripe holds data for reads returning from disk and for data that needs to be temporarily saved before being written to the disk. Each disk buffer has observable states (Empty, Want, Dirty, Clean) and state transitions perform operations on stripes.

State transitions

State transitions perform operations on disk buffer stripes to exchange data between the disk and the RAID controller. Examining the states of each disk is how the RAID controller checks the disk array for failed devices, devices which recently completed writes (ready for return), and devices that are ready to be read.

These state transitions are managed in the `handle_stripe5()` method. Every time `handle_stripe5()` is called, it examines the state of each disk's buffer stripe and calls the correct operations for each buffer stripe. Functions set the `UPTODATE` and `LOCK` flags to indicate the state of the stripe buffer and to facilitate concurrent execution by marking buffers as locked, up-to-date, wanting a compute, and wanting a read. The header file `raid5.h` enumerates the states and the corresponding spin locks.

The dirty state - write selection / scheduling

The dirty state first chooses a write type using the small/large-write selection algorithm and then schedules the write to disk. All new write requests must be considered by the dirty state before being written to disk. When a buffer stripe is transitioned to the dirty state, the buffer must contain valid data that is either being written to disk or has already been written to disk. The `handle-stripe-dirtying5()` method handles transitions in and out of the dirty state.

The write selection algorithm must decide whether to perform a read-modify-write (r-m-w) or a reconstruct-write (r-c-w) parity calculation. The read-modify write is

analogous to a small-write parity calculation and involves reading the old parity and the old data before calculating the new parity. The reconstruct-write is analogous to the large-write parity calculation and is performed by reading the buffers of disks which are not being written to before calculating the new parity. This write selection algorithm decides which parity calculation to use based on the number of reads needed for a r-m-w and the number of writes needed for a r-c-w.

To count the disks for each write, the algorithm cycles through the disks and keeps track of the "predicted" write counts in two variables: `rmw` and `rcw`. These indicate the read-modify-write and reconstruct-write overhead. The flags for each buffer stripe indicate whether the disk needs to be read or not. If the disk in question is going to be written in the next cycle, `rmw` is incremented by one. If the disk in question is not going to be written in the next cycle, `rcw` is incremented by one.

Recall that a r-m-w reads the old data, so checking to see if the device is going to be written to is equivalent to asking if the device in question needs to be read for a r-m-w. Note that the parity drive will always increment `rmw` since the parity drive needs to be read for a read-modify write. Figure 4.5 shows a model of the typical disk counting for a 128KB and 64KB writes. The debugging output for the r-m-w and r-c-w calculations is shown in Figure 4.6. The debugging output first shows how the `handle_stripe_dirtying5()` function first examines the requests of each disk (states) and then it shows how the `rmw` and `rcw` variables are incremented.

Once these tallies are made, the most efficient write is chosen. The most efficient

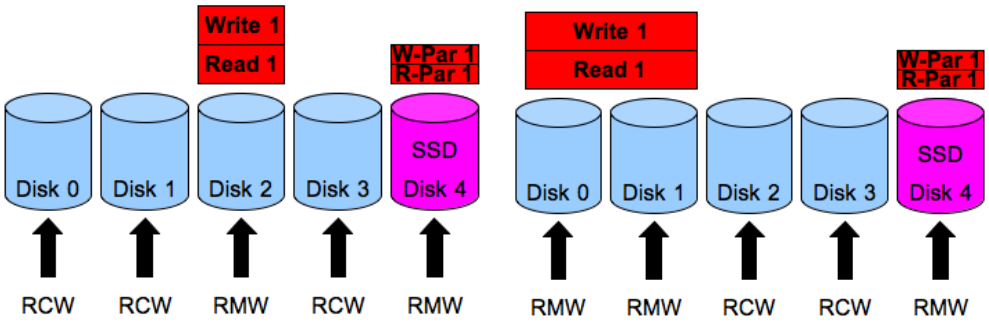

Figure 4.5: Counting the drives that would be needed for a r-m-w and a r-c-w for a 64KB write (1 drive) and a 128KB write (2 drives). Notice that any drive that is not marked as a drive that needs to be read for r-m-w is automatically marked as a r-c-w read drive. The parity drive is always marked as a r-m-w drive.

```
23:52:32 ssd kernel: [15755.395827] check disk 4: state 0x0 toread (null) read
(null) write (null) written (null)
23:52:32 ssd kernel: [15755.395832] check disk 3: state 0x0 toread (null) read
(null) write (null) written (null)
23:52:32 ssd kernel: [15755.395835] check disk 2: state 0x0 toread (null) read
(null) write (null) written (null)
23:52:32 ssd kernel: [15755.395839] check disk 1: state 0x0 toread (null) read
(null) write (null) written (null)
23:52:32 ssd kernel: [15755.395842] check disk 0: state 0x4 toread (null) read
(null) write ffff880070d56e40 written (null)
23:52:32 ssd kernel: [15755.395845] Consider new write: handle-stripe-
dirtying()
23:52:32 ssd kernel: [15755.395847] disk 4 –>RMW++
23:52:32 ssd kernel: [15755.395848] disk 3 –>RCW++
23:52:32 ssd kernel: [15755.395850] disk 2 –>RCW++
23:52:32 ssd kernel: [15755.395851] disk 1 –>RCW++
23:52:32 ssd kernel: [15755.395853] disk 0 –>RMW++
23:52:32 ssd kernel: [15755.395855] Sector 21105024, rmw=2 rcw=3
```

Figure 4.6: Debugging output for a 64KB write on RAID4S. The write selection algorithm chose a r-m-w parity calculation.

write is the write type that requires the smallest number of reads. Therefore, the write selection algorithm threshold is set by examining the tallies incurred in the previous step. If $rmw < rcw$ and $rmw > 0$, then the algorithm concludes that the small-write parity calculation it optimal. The function then reads all the drives to be written in prepararation for the parity calculation. On the other hand, if $rcw <= rmw$ and $rcw > 0$, then a large-write parity calculation is the more efficient computation. In this case, all the drives that are not going to be written to are read to prepare for the parity calculation. The threshold checks whether $rmw > 0$ and $rcw > 0$ to make sure that the stripe is active and not locked for an earlier parity computation.

This threshold is set by two conditionals in `raid5.c`. They have been reproduced in Figure 4.7. Once the proper drives are read, a reconstruction is scheduled. This has a pipelining effect because new write requests can be considered while older write requests are being passed to the write scheduler. Once the write type has been determined, the function locks the drives that need to be read, which also stops the scheduler from scheduling any other write requests.

The condition `rcw` == 0 is passed to the scheduler so the scheduler can determine the write type. If a r-c-w write is chosen as the write type, the previous step will lock non-write drives for reading so the `rcw` variable will be 0. If r-m-w write is chosen as the write type, the previous step will lock all the write drives for reading so the `rmw` variable will be 0. `schedule_reconstruction()` locks bits and drains buffers as it puts the type of write request into the queue.

```
if (rmw < rcw && rmw > 0) {
        /*
         * -- Block 1 --
         * Prepare for r-m-w
         *
         * Decides if the drive
         * needs to be read for
         * a r-m-w and reads it
         * if it is not locked
         *
         */
}
if (rcw <= rmw && rcw > 0) {
        /*
         * -- Block 2 --
         * Prepare for r-c-w
         *
         * Decides if the drive
         * needs to be read for
         * a r-c-w and reads it
         * if it is not locked
         *
         */
}
```

Figure 4.7: The raid5.c write selection algorithm threshold.

Loadable Kernel Module (LKM)

The Ubuntu Linux 2.6.32 kernel is used for the implementation and testing of RAID4S-modthresh. The `raid5.c` compiles into a .ko file and is linked into the kernel through the `raid456.ko` kernel module. After stopping all the RAIDs, the LKM is loaded into the kernel by first removing the old `raid456.ko` and loading the new `raid456.ko` from the offline source tree.

Debugging

All debugging is written to `/var/log/kern.log` and can be accessed either by reading this file or through `dmesg`. Scripts are used to extract the relevant debugging output for each run. Typical debugging output is shown in Figure 4.6 and the output consists of the disk buffer states, the stripes being examined, the `rmw`/`rcw` counts, the stripe operations being performed, the states of various buffer stripe flags, and the type of writes being performed. The debugging is also used to determine which functions are being called when and how the `handle_stripe()` method chooses and schedules stripe operations.

The debugging output also proved to be extremely valuable when changing the RAID4S-modthresh threshold for kernel bugs. The `raid5.c` code is very exhaustive when making sure that the proper buffers have data and that the correct flags are set. `BUG_ON()` checks the buffer stripes for the correct data and flags and will throw a Kernel BUG for the invalid opcode if the states are not initialized correctly. It then halts all affected modules and logs the failure in `/kern/log/kern.log`. It then freezes the system, which can only be remedied by killing the xdd process, rebooting the system, and re-configuring all the RAIDs.

4.3 Implementation

We modify the write selection algorithm in the dirty state in order to improve the small-write performance enhancements of RAID4S. This is done by changing the threshold in the `handle_stripe_dirtying5()` function of the `raid5.c` loadable kernel module. The `handle_stripe_dirtying5()` function handles the write selection algorithm and passes the chosen write to the `schedule_reconstruction()` function, as discussed in section 4.2.

Two implementations are made to change the write selection algorithm:

1. RAID4S-modified: schedule only small-writes

2. RAID4S-modthresh: schedule more small-writes

For both implementations, the r-m-w and r-c-w disk counting, discussed in section 4.2, needs to be left intact for the `schedule_reconstruction()` method. The `schedule_reconstruction()` method relies on the `rmw` and `rcw` variables for write selection and will not proceed until the proper disks have been read for the parity calculation. The indication that these reads are in flight occurs when either the `rmw` or `rcw` variable is 0, since the drives have been locked for reading.

Modifications are made to the two conditionals shown in Figure 4.7, which specify the drives that need to be read.

RAID4S-modified: force all small writes

For this first implementation, the `raid5.c` software RAID controller code is modified to force all write requests to be treated as small-writes. This will force all writes to calculate parity using the small-write parity calculation instead of the large-write parity calculation.

The RAID controller needs to be modified to satisfy two conditions:

1. the correct drives are read for the parity calculation

2. the write scheduler always calls for a r-m-w

To satisfy condition (1) changes are made in the second `if` condition (Block 2) of Figure 4.7 so that in the r-c-w calculation, the drives to be written are read for the parity computation. With this change, the drives to be written are always read for the parity computation, independent of the write type chosen. To satisfy condition (2), the write scheduler parameter is forced to call a r-m-w by changing the `rcw` == 0 parameter to 1. The following two lines of code achieve these requirements:

```
/*
 * Implementing RAID4S-modified
 *
 * Satisfy condition (1)
 * In the method handle_stripe_dirtying5()
 * In Block 2 of Figure 8
 * Comment out the original code
 * Insert new beginning to "if" statement
 */

// ...
```

```
//if (!test_bit(R5_OVERWRITE, &dev->flags)&&
//      i != sh->pd_idx &&
if ((dev->towrite || i == sh->pd_idx) &&
        !test_bit(R5_LOCKED, &dev->flags)&&
        !(test_bit(R5_UPTODATE, &dev->flags)||
        ...) {

// ...

/* Satisfy condition (2)
 * At the end of handle_stripe_dirtying5()
 * Comment out the original code
 * Insert new schedule_reconstruction()
 */

 // ...

//schedule_reconstruction(sh, s, rcw==0, 0);
schedule_reconstruction(sh, s, 0, 0);

//...
```

If either condition (1) or (2) of the first implementation are skipped, the algorithm behaves incorrectly. Skipping condition (1) causes a crash because when the disks to be written are read, there is no data in the buffer stripe. Skipping condition (2) does not force the algorithm to choose r-m-w for every write type. Because of these conditions, it is not sufficient to change the `if` statements to always force the program flow into Block 1 of Figure 4.7.

RAID4S-modthresh: modify threshold to schedule more small-writes

For the second implementation, the `raid5.c` software RAID controller code is modified to allow more write requests to be treated as small-writes. This will force medium-writes to calculate parity using the small-write parity calculation instead of the large-write parity calculation.

The code needs to be modified so that writes that span half the drives are classified as small-writes instead of large-writes. To accomplish this, the two conditionals from Figure 4.7 need to be modified so that Block 1 accepts larger writes and Block 2 accepts the rest (anything that is not a small-write). Changes are made to the following `if` statements :

```
/*
 * Implementing RAID4S-modthresh
 *
 * In the method handle_stripe_dirtying5()
 * Comment out the original code
 * Insert new "if" statements
 */

// ...

//if (rmw < rcw && rmw > 0) {
if (rmw <= (rcw+1) && rmw > 0) {
        /*
         * -- Block 1 --
         * Prepare for r-m-w
         *
         * Decides if the drive
         * needs to be read for
         * a r-m-w and reads it
```

```
          * if it is not locked
          *
          */
}
//if (rcw <= rmw && rcw > 0) {
if ((rcw+1) < rmw && rcw > 0) {
         /*
          * -- Block 2 --
          * Prepare for r-c-w
          *
          * Decides if the drive
          * needs to be read for
          * a r-c-w and reads it
          * if it is not locked
          *
          */
}

// ...
```

This has the effect of sliding the write threshold to the right in order to accept more writes. Now writes that span less than or equal to half the drives will be classified as small-writes instead of writes that span strictly less than half the drives.

It should be noted that `rcw` is replaced by `rcw` + 1. This is because of the way that the `handle_stripe_dirtying5()` function counts r-m-w and r-c-w drives. The additional drive for the r-m-w calculation is the parity drive and this tilts the balance in favor of r-c-w when the number of drives to be written and number of remaining drives are the same. This means that without the `rcw` + 1, the drives r-m-w drives will still outnumber the r-c-w for drives that span half the disks. The following example illustrates the reason for using `rcw` + 1.

RAID4S vs. RAID4S-modthresh example

Consider an example: a 5 disk RAID4S comprised of 4 data disks and 1 parity disk with data striped at 256KB (64KB data chunk for each disk). In the traditional writeselection algorithm, any write sizes of 64KB or less will constitute a small-write since it only spans 1 drive. Our modified write selection algorithm seeks to extend this threshold to include any write sizes that span 2 disks or less (i.e. 128KB or less).

The `rmw` and `rcw` count for a 64KB write is shown in Figure 4.6. In this debugging log, disk 0 has a write request. The write selection algorithm classifies disk 0 and disk 4 (parity disk) as r-m-w's because they will need to be read if the algorithm were to choose r-m-w. The remaining disks would need to read for a r-c-w. Both the original RAID4S write selection and the RAID4S-modthresh write selection algorithm will select r-m-w because $rmw < rcw$.

The `rmw` and `rcw` counts for a 128KB write are shown in Figure 4.8. In this debugging log, disk 0 and disk 1 have a write request. The write selection algorithm classifies disk 0, disk 1, and disk 4 (parity disk) as `rmw`s. The remaining disks would need to be read for a r-c-w. The RAID4S write selection algorithm will select a r-c-w parity calculation because $rcw <= rmw$. The RAID4S-modthresh write selection algorithm selects a r-m-w parity calculation as is evident by the last three lines of the debugging output. RAID4S-modthresh selects a r-m-w parity calculation because $rmw <= (rcw + 1)$.

It should now be clear that the `rcw` variable needs to be increased by 1 to account

for the parity when half the data drives are being written; the `rmw` and `rcw` values need to be skewed to account for the parity disk. In this implementation, the `schedule_reconstruction()` method is set back to $rcw == 0$ for the write-type, since we still want to execute large-writes if the threshold is overtaken.

```
21:10:08 ssd kernel: [178429.097858] check disk 4: state 0x0 toread (null)
read (null) write (null) written (null)
21:10:08 ssd kernel: [178429.097862] check disk 3: state 0x0 toread (null)
read (null) write (null) written (null)
21:10:08 ssd kernel: [178429.097865] check disk 2: state 0x0 toread (null)
read (null) write (null) written (null)
21:10:08 ssd kernel: [178429.097869] check disk 1: state 0x4 toread (null)
read (null) write ffff88007028fa40 written (null)
21:10:08 ssd kernel: [178429.097875] check disk 0: state 0x4 toread (null)
read (null) write ffff88007028ecc0 written (null)
21:10:08 ssd kernel: [178429.097879] Consider new write: handle_stripe_dirtying()
21:10:08 ssd kernel: [178429.097880] disk 4 –>RMW++
21:10:08 ssd kernel: [178429.097882] disk 3 –>RCW++
21:10:08 ssd kernel: [178429.097884] disk 2 –>RCW++
21:10:08 ssd kernel: [178429.097885] disk 1 –>RMW++
21:10:08 ssd kernel: [178429.097887] disk 0 –>RMW++
21:10:08 ssd kernel: [178429.097890] Sector 21105128, rmw=3 rcw=2
21:10:08 ssd kernel: [178429.097893] Read_old block 4 for r-m-w
21:10:08 ssd kernel: [178429.097895] Read_old block 1 for r-m-w
21:10:08 ssd kernel: [178429.097896] Read_old block 0 for r-m-w
```

Figure 4.8: Debugging output for a 128KB write on RAID4S-modthresh. The write selection algorithm chose a r-m-w parity calculation, as is evident from the last three lines.

Analysis of RAID4S-modified and RAID4S-modthresh implementations

Both implementations are included because they produced interesting results and provide insight into the RAID controller setup and program flow. RAID4S-modified was implemented as a precursor to RAID4S-modthresh but peculiar results suggest that the RAID4S-modthresh should be extended further to include even more small-write parity calculations.

Comparing RAID4S-modified and RAID4S-modthresh forced us to think deeply about what the RAID controller was really doing and how it may be better to always test rather than assume that a theoretical exploration of a concept is sound. RAID4S-modified throws errors for very large-writes, which are not shown in this chapter because the focus is medium-writes, but these failures served to help us learn the inner-workings of `raid5.c` and to show that further study of forcing small-writes can only accelerate the learning for future, related topics.

4.4 Evaluation and Results

To test the performance speedup of RAID4S-modthresh, random write tests were performed on three RAID configurations: RAID4S, RAID5, and RAID5S. These configurations are shown in Figure 4.9. The XDD [67] tool is a command-line benchmark that measures and characterizes I/O on systems and is used in these experiments to

measure performance. Throughput measurements show the improved performance that RAID4S-modthresh has over RAID4S, RAID5, and RAID5S for medium-writes.

The data is striped into 256KB blocks and each drive is assigned 64KB blocks. The data is written randomly to the target RAID system in various seek ranges, depending on the write size. The maximum value in the seek range is calculated by using:

$$\text{range}_{\text{seek}} = \frac{\text{size}_{\text{device}}}{\text{size}_{\text{block}}}$$

where $\text{range}_{\text{seek}}$, $\text{size}_{\text{device}}$, and $\text{size}_{\text{block}}$ are all measured in bytes. For these experiments,

$$\text{size}_{\text{device}} = 49153835008$$

and

$$\text{size}_{\text{block}}\text{bytes} = 1024 * \text{size}_{\text{block}}\text{KB}$$

The `queuedepth`, which specifies the number of commands to be sent to 1 target at a time, is 10. For every experiment, a total of 128MB is transferred on each pass and each run consists of 3 passes. For each run, the average of the three passes are plotted.

All tests are run on an Ubuntu Linux 10.0.04 namespace running on an Intel Core i7 CPU with 2GB RAM and 9GB swap memory. The software RAID controller that is modified and loaded is from the `mdadm` drivers package in the Linux kernel source version 2.6.32. The hard drives are 640GB 7200 RPM Western Digital Caviar Black SATA 3.0Gb/s hard drives with 32MB cache and 64GB Intel X25-E SSDs. Both drives use 12.25 GB partitions.

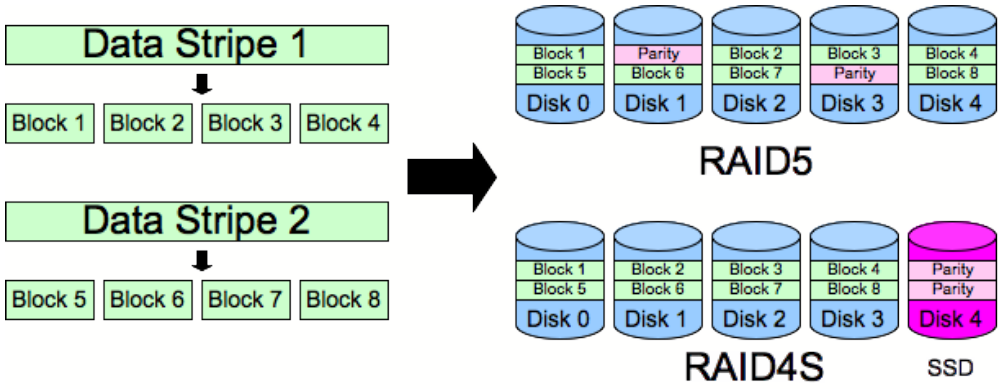

Figure 4.9: The RAID5 and RAID4S setups. RAID5S has the same topology of RAID5 but with the drives all replaced with SSDs. RAID4S has a dedicated parity SSD while RAID5 schemes rotate the parity amongst the drives.

The Direct I/O (`-dio`) option is activated for all XDD runs to avoid caching effects. In many file systems there is a system buffer cache which buffers data before the CPU copies the data to the XDD I/O buffer. Without activating Direct I/O, results would be skewed by the memory copy speed of the processor and the transfer of data from the file system buffer instead of the disk.

Block unaligned writes - medium writes

To test the performance of medium-writes, XDD is configured to perform writes in the range from 40KB to 256KB in increments of 4KB. Since all the RAID configurations are set up with 5 drives, the RAID4S write selection algorithm will select large-write parity computations when the writes span more than 1 drive and the RAID4S-modthresh write selection algorithm will select large-write parity computations when the writes span more than 2 drives. Hence, the range of writes tested is adequate to properly

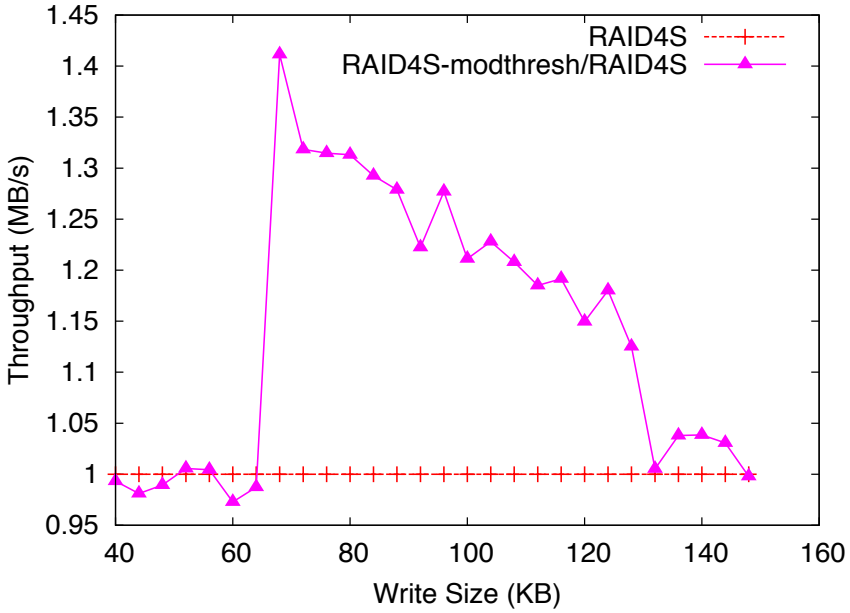

Figure 4.10: Comparing RAID4S-modthresh to RAID4S. The RAID4S-modthresh values have been divided by RAID4S to get a normalized speedup of RAID4S-modthresh over RAID4S.

illustrate the benefits of the RAID4S-modthresh when compared to RAID4S, RAID5, and RAID5S. Since sector sizes are 4KB, writing to sizes that are not multiples of 4KB will result in degraded and unpredictable performance.

Figure 4.1 compares the throughput of RAID4S and RAID4S-modthresh for randomly generated writes in the 40KB to 168KB range. From 40KB to 64KB, the write throughputs are identical because both write selection algorithms choose small-write parity computations. Similarly, writes larger than 128KB are also identical in both RAID4S and RAID4S-modthresh because both write selection algorithms choose large-

write parity computations. In this case, RAID4S-modthresh calculates `rmw` = 4 and `rcw` = 1, which does not satisfy the threshold condition for small-writes.

The most interesting results are what lies between, in the 64KB to 128KB medium-write range. In the medium-write range, RAID4S-modthresh shows a noticeable difference in the write throughput over RAID4S. The normalized speedup of RAID4S-modthresh over RAID4S can be seen in Figure 4.10. The throughput increases by 2MB/s for all the unaligned and aligned writes less than or equal to 128KB.

Note that in RAID4S-modthresh, writes larger than 128KB, although still considered medium-writes, are calculated as large-writes because `rmw` = 4 and `rcw` = 1. The results of this graph suggest that the threshold can be pushed even farther to the right, because RAID4S-modthresh still outperforms RAID4S up to about 144KB sized writes.

Figure 4.11 compares the throughput of RAID4S-modthresh to that of RAID5 and RAID5S. The results show significant speedup for RAID4S-modthresh for medium-writes over both RAID5 configurations and is finally overtaken by RAID5S after the 128KB write size. This is expected since RAID5S large-writes are faster than RAID4S because of the extra SSDs used in RAID5S. Even the 128KB write for RAID4S-modthresh compares favorably with RAID5S.

Finally, Figure 4.12 compares RAID4S-modified to RAID4S-modthresh. RAID4S modified alters the write selection algorithm to always select small-writes. The predicted effect is that small-write computation will perform worse than large-write computation because small-writes need to read more drives than larger write-sizes. The results show

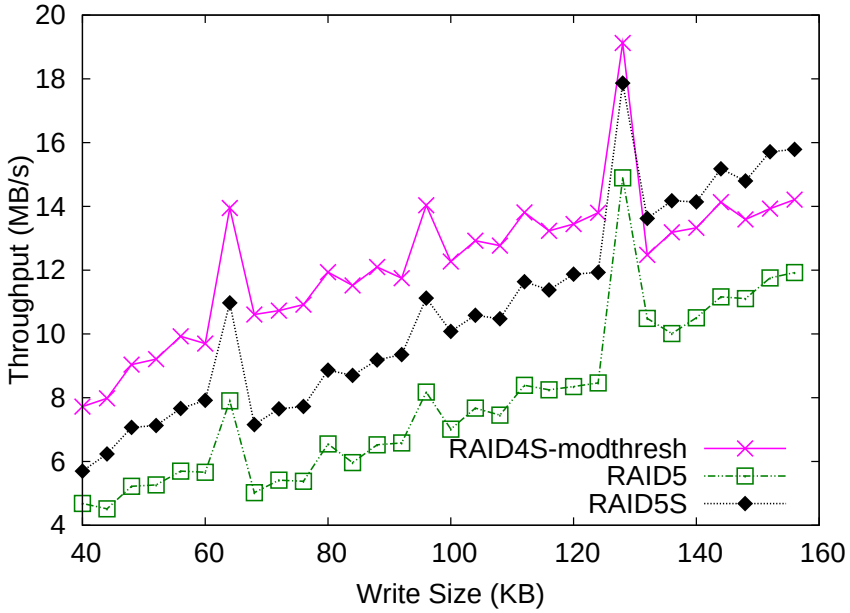

Figure 4.11: Comparing RAID4S-modthresh to RAID5 and RAID5S. The speedup is noticeably improved for medium-writes.

that immediately after 128KB, the RAID4S-modified throughput is still better then the RAID4S-modthresh throughput by a small margin. RAID4S-modthresh eventually achieves higher throughput than RAID4S-modified around 164KB, but the lower performance immediately after 128KB is still peculiar. These results, like Figure 4.10 suggest that the small/large-write threshold can be pushed even farther to the right to realize even more of a performance gain.

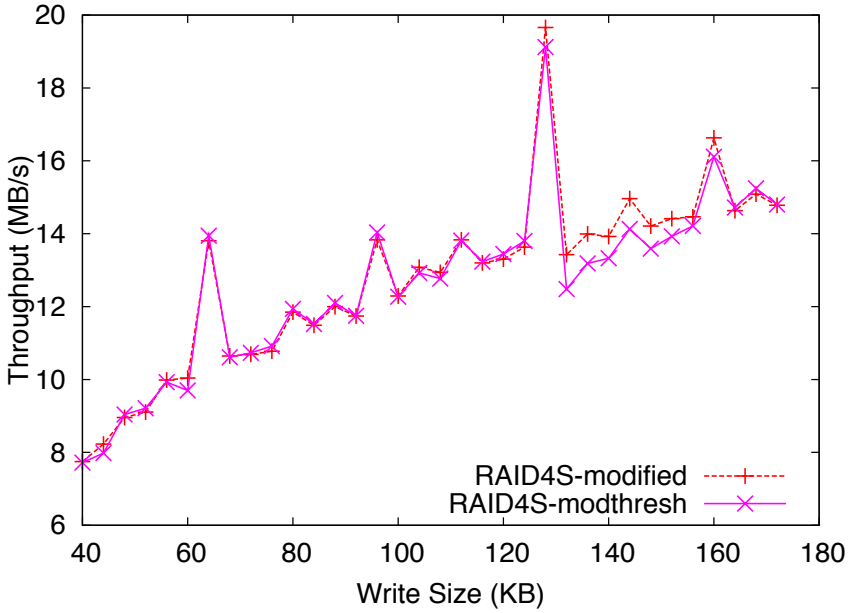

Figure 4.12: The throughput of RAID4S-modified (all small-writes) and RAID4S-modthresh have interesting results. It was expected that throughput would be the same for all writes up to and including 128KB but performance varies. Even more interesting is the trend immediately after 128KB, where RAID4S-modified has better performance than RAID4S-modthresh in throughput.

Block aligned writes: powers of 2

To test the overall performance of the system, XDD is configured to perform block aligned writes, which are write sizes that are powers of 2 (2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB). This experiment includes medium-writes, but unlike the previous experiment, this range also covers very small and very large-write sizes for a 256KB stripe. The block aligned write size experiments do not show as pronounced a speedup as block unaligned writes (section 4.4) because these experiments focus on the

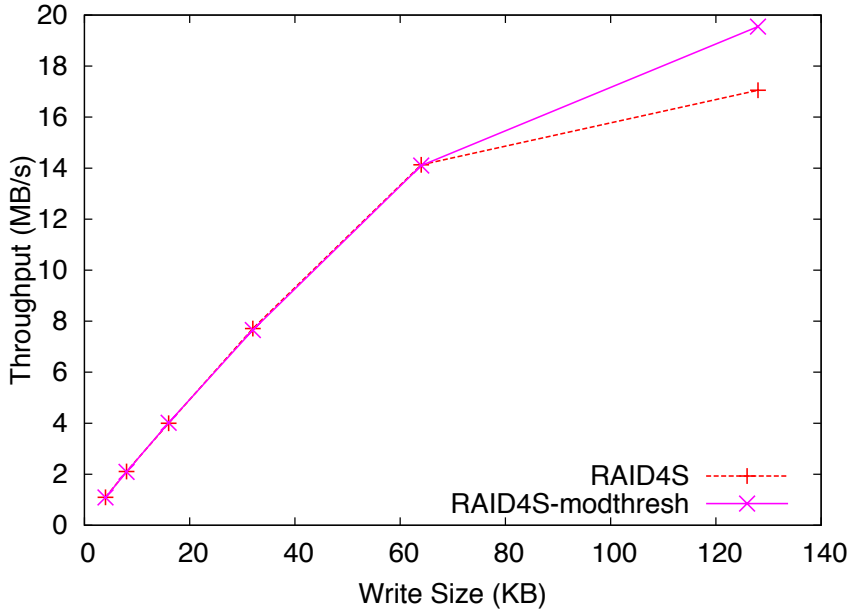

Figure 4.13: Comparing RAID4S-modthresh to RAID4S on a straight scale without the 256KB write-size. The 256KB write-size skewed the graph because it has such a high performance relative to the medium and small-write throughputs. For medium-writes, RAID4S-modthresh provides a noticeable speedup over RAID4S.

system as a whole. The purpose of these experiments is to show that RAID4S-modthresh will not degrade the performance of RAID4S.

Figure 4.13 shows the performance gain that RAID4S-modthresh experiences for 128KB sized writes compared to RAID4S. All smaller writes are the same since both systems select the same small-write computation for parity calculation. The write-performance for 128KB RAID4S-modthresh was predicted to be higher, judging from the slope of the line but the graph shows a degradation in performance relative to the expected performance. Notice that the throughput for RAID4S-modthresh steadily

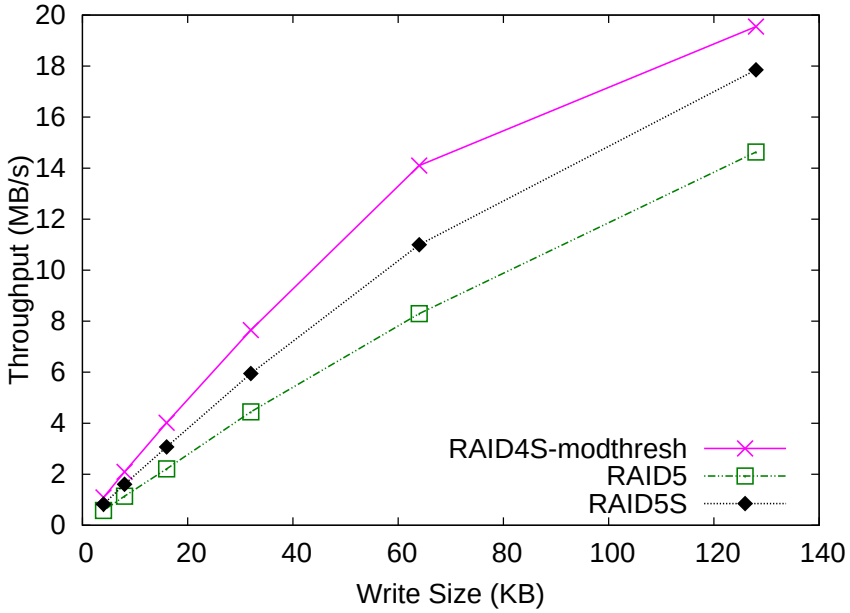

Figure 4.14: Comparing RAID4S-modthresh to RAID5 and RAID5S on a straight scale without the 256KB write size. The 256KB write size skewed the graph because it has such a high performance relative to the medium and small write throughputs. For medium writes, RAID4S-modthresh provides a noticeable speedup over the two other RAID configurations.

increases at a linear rate but veers off to a much lower value than interpolation would suggest. This behavior can probably be attributed to the additional disk read and its unanticipated overhead.

Figure 4.14 shows the throughput speedup of RAID4S-modthresh compared to RAID5 and RAID5S. Classifying and computing the 128KB write as a small-write permits RAID4S-modthresh to enjoy more of a speedup than RAID4S when compared to RAID5 and RAID5S. Although the performance is better than RAID4S, it is still far

less than expected, as discussed in the observations above.

Finally, Figures 4.15 and 4.16 show the minimal improvement that RAID4S-modthresh has on the entire RAID system. The speedups are evident but not overwhelming. The takeway from these graphs is not that RAID4S-modthresh facilitates a large performance increase, but rather that a small performance increase can be achieved by making a small change to the RAID controller without negatively impacting the rest of the system.

Although the measurements show that RAID4S-modthresh enjoys a small performance gain for medium-writes, the more important point that these experiments illustrate is that RAID4S-modthresh does not negatively impact either end of the write size spectrum. RAID4S-modthresh is an appealing implementation option for RAID4S systems because it moderately improves performance without incurring any noticeable performance degradations.

Future work and implementations

In addition to showing encouraging speedups for RAID4S-modthresh, the experiments show some interesting possibilities for future work. As noted in section 4.4, peculiarities in the comparison medium-write throughputs of RAID4S, RAID4S-modified, and RAID4S-modthresh show that the threshold for small-write computations can be extended to even larger medium-write sizes.

An interesting extension of RAID4S-modthresh would be to examine the exact

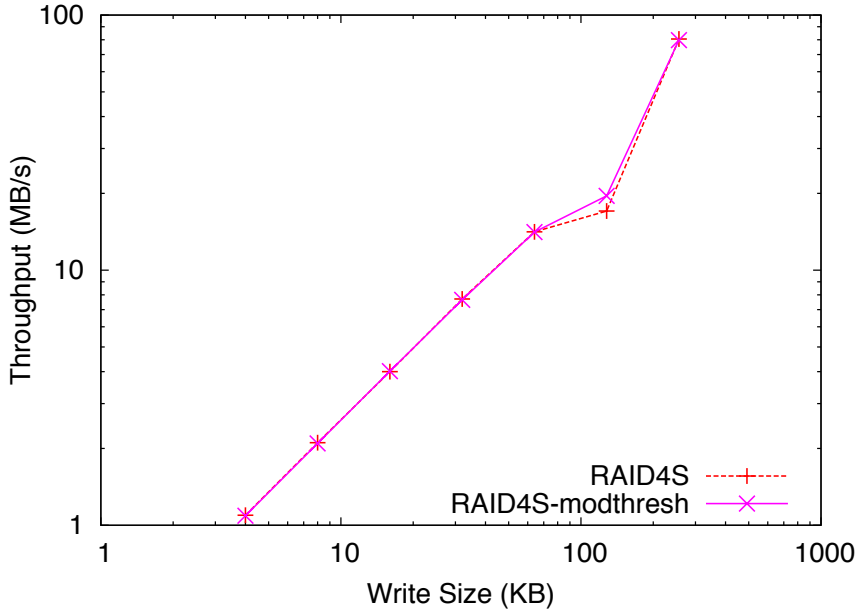

Figure 4.15: Comparing all the write sizes for block aligned writes (powers of 2) shows that RAID4S-modthresh increases the throughput of medium-sized writes with minimal impact on the rest of the RAID4S write sizes. Overall, the throughput for all write sizes is the same as the measurements in Figure 4.2 but with a small improvement in the 128KB medium-write.

write size, rather than how many drives are spanned, to select the write type. When examining fine-grained block unaligned write sizes, the current RAID4S-modthresh disk counting for the `rmw` and `rcw` variables seems awfully clunky and inadequate. To achieve fine precision and minor performance upgrades, it would be beneficial to re-examine the write-type calculation to see if the disk counting can be replaced by a more accurate measurement.

Another possibility for future work is to consider the write selection algorithm for

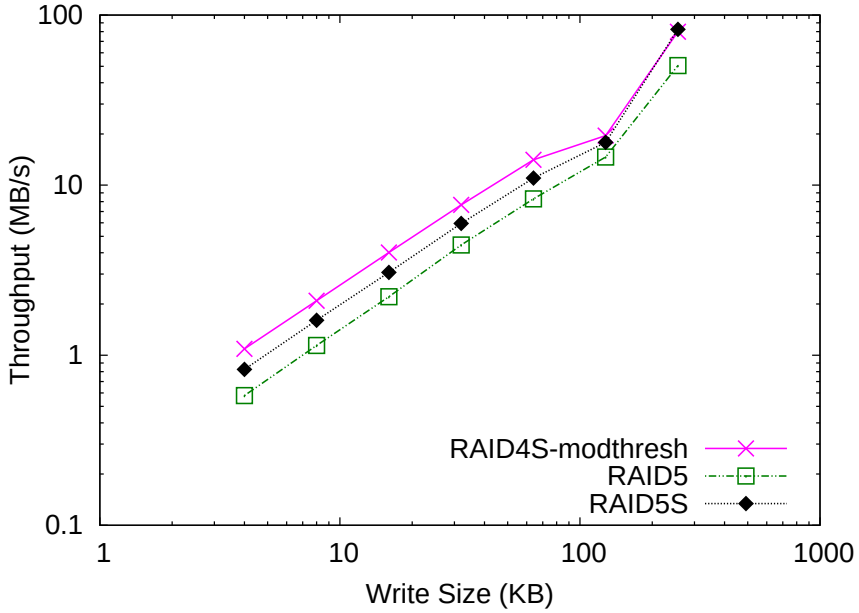

Figure 4.16: Block aligned writes (powers of 2) show that RAID4S-modthresh still performs better than RAID5 and RAID5S. The throughput for all write sizes is about the same as the measurements in Figure 4.2 but with a small improvement in the 128KB medium-write where RAID4S-modthresh utilizes the reconstruct-write algorithm.

RAID4S in choosing a better threshold. It would be beneficial to take a closer look at RAID4S-modthresh and RAID4S-modified, as shown in Figure 4.10 to figure out why RAID4S-modified performed better for some of the smaller large-writes. This might disprove our preconceived notion that large-writes are always better than small-writes. Debugging the two systems side by side to see what RAID4S-modified does that RAID4S-modthresh doesn't do would lead to an explanation of the speedup that might disprove some of the assumed characteristics of small and large-writes.

4.5 Related Work

Related research has similar resources and injects them into the RAID4 and RAID5 systems to achieve different levels of performance and reliability. Many of these solutions are complicated and require extra storage overhead which makes them less desirable from a pure performance perspective than RAID4S and RAID4S-modthresh.

HDD solutions

Parity logging [74] is a common solution to the small-write problem. For each small-write, parity logging saves the parity update image, which is the resulting XOR bitwise value of the old data XORed with the new data, called the parity update image, in a fault-tolerant buffer. When the buffer is filled with enough data to constitute an "efficient" disk transfer, the logged parity update image is applied to the in-memory image of the out-of-date parity and the new parity is written sequentially to the RAID setup. Although parity logging improves performance without the added cost of an SSD and can be expanded to RAID5, it also incurs a storage overhead that increases with the striping degree. It also does not address the small-write and large-write relationship and the possibility that making a small transformation to the threshold might not necessitate the need to alter the algorithm or storage implementation of the entire system. Finally, the irregularity of writes and the chance that a write can be unpredictable and may not fit the best sizes, may affect performance in unpredictable ways. Parity logging is great for disks of equal speeds but does not take advantage of SSD technology and the ability

to access multiple disks in parallel with many parity log writes.

LFS [65] optimizes writes by destroying locality, using a segment cleaner, and keeping track of data and addresses the overhead of small-writes to data. For small file workloads, the LFS converts the small disk I/Os into large asynchronous sequential transfers. Similarily, Write Anywhere File System (WAFL) [37] is designed to improve small-write performance. This is achieved by letting WAFL write files anywhere on disk. Hence, WAFL improves RAID performance because WAFL minimizes seek time (write to near blocks) and head contention (reading large files). These two methods assume that small-writes are poor and transform large writes by logging the data. By contrast, RAID4S-modthresh takes advantage of the RAID4S layout to improve small-write performance rather than trying to eradicate the small-writes completely.

SSD solutions

Heterogenous SSD based RAID4 [57] injects SSDs into a RAID4 setup by replacing all the data drives with SSDs. They measured performance in terms of life span and reliability instead of cost and throughput. They argue that using the SSD as a parity drive is subject to unacceptable wear caused by the frequent updates to the parity. This wear, up to 2% higher than the heterogenous RAID4, can cause an increase in error rates, data corruption, reduced reliability, and a reduction of SSD lifecycles. This RAID4 system values different performance measurements: reliability and life span. They acknowledge the small-write bottleneck but do not address throughput in their

evaluation, which is the purpose of RAID4S and, by extension, RAID4S-modthresh. RAID4S and RAID4S-modthresh focus on improving small-write performance without incurring a large cost penalty.

The Hybrid Parity-based Disk Array (HPDA) solution [51] uses SSDs as the data disks and two hard disks drives for the parity drive and a RAID1-style write buffer. Half of the first drive is the traditional parity drive while the remaining half of the first HDD drive and the whole of the second HDD drive is used to log small-write buffers and as a RAID1 backup drive. This solution facilitates high performance for small-writes and avoids the flash wear-out and erase-before-write SSD problems. RAID4S experiences more wear then HPDA, but costs less and requires less storage space because it does not require $n - 1$ SSD drives. RAID4S-modthresh is also simpler, cheaper, and provides similar speedups.

Delayed parity techniques [42] log the parity updates, instead of the small-writes, on a partial parity cache. Partial parity takes advantage of the flash properties to read only when necessary. The process has two states, partial parity creation / updating and partial parity commit. The delayed parity reduces the amount of small-writes. The create/update alters the state of the partial parity cache by inserting the parity if it absent and determining if an existing parity is redundant, i.e. whether it needs to XORed with the old data. This reduces the number of reads, since the full parity is not stored and reduces the resulting parity generation overhead. The problem with this is that it increases the commit speed overhead, introduces an extra calculation overhead

(what needs to be read), and costs more because it is a RAID5S system. RAID4S and RAID4S-modthresh introduces a solution to the small-write problem to increase performance without the same storage and complexity overhead.

4.6 Conclusions

RAID4S provides a significant throughput speedup for small-writes while striving to maintain a low-cost and reliable configuration. RAID4S-modthresh is a modification to the software RAID controller (`raid5.c`) for a RAID4S system that modifies the write selection algorithm to use the small-write parity computation for medium sized writes. Speedups of 2MB/s, which has a maximum speedup factor of 1.42X, are observed when comparing RAID4S-modthresh to RAID4S. With minimal changes to the software RAID controller and the RAID setup, RAID4S-modthresh takes advantage of the small-write speedup of RAID4S to provide moderate throughput improvements, which are most noticeable in block unaligned writes.

Future work includes further comparisons between the RAID4S-modthresh and RAID4S-modified (selecting all small-writes) algorithms to determine where to set the most optimal small/large-write threshold and how to utilize drive workloads and the exact write size request to construct a more accurate write selection algorithm.

Chapter 5

RAIDE: Optimized Stripe Edge Workload Heterogeneity

Random write workloads in RAID5 result in a heterogeneous workload distribution across the array. Specifically, the edge devices perform more reads (due to read-modify-write activity) than the middle devices. This chapter shows an example of this phenomenon, provides an analysis of why it happens, and finally presents RAIDE results.

Figure 5.1 displays the Iostat read activity from a small write workload on a 4+1 RAID5 array with 64KB chunk size. The data indicate significant read workload imbalance, particularly at write sizes 320KB, 640KB, and 960KB. These write sizes do not represent the same increment away from full stripe writes. Specifically, 320KB is a 1.25X stripe width, 640KB is a 2.5X stripe width, and 960KB is a 3.75X stripe

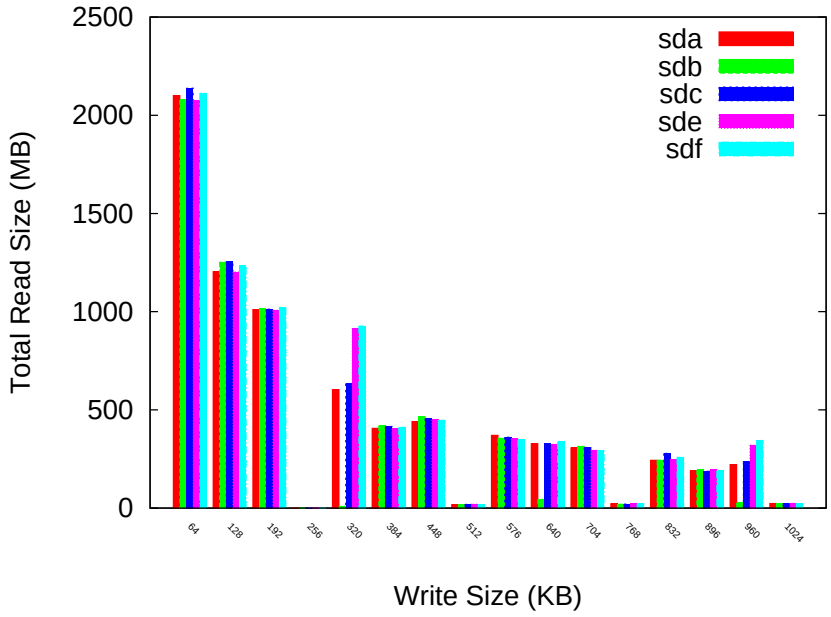

Figure 5.1: RAID5 small writes show unevenness at 320KB, 640KB, and 960KB.

width. The other point to make is that there is imbalance at sizes above small writes. Even when writes are coalesced into larger array writes, there is imbalance as long as the write is not aligned to the array (i.e. small writes are part of the larger write). While this can be avoided with careful workload design, it's likely that some writes will have this behavior.

Let us define a stripe edge as the boundary between stripes in an array. Each device that straddles the edge is considered an edge device. Consider the example array in Figure 5.2. Each stripe 0-4 starts at device 0 and ends at device 4, where the data wraps to the next stripe at the edge. In this array, devices 0 and 4 are edge devices since they

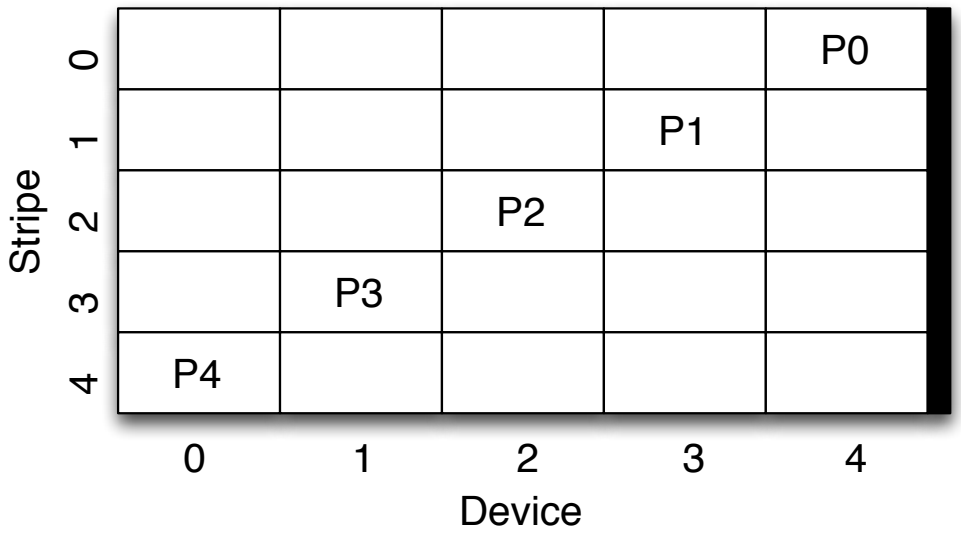

Figure 5.2: The RAID edge is the boundary between consecutive stripes. The thick line on the right side indicates the edge of the array. Edge devices are the devices next to the edge (devices 0 and 4 in this example).

are the boundary between stripes. We will see more specifically why edge devices are significant in Section 5.1. For now, recognize that a write that spans at least two data devices may spread across two stripes. When a write spans two stripes, it requires updates to two different parities. This results in an uneven workload at the edges because of these extra edge updates.

5.1 RAID4 Uneven Workload

Small write workloads require data and parity reads in addition to the specific data requested to be written. The data that needs to be read depends on the size of the write and the alignment within a full stripe. When a write spans across more than one stripe, it may require up to two small writes.

i	j	size	rmws	rcws
0	2	2	0	1
1	3	2	0	1
2	4	2	0	1
3	0	2	2	0

Table 5.1: Read-modify-write and reconstruct-write counts for writes spanning two devices of an array. The last write mapping breaks the write into two smaller writes, resulting in an increased read workload on the edges of the array.

Each write size and position is analyzed to show how many of each small write type is required to complete that specific write. Writes of size 1 are straightforward: every write that fits on one device requires a single read-modify-write (rmw).

Table 5.1 looks at the example of a small write of size two (i.e. two devices) on an array of size 4. Depending on the start location of the write, we see one reconstruct-write or two read-modify-writes. Each of these cases will stress the array in a different way. The reconstruct writes will sum up to a balanced workload, in a fully random workload. The read-modify-writes, however, will produce extra reads on devices 0 and 3. This case motivates placement of faster hardware at the edges of RAID arrays.

The next part looks at the small write activity required by any write, given its start position and size. This is to generalize the example in Table 5.1. There are five basic cases:

1. Full stripe write(s).
2. Single stripe small write.
3. Full stripe write(s) then small write.
4. Small write then full stripe write(s).

5. Small write, full stripe write(s), then small write.

The first task is to compute the end position of the write.
Given: Write position (i), size (k), array size (n). Find end position (j).

```
if (k <= n-i)
     j = n-i+k;
else
     j = (k-i)%n;
```

With i, j, k, and n, we perform the algorithm in Figure 5.3 to determine the number of each type of partial-stripe write. This function determines the small write activity required by any write, given its start position and size. It uses conditionals to separate small writes into each possible array spanning (e.g. one stripe vs. two stripes) and determines the size of each small write.

5.2 RAIDE Design

RAIDE recognizes that small write workloads introduce an imbalance in the workload across data devices in RAID4 and RAID5. When a write spans more than one stripe and one stripe contains a small write, that small write will introduce an extra read and write to that edge device. In a random workload, these add up to a significant fraction of the workload to those devices.

RAIDE handles this additional workload by placing faster devices at the edge of the array. We chose to use a RAID5 parity placement strategy to differentiate between RAID4S and RAIDE performance improvements. The ideal speed difference between

```
int small-write-1-size = 0;
int small-write-2-size = 0;
if (n-i >= k) { // Write fits in a single stripe
    if (n == k) {
        // Full stripe
        small-write-1-size = 0;
        small-write-2-size = 0;
    } else { // Single stripe small write
        small-write-1-size = k;
        small-write-2-size = 0;
    }
} else { // Write spans two stripes
    // small write, full stripe
    if (j == n) { // Small write, then full stripe
        small-write-1-size = n - i;
        small-write-2-size = 0;
    } else if (i == 0) { // Full stripe, then small write
        small-write-1-size = 0;
        small-write-2-size = j;
    } else { // Small write, then small write
        small-write-1-size = n - i;
        small-write-2-size = j;
    }
}
```

Figure 5.3: Analysis of small write workloads shows how to determine the small writes required for a given write. This information could be used by the RAID controller to remap writes that are split across two stripes into a single stripe.

edge and non-edge devices should match the workload difference seen in the Iostat data for a typical workload on the system. For testing purposes, the array contains two SSDs at the edges and three HDDs in the middle.

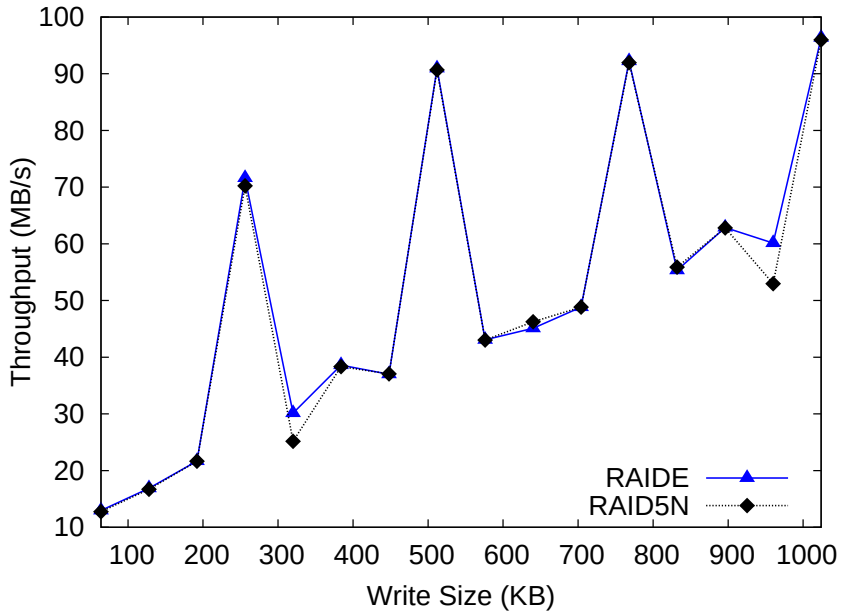

Figure 5.4: RAIDE random write throughput with 64KB chunk size shows that RAIDE nearly always performs equal to or better than RAID5N with the same hardware in a different configuration.

5.3 Results

Our test bed is the same machine and Linux configuration described in Chapter 3. We place an SSD at each edge of the array, at positions 0 and 4, and three HDDs in the middle, at positions 1-3. For evaluation, we consider the naïve placement strategy of RAID5N. We place an HDD at each edge of the array and at the center position 2, while placing the two SSDs at positions 1 and 3.

The random write experiment was run for a total of 8192MB of writes at each write size. Figure 5.4 shows the result. RAIDE performs better than RAID5N at 320KB

and and 960KB. These two points are 1.25 stripes and 3.75 stripes, respectively. The near-full stripe write sizes always have small write segments in addition to the full stripe aligned write portion.

5.4 RAIDE Variants

There are several alternative layouts that utilize the uneven workload observed in this chapter. RAIDE is the closest to RAID5S, but in fact any striped array is likely to benefit from placing faster devices at the edges of the array. For future work, it may be useful to combine RAIDE with RAID4S and RAID4S-modthresh. This experiment will require at least three faster devices. Ideally the parity device will be faster than the edge devices since the parity workload is higher than the edge workload.

5.5 Conclusions

This chapter presents an analysis of several small write workloads and shows that the edge devices in RAID arrays are busier than the middle devices. The experimental results show a throughput improvement when faster devices are placed at the edges of an array. The heterogeneous workload we observed in this chapter directly motivated RAIDE while also encouraging us to look at further array heterogeneity. Chapter 6 examines heterogeneous hardware in RAID arrays and presents an algorithm that stores extra parity on faster devices.

Chapter 6

RAIDH: Performance-Enhanced Arrays of Heterogeneous Devices

RAID arrays typically start out with homogeneous devices, but may become heterogeneous as failed devices are replaced and to upgrade the array. We examine a heterogeneous storage hardware configuration called RAIDH. The goal of this chapter is to illustrate the shortcomings of RAID in the context of heterogeneous hardware and present a hybrid layout that appreciates heterogeneous device performance across an array. RAIDH improves the small write performance of RAID arrays composed of mixed hardware, enabling benefits that typical RAID layouts do not provide.

6.1 Parity Placement Algorithm

The location of parity in each stripe is calculated using a deterministic weighted parity placement function [28]. Each device is weighted individually, allowing full control over how much parity goes to each device.

Figure 6.1 displays an implemention of the parity placement function. The individual device weights are stored in the weights[5] array, shown here as 3, 2, 1, 1, 1. Each placement for the parities[weightSum] array is chosen as the maximum value in value[5]. The value[] array is updated by subtracting weightSum from the chosen parity location and adding the corresponding weights[k] to value[k] for all k. This function produces the parities[] array that is input into the RAID placement code described in Section 6.3.

Parity weight examples

The first example places parity in a RAID4 layout. Each data drive has parity weight 0, while the last drive has parity weight 5. The parity weight array for RAID4 is `{0, 0, 0, 0, 5}`. The parities array indicating where to store parity for a given index within the size 5 cycle is `{4, 4, 4, 4, 4}`.

RAID5 parity distribution is also easy to construct using the following weight distribution array. Each device is selected once to store parity, thus dividing parity evenly across the array. For RAID5, the parity weight array is `{1, 1, 1, 1, 1}`. The parities array for RAID5 is `{0, 1, 2, 3, 4}`.

```
public class weightedRAID {
    public static void main(String[] args) {
        int[] weights = new int[] {3, 2, 1, 1, 1};
        int[] value = new int[weights.length];
        for (int i=0; i<value.length; i++)
            value[i] = 0;
        int weightSum = 0;
        for (int i=0; i<weights.length; i++)
            weightSum += weights[i];
        // Location of parity for first weightSum stripes.
        int[] parities = new int[weightSum];
        for (int i=0; i<weightSum; i++) {
            int j = argmax(value);
            parities[i] = j;
            value[j] -= weightSum;
            for (int k=0; k<value.length; k++) {
                value[k] += weights[k];
            }
        }
    }
    private static int argmax(int[] x) {
        int argmax = 0;
        int max = x[0];
        for (int i=1; i<x.length; i++) }
            if (x[i] > max) {
                max = x[i];
                argmax = i;
            }
        }
        return argmax;
    }
}
```

Figure 6.1: Java implementation of weightedRAID parity placement algorithm example.

6.2 Heterogeneous Array Performance Analysis

Evaluating potential heterogeneous arrays is challenging because the hardware may not be available and the setup cost of each RAID array is high (physical hardware must be installed, RAID must be initialized with data, etc.). It is better to design the several possible array configurations on paper and analyze their expected performance before building any systems. With this data, the best configuration is chosen.

Let us consider a typical set of hardware and several possible array choices. Suppose we have two fast hard drives, two slow hard drives, and one fast SSD. The devices all have the same capacity. The speed of the faster devices should be described normalized to the slowest device in the system. Thus the slow device has speed k, and in our example, the fast hard drives have speed $2k$, and the SSD has speed $4k$. The default RAID5 setup would store the same amount of data and parity on each device. To utilize the speed differences, the parity weight array could be `{0, 0, 1, 1, 2}`. This places no parity on the slow HDDs and splits the rest of the parity between the remaining faster devices such that the SSD gets twice as much as the fast hard drives. With this parity weight array, the locations to store parity given by the parities array is `{4, 2, 3, 4}`.

The performance of this RAIDH array depends on its workload. A read-only workload to this configuration would prioritize the slower devices, thus this array is best for write workloads. The cycle size is 4, which means that any given write will fall into one of 4 possible parity placement stripes within the cycle (i.e. the parity may fall on

	Device Reads					Device Writes				
Size	0	1	2	3	4	0	1	2	3	4
1	12.5	12.5	21.9	21.9	31.2	12.5	12.5	21.9	21.9	31.2
2	15.4	15.4	21.1	21.1	26.9	15.4	15.4	21.1	21.1	26.9
3	22.7	9.1	15.9	22.7	29.5	17.9	14.3	19.6	21.4	26.8
4	23.9	8.5	16.9	22.5	28.2	17.3	16.5	22.0	22.0	22.0

Table 6.1: Percentage of total read and write workloads handled by each device in the example RAIDH array. The size of each write is in multiples of the array chunk size.

any of the available 4 parity locations). Each specific workload also has several possible placements within that parity choice (i.e. the data may start anywhere in this stripe or up to $size - 1$ away and but might split across the parity if it is larger than a single device write). Table 6.1 outlines the percentage of reads and writes going to each device with a random write workload of a given size. Each percentage was determined by counting the number of reads and writes to each device for every possible data placement and parity placement. The data indicates that our parity weight choices might not put enough parity on the faster devices. For the size 1 case, for example, devices 2-3 perform 1.75X more reads and writes than devices 0-1, below the performance difference of 2X. The data also shows evidence of the workload imbalance at the edges of the array for sizes 3-4. This phenomenon is described in more detail in Chapter 5.

The workload percentages in Table 6.1 provide insight into what the array will do to complete each workload. Using the performance numbers (SSDs are 2X faster than fast hard drives which are 2X faster than slow hard drives), we determine the expected speedup from RAIDH over RAID5. RAID5 would spread data and parity evenly over all devices. Focusing only on writes of size 1, we can ignore the edge workload imbalance.

For these writes, the RAID5 workload is evenly distributed across the array. On the other hand, RAIDH will perform 1.75X more reads on the 2X faster hard drives and 2.5X more on the SSDs. Because the fast hard drives are 2X faster than the slow ones, these fast hard drive I/Os will complete faster than the I/Os on the slow hard drives. The same argument about the SSD workload follows the same way. Thus the most important workload is the one to the slow hard drives. The total workload to these two devices is 25% of the total workload. The same devices in a RAID5 array would be handling 40% of the workload. RAIDH offloads 15% of the workload from slow devices to faster devices.

Predicting RAIDH vs. RAID5 array performance

There are two natural parameters that help predict RAIDH array performance compared to RAID5. The first is the heterogeneity of devices present in the array. The second is the width of the array (larger arrays will also have more likelihood of heterogeneity).

Increasing array width

The choice of array width directly impacts array performance and reliability. Larger arrays provide higher device parallelism and thus higher performance. They also reduce reliability unless additional parity is computed and stored. Figure 6.2 shows four different array configurations, varying from two devices to five devices. For RAIDH, increasing the array width also increases the relative heterogeneity. As long as the faster

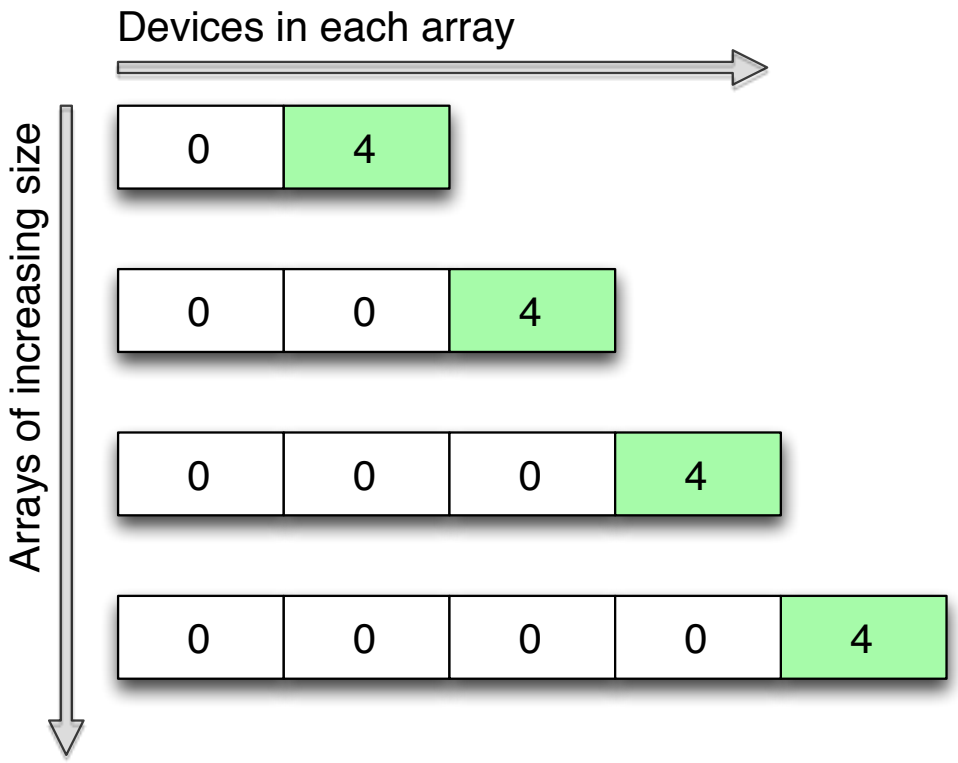

Figure 6.2: Increasing the stripe width while maintaining heterogeneity (one device is faster than others) results in increased benefits of RAIDH. This is sustained until the faster device is no longer fast enough to support the rest of the array.

device is fast enough to support the additional parity workload, larger arrays will result in greater benefits from RAIDH.

Decreasing array heterogeneity

The amount of heterogeneity of the devices in the array clearly impacts the performance benefits of RAIDH. The RAIDH algorithm will chose a RAID5 layout for completely heterogeneous arrays. Figure 6.3 varies heterogeneity from high to zero. The high heterogeneity array has one device with parity weight 8 and the rest of the array with

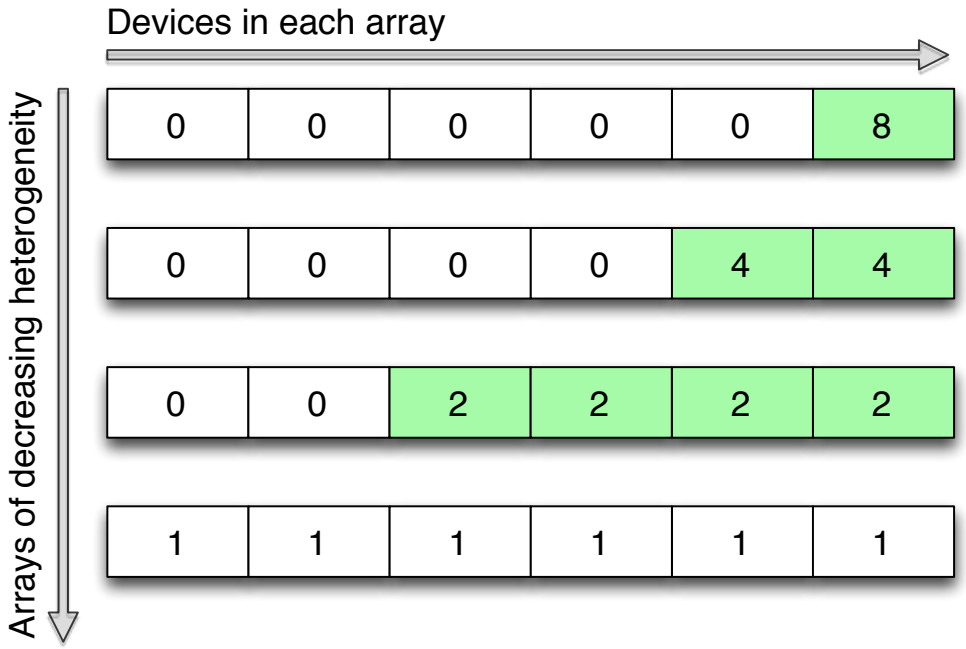

Figure 6.3: Decreasing the heterogeneity of the array while maintaining stripe width of six devices results in decreased benefits of RAIDH. As heterogeneity is reduced, the array becomes more homogeneous and RAID5 becomes a better layout.

parity weight 0. This indicates that the faster device should store all parity. This layout is also known as RAID4S, which was described in Chapter 3. The next array has two devices, each of parity weight 4, in an array of devices with parity weight 0. In this case, the faster devices will split the parity and each will store half parity and half data. Because the fast devices are now storing some data, the array is more similar to RAID5 (which would store equal amounts of data and parity on all devices). Thus the performance difference between RAIDH and RAID5 will decrease as array heterogeneity decreases.

```
int size_of_box = weight_sum * data_disks;
int offset_within_box = stripe2 % size_of_box;
int parity_idx = sector_div(offset_within_box, weight_sum);
switch (algorithm) {
    case ALGORITHM_HETEROGENEOUS:
        pd_idx = parity_indexes[parity_idx];
        if (*dd_idx >= pd_idx)
            (*dd_idx)++;
        break;
/* ... other RAID5 algorithms ... */
}
```

Figure 6.4: Code to compute the locations of data and parity for RAIDH.

6.3 Implementation

RAIDH is implemented in the Linux RAID kernel module. The implementation is added to the existing RAID5 software, defined in `/usr/src/linux.source.2.6.32/drivers/md/raid5.c`. An overview of the existing functionality is described in Section 4.2.

The RAIDH algorithm allows flexibility in parity placement in place of existing parity-based RAID algorithms. Many different parity weightings are possible; our experiments look at several specific choices that improve performance for our hardware.

The most significant software changes to enable RAIDH are located within the function `raid5_compute_sector()`. This function essentially computes data and parity location for a given sector. Recall that the set of parity device locations is the size of the sum of weights; thus we store the parity locations in array parity_indexes[weight_sum]. The code snippet in Figure 6.4 finds the locations using the

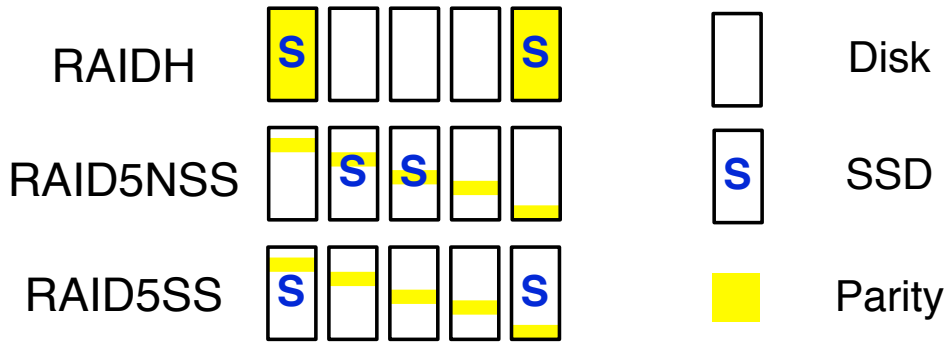

Figure 6.5: All layouts have two SSDs and three hard drives. RAIDH stores parity on SSDs and places them at the edge of the array. RAID5NSS and RAID5SS use a RAID5 layout but place the SSD in the middle or at the edges.

geometry of the array and parity for heterogeneous RAID.

The process to install the modified kernel module is described in Section 4.2.

6.4 Results

The basic system setup used for these experiments is the same described in Chapter 3. Figure 6.5 pictorially shows the hardware and parity setup for each RAID layout. The hardware setup for RAIDH is two SSDs and three HDDs, with the HDDs in the middle positions.

We combined the RAID4S-modthresh optimization with RAIDH so that half-stripe writes will read the parity devices. The layouts for comparison are RAID5SS, where the same hardware is used in the same placement with RAID5, and RAID5NSS, where the hardware is the same but SSDs are shifted to the center positions. RAID5NSS removes the extra benefit of placing faster devices at the edges of an array to handle the additional

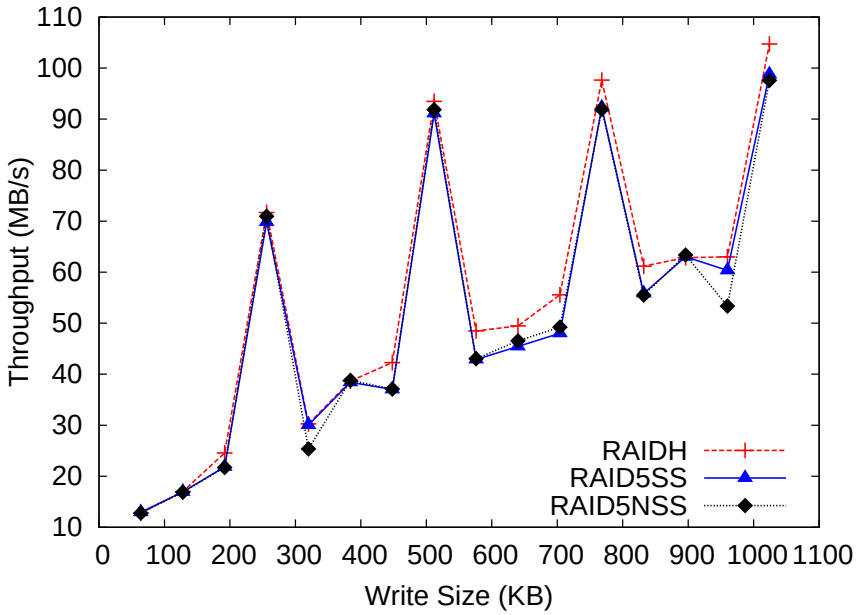

Figure 6.6: Random write throughput of RAIDH, which stores parity divided between two SSDs shows better performance than RAID5 variants at critical points. All three experiments use the same hardware: two SSDs and three HDDs. The RAID5SS array contains SSDs on the edges (positions 0 and 4), optimizing small read performance on some rmw-writes. The RAID5NSS (naïve) array contains SSDs in the middle (positions 1 and 2) and avoids the edge read optimization.

workload. More details on the edge device workload can be found in Chapter 5.

XDD random write

Figure 6.6 compares the throughput of RAIDH to RAID5SS and RAID5NSS. The array is configured with 64KB chunks and 4+1 parity. RAIDH throughput matches or beats both RAID5 configurations. This is a promising result, especially that RAIDH continues to perform better for larger write sizes. This indicates that RAIDH is versatile and

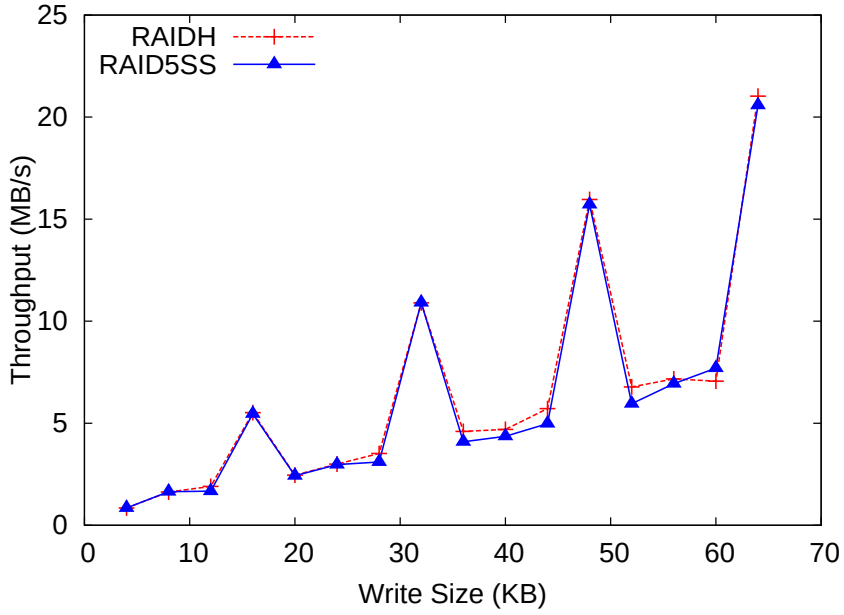

Figure 6.7: With a 4KB chunk size, heterogenous RAID performs better than RAID5SS at most write sizes.

provides a benefit for many write workloads.

For the next experiment, we investigate a smaller write workload and configure the RAIDs with a smaller chunk size to match. The RAIDs are configured with 4KB chunks and the workloads range from 4KB to 64KB in increments of 4KB. Figure 6.7 shows the random write performance of RAIDH with this 4KB chunk size. The throughputs are all lower than in Figure 6.6 because hard drives perform better as write sizes get larger (because there are fewer seeks per Byte). Because the throughputs are smaller, so are the differences between the two configurations. RAIDH performs slightly better than RAID5SS for most write sizes, aside from an anomaly at 60KB.

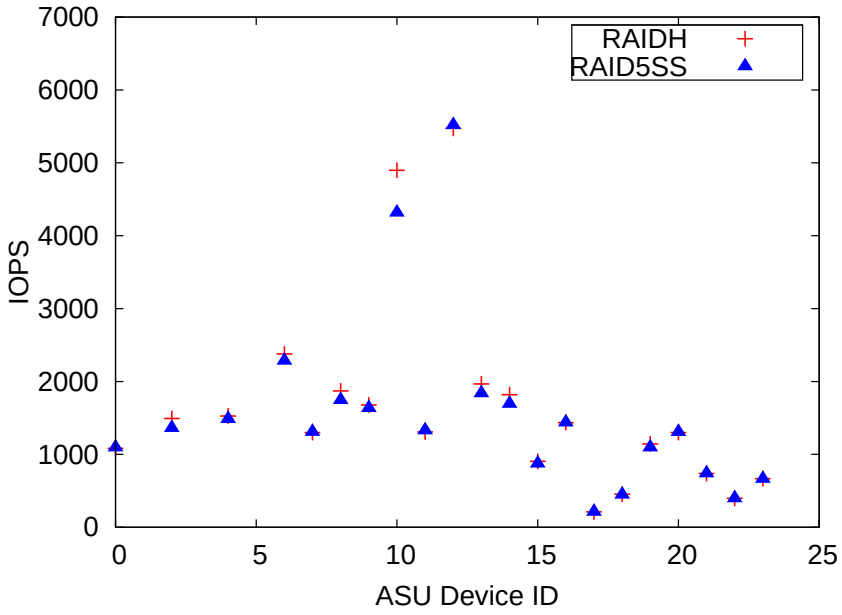

Figure 6.8: RAIDH Financial replay shows improvement for ASU Device ID 10. The other workloads don't show a big difference in raw throughput.

Financial trace replay

This section of the results uses the Financial traces from the Storage Performance Council [1]. The two traces, Financial1 and Financial2, were collected at two separate financial institutions. The replay here is of Financial1 and replays 21 of the 24 devices (we ignored three that require a larger block range than we can support).

The replay results are shown in Figures 6.8 and 6.9. RAIDH is compared to a basic RAID5 (RAID5SS) with the same hardware. Because the Financial1 trace contains mostly small ($< 64KB$) write sizes, the RAIDs are configured with a 4KB chunk size. This means that the writes will be distributed between small and large writes rather

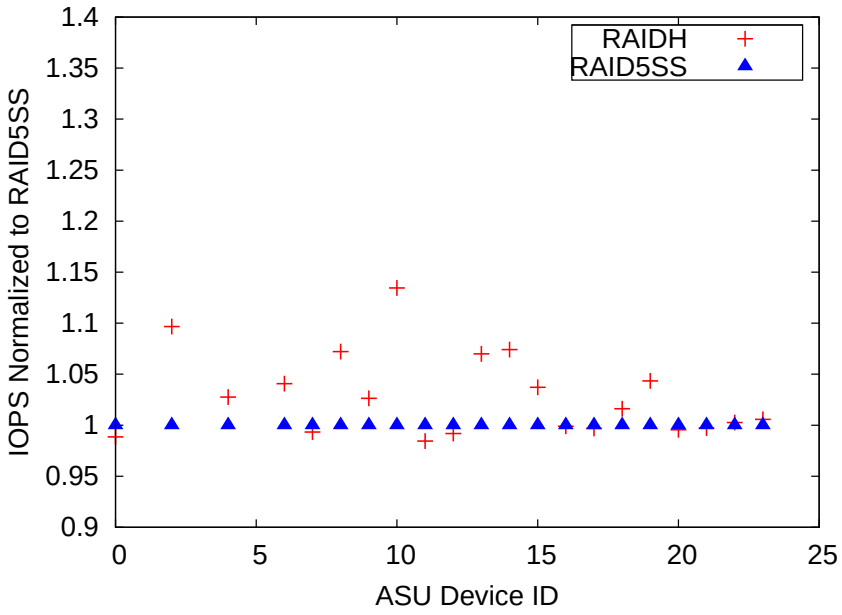

Figure 6.9: RAIDH Financial replay normalized to RAID5SS shows a small throughput improvement for most workloads.

than all fitting on a single device as read-modify-writes. The results show a minor improvement in some device workloads with occasional lower performance. The points where RAID5SS performs better are likely because RAID5SS benefits from SSDs at the edge of the data stripes.

The financial replay results have some of the best throughput improvements for RAID4S. Figure 6.10 summarizes the results of RAID4S with RAIDH. We see that RAIDH has the highest performance in many cases, with over 2.5X improvement over RAID5 for ASU 10. This final result highlights the benefit of incorporating ideas from each of the previous chapters into RAIDH.

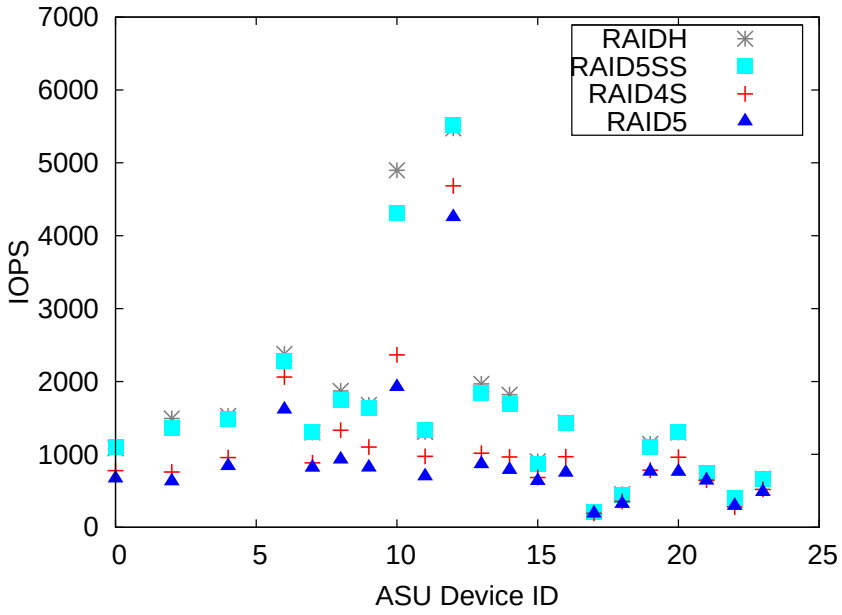

Figure 6.10: RAIDH Financial replay shows improvement for ASU Device ID 10. The other workloads don't show a big difference in raw throughput.

6.5 Performance Model

By making several simplifying assumptions about the I/O workload, we can predict RAIDH performance in arbitrary hardware configurations. First we assume a small random write workload, since placing parity on faster devices is best for this workload. Note that random workloads are a reasonable real-world case to assume, as even sequential workloads can become fragmented and random as storage fills up. Next, we assume that individual writes currently queued will not randomly reside in the same stripe. This allows us to assume that all writes can be parallelized. Last, we assume

that the array is fully utilized and there is no idle time. Our workload is a single write randomly assigned to each stripe in the array, thus getting a full distribution of possible writes. Now the workload will be completed as fast as the slowest drive finishes its writes. Note that when offloading I/Os to faster devices, we don't want to offload so many that the faster device becomes a bottleneck.

Let's consider a workload of X small writes. Including parity reads and writes, we have a total of $4X$ I/Os. Let $Y = 4X$. For an array of N devices with RAID5, this distributes to $\frac{Y}{N}$ I/Os per device.

Now suppose one device has speedup factor F over the other devices. For example, a device that is twice as fast as the rest would have $F = 1$. This is the same as having Y I/Os on $N + F$ devices. The extra workload can be seen as offloaded to a device with speed F, since we assume optimal workload parallelization. Now we have the following equation for the number of I/Os per device $\frac{Y}{N+F}$.

The reduction in I/Os per device is:

$$= \frac{\frac{Y}{N} - \frac{Y}{N+F}}{\frac{Y}{N}}$$

$$= \frac{N\left[\frac{Y}{N} - \frac{Y}{N+F}\right]}{Y}$$

$$= \frac{\frac{YN}{N} - \frac{YN}{N+F}}{Y}$$

$$= 1 - \frac{N}{N+F}$$

Let's validate this result with a couple examples.

1. $N = 5$ and $F = 0.5$ (i.e. one device completes small writes 1.5X as fast as the rest of the array). The reduction in I/Os per device is $1 - \frac{5}{5.5} = 9\%$.

2. $N = 5$ and $F = 3$ (i.e. 4X). The reduction in I/Os is $1 - \frac{5}{8} = 37.5\%$.

Here we convert a reduction in workload to an increase in throughput. The throughput for workload Y is $\frac{Y}{T}$, where T is the time to complete the workload. If T is reduced by $Z\%$ (which is the same as reducing the number of I/Os by $Z\%$), the increase in throughput is:

$$= \frac{\frac{Y}{T(1-Z)} - \frac{Y}{T}}{\frac{Y}{T}}$$

$$= \frac{T}{Y}\left[\frac{Y}{T(1-Z)} - \frac{Y}{T}\right]$$

$$= \frac{1}{1-Z} - 1$$

Let's look at a couple examples to verify this result. If time is decreased by 50%, throughput should increase 100%. If time is decreased by 9%, throughput should incrase 10%. These results make sense when we consider that we don't want to offload so many

I/Os that the faster devices become a bottleneck. If we had offloaded the amount that the faster device can handle, then it would have taken as long as the slower devices took to complete the full RAID5 workload. Thus the workload would have taken the same amount of time with no speedup. By offloading just enough so that all devices complete at the same time, we get a better speedup.

These predicted results will vary for real workloads. The actual speedup will be proportional to the percentage of small writes sent to the RAID. Larger writes that are unaligned may still provide high speedups over RAID5, but fully sequential aligned writes will perform similarly to RAID5. Read workloads will perform proportionally worse, because read workloads do not use parity unless the array is in degraded mode (i.e. a device has failed). The faster devices will be underutilized since they store extra parity that is not used in read workloads.

In order to validate the model with experimental data, consider the results of Figure 4.14. For write size 64KB, RAID5 throughput was about ~8MB/s and RAID4S-modthresh throughput was about ~14MB/s, an increase of ~75%. With $N = 5$ and $F = 3$, for that experiment the model predicts a throughput of $13\frac{1}{3}$MB/s, an increase of 62.5%, very close to our observation—close enough to validate our model. We attribute the slightly higher throughput of RAID4S to the fact that RAID4S is somewhat more likely to be able to parallelize multiple small I/Os than RAID5, whereas our model assumes that both RAID5 and RAID4S always parallelize all I/Os equally.

Aside from the optimistic assumption about parallelization (which slightly under-

estimates real world performance of our system), the model is otherwise a best-case estimate of possible throughput increases. On real workloads, the improvement will be proportional to the ratio of small writes in the workload.

6.6 Related Work

Heterogeneous devices are increasingly common in storage arrays, but there is not a lot of work to address the issue in RAID systems. One direction taken is to address the size heterogeneity by virtualizing physical disks into logical disks [95]. This approach breaks up disks into chunks, divides the chunks into parity groups, then merges them into logical arrays. The work enables very high reliability of the logical arrays, but does not address array performance.

AdaptRAID0 and AdaptRAID5 are algorithms that improve RAID performance by remapping blocks in heterogeneous arrays [27]. The implementations both take advantage of faster devices by using uneven stripe widths. The faster devices are present in all stripes while the slower devices are left out of some stripes. This resulted in higher IOPS in many cases, but does not provide any benefit for a single faster device.

6.7 Conclusions

This chapter presents heterogeneous RAID, where parity is stored in a weighted fashion across any set of heterogeneous devices. The algorithm to place parity allows a small

array of parity locations (its size is the sum of weights) which repeats. The results show some improvement over RAID5 with the same hardware. It is challenging to design experiments that compare heterogeneous RAID without optimizing for the uneven workloads presented in Chapter 5. Because we compared RAIDH to RAID5SS, we were using the optimized hardware layout for RAID5. It might have been more fair to RAIDH to compare to RAID5NSS, but we ultimately chose to use the best non-RAIDH configuration.

Chapter 7

Conclusions

Disk-based storage systems suffer inherent limitations of high latencies, high power requirements, and limited number of power cycles. SSDs offer improvements in each of these areas and thus are a natural candidate for replacing disks. We propose several modifications to basic parity-based RAID that improve results in heterogeneous arrays. Each of these systems was implemented in the Linux software RAID kernel and experiments were run on a single-node system with 5 storage devices. First we presented RAID4S, a RAID storage system which stores data in a hybrid layout with hard drives and a small number of SSDs. Because parity is isolated on SSD, the RAID4S arrangement performs up to 3.3X better than RAID4 at a relatively small increase in system cost. Next we presented an optimization for RAID4S called modthresh, which converts reconstruct-writes into read-modify-writes to better utilize the SSDs. Next we presented a further analysis of the unevenness of small write workloads across arrays,

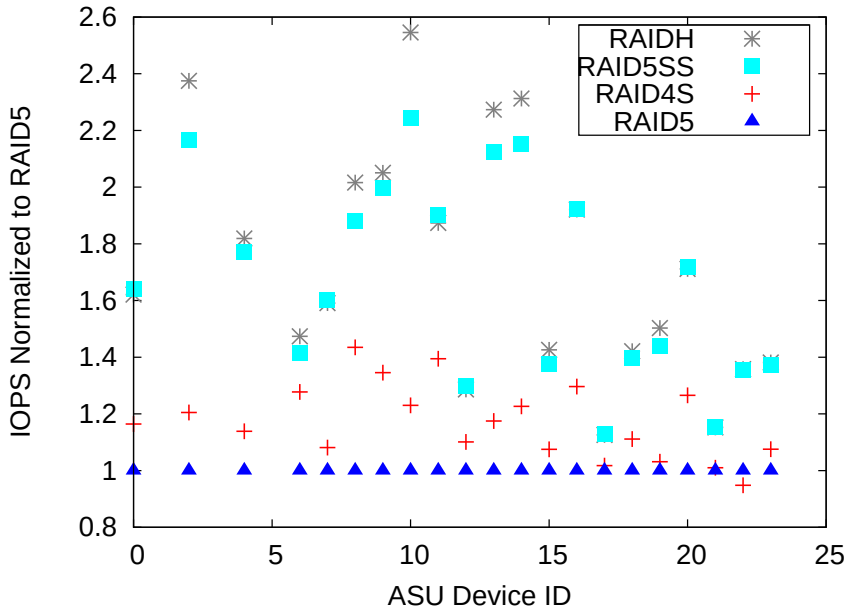

Figure 7.1: RAIDH Financial replay normalized to RAID5 shows a throughput improvement for most workloads.

which motivates RAIDE. In particular, placing faster devices at and near the edges of arrays will improve small write performance in some cases. Lastly, we implemented a fully heterogeneous parity placement design that allows for greater flexibility in parity storage. Based on the performance of each device in an array, parity weights are chosen such that faster devices store more parity. Performance improved over standard RAID5 with the same hardware, and the greater flexibility in configuration may work well for very heterogeneous hardware setups.

The RAID layouts described in this dissertation were chosen to mitigate certain performance issues inherent in basic RAID. Figure 7.1 shows the normalized performance

of replaying the financial trace workload on the best configurations. RAIDH performs up to 2.5X better than RAID5. RAID4S performs best for small write workloads where the writes fit on less than half an array (in the 4+1 arrays of our experiments, writes must fit on a single device). RAID4S-modthresh increases the benefit of RAID4S by changing the threshold for the reconstruct-write optimization to improve writes that fit on half the devices in an array. RAIDE recognized higher read workloads on edge devices in small write workloads. This uneven workload is present when read-modify-writes occur. Lastly, RAIDH enables heterogeneous arrays to perform better than their homogeneous counterparts. Specifically, parity placement concentrated on faster devices improves performance. Even at larger write sizes, RAIDH performs well in any size that is not a multiple of the array width. All these techniques perform best for small write workloads, which includes unaligned large write workloads. Read-only workloads do not benefit from these techniques, but mixed workloads even with a small write percentage show improvements by storing parity on SSDs.

There are several areas to explore for future work. The most obvious is to incorporate more than one parity (RAID6). Two SSDs could be used to store parity by implementing Row-Diagonal Parity in Linux software RAID. The performance improvements over erasure-based RAID6 are expected to be similar to RAID4S. This is a natural extension to RAIDH, as the second parity placement for each stripe would use the same weights.

Another area of future work is a reorganizing layout. This system could detect

changes in the workload and adjust the RAID layout to accommodate and provide better performance. For example, read-only data could be moved to faster devices while data that is frequently updated could be moved so that its parity is on faster devices. It could also detect hardware changes by monitoring device throughput and moving parity to faster devices. The RAID layout could be on a per-stripe basis or with larger chunks depending on performance experiments and the monitoring space and complexity overhead.

Bibliography

[1] http://traces.cs.umass.edu/index.php/Storage/Storage.

[2] Samsung SSD PB22-J. http://www.samsung.com/global/business/semiconductor/products/flash/ssd/2008/down/PB22-J.pdf, May 2009.

[3] Samsung SSD SATA 3.0Gb/s(SLC). http://www.samsung.com/global/business/semiconductor/products/flash/ssd/2008/product/lineup.html, April 2009.

[4] Western Digital WD Caviar Green WD20EADS. http://www.wdc.com/en/products/products.asp?driveid=576, April 2009.

[5] Triton storage backup policy. http://tritonresource.sdsc.edu/storage_backup.php, June 2012.

[6] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[7] Ahmed Amer, Jehan-François Pâris, Thomas Schwarz, Vincent Ciotola, and James Larkby-Lahet. Outshining mirrors: MTTDL of fixed-order SSPiRAL layouts. In *Fourth International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2007)*, 2007.

[8] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD reliability. In *Fifth ACM European Conference on Computer Systems*, 2010.

[9] Brett Battles, Cathy Belleville, Susan Grabau, and Judith Maurier. Reducing data center power consumption through efficient storage. NetApp white paper WP-7010-0207, February 2007.

[10] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *SIGOPS Operating Systems Review*, 41(2):88–93, April 2007.

[11] Timothy Bisson. *Improving Hard Disk Power Consumption, Performance, and Reliability through I/O Redirection*. PhD thesis, University of California, Santa Cruz, June 2007.

[12] Dina Bitton and Jim Gray. Disk shadowing. In *Proceedings of the 14th Conference on Very Large Databases (VLDB)*, pages 331–338, 1988.

[13] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai 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.

[14] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, February 1995.

[15] Mario Blaum, Jehoshua Bruck, and Alexander Vardy. Binary codes with large symbols. In *Proceedings of the 1994 IEEE International Symposium on Information Theory*, 1994.

[16] A. Brinkmann, S. Effert, F. Meyer auf der Heide, and C. Scheideler. Dynamic and redundant data placement. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, 2007.

[17] André Brinkmann, Michael Heidebuer, Friedhelm Meyer auf der Heide, Ulrich Rückert, Kay Salzwedel, and Mario Vodisek. V:Drive costs and benefits of an out-of-band storage virtualization system. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2004.

[18] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2000.

[19] André Brinkmann, Kay Salzwedel, and Mario Vodisek. A case for virtualized arrays of RAID. In *International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI '04)*, 2004.

[20] John A. Chandy and Sumit Narayan. Reliability tradeoffs in personal storage systems. *SIGOPS Operating Systems Review*, 41(1):37–41, January 2007.

[21] 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.

[22] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD Record*, 39:5–10, February 2011.

[23] Don Clark. Memory-chip alternative is unveiled. Wall Street Journal, May 19, 2009.

[24] Jimmy Clidaras, David W. Stiver, and William Hamburgen. Water-based data center. United States Patent 7,525,207, April 2009.

[25] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*, 2002.

[26] 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 Third USENIX Conference on File and Storage Technologies*, 2004.

[27] T. Cortes and J. Labarta. Taking advantage of heterogeneity in disk arrays. *Journal of Parallel and Distributed Computing*, 63(4):448–464, 2003.

[28] Adam Crume. Personal communication, 2012.

[29] Richard Freitas, Winfried Wilcke, and Bülent Kurdi. Storage class memory, technology, and use. Tutorial Slides, presented at *6th USENIX Conference on File and Storage Technologies*, 2008.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[31] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disc arrays: Low-cost reliable storage with acceptable throughput. In *Proceedings of the 16th VLDB Conference*, 1990.

[32] Jim Gray and Catharine van Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft Research, December 2005.

[33] Kevin M. Greenan, Ethan L. Miller, and Thomas J. E. Schwarz, S.J. Optimizing Galois field arithmetic for diverse processor architectures and applications. In *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, September 2008.

[34] Aloke Guha. Solving the energy crisis in the data center using COPAN systems' enhanced MAID storage platform. Copan Systems white paper, December 2006.

[35] James Lee Hafner. WEAVER codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[36] Lisa Hellerstein, Garth A. Gibson, Richard M. Karp, Randy H. Katz, and David A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12:182–208, 1994.

[37] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

[38] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, 1992.

[39] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Journal of Distributed and Parallel Databases*, 2(3):295–335, July 1994.

[40] Bo Hong, Feng Wang, Scott A. Brandt, Darrell D. E. Long, and S. J. Thomas J. E. Schwarz. Using MEMS-based storage in computer systems—MEMS storage architectures. *ACM Transactions on Storage*, 2(1):1–21, February 2006.

[41] R. J. Honicky and Ethan L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)"*, 2004.

[42] Soojun Im and Dongkun Shin. Delayed partial parity scheme for reliable and high-performance flash memory ssd. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[43] Soojun Im and Dongkun Shin. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *Computers, IEEE Transactions on*, 60(1):80–92, January 2011.

[44] Jeff Janukowicz and Dave Reinsel. Evaluating the SSD total cost of ownership. IDC white paper, November 2007.

[45] Hyojun Kim and Seongjun Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. *6th USENIX Conference on File and Storage Technologies*, February 2008.

[46] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. HybridStore: A cost-efficient, high-performance storage system combining SSDs and HDDs. In *2011 IEEE 19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 227–236, July 2011.

[47] Mingqiang Li, Jiwu Shu, and Weimin Zheng. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Transactions on Storage (TOS)*, 4(4), January 2009.

[48] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *MSST/SNAPI 2012*, Pacific Grove, CA, April 2012.

[49] Diego R. Llanos. TPCC-UVa: an open-source TPC-C implementation for global performance measurement of computer systems. *ACM SIGMOD Record*, 35:6–15, December 2006.

[50] Peter Lyman, Hal R. Varian, Peter Charles, Nathan Good, Laheem Lamar Jordan, and Joyojeet Pal. How much information? 2003. `http://www.sims.berkeley.edu/research/projects/how-much-info-2003/`.

[51] Bo Mao, Hong Jiang, Dan Feng, Suzhen Wu, Jianxi Chen, Lingfang Zeng, and Lei Tian. HPDA: A hybrid parity-based disk array for enhanced performance and reliability. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[52] Jai Menon and Dick Mattson. Comparison of sparing alternatives for disk arrays. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 318–329, 1992.

[53] Ross Miller. Samsung's PRAM chips hit mass production in June. http://www.engadget.com/2009/05/05/samsungs-pram-chips-go-into-mass-production-in-june/, May 5, 2009.

[54] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. *Proceedings of the fourth ACM European conference on Computer systems*, pages 145–158, 2009.

[55] Device prices and specifications. http://www.newegg.com, August 2011.

[56] Spencer W. Ng and Richard L. Mattson. Maintaining good performance in disk arrays during failure via uniform parity group distribution. In *Proceedings of the First International Symposium on High-Performance Distributed Computing (HPDC-1)*, 1992.

[57] Kwanghee Park, Dong-Hwan Lee, Youngjoo Woo, Geunhyung Lee, Ju-Hong Lee, and Deok-Hwan Kim. Reliability and performance enhancement technique for SSD array storage system using RAID mechanism. In *9th International Symposium on Communications and Information Technology, 2009. ISCIT 2009*, pages 140–145, September 2009.

[58] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). 17:109–116, June 1988.

[59] Eduardo Pinheiro, Ricardo Bianchini, and Cezary Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *Proceedings of the Joint International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, June 2006.

[60] 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 (FAST)*, 2007.

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

[62] James S. Plank. The RAID-6 liberation codes. In *6th USENIX Conference on File and Storage Technologies*, San Jose, CA, February 2008.

[63] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: Coordinated multi-level power management for the data center. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 48–59, Seattle, WA, March 2008.

[64] A. L. Narashimha Reddy and P. Banerjee. Gracefully degradable disk arrays. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS '91)*, 1991.

[65] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems*, volume 10, pages 26–52, February 1992.

[66] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, March 1994.

[67] Thomas M. Ruwart. *Xdd User's Guide*. I/O Performance, Inc., 6.5.091706 edition, December 2005.

[68] Jiri Schindler, Sandip Shete, and Keith A. Smith. Improving throughput for small disk requests with proximal I/O. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[69] Patrick Schmid and Achim Roos. New desktop hard drives: Speed or capacity? `http://www.tomshardware.com/reviews/2tb-hdd-caviar,2261.html`, April 2009.

[70] 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 (FAST)*, 2007.

[71] Thomas J. E. Schwarz, S.J., Qin Xin, Ethan L. Miller, Darrell D. E. Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *Proceedings of The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, 2004.

[72] Michael Sevilla, Rosie Wacha, and Scott A. Brandt. RAID4S-modthresh: Modifying the write selection algorithm to classify medium-writes as small-writes. Technical Report UCSC-SOE-12-10, University of California, Santa Cruz, 2012.

[73] William Slawski. Search on the seas: Google water-based data center patent granted. `http://www.seobythesea.com/?p=1357`, April 2009.

[74] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging: Overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 64–75, 1993.

[75] Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth A. Gibson. A redundant disk array architecture for efficient small writes. Technical report, Carnegie Mellon University, 1993.

[76] Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth A. Gibson. Parity-logging disk arrays. *ACM Transactions on Computer Systems*, 12(3):206–235, 1994.

[77] Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth A. Gibson. A redundant disk array architecture for efficient small writes. Technical Report CMU-CS-94-170, Carnegie Mellon University, 1994.

[78] Mark W. Storer, Kevin M. Greenan, and Ethan L. Miller. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *6th USENIX Conference on File and Storage Technologies*, San Jose, CA, February 2008.

[79] Alexander Thomasian and Jai Menon. RAID5 performance with distributed sparing. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):640–657, 1997.

[80] Lei Tian, Dan Feng, Hong Jiang, Ke Zhou, Lingfang Zeng, Jianxi Chen, Zhikun Wang, and Zhenlei Song. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, February 2007.

[81] Mustafa Uysal, Arif Merchant, and Guillermo A. Alvarez. Using MEMS-based storage in disk arrays. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, April 2003.

[82] Nitin H. Vaidya. A case for two-level distributed recovery schemes. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):64–73, May 1995.

[83] Rosie Wacha, Scott A. Brandt, John Bent, and Carlos Maltzahn. RAID4S: Adding SSDs to RAID arrays. Poster at 5th European Conference on Computer Systems (EuroSys 2010), 2010.

[84] Charles Weddle, Mathew Oldham, Jin Qian, An-I Andy Wang, Peter Reiher, and Geoff Kuenning. PARAID: A gear-shifting power-aware RAID. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, February 2007.

[85] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A reliable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[86] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC '06)*, 2006.

[87] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *6th USENIX Conference on File and Storage Technologies*, San Jose, CA, February 2008.

[88] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 1996.

[89] Suzen Wu, Hong Jiang, Dan Feng, Lei Tian, and Bo Mao. WorkOut: I/O workload outsourcing for boosting RAID reconstruction performance. In *FAST '09: 7th USENIX Conference on File and Storage Technologies*, pages 239–252, 2009.

[90] Intel X25-E SATA solid state drive datasheet. `http://download.intel.com/design/flash/nand/extreme/319984.pdf`, May 2009.

[91] Qin Xin, Ethan L. Miller, and Thomas J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 172–181, Washington, DC, USA, 2004. IEEE Computer Society.

[92] Qin Xin, Ethan L. Miller, S.J. Thomas Schwarz, Darrell D. E. Long, Scott A. Brandt, and Witold Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[93] Lihao Xu and Jehoshua Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, January 1999.

[94] Mike Yan and Jon Ehlen. Open vault storage hardware v0.5, May 2012.

[95] Roger Zimmerman and Shahram Ghandeharizadeh. HERA: Heterogeneous extension of RAID. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, 2000.