ACCURATE HARDWARE RAID SIMULATOR

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Darrin Kalung Weng

June 2013

COMMITTEE MEMBERSHIP

TITLE:	Accurate Hardware RAID Simulator
AUTHOR:	Darrin Kalung Weng
DATE SUBMITTED:	June 2013
COMMITTEE CHAIR:	Chris Lupo, Ph.D. Assistant Professor Computer Science Department
COMMITTEE MEMBER:	John Oliver, Ph.D. Assistant Professor Electrical Engineering Department
COMMITTEE MEMBER:	Phillip Nico, Ph.D. Assistant Professor Computer Science Department

Abstract

Accurate Hardware RAID Simulator

Darrin Kalung Weng

Computer data storage is growing at an astonishing rate. With cloud computing and the growth of the Internet enterprise storage has been predicted to grow at rates as high as 300% per year. To fulfill this need technologies such as Redundant Array of Independent Disks or RAID are being used in industry today. Not only does RAID increase I/O performance but also provides redundancy measures to protect against hardware failure. Even though RAID has existed for some time now and is well understood, proprietary optimizations such as command scheduling and cache strategies that are employed by current RAID controllers are not well known. This thesis presents a model for RAID 5 that incorporates these features and describes the overall function of hardware RAID controllers. Also a python implementation of this model, Accurate Hardware RAID Simulator (AHRS) is presented and validated against a current hardware RAID controller. It is shown that AHRS can reproduce the behavior of a hardware RAID system with an accuracy of 97.92% on average compared to a LSI hardware RAID controller.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data storage has been a fundamental problem for computers since their creation. Many different technologies and techniques have been employed to allow a computer to accomplish the simple task of storing away information and then recalling it at a later time. Without being able to do this computers would not be the useful tools that they have become today. Not only does data need to be available but for most systems overall performance is greatly impacted by how quickly data can be accessed by the storage device. To serve both of these needs RAID or Redundant Array of Independent Disks technologies are commonly deployed in computer systems.

The growth of enterprise storage is predicted to be between 30% and 40% per year or even as high as 300% from certain studies [18]. A large portion of this growth comes from the emerging cloud storage market. RAID has become the industry standard to provide systems with fast and reliable storage. With this explosive growth RAID systems have become even more important to enterprise storage strategies.

RAID relies on using multiple storage devices together in parallel to increase performance and provide redundancy. By combining the efforts of multiple hard drives many of the data heavy applications of today are possible. These systems contain complex interactions which must be well understood so that tuning of these systems can be most effective. With the proprietary algorithms that are used by modern RAID controllers it is not very well known how individual drives are accessed within the array. One configuration in the RAID standard, RAID 5, is commonly used throughout enterprise storage. A model is needed to provide an understanding of how RAID 5 behaves in real systems and an easy way to experiment on such systems. Many existing models of RAID 5 are statistical in nature or only view the storage system from a high level. The approach used in this thesis is from the bottom up and seeks to build understanding from observations of real live systems instead of attempting to approximate behavior.

1.1 Contributions

There are three major contributions made by the work presented by this thesis. First is the RAID 5 Spatial model which accurately describes where data and parity are located within an array. Next is the RAID 5 performance model which describes the actions taken by a hardware RAID controller to access data on the array. Also this model describes the effects and behaviors of different parameters made available to these types of controllers. Lastly Accurate Hardware RAID Simulator (AHRS) is a python implementation both of the above two models is presented and validated against a current hardware RAID 5 system.

1.2 Outline

In Chapter 2 supporting information on how hard disk drives and RAID work is given. Next in Chapter 3, two models describing the behavior of a hardware RAID controller are presented. Also in Chapter 3, a description of the python implementation of these models, AHRS is given. In Chapter 4, AHRS is evaluated against a real hardware RAID controller and the results of this evaluation are discussed. Lastly in Chapter 6, a conclusion and summary of this thesis is presented.

Chapter 2

Background

2.1 Hard Disk Drives

Hard disk drives have been the dominant storage medium for computers since the 1960s. These devices store data on rotating platters of ferromagnetic material in a manner much like a vinyl record player. Instead of a needle feeling the surface of a record track spinning around, a hard drive uses a magnetic head to detect changes in magnetic direction. In a hard drive the disk that is spun is called a platter. These are spun under to allow the head to read a concentric track of magnetic transitions which are translated into binary data. Unlike a vinyl record these magnetic patterns can be modified on the fly by the head allowing data to be written and rewritten to the media. Multiple platters and heads can be combined into a single drive to increase the total capacity of the whole unit.

Modern hard drives are block devices from the operating system's point of view. This means they are only block addressable and cannot access single bytes at a time. The minimum accessible block size for today's hard drives is 512

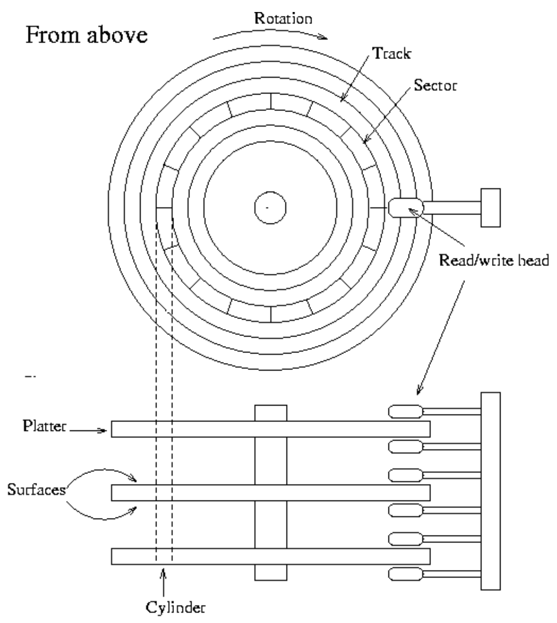


Figure 2.1: Head, Cylinder, Sector Addressing [1]

bytes. Originally these blocks were addressed by their actual location on the disk specified by the following three parameters; head, cylinder, and sector. A cylinder specifies how far from the center of the platter the data is located and spans vertically through all the platters. Next the sector is combined with the cylinder to form an arc containing the data. Lastly the head number specifies the surface the data exists on. Tracks closer to the outside diameter of the platter have space for more sectors than those located near the inside. This fact is used to improve density of hard drives but increases the difficulty of addressing using the original head, cylinder, sector scheme. Therefore this method of addressing was dropped in favor of Logical Block Addressing. This is where the operating system views the drive as a sequential volume of blocks addressed by Logical Block Addresses or LBAs. A hard drive takes a LBA and translates it into the actual location on the drive. This translation can be different for each hard drive manufacturer, abstracting the format of the drive from the host system. Even so by convention it can be assumed that the addressing starts at the outer perimeter where the heads first hit the disk.

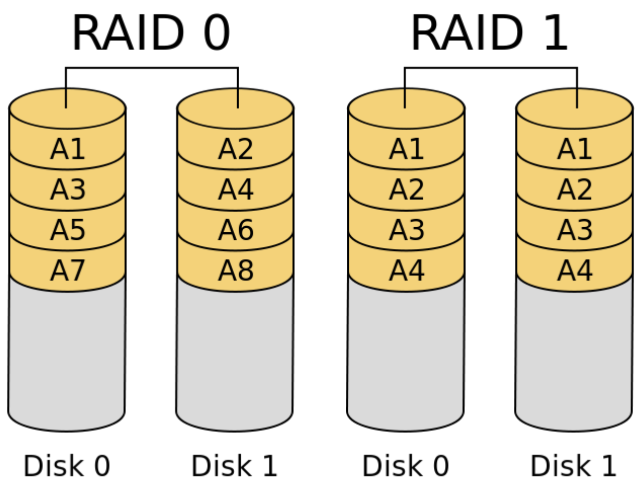
The performance of modern hard drives is limited by seek times and the speed which data can travel under the heads. Any time data is accessed on a drive the head must be positioned on the correct track. This is a mechanical process which on average takes 9 ms for most consumer products [20] and 4 ms for high performance enterprise level drives [21]. Even after seeking the head to the correct position the drive may also have to wait for the requested data to spin around the platter to come under the head. This is known as rotational latency. Once the requested data is actually under the head the speed that data can be accessed is determined by how fast the disk is spinning and how densely data is packed on the platter. Most consumer drives spin at either 5400 or 7200 RPM, while higher end enterprise drives operate at much greater speeds at 10,000 or 15,000 RPM. Bit densities have progressively increased at a steady rate each year. The sustained sequential performance difference between consumer and enterprise drives is about 154 MB/s to 204 MB/s [20, 21]. The less seeks required the better performance a hard drive will have. Hard drives perform best with very sequential workloads where the heads do not need to seek much [17]. Random workloads introduce a large amount of seeks and the drive spends most of its time moving the head instead of actually accessing data.

2.2 RAID

RAID which stands for Redundant Array of Independent Disks [16] is a technology used in many of today's computer storage solutions. The main weakness of hard drives is that it takes a significant amount of time (in the millisecond range) for the head to seek to where data is located on the disk. Also a major concern is how to recover data if a hard drive fails. To combat these threats multiple disks were combined into a single volume so that many drives may work in parallel to provide greater performance and/or redundancy [8].

There are a few important terms to define when describing RAID systems. Figure 2.2 shows two basic RAID configurations which will be explained later. A *stripe* is a sequential section of user data that is stretched across all the drives in an array. A *chunk* is the division of a stripe that resides on a single drive. These also are called a strips by some but for clarity this work will always refer to these as chunks. In Figure 2.2, a chunk is each section prefixed with an A and a stripe is all the sections at the same level. For example A1 and A2 are the chunks that make up the first stripe in the RAID 0 configuration in Figure 2.2. Every stripe is made of evenly sized chunks and a RAID volume is made up of many evenly sized stripes. The sizes of these data divisions combined with the number of drives in the array affects the overall performance of a RAID volume [2]. The host is the system which the RAID array is connected to and views the array as a continuous storage volume. When a drive fails in a RAID array, the array enters what is called a degraded state. In this state the redundancy measures must be employed to recover the lost data and restore the array before returning back to an operational or optimal state.

A RAID array can be controlled by either hardware or software. A hardware RAID controller uses separate hardware to manage and issue the correct commands to the drives of the array. This offloads the burden of managing the array from the CPU of the system and increases performance. Most hardware controllers also contain their own memory and caches allowing even larger increases in performance. Another benefit to a hardware controller is that the drives and



(a) RAID 0 (b) RAID 1

Figure 2.2: RAID Levels 0 and 1 [7]

the controller don't have to be directly connected to the host and can be located in a separate enclosure connected via SAS, Fiberwire, etc. Software RAID setups are controlled by software running on the host CPU. In this case no special hardware is used to connect the drives or to assist with any of the calculations required to operate. This software can reside as a part of the operating system or can be a feature of a filesystem. Linux's mdadm and LVM (Logical Volume Management) are examples of the first and filesystems like ZFS and BTRFS support RAID volumes. These types of setups tend to be simpler to setup, but incur a CPU overhead cost. This overhead increases when an array needs to be rebuilt or recovered. Also because the software needs to be running before allowing access to the array many Software RAID implementations do not support booting or can have difficulty doing so.

Because there are many ways to organize multiple disks into a single volume standard RAID levels are defined. RAID 0 is the first level to be defined and is the most basic. Figure 2.2 shows how data is organized. This level stripes data across every drive in the array. As shown in Figure 2.2, this means the first chunk of data is on Drive 0, and then Drive 1 and so on. After reaching the last drive in the array the data wraps back around to Drive 0. All the drives in the array are assumed to have the same capacity but it is possible to form a RAID 0 array with drives of different sizes. In this case the usable size of each drive is clamped to the size of smallest added to the array providing the illusion of having all the same drives. The goal of RAID 0 is to increase performance and but provides no redundancy. If a drive fails all the data contained by the array is lost and cannot be recovered. Assuming that a host system wants to read large amounts of data quickly, RAID 0 allows multiple drives to access data at the same time, increasing throughput. Assuming the RAID controller can keep up and a large sequential request is issued, the maximum throughput is the maximum individual drive throughput multiplied by the number of drives in the array or $(MAXDriveThroughput * NumberofDrives)$. Another benefit is that both drives can access different LBAs simultaneously and serve multiple requests at once. Unfortunately this is only possible if the requests are smaller than the chunk size, otherwise data from multiple drives is required and must be synchronized.

RAID 1 is a simple scheme where two drives are mirrors of each other and is shown in Figure 2.2. This means that both disks are exact copies of each other and extra commands must be issued to maintain this state. Write requests from the host will cause both drives to write simultaneously to maintain the mirror image of the volume. Read requests will only be sent to a single drive in the array as there is no point in sending multiple copies of the same data back to the host. Opposite of RAID 0, the goal of RAID 1 is to purely provide data redundancy and offers no performance benefit. If a single drive fails no data is lost and is

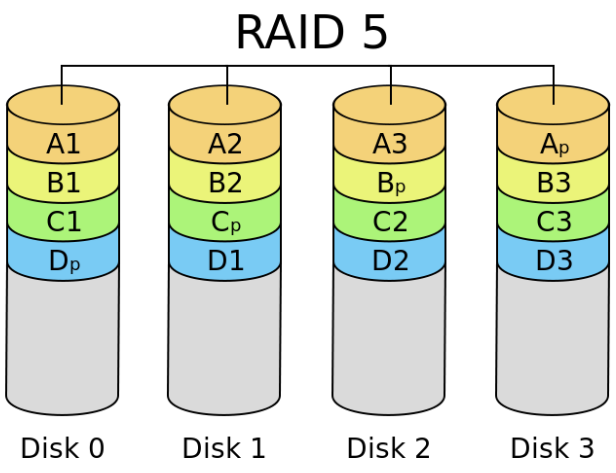
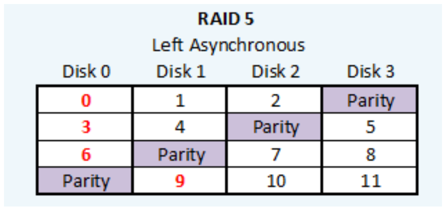


Figure 2.3: RAID Level 5 [7]

still accessible. This forces the array into what is called a degraded state which needs to be rebuilt to regain its redundancy properties. Once the damaged drive is replaced then all the data present on the good drive must be copied to the replacement drive. Once this is completed the array can be brought out of the degraded state and be operational again. Even though the array may be in a degraded state, data can still be available to the host but with lower performance as a portion of the system's resources must be spent on rebuilding the array.

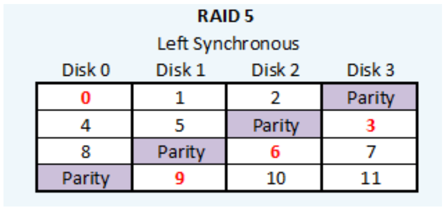
It is especially important in server environments to have both high performance and redundant storage. Both RAID 0 and 1 can only provide either one or the other but not both at the same time. RAID 5 combines data striping with distributed parity to fill this need and is shown in Figure 2.3. Similar to RAID 0 data is divided into stripes across the array with a chunk of data allocated sequentially across each drive in the array. The difference is for every stripe a single chunk is used to store parity instead of data. This chunk stores the result from XORing the data from all the the other chunks in the stripe. If a drive fails its data can be recreated from the remaining chunks and the parity chunk by a simple XOR. Every time data is written to the volume the corresponding parity must be updated to keep the array's redundancy property. This causes writes to follow a read-modify-write pattern introducing a significant overhead. To prevent parity access from becoming a bottleneck it is distributed across all the drives in the array. The available storage for a RAID 5 array is $(N - 1) * DriveSize$ as one drive worth of space is used up to store parity data.

There are multiple ways data and parity can be ordered in a RAID 5 array. Commonly the parity chunk is rotated around each drive to ensure an even distribution. For example for the first stripe the parity chunk will be on the first drive, second for the next and so on until it wraps around again to the first drive. This rotation is known as Right as the first parity chunk starts on the rightmost drive, if you reverse this order you get Left. Now data can either be laid out in a synchronous or asynchronous fashion. Synchronous means that each successive data chunk is sequentially numbered, always starting with the first chunk after parity. Asynchronous data chunks always start with the first drive in the array or the second if it taken up by parity data. It has been proven that a synchronous layout has slightly better performance than the alternatives experimentally by [13]. The model produced in this thesis uses a Left Synchronous layout that is common to most configurations.


RAID 5
Left Asynchronous

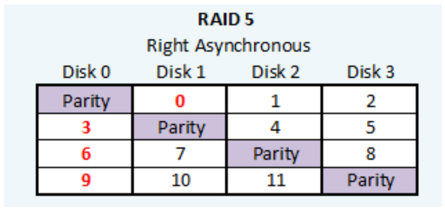
Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	Parity
3	4	Parity	5
6	Parity	7	8
Parity	9	10	11

(a) Left Asynchronous


RAID 5
Left Synchronous

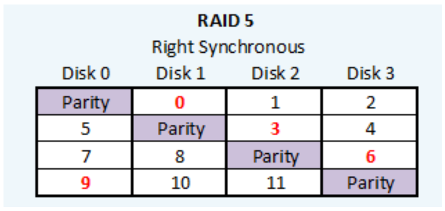
Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	Parity
4	5	Parity	3
8	Parity	6	7
Parity	9	10	11

(b) Left Synchronous


RAID 5
Right Asynchronous

Disk 0	Disk 1	Disk 2	Disk 3
Parity	0	1	2
3	Parity	4	5
6	7	Parity	8
9	10	11	Parity

(c) Right Asynchronous


RAID 5
Right Synchronous

Disk 0	Disk 1	Disk 2	Disk 3
Parity	0	1	2
5	Parity	3	4
7	8	Parity	6
9	10	11	Parity

(d) Right Synchronous

Figure 2.4: RAID 5 Layouts [3]

Chapter 3

Hardware RAID 5 Models

The two models that form the core of this work are the RAID 5 spatial and performance models. The goal of these models is to closely emulate a real RAID controller used in industry today and expose the behavior of such hardware. Each one of these models represents a different aspect of a RAID 5 system and individually do not describe how this type of system works. The spatial model provides a set of equations that describe the location of data and parity on the drives of the array. While this is useful, the spatial model does not give any information on how these locations are accessed by the RAID controller. The performance model describes how data is accessed by the controller and what is the expected performance of these actions. The performance model uses the locations provided by the spatial model to generate the commands that are issued to the drives. The LBAs of these commands are critical so that an accurate model of individual drive performance can be formed and therefore the overall system performance can also be estimated.

3.1 Hardware RAID 5 Spatial Model

The spatial RAID 5 model is the first model that will be described in this thesis. This model describes the exact locations where data and parity reside within the RAID array for the LSI 9260-8i. This RAID controller is an entry level enterprise controller with 512 MB of memory that is connected to the host by PCI express 2.0. Even though this model is simple, it is necessary as a foundation to support the performance model that will be described in 3.2. As noted before this model only supports a left-symmetric parity layout.

Three equations are combined to form this model. All of these take in the following parameters, host LBA, chunk size, and Number of Drives. The host LBA is the address into the RAID volume from the host system. Next, the chunk size and Number of Drives (N) are parameters for the array to be modeled. For example, a chunk size of 128 KB and 6 drives could be used. For most RAID controllers the chunk size is a range from 8KB to 1MB but this model supports any desired chunk size greater than a single sector or 512 bytes. Least 3 drives must be used. This is not a limitation of the model but a part of the RAID 5 specification. The outputs from this model are tuples of the form $(DriveLBA, Drive)$ signifying the LBA and drive of the desired data or parity. The units used by these three equations are in blocks, the minimum addressable space for this type of storage.

$$\begin{aligned} DriveLBA = & \left\lfloor \frac{HostLBA}{ChunkSize \times (N-1)} \right\rfloor \times ChunkSize \\ & + HostLBA \bmod ChunkSize \end{aligned} \tag{3.1}$$

$$DriveNumber = \frac{HostLBA}{ChunkSize} \bmod N \tag{3.2}$$

$$ParityDrive = N - 1 - \left[\left(\frac{HostLBA}{ChunkSize \times (N-1)} \right) \bmod N \right] \tag{3.3}$$

Equation 3.1 determines the drive LBA for a host LBA. It divides the host address by the amount of data per stripe, the Chunk size multiplied by the number of drives minus one. This number is then rounded down to the lowest integer which is the stripe number in which the request exists on. Since stripes are ordered sequentially in the array, it is the same if only considering a single drive as well. This information combined with the chunksize allows the calculation of the base address of the chunk on the drive. To determine the offset into this chunk the modulus of the host LBA by the Chunk size is taken and added to the base address.

Equation 3.1 alone does not give the location of data in the array. What is missing is which drive in the array that the data is located on. Equation 3.2 is a simple equation where the host LBA is divided by the Chunk size and then the modulus with respect to number of drives is taken of this result. This gives an index number of the drive where the specified host LBA exists. By using the two equations described above a transformation from host LBA into the tuple $(DriveLBA, DriveNumber)$ is modeled for accessible data.

While the above two equations will give the exact location for data located on the array, locations of parity have not yet been covered. Equation 3.3 determines the drive in which parity resides for a specified host LBA. Again this equation first determines the stripe number for which the host LBA exists on the same way as Equation 3.1. The modulus of the stripe number and the number of drives is subtracted from the number of drives minus one to get the result. Each LBA in the parity chunk is responsible for the same LBA for all the data chunks in

Parameter Name	Purpose
Chunk Size	Size of each Chunk in LBAs
N	Number of Drives in the Array
Host LBA	The address from the host perspective

Table 3.1: List of Parameters for the Spatial Model

the same stripe. Since this is the case Equation 3.1 may be reused to give the drive LBA of the parity. Equation 3.3 is combined with Equation 3.1 to form $(DriveLBA, ParityDriveNumber)$.

3.2 Hardware RAID 5 Performance Model

While the spatial model presented in Section 3.1 gives the data and parity layout of the LSI 9260 when managing a RAID 5 array, how these locations are accessed by the controller needs to be modeled. The performance model fills this gap and simulates the actions the controller takes when a host system issues commands to it. By using information provided by the spatial model and the performance model, an accurate model can be formed.

3.2.1 Overview

A few assumptions have been made to make the performance model more manageable. First all requests made by the host must be aligned to chunk boundaries. The performance difference between chunk aligned access and non-aligned can be emulated by issuing multiple requests to access adjacent chunks. Next the size of all requests must be a multiple of the chunk size. When writing a request smaller than the chunk size calculations must be done to determine what areas

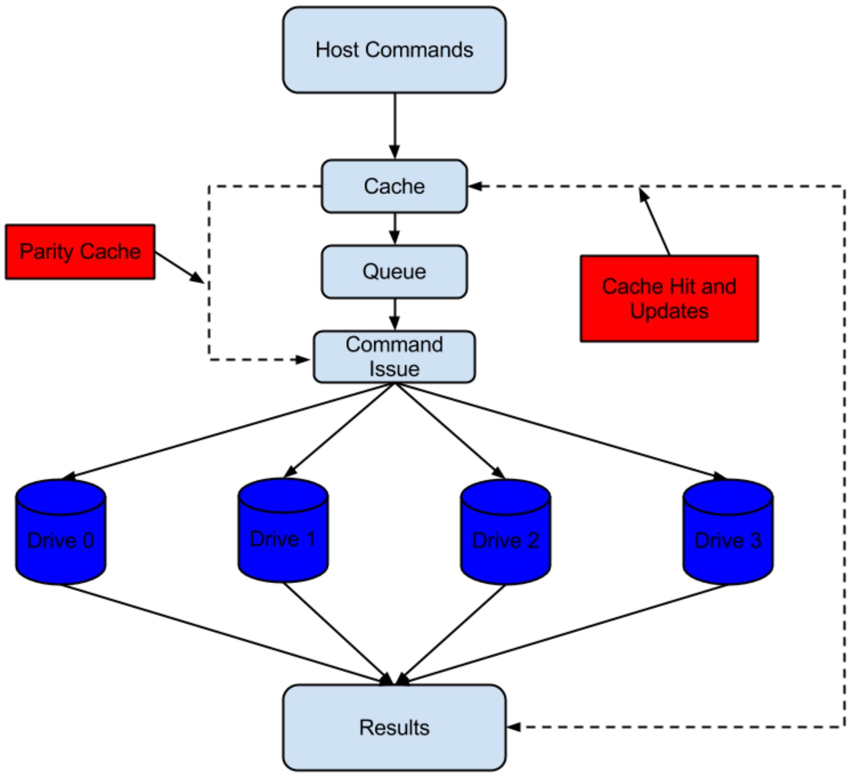


Figure 3.1: Block Diagram of the Performance Model

must be accessed to calculate parity, which adds unnecessary complexity. These two assumptions make the performance model simpler without sacrificing model accuracy. Lastly it is assumed the hard drives in this model do not have command queues and only process a single command at a time. Even though modern hard drives have this feature, modeling queuing behavior is very complicated and the overall behavior of the RAID controller is not affected by this simplification.

The input to the performance model is a list of requests. These requests have the following three fields, type, LBA, and length. Type can either be read or write, LBA is the host address, and length is how many 512 byte blocks are requested. All the parameters match the information that a host system would send to the RAID controller. This list of commands is what makes up a workload

Host LBA	Length	Type (Read/Write)

Table 3.2: Format of a Host I/O Request

Parameter Name	Purpose
Chunk Size	Size of each Chunk in LBAs
N	Number of Drives in the Array
Cache Mode	Which Cache strategy is selected (Direct/Cached)
Write Cache Policy	How the write cache behaves (Write Through/Write Back)
Cache Size	How many entries can the cache hold at once
Disk Access Rate	How fast can data be accessed from a disk
Min Seek Time	Amount of time to seek one track
Max Seek Time Rate	How long it takes to seek from OD to ID
Avg Seek Time Rate	Average seek time for the drives

Table 3.3: List of Parameters for the Performance Model

and allows the model to support any type of input that could be found in a real system.

Similar to the spatial model, the performance model requires a set of parameters. These parameters need to be defined before any input is introduced to the model and must stay consistent throughout the processing of an input workload. The parameters for this model will be explained within the description of the stage in which they take effect.

The performance model is made up of multiple stages which process a part of the model and then pass results to the next stage. The stages in this model are Cache, Command Queue, Command Issue, Drives, and Results. As shown by Figure 3.1 each stage feeds into another to produce the output. Each one of these stages can be individually modified without affecting the others. This allows the modeling of different settings on the LSI 9260 such as different caching strategies or queue behaviors. Also this supports a modular system where these stages can be replaced or revised if new techniques are discovered.

3.2.2 Cache Stage

Caching is an important performance feature of hardware RAID cards and its behavior is modeled in the first stage. The caching stage maintains a list of all the currently cached entries. An entry is represented by the host LBA of the corresponding chunk in the array and a timestamp to mark the entry's arrival. It is considered a cache hit when all the chunks in a read request are present in the list. This stage takes the input and determines which requests are cache hits and routes them straight to the results stage. If the request is a cache miss, it is passed to the command queue stage instead. It is possible for a read request where only a few of the desired chunks are in the cache while the rest are not. When the cache list becomes full and a new entry is to be added the entry with the oldest timestamp will be replaced. This follows a Least Recently Used or LRU cache replacement algorithm. This is a widely used algorithm and performs well for storage accesses. The size of the cache is an important parameter that must be defined and limits the maximum number of entries in the cache list. The cache can be defined by an actual size instead of a number of entries by dividing the desired cache size in bytes by the chunk size.

The cache stage has a parameter that can be either one of two modes, direct or cached. As quoted from the LSI 9260 manual for direct Mode,

> "All read data is transferred directly to host memory bypassing RAID controller cache. All write data is transferred directly from host memory bypassing RAID controller cache"

[15]. In direct mode the cache is always bypassed for host data and only used internally to reduce parity reads. A block diagram showing how requests pass through the cache stage in direct mode is shown in Figure 3.2. Data is still cached to assist with write operations but is not filled by host read requests. In direct

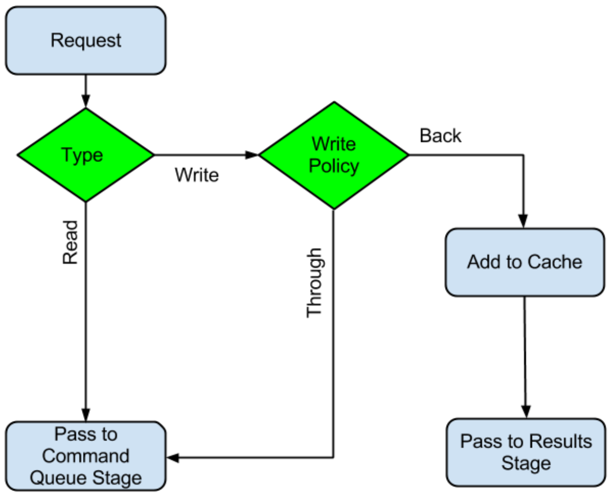


Figure 3.2: Block Diagram of Direct Mode

mode the cache list is not used. Instead a list named the parity cache is made in the command issue stage to hold entries to assist with parity calculations. How this list improves write requests will be described in Section 3.2.4. The parity cache has the same format and size as the cache list and also uses LRU to replace entries. Insertion into the parity cache is done by the results stage. When a write request is completed all the chunks of the accessed stripes will be added. Since in this mode a host request cannot trigger a cache hit, there is only a small performance increase for write heavy workloads. This means that no requests will ever be directly forwarded to the results stage.

Cached mode operates as a traditional cache where data will be added to it when requests pass through. Both read and write requests contribute to the cache list. Figure 3.3 outlines the operation of cached mode. If there is a cache hit for a read request the timestamp of those entries will be updated, and the request passed to the results stage. Otherwise the request is passed on to the command queue stage. Write requests are first added to the cache and then are

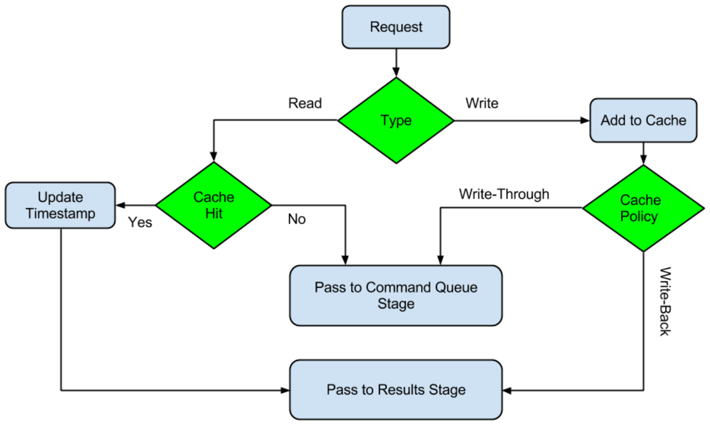


Figure 3.3: Block Diagram of Cached Mode

routed depending on the selected write cache policy. The write policy can be either Write-Through or Write-Back and are settings available on the LSI 9260. These cache policies operate as normally defined, write requests will leave their data in the cache on their way to the disk for Write-Through and Write-Back will store the data in the cache before flushing them to disk. This is modeled by passing the Write-Through requests to the command queue stage while the Write-Back are sent to the results stage.

In cached mode, insertion into the cache list behaves in the following way. The result stage will notify the cache stage when read requests complete. When a request is received by this stage the chunks that belong to it will be inserted into the cache list. As stated before, LRU will be employed to replace any old entries. All write requests add their chunks to the cache list while passing through the cache stage. This means that chunks that have been recently written will provide a cache hit for following reads, if not evicted.

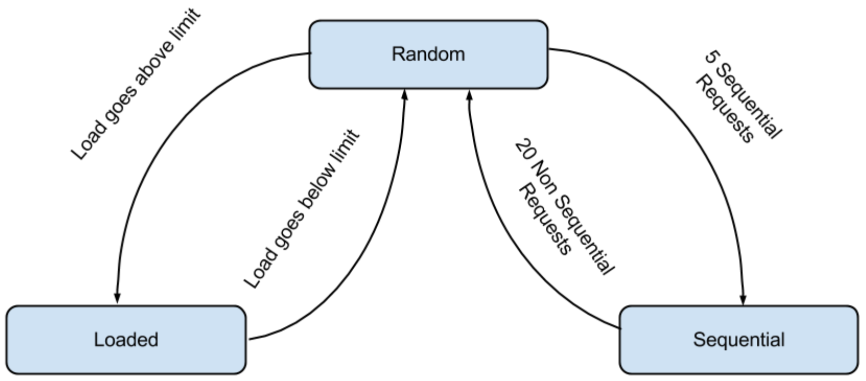


Figure 3.4: Block Diagram of the Command Queue Stage

3.2.3 Command Queue Stage

The command queue stage is meant to emulate the queuing and scheduling algorithms of the RAID controller. This stage takes I/O requests passed from the cache stage and schedules the commands and pushes them to the command issue stage. The algorithms used by this stage are much more simple than what is actually used by the LSI 9260 but approximates its behavior. There are three different states the queue may reside in, Passthrough, Sequential, and Loaded. Each one of these states handles and schedules requests differently and can transition into different states during a workload depending on the how requests are organized. Figure 3.4 is a state diagram for this stage.

The initial state is Passthrough of the queue stage and the most basic. This mode passes all requests through without any reordering or merging. Even though this seems to be a bad strategy for the controller to take, it does have a few merits. First is that since no actions need to be taken, commands can be sent to the drives very quickly. If there is only a small load on the array the overhead of scheduling commands may not be worth it. Even though it is assumed in this model that the drives do not have command queues, modern drives do have them. This means the drive can only process a single command at a time. When these queues become full the drive cannot accept any more commands. The controller backs off once these queues are close to being filled. It is better for small random loads that the controller simply allow the commands to pass through until saturating the drive queues and this behavior must be accounted for by this model.

If a part of the input trace is detected to be sequential the command queue shifts into the Sequential state. This mode will allow commands to be queued up, and if a few commands come in out of order they are inserted into their correct location. When requests are received by the command queue stage they are inserted into a queue in order of increasing LBA. If it is detected that the last few requests do not match a sequential pattern then the queue is flushed and the command queue stage shifts back to Passthrough. These scheduled requests are sent off to the command issue stage.

The last state the command queue can have is called the Loaded state and represents the controller under a heavy load. Its operation is shown by Figure 3.5 Once the controller has enough requests to saturate the drives of the array it must buffer the requests in its own memory. When this buffer starts to fill there are opportunities for the controller to reorder commands. The buffer is broken up into two segments and requests are distributed to either segments depending on the LBAs of the requests and how many commands are currently in the segment. The segment at the beginning of the queue fills up first. If a request that comes in is located far from those in the current segment then the request is inserted into the next segment. When requests are inserted they are put in an order so that

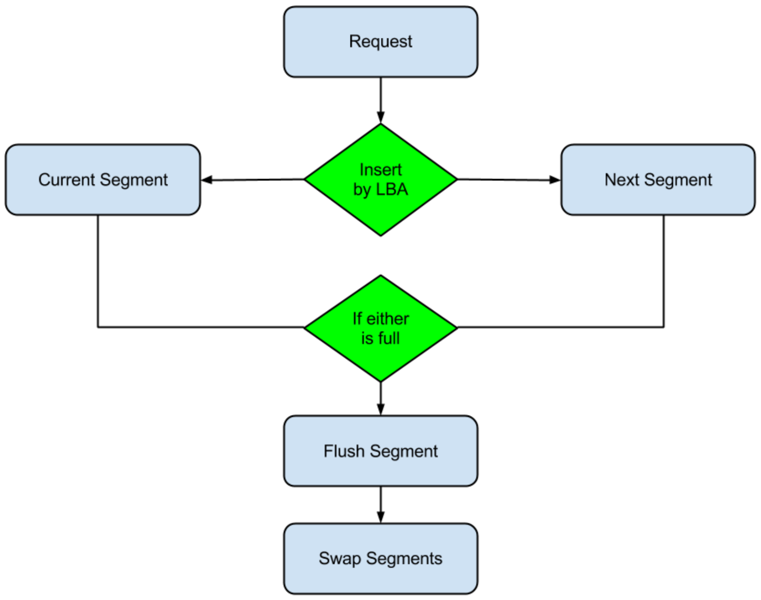


Figure 3.5: Block Diagram of Loaded State in the Command Queue Stage

all the LBAs are all increasing or decreasing to make sure there are only seeks in one direction. Once the segment becomes full or the timeout has occurred it is pushed to the command issue stage.

3.2.4 Command Issue Stage

The command issue stage takes I/O requests and issues the commands to the drives in the array. This stage takes its input from the command queue in the form of host requests. The process this stage undertakes is outlined in Figure 3.6. After the commands are produced by this stage they are passed to the drive stage. The drive commands have a format of (Type, Drive LBA, Drive) as shown by Table 3.4.

All actions on the array will always occur on a per stripe basis. Even if a

Drive LBA	Drive Number	Type (Read/Write)

Table 3.4: Format of a Disk Command

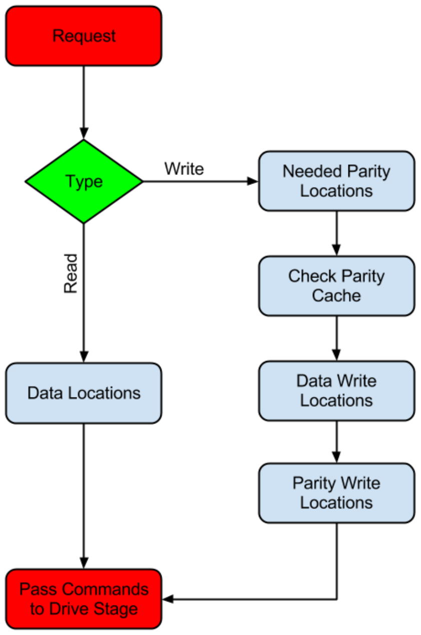


Figure 3.6: Block Diagram of the Command Issue Stage

request spans across multiple stripes, commands will be issued for each stripe instead of merging the individual drive commands together. RAID arrays take this approach to help reduce the amount of lost or corrupted data if a power failure occurs as the chance of a command only partially completing is very low for short commands. Also this keeps all the drives in the array synchronized with their angular and head positions so that no one drive should lag behind any other drive reducing the amount of idle waiting time.

For read requests the issue stage uses the equations outlined by the spatial model defined in Section 3.1 to calculate the LBAs and drives needed to fulfill a host request. The length of the request is split up by the chunk size and (Drive LBA, Drive) is calculated for each resulting chunk. After all these calculations are completed for a stripes worth of commands they are passed off to the drive stage.

Write requests follow a more complex path through the command issue stage. In RAID 5, writes follow a Read-Modify-Write pattern where data needed for parity calculations is read, parity is calculated and finally the data and parity are written to disk. As with read requests, a write request is first split into smaller pieces by stripe boundaries. As shown in Figure 3.6 for each stripe the following steps are taken. First the host LBAs of all the non-written chunks in the stripe are determined. The data contained by these chunks must be read to calculate the value of the parity chunk of the stripe. Even though parity could be calculated by only reading the parity data and using the incoming write data the controller uses the method explained above. Next, depending on which cache parameters are selected, the parity cache is checked to see if there are hits on these data chunks. A hit indicates that these chunks already reside within the controller and do not need to be read. With the parity reads taken care of, the locations of the actual writes are calculated the same way the read commands are calculated with Equations 3.1 and 3.2. Lastly the location of the parity chunk is determined with Equations 3.1 and 3.3. All of these commands are then passed to the Drive stage for processing.

3.2.5 Drive Stage

The drive stage models the actions of the drives connected to the RAID controller. This stage takes commands from the command issue stage and organizes them by each drive in the array. Next this stage calculates the amount of time needed for each of these commands to complete. When commands are completed

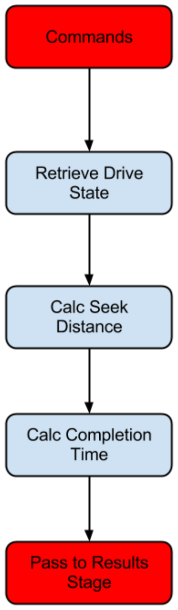


Figure 3.7: Block Diagram of the Drives Stage

they are passed to the results stage. The model used for the individual drives is simple and is meant to give a good estimation of performance rather than absolute accuracy. There are many assumptions in the drive stage that do not match the behavior of real hard drives but have been chosen to simplify the model.

Only completion times of commands sent to the drives are important to the overall model. To avoid adding too much complexity to the model the following assumptions are made. First is that the time for a drive to complete a command is determined by the amount of time it takes to seek to a location plus a fixed time for the data to pass under the head. Because every drive command that can be issued by this model is fixed to the size of a chunk, we are assuming that for every disk access the time the data spends under the head is the same. This time is determined by the parameter sequential access rate which is the max speed in which data can be read or written to disk. Combining this information with the chunk size allows the constant access time to be added to each command to be calculated by dividing $ChunkSize/AccessRate$. We assume that drives do not contain their own command queues and can only process a single command at a time. This assumption is made because modern hard drives have command queues and complex scheduling algorithms that are very difficult to model. Therefore removing these features from the modeled drives makes things much more manageable. Lastly it is assumed that all the drives contained by this model are homogeneous and exhibit the same performance characteristics.

$$f(n) = \begin{cases} 0 & \text{if x is 0} \\ a\sqrt{x-1} + b(x-1) + c & \text{if x} < 0 \end{cases} \tag{3.4}$$

$$\begin{aligned} a &= \frac{(-10minSeek + 15avgSeek - 5maxSeek)}{3\sqrt{numCyl}} \\ b &= \frac{(7minSeek - 15avgSeek + 8maxSeek)}{3numCyl} \\ c &= minSeek \end{aligned} \tag{3.5}$$

Seek times are the dominant factor for determining the completion times of commands. To model these times accurately Lee's seek equation is employed [14]. The Lee seek equation shown in Equation 3.4 requires the parameters min, average, and max seek times for a drive. These parameters are used to calculate the variables a, b, c using Equation 3.5, the original equation takes the seek distance as a number of cylinders. Since this model is not attempting to model complex modern disk formats, a parameter sectors per cylinder must be defined. This is used to calculate how many LBAs must be traveled before moving on to the next cylinder. This value is constant throughout the whole platter and doesn't follow real drive formats. The current position of the head is stored and used to calculate the seek distance between two different commands. This value is initialized to zero or at the outer most point on the platter.

The path of commands through the drive stage is outlined in Figure 3.7. When a command reaches the drive stage the current drive head position for the destined drive is retrieved. Next the seek time is calculated between the current position of the head and the new position required by the next command. The fixed access time described by *ChunkSize/AccessRate* is added to the result computing the command completion time. This information is passed along with the (Drive LBA, Drive) address to the results stage.

3.2.6 Results Stage

The last stage in this model is the results stage. The results stage is responsible for receiving completed drive commands and then calculating overall performance of the system. This stage passes information of completed requests back to the cache stage to insert new entries or replacements into the cache list. The following performance metrics are computed from the data passed into this stage, overall throughput, drive throughput, total commands issued, commands issued for each drive and trace completion time. Commands that are satisfied directly from the cache stage are assumed to be passed instantly and do not contribute to any of the time metrics.

$$Throughput = \frac{\sum Command.Length}{CompletionTime} \tag{3.6}$$

$$CommandTotal = \sum Command \tag{3.7}$$

$$CompletionTime = MAX(\forall Drive \sum CompletionTime) \tag{3.8}$$

Total completion time is determined by summing the completion time of all the commands sent to each drive and taking the maximum result. Throughput is calculated by summing the total amount of data and dividing by the total completion time as shown by Equation 3.6. Commands are organized by type before being summed together so that read and write performance are both determined. This process is done both on a per drive basis and for the whole array as well. Total commands issued is the sum of all the commands received by the results stage as shown by Equation 3.7. The trace completion time is determined by the drive which took the longest to complete all of the commands sent to it. To calculate the completion time for each drive the sum of all commands on that drive is calculated. Then the max of these is selected as the trace completion time as shown by Equation 3.8. Lastly the average command completion time is simply the mean average of every command received by the results stage.

3.3 Accurate Hardware RAID Simulator (AHRS)

The performance and spatial models described by Chapter 3 describe the behavior and data layout of the LSI 9260 hardware RAID controller. It takes significant time to validate a workload through the model by hand. Therefore a simulator based on both of these models was created.

The Accurate Hardware RAID Simulator was written in Python version 3.3. It is designed to take text input generated by the linux utility blktrace [5] which captures I/O requests as they travel through the Linux storage stack. The only deviation between AHRS and the models described in Sections 3.1 and 3.2, is the loaded mode in the command queue stage was not implemented. The text input is parsed for requests issued to the storage device and form the workload to feed into the simulator. All of the parameters listed for both the spatial model and performance model are given through a configuration file. There are two different files output from a simulation. First is the performance figures calculated by the results stage described in Section 3.2.6 and the amount of read and write data transfered in simulation. The second output is a text file representing the simulated trace. The commands issued to each drive are contained in an easily parsed text format.

Chapter 4

Results and Validation

4.1 Test Setup

The test machine used to verify the work presented in this thesis has the following hardware specifications. An AMD Opteron 6200 processor with 8 cores running at 2.6 GHz with 32 GB of RAM drives the system. Next the LSI 9260-8i was selected as the hardware RAID controller. The drives used in the experiments are four Western Digital enterprise 6 Gbit/s SAS drives rotating at 10,000 rpm. Lastly a SAS expander was placed in between the controller and the drives. A SAS expander is a piece of hardware that routes SAS traffic much like an Ethernet router. There is a single SAS connection between the expander and controller and then the expander is connected to each drive in the array as shown by Figure 4.1. Adding the expander to the experimental system has very little impact on performance and was only deployed to allow the capture of all the traffic in these experiments.

This hardware ran Arch Linux with the Linux kernel version 3.9.3-1. None of

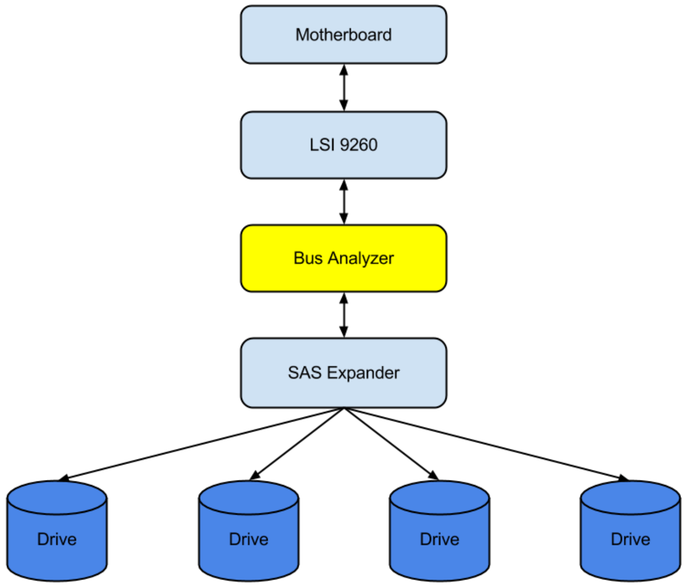


Figure 4.1: Hardware Test Setup

the experiments were carried out on any type of file system and instead always used raw partitions on the RAID 5 volume. This was done to eliminate any influence or interference a file system may have on storage workloads. Most modern file systems use techniques such as journaling that add complexity and additional load to the storage system. Also many file systems behave differently depending on how full or empty the partition they reside on is. Again this adds an unnecessary distraction from discovering the RAID controllers behavior. Along with raw partitions the Noop I/O scheduler from the Linux kernel was used instead of the Completely Fair Queuing scheduler. Noop is the simplest I/O scheduler available to the Linux kernel and is a FIFO or First In First Out algorithm [12]. Unlike the other schedulers in the Linux kernel Noop does not reorder requests and lets them pass through in the order they are issued by the host system. The only action it takes on incoming requests is to merge adjacent requests into large ones, if possible. Since Noop doesn't change the ordering of requests like most other disk schedulers we can accurately send workloads without the influence by the operating system. Lastly the command queues on the hard drives were disabled.

To capture the commands sent to the drives of the array a SAS bus analyzer is used. The bus analyzer sits in between two different devices on a SAS network and can capture all the traffic that is passed over the bus. It is designed to not interfere or affect any of the transmissions over the SAS connection and does not introduce any extra latency. In this work only the SAS commands related to reading or writing data were examined. A timestamp, the LBA and length of commands were the fields extracted and used for this work.

4.2 Validation Framework

The spatial model forms the foundation for the work presented in this thesis and must be validated. Two tests were performed to confirm the models accuracy for every available chunk size setting on the LSI 9260. It takes approximately six hours access every single LBA on the RAID volume and this would have be redone for each available chunk size. Therefore for the first test the following approach was taken. Instead of accessing every possible location on the volume, many samples consisting of a single LBA were taken across the whole volume. Approximately 10 million samples were taken to validate the spatial model. The locations for these samples were chosen to test accesses that not only align nicely within chunks but also next to chunk boundaries to ensure correctness. The other test consisted of small sequential reads spread around the volume. To confirm the model's correctness, the LBAs read by the controller need to contain the same data locations and have gaps where the parity data located. Since the requested host LBAs are known it is easy to compare and confirm the accuracy of the model with these two tests. The spatial model passed both of these tests without any deviations.

For each experiment the following criteria will be used to compare to the output produced by AHRS to the traces generated by the actual RAID controller. The data used for these comparisons are the drive command traces produced by AHRS and the commands issued by the RAID controller captured via the bus analyzer.

Accuracy of AHRS will be determined by two factors. Both of these methods produce a similarity measure between the two outputs. The first is the Jaccard index which is a ratio between 0 and 1 measuring the similarity of two sets. The ratio of the intersection over the union of two input sets is how this ratio is calculated and is shown by Equation 4.1 [11]. The higher this ratio the more similar the two sets are. The ratio will be displayed as a percentage in these results meaning a result of 100% is desired. In our case this gives a metric showing how many of the same commands exist in each output. This only accounts for the content of the traces, not the ordering of the commands. Therefore the Jaccard index will be the same for traces that are inverses of each other. Since the calculation of the Jaccard index relies on sets which only contain unique elements to account for repeated commands the following is done. If a command is repeated multiple times within a trace it is inserted with a count attached to its element. For example if the command (Read LBA: 0 Length 16) was repeated twice the second element would be inserted into the set as (Read LBA: 0 Length

16 1).

$$J(X,Y) = \frac{|X \bigvee Y|}{|X \bigwedge Y|} \tag{4.1}$$

The second measure of accuracy used is the Damerau-Levenshtein distance [9]. This metric is the edit distance between two strings or the number of insertions, deletions, substitutions, or transpositions of adjacent characters needed to transform one string into another. In this case instead of a string full of characters, strings of commands are compared against each other. The results from this comparison will be displayed in the following manner. The edit distance will be followed by the number of edits compared to the overall size of the trace. This is displayed as a percentage and shows how much the trace is different. An edit distance of 0 means a complete match and a larger distance is an indication that the traces are more different.

Lastly the amount of data transferred between the host and the drives in each trace will be compared. Read and write commands will be separated into different sums and compared. A percent error will be included in the data presented. Tables 4.3 and 4.5 contain the values of these amounts for both direct and cached mode respectively.

Performance predictions made by AHRS will not be included in this work as it was found that the simple disk model employed in Section 3.2.5 is not sufficient enough to accurately model the test machine. Unfortunately too many of the assumptions made by the Drive stage do not match the real behavior of the drives used in the experiments and therefore the results are not reliable.

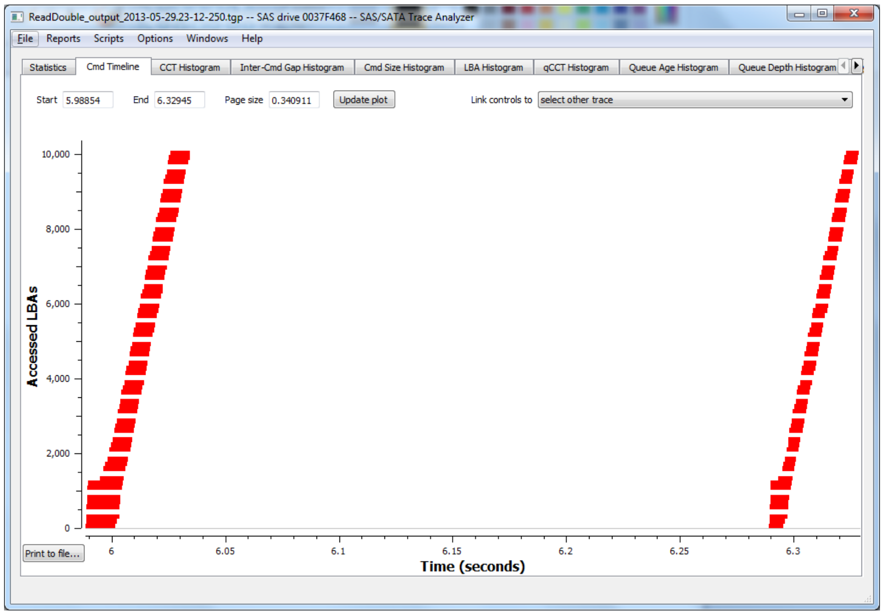
Parameter Name	Value
Chunk Size	128 LBAs
N	4
Write Cache Policy	Write-Through
Cache Size	8192 Entries (512 MB)

Table 4.1: List of Parameters Used for All Experiments

4.3 Experiments

There were a set of five experiments that were performed to test against the AHRS. Each on of these test different types of I/O workloads that are commonly seen by RAID systems. All the experiments were done with both the direct and cached mode selected as a controller parameter. These are the cache parameters described in Section 3.2.2. As stated in the assumptions made in Section 3.2, all I/O requests are aligned to chunks in the array. For each one of these experiments a visualization was generated that shows the SAS bus trace captured as well as a comparison between the simulated and actual traces. Only the visualization from one drive will be shown as the variation between each in the array is very minimal. The red dots or lines signify reads and blue represents write commands.

The Flexible I/O tester or fio is a Linux tool used to generate I/O workloads [4]. All the experiments relied on fio to produce the input workloads. The following was true for each experiment. A total of 30 MB of data was transferred to or from the RAID volume. Direct I/O was used to bypass the Linux kernel's write buffer and the Linux asynchronous I/O library known as the libaio engine in fio was selected. Lastly as stated in Section 3.2 all I/O was aligned to chunk boundaries.



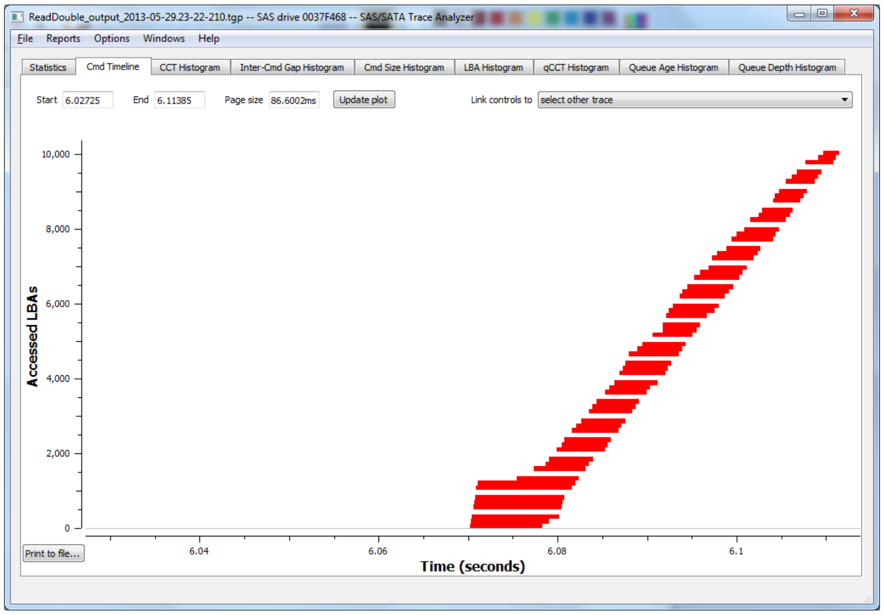
Start 5.98854 End 6.32945 Page size 0.340911

Link controls to select other trace

Figure 4.2: Visualization of ReadDouble Capture in Direct Mode

4.3.1 Read Double

Read Double is the first experiment that tests a sequential read that is repeated twice on the same data. This tests not only a simple sequential workload but is meant to expose how the controller caches read data. Figure 4.2 shows the visualization from the direct mode capture, while Figure 4.3 shows the capture when using cached mode. Comparing these two figures you can clearly see the effects of the cached mode where the second sequential read is missing. Since the data is in the cache after the first read, no other commands need to be issued in cached mode. A Jaccard percentage of 100% and an edit distance of 0 for both direct and cached mode resulted from this experiment meaning AHRS is completely accurate for this type of workload.


ReadDouble_output_2013-05-29.23-22-210.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

Start 6.02725 End 6.11385 Page size 86.6002ms

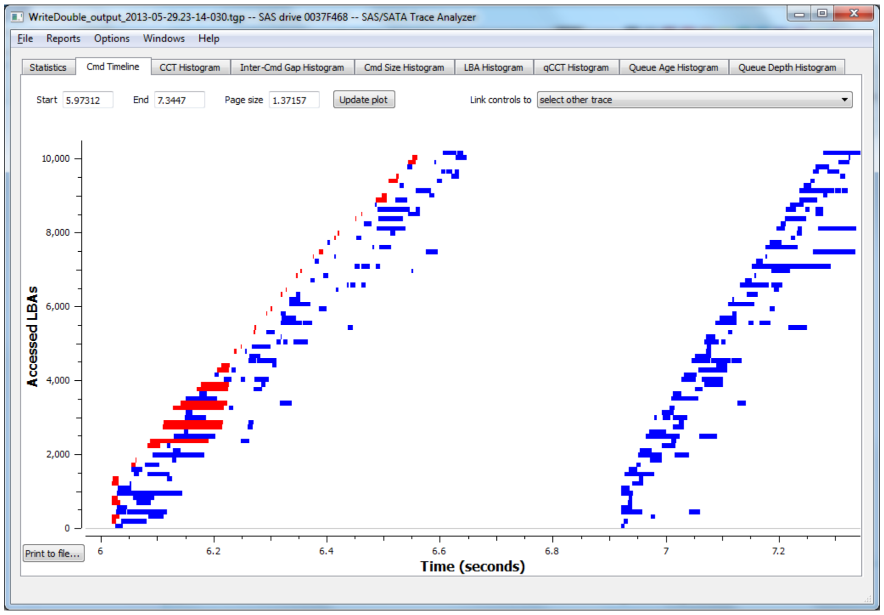
Link controls to select other trace

Figure 4.3: Visualization of ReadDouble Capture in Cached Mode

4.3.2 Write Double

Write Double is very similar to Read Double where there are two sequential writes done right after each other. As known with RAID 5 writes require extra data to be read so that parity calculations can be done. Figures 4.4 and 4.5 show the visualizations for this experiment. Both these traces are almost exactly the same. During the first sequential write there are reads for parity data and this data is cached for the second write in both modes. This is why there are no reads during the second write.

Unlike with Read Double, AHRS doesnt produce an exact match against the output of the real controller. As represented by the Jaccard percentage of 99.73% for both direct and cached mode, AHRS produces almost the same commands in both cases. The difference was caused by a few extra reads issued by AHRS for parity calculations. This is backed up by the positive read percent difference value in Table 4.3. Ordering of commands is the factor that causes the most error


WriteDouble_output_2013-05-29.23-14-030.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

Start 5.97312 End 7.3447 Page size 1.37157

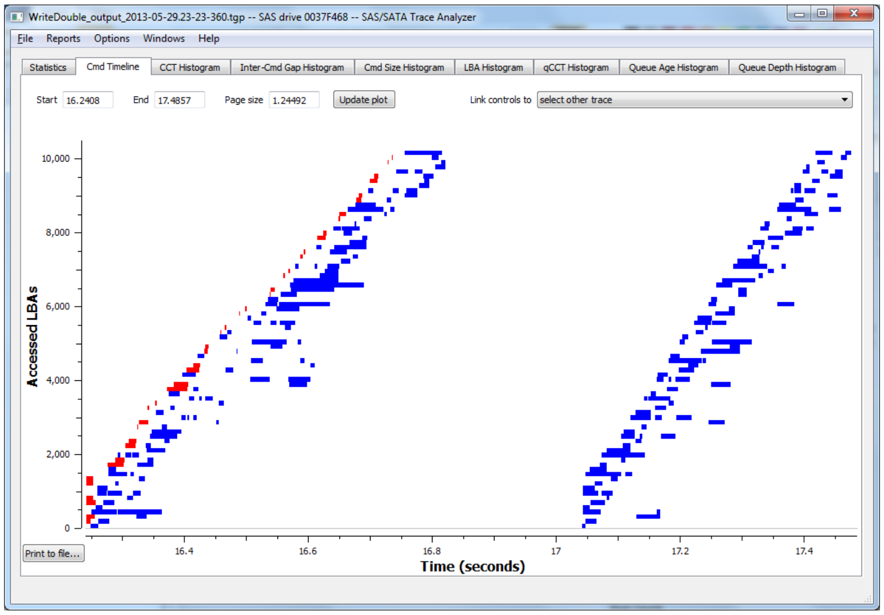
Link controls to select other trace

Figure 4.4: Visualization of Write Double Capture in Direct Mode

as shown by the edit distance results. For direct mode a total of 901 edits are needed to produce a match while cached mode needed 890. Since these numbers are so close it can be concluded that the same write scheduling algorithm is used in these modes. Looking into the traces more closely it seems that the real controller queues a small amount of writes together and issues all the necessary parity reads before issuing any write commands. Once there are cache hits the writes are still grouped in the same way but the unneeded reads are not issued.

4.3.3 Random Read

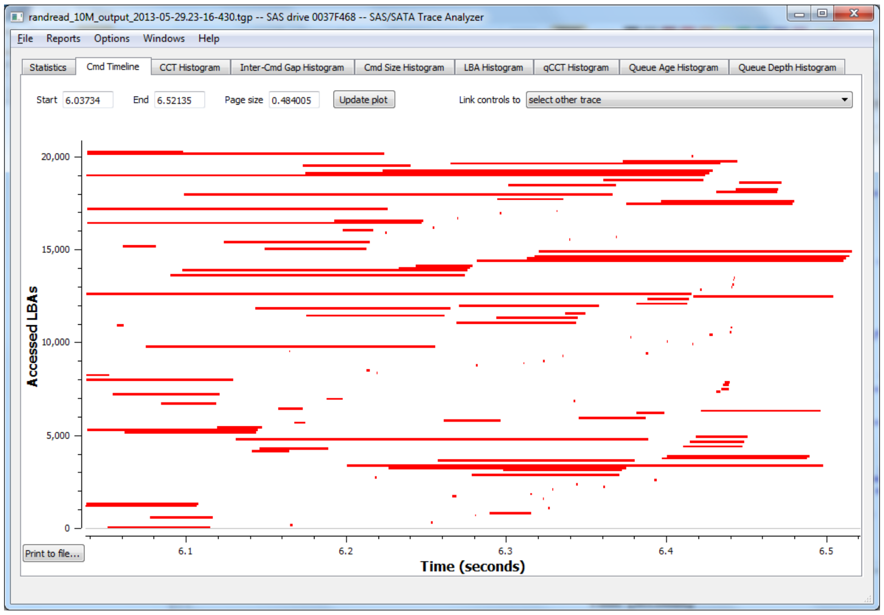
Random Read simulates a random access pattern with only read requests. For normal hard drives a random workload such as this is one of the worst cases for performance. Figures 4.6 and 4.7 show the visualizations for this experiment. Because the access pattern is random the chance of a cache hit is very low, so the output of both workloads is very similar.


WriteDouble_output_2013-05-29.23-23-360.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

Start 16.2408 End 17.4857 Page size 1.24492

Link controls to select other trace

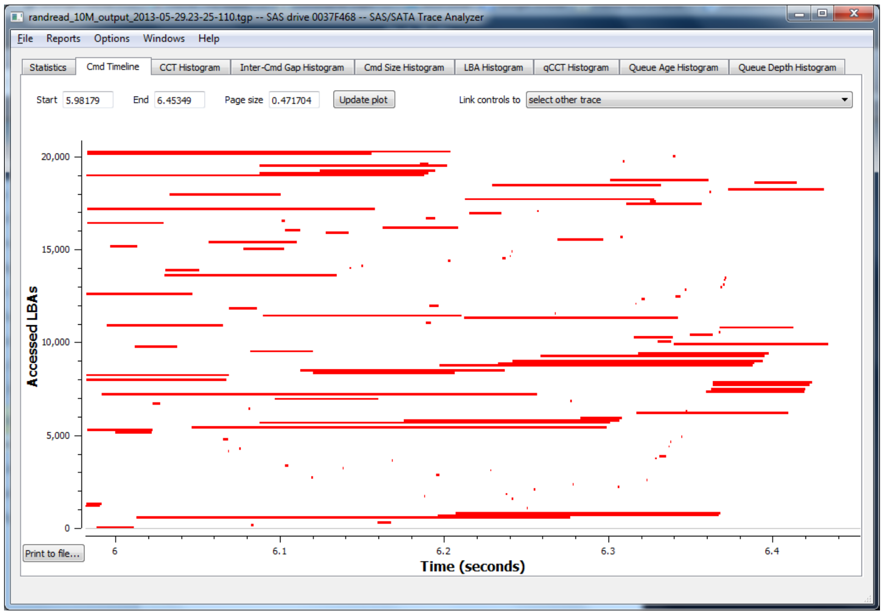
Figure 4.5: Visualization of Write Double Capture in Cached Mode


randread_10M_output_2013-05-29.23-16-430.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

Start 6.03734 End 6.52135 Page size 0.484005

Link controls to select other trace

Figure 4.6: Visualization of Random Read Capture in Direct Mode



randread_10M_output_2013-05-29.23-25-110.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

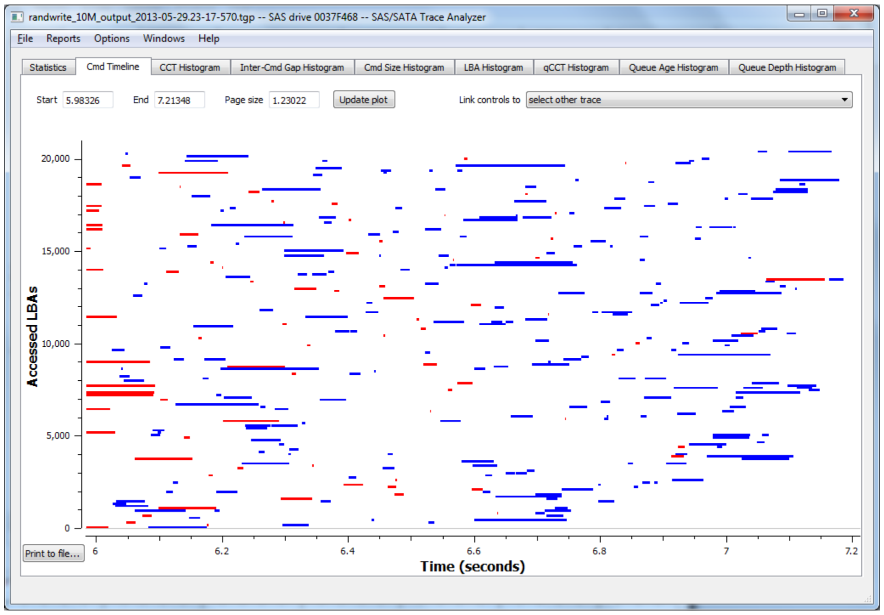
Start 5.98179 End 6.45349 Page size 0.471704 Link controls to select other trace

Figure 4.7: Visualization of Random Read Capture in Cached Mode

AHRS produces all the same drive commands as the actual controller as shown by a Jaccard percentage of 100% for both modes. Also the amount of data transfered is the same as the controller as shown in Tables 4.3 and 4.5. The edit distances were calculated to be 49 for direct mode, and 20 for cached mode. When looking at the individual commands in the traces the cause of edits were found. The edits were caused by a few commands being swapped for each other. For the most part the commands are either groups of one or two but there are a few groups of around five commands that are in a different order. The pattern of these swaps seem to be unpredictable and doesn't expose any new information about the controller's queuing algorithm.

4.3.4 Random Write

Random Write repeats the same type of random access pattern except only writes are issued. Figures 4.8 and 4.9 show the visualization for this experiment.

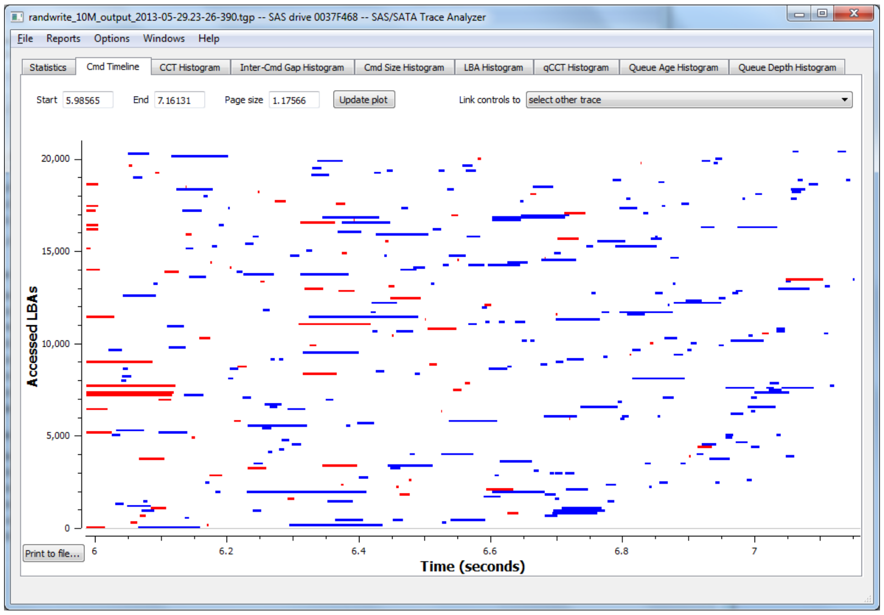

randwrite_10M_output_2013-05-29.23-17-570.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

Start 5.98326 End 7.21348 Page size 1.23022

Link controls to select other trace

Figure 4.8: Visualization of Random Write Capture in Direct Mode

For a RAID 5 system this is the worst case scenario as both the read-modify-write pattern and long hard drive seeks are involved. For both direct and cached mode AHRS is caching more data to use for parity reads than the actual controller as shown by the negative percent difference for reads in Tables 4.3 and 4.5. This is a result of a more aggressive caching policy used in AHRS than the controller. To combat this worst case workload, the real RAID controller reorders commands very aggressively. One would expect to see an elevator algorithm where commands are ordered in ascending and descending sections attempting to service the most requests within sweeps of the head. This type of scheduling is commonly used by I/O schedulers for its simplicity and performance but there is little to no evidence of one being employed by the controller. Because of the undiscovered scheduling write algorithm, AHRS cannot predict the output very accurately for this type of workload.


randwrite_10M_output_2013-05-29.23-26-390.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

Start 5.98565 End 7.16131 Page size 1.17566

Link controls to select other trace

Figure 4.9: Visualization of Random Write Capture in Cached Mode

4.3.5 Random Workload 30% Writes

Random 30% Writes combines both random read and write into one workload where 30% of the I/Os are writes and the rest are reads. This is meant to emulate a workload a typical database might generate when at a significant load. Even with this complex workload, AHRS produces almost the same commands as the controller. As seen by the results in Tables 4.2 and 4.4, a Jaccard index of 98.2% in direct mode and 90.72% in cached mode. Unfortunately as with the other workloads with write commands there is a high edit distance of 410 for direct mode and 439 in cached mode. As evidenced by the other experiments, AHRS has difficulty ordering write commands correctly and is the greatest source of error in this workload. Also as seen in the Random Write experiment in Section **??**, AHRS is more aggressive when it comes to caching data. In direct mode this extra caching is slightly greater by a 2.74% percent compared to the controller. This value was lower when cached mode was used, a difference of .55% resulted.

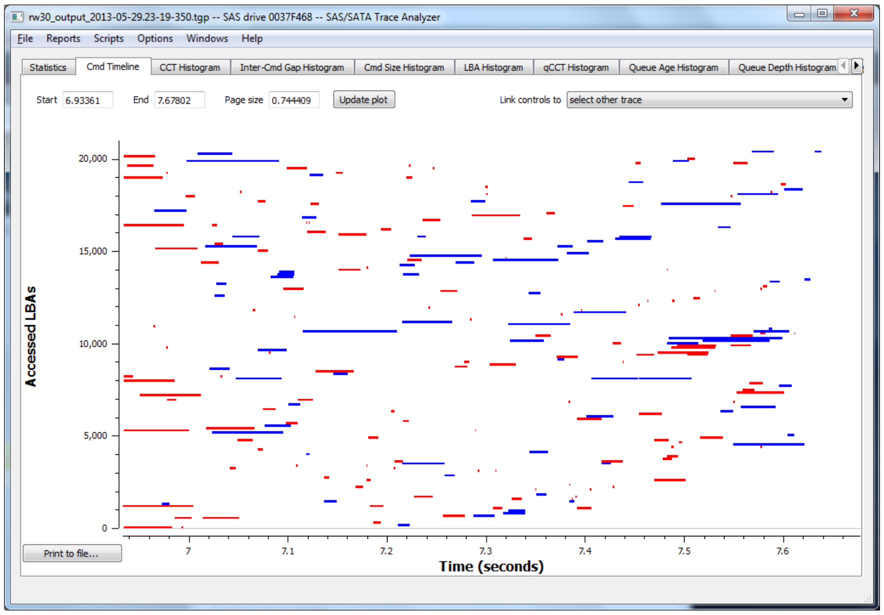


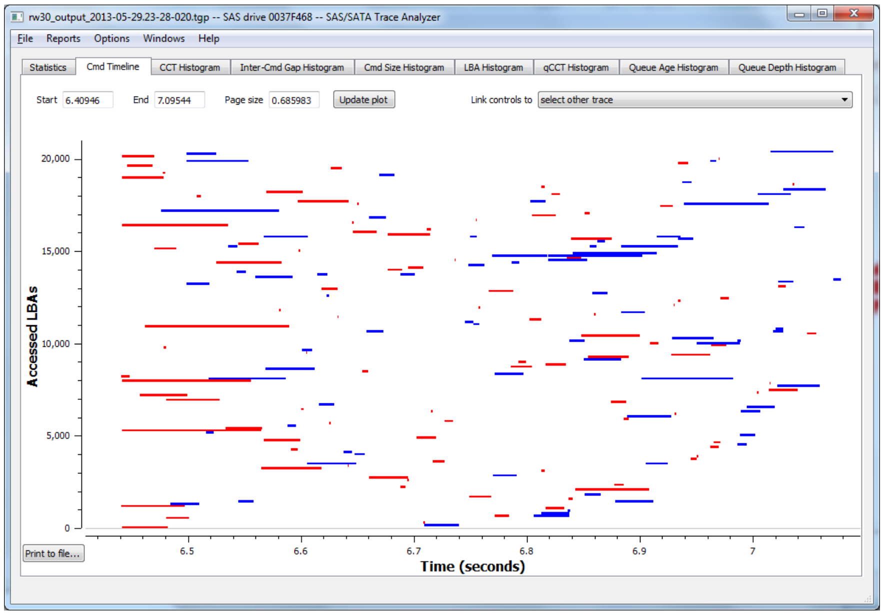
Figure 4.10: Visualization of Random 30% Write Capture in Direct Mode

Experiment	Jaccard	Damerau–Levenshtein Distance
Read Double	100%	0 (0%)
Write Double	99.73%	901 (79.86%)
Random Read	100%	49 (10.04%)
Random Write	97.03%	729 (56.6%)
Random 30% Write	98.2%	410 (48.8%)

Table 4.2: Command Issue Accuracy Results in Direct Mode

Experiment	AHRS Ouput		Controller Ouput		(Values in MB)	
	Read	Write	Read	Write	Diff Read	Diff Write
Read Double	30	0	30	0	0%	0%
Write Double	10	60	10	59.81	0%	0.31%
Random Read	30	0	30	0	0%	0%
Random Write	17.63	60	20	60	-1.19%	0%
Random 30% Write	33.31	17.75	34.25	17.75	-2.74%	0%

Table 4.3: Total Commands Issued by AHRS in Direct Mode


rw30_output_2013-05-29.23-28-020.tgp -- SAS drive 0037F468 -- SAS/SATA Trace Analyzer

Start 6.40946 End 7.09544 Page size 0.685983

Link controls to select other trace

Figure 4.11: Visualization of Random 30% Write Capture in Cached Mode

Experiment	Jaccard	Damerau–Levenshtein Distance
Read Double	100%	0 (0%)
Write Double	99.73%	890 (78.9%)
Random Read	100%	20 (4.1%)
Random Write	93.75%	766 (59.47%)
Random 30% Write	90.72%	439 (65.82%)

Table 4.4: Command Issue Accuracy Results in Cached Mode

Experiment	AHRS Ouput		Controller Ouput		(Values in MB)	
	Read	Write	Read	Write	Diff Read	Diff Write
Read Double	15	0	15	0	0%	0%
Write Double	10	60	10	59.81	0%	0.31%
Random Read	30	0	30	0	0%	0%
Random Write	15	60	20	60	-25%	0%
Random 30% Write	23.31	17.75	23.44	17.75	-.55%	0%

Table 4.5: Total Commands Issued by AHRS in Cached Mode

Chapter 5

Related Work

At this point in time there have been a few attempts to try to model hard drive arrays. Two of these will be described, A Modular, Analytic Performance Model of Disk Arrays [19] and DiskSim [6]. The first provides a different mathematical approach to the problem presented in this thesis while DiskSim is a hard drive simulator with RAID capabilities.

5.1 DiskSim

DiskSim was created to be an efficient, accurate, and highly configurable storage system simulator [6]. It was developed by the Parallel Data Laboratory at Carnegie Mellon University to support research of existing and upcoming storage technologies. Its main focus is simulating modern hard drives precisely and has been validated against existing production drives. The whole project has been written in C and has no dependencies on any other software. The input to the simulation can be an externally generated trace or be created internally by the synthetic workload generator. Included as a part of DiskSim are two libraries that allow the use and expansion of the simulator; diskmodel contains the mechanical and layout simulation code and libparam which is the input parameter parser allowing other projects to easily use the same input format. DiskSim has been used in a variety of academic projects with great success [10, 22].

The simulator is organized into modules that are configurable and linked together to form a storage system. These modules include device drivers, buses, controllers, adapters, and the drives. The driver module is the topmost module in the system and defines the behavior of the operating system's interaction with the storage system. Most of the parameters of this module set the scheduling and queue techniques, for example choosing between a First-In-First-Served or a Shortest Seek First algorithm. Communication channels between devices in the simulation are defined by the Bus module. Parameters for this module include transfer speeds, and arbitration strategies for shared buses. One would set these to match a current technology such as SATA or SAS and declare new bus modules for every connection in the storage system.

The controller module is placed between drives and connects the resulting buses. There are three types, passthrough, driver managed, and complex. The passthrough controller simply allows commands to pass from one bus to another. Next the driver managed controller emulates a simple SCSI controller which takes commands from one type of bus and translates and puts them onto another bus in route to the drives. This type introduces the overhead of the operations necessary to perform this. A modern example of this type of controller would be a SAS expander placed in between a host and a storage enclosure. Finally the complex controller model introduces a command queue and can enable a cache located on the controller. The same type of queuing algorithms are available on this module as the device driver module described before. If caching is desired there are parameters such as write through or write back policies, discard policies like LRU or FIFO, size etc.

DiskSim contains two different models for simulating hard drives, Disks and SimpleDisk. The SimpleDisk model is a simple model where all accesses take a constant time. This eliminates calculations for the mechanical actions of a drive and also assumes a fixed rate of data transfer. Even with these simplifications this model still has parameters for the scheduling and queuing of commands. Even though this model is not very useful for simulating real hard drives its code gives an example on how to create new storage modules for new technologies for DiskSim and also emulates the basics of Solid State Drive behavior.

For a realistic simulation the Disks model must be used. This module is the core of DiskSim and is the most complex module that exists in the system. A large portion of the parameters for this model define the layout and mechanical properties of the drive. Currently there are three different layouts available in DiskSim, G1, G2, and G4. Each one of these represents a different generation of hard drives with increasing complexity. For all these layouts the number of data surfaces and number of cylinders must be specified. Both of the G1 and G2 models can be defined with multiple density zones while the G4 model does away with this to implement modern zoneless formats. A mechanical model must also be defined with a layout model. Currently there only exists the G1 mechanical model. This is where the simulation of all mechanical motion of the drive is calculated. Parameters such as disk RPM, average seek times, settle time, head switch times are important for an accurate simulation. Lastly characteristics of the drive's buffer need to be defined. Options such as buffer size, pre-seeking, and caching behaviors have the greatest amount of impact.

The G1 model uses the same seek equation as the Drive stage described in

AHRS. The inspiration of using Lee's seek equation to estimate drive performance came from DiskSim. Unfortunately our work makes too many simplifying assumptions and cannot produce an accurate simulation of a hard drive's performance.

DiskSim can also form and simulate many types of RAID arrays. Unlike most of the other components included with DiskSim the Disk Array module is very basic and only includes a small number of parameters. The most important of these are the redundancy scheme, stripe unit, parity rotation type, and the devices included in the array. To simulate RAID 5, the redundancy scheme of Parity Rotated must be selected.

Now this module does not have any intelligence and cannot simulate any well known optimizations done by modern RAID controllers. One of these is the caching of parity reads during writes. Also controller caching and reordering cannot be controlled by this module making it very difficult to accurately emulate existing hardware. AHRS on the other hand is able to simulate such behaviors found on real RAID controllers.

5.2 A Modular, Analytic Performance Model of Disk Arrays

A Modular, Analytic Performance Model of Disk Arrays aimed to produce a throughput model for RAID 1/0 [19]. RAID 1/0 is known as a hybrid RAID level where two standard RAID levels are nested within each other. In this this case 1/0 means that first level is RAID 0 and data is striped. Then each stripe is mirrored as shown by Figure 5.1. This model predicts the amount of

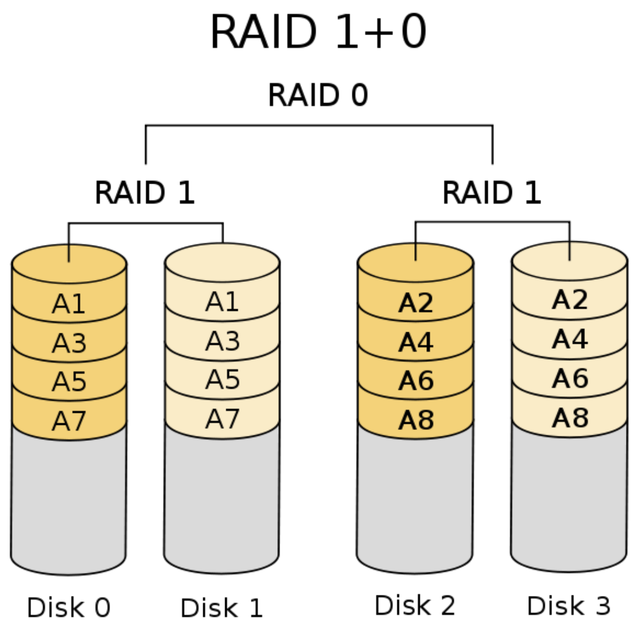


Figure 5.1: Hybrid RAID Level 1/0 [7]

I/O operations per second or commonly abbreviated as IOPS for a given storage workload. As with the work in this thesis the model incorporates the effects of caching and controller scheduling algorithms.

Instead of representing workloads as lists of host I/O requests they are described by a group of objects. All objects contain a set (Attribute, Value) pairs and sub objects which make up a workload action. Two types of objects were defined, store and stream. Store objects simply describe a continuous section of data. Stream objects hold information on the temporal and behavioral aspects of a workload. These objects contain fields defining request rate and size, along with the mean number of consecutive accesses for both reads and writes.

Similar to our work, their model is made up of different components, each modeling a different aspect of the system. Three components make up the overall model, cache, controller and disk. Each one of these feed into each other in the order specified. The cache receives stream objects and modifies them to reflect the effect of cache hits. Removed requests are based on a probability calculated from the size of the cache and how frequent a stream accesses the same data. The controller component takes streams and splits them into new streams destined for each drive in the array. For both read and write requests the average amount of disk accesses are calculated for each stream. These accesses are passed along to the disks component. Unlike with our work, drives are modeled to have their own individual caches. Similar to the cache component, the probability of a cache hit on the drive is used to reduce the amount of requests in the incoming stream. The remaining requests are used to calculate the disk service time which is determined by the mean disk seek time and the transfer speed of the drive.

When compared to a real RAID 1/0 array their model had the following results. For the workloads experimented on about the relative error was about 15% between the actual and predicted throughput. There were worst case workloads where this number increased to 40%. Since it was found that our disk model was not significant enough to accurately predict throughput a direct comparison cannot be made between the two works. Even so since AHRS can predict the actual commands the controller will produce, more in depth understanding and experimentation can be accomplished. Another factor that must be considered when comparing these two works is that RAID 1/0 contains no parity data and is much easier to simulate. Most of the research done in our work was to discover the behavior caused by parity reads and writes.

The core of the model presented by A Modular, Analytic Performance Model of Disk Arrays is based on calculated averages and probability. These figures cannot represent all workloads accurately. This is a trade off between accuracy and model complexity. Our work takes the opposite approach and tries to accurately predict the output of almost all workloads by increasing the amount of information tracked by the model. A side effect of this is that the actual behavior of the RAID controller is more exposed and can be more thoroughly understood. Also this allows different parts of the model to be tweaked or redesigned to further research on this type of storage. For example a stage in our model can be modified to experiment with new behavior to see if performance can be improved. Their model on the other hand is limited in this regard and can only be modified to more accurately predict a system that already exists.

Chapter 6

Conclusion

In this thesis we have described the construction of, the RAID 5 spatial and performance models. Together these form a foundation for the understanding of RAID 5 behavior with current controller hardware. This understanding is necessary for the further development of this vital storage technology.

The spatial model provides the layout of RAID 5 and allows the transformation of host addresses into the precise location of data on the drives of the array. This model also can determine the location of parity data responsible for maintaining redundancy. All this information is used by the performance model to form the correct commands sent to the drives. Two different tests confirmed that this model is accurate when compared to the layout used by the LSI 9260.

The performance model generates the commands needed to access data on the array. It also emulates the command queuing and caching behavior common with hardware RAID controllers. Each of the stages of this modular model explains a different aspect of host request processing and contains many configurable parameters to model many different RAID 5 environments. When validated against the LSI 9260 for accuracy, Jaccard indexes of 100% for a purely read workloads, and high 90% for purely write workloads. The worst result came from a mixed read/write workload with a index of 90.72%. The overall average of all experiments was 97.92%. It was found that the simple drive model used by the performance model was inadequate to emulate a real system's performance.

Lastly AHRS is a working implementation of the performance and spatial models. This simulation validates the logic behind these two models and also exposes areas where the model can be improved. Read only workloads are explained very well by this simulator. Writes are more approximate but still match the actions of a real system. Even when the ordering of commands are not exact the same amount and type of commands are generated.

6.1 Future Work

The largest weakness of AHRS is the command queue stage. Currently this stage only implements basic command scheduling algorithms and has difficulty emulating the controller's behavior under more complex workloads. Research focused on this aspect of RAID 5 could not only improve this implementation but also reveal new scheduling techniques. AHRS current implementation language python is easy to work with but lacks the performance of more low level languages such as C or C++. For the small workloads processed in this thesis the performance of python was sufficient. By rewriting AHRS in such a language, very large workloads could be processed and integrated with existing systems. One of these systems could be DiskSim and would increase the accuracy of the overall model greatly. This would introduce an even higher level of precision to the AHRS and would allow the simulation of complex drive behavior such as native command queuing. Lastly the spatial model could be expanded to include the layouts for other RAID level such as 6 or 0/1. Again this would increase the versatility of AHRS and allow more diverse research on storage systems.

Bibliography

[1] The linux system administrator's guide. http://www.tldp.org/LDP/sag/html/hd-schematic.png.

[2] Raid 5 scaling tests with up to eight drives. http://www.tomshardware.com/reviews/raid-5-scaling-tests-drives,852.html.

[3] RAIDs utilizing parity function(s). http://www.freeraidrecovery.com/library/raid-5-6.aspx.

[4] J. Axboe. Fio-flexible IO tester. *http://freecode.com/projects/fio*, 2008.

[5] J. Axboe and A. D. Brunelle. Blktrace user guide, 2007.

[6] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The DiskSim simulation environment version 4.0 reference manual. *Parallel Data Laboratory*, page 26, 2008.

[7] C. M. Burnett. Standard RAID levels. http://en.wikipedia.org/wiki/Standard_RAID_levels.

[8] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[9] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[10] G. R. Ganger and Y. N. Patt. The process-flow model: examining i/o performance from the system's point of view. In *ACM SIGMETRICS Performance Evaluation Review*, volume 21, pages 86–97. ACM, 1993.

[11] P. Jaccard. *Etude comparative de la distribution florale dans une portion des Alpes et du Jura*. Impr. Corbaz, 1901.

[12] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 295–304. ACM, 2009.

[13] E. K. Lee and R. H. Katz. Performance consequences of parity placement in disk arrays. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 190–199. ACM, 1991.

[14] E. K. Lee and R. H. Katz. An analytic performance model of disk arrays. In *ACM SIGMETRICS Performance Evaluation Review*, volume 21, pages 98–109. ACM, 1993.

[15] LSI Digital Corp. *LSI MegaRAID Controller Benchmark Tips*, January 2013.

[16] D. A. Patterson, G. Gibson, and R. H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[17] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–103, 2006.

[18] J. Toigo. 2013 storage trends. *http://esj.com/Articles/2012/12/17/2013-Storage-Trends.aspx?Page=1*, 2013.

[19] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pages 183–192, 2001.

[20] Western Digital Inc. *WD Black Desktop Hard Drives*, January 2013.

[21] Western Digital Inc. *WD XE SAS Hard Drives*, January 2013.

[22] B. L. Worthington. *Aggressive Centralized and Distributed Scheduling of Disk Requests*. PhD thesis, The University of Michigan, 1995.