Implementing a Statically Adaptive Software RAID System

Matt McCormick
mattmcc@cs.wisc.edu

Master's Project Report

Computer Sciences Department
University of Wisconsin–Madison

Abstract

Current RAID systems are limited by the performance of the slowest disk drive in the array. This paper proposes several new techniques for mapping data to the different drives in such a way that the higher performance disks receive a greater share of a files data than the lower performing disks. The techniques presented here statically determine the mapping strategy for the entire RAID. This determination is made at the creation of the RAID and remains the same through-out the life of the device. Preliminary studies show a modest 7% improvement for reading and writing large files (256 MB) in their entirety. More suprising is the reduction in bandwidth due to system overhead. The SCSI bus in the system studied is capable of maintaining 80 MB/s of traffic, but our results show a maximum read bandwidth of 65 MB/s and a maximum write bandwidth of only 45 MB/s.

1 Introduction

The computer industry is a place where advancements in technology and performance are measured in terms of months and the top of the line products today will be tomorrows bargain sale to make room for the new and improved versions. Disk storage is no exception to this rule. The potential effects this has on RAID systems is obvious. The initial disks in the system will quickly be out of date and faster, more reliable, and higher capacity disks will soon be available. The failure of disks in the current system or the need to expand the overall storage capacity of a system means that system administrators are going to have the option to place higher performance disks into a system with these lower performing drives. However, current RAID implementations assume that all the disks in a system perform at an equal level. This means the slower disks will dictate the overall performance of the system and the potential benefits of upgrading to a higher performance disk will be lost.

This paper is intended to investigate the potential benefits of considering the performance of individual disks when building a RAID system. The theory is that putting more data on a disk that provides a higher bandwidth between

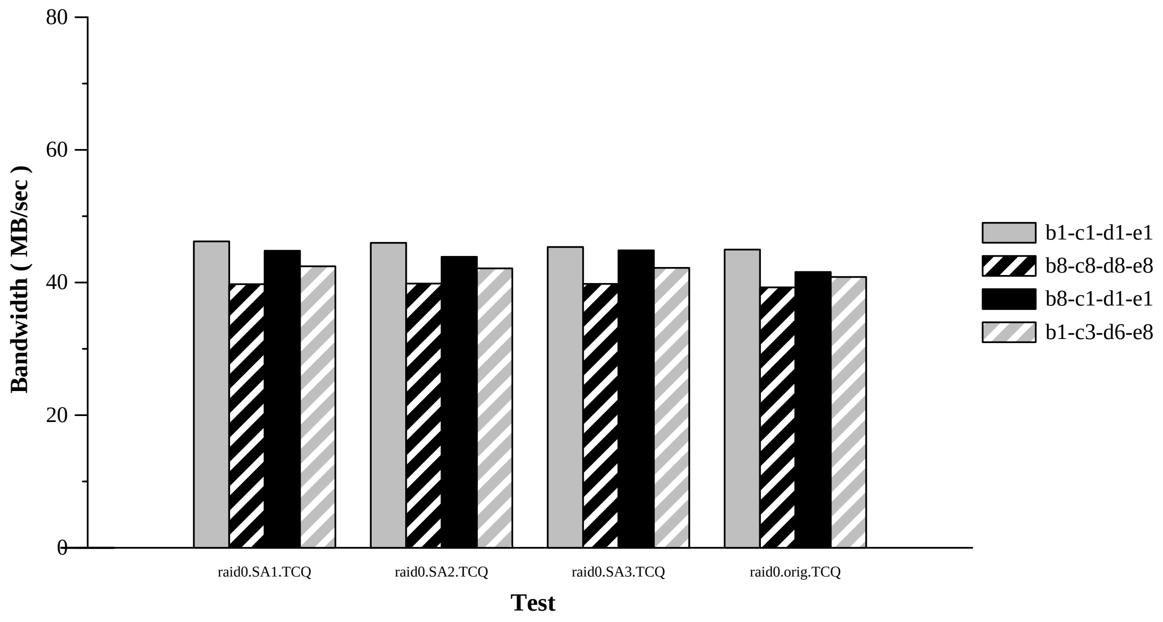


Figure 1: **Write Bandwidth Results for Test and 4 KB block size and 262144 KB file size**

the platters and the host computer will result in a higher performance system. This research proposes and evaluates three different adaptive techniques for mapping file data to the disks in a RAID. All of the techniques presented here implement RAID-0. All three of the techniques base their mapping strategy on the overall performance of each of the disks. As a result, each of the techniques presented will map the exact same amount of data from a file to each disk. The difference between them is the ordering of that mapping. All of the methods are compared to the original RAID-0 implementation in the Linux-2.2.19 kernel.

Section 2 looks in depth at the design and implementation of each of the 3 mapping algorithms. Section 3 examines the results of reading and writing one large file using these techniques. Section 4 examines possible future work in this area. Section 5 gives the conclusions of this research.

2 Implementation

2.1 Current RAID 0 Implementation

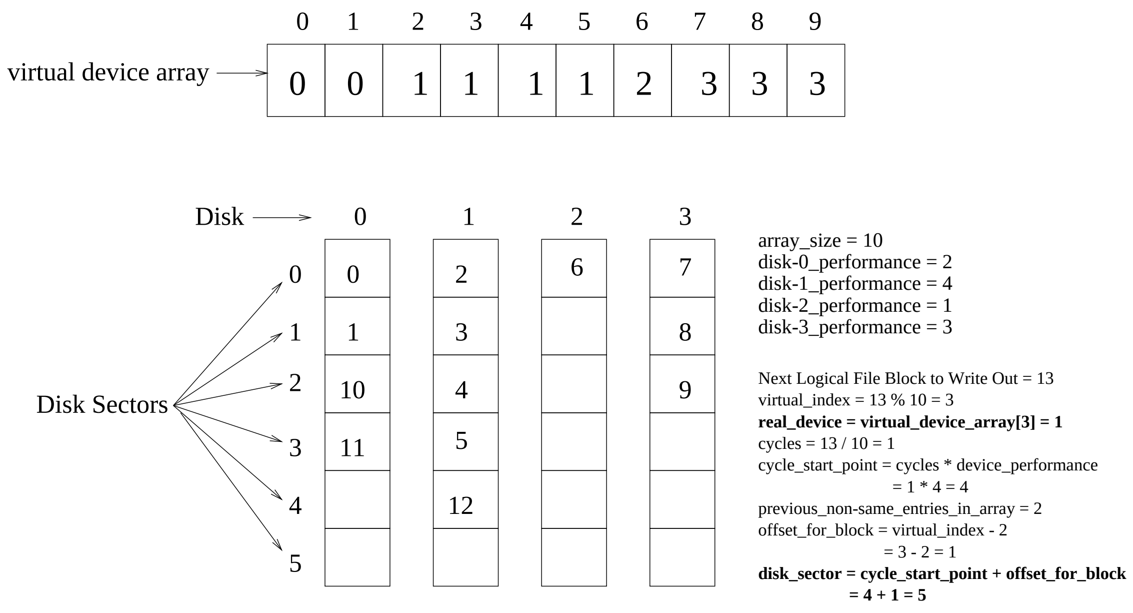
Like all of the other software RAID techniques in the Linux kernel, the existing RAID-0 implementation assumes that all disks perform exactly the same. The technique for selecting where to put data is based on a very simple algorithm. The mapping algorithm works on logical blocks of a file. In the system examined here, each of these logical blocks is 4 KB in size. The disk to access for this logical block is determined by taking the modulo of the block number by the number of disks in the RAID configuration. The sector on the disk is determined by doing integer division between the logical block number and the number of disks. While this technique is very simple to implement, the overall performance of the system is limited by the bandwidth of the slowest disk. This can be seen by examining test raid0.orig.TCQ (this is the original RAID-0 implementation) in Figure 1. The performance when all disks are using the fastest partition (b1-c1-d1-e1) performs approximately 15% faster than when all of the partitions are from the slowest portion of the disks (b8-c8-d8-e8). When just one of the disks is converted from a fast partition to the slow partition (b8-c1-d1-e1), the system now only performs about 6% better than the all slow configuration.

This occurs because the same amount of data is being written to each disk. This means the fast disks will finish first but the entire operation will have to wait for the slow disk. This not only leads to a slower transaction, but also wastes the bandwidth of the faster disks. It makes sense then to attempt to write more data to the faster disks and less data to the slow disk. If the proper amount of data is written to each, all of the disks should be done with their share of data at the same time. This would reduce the total time for the operation in the case where one of the disks is slower than the rest.

The difficulty is in determining the correct amount of data for each disk. Whatever mapping strategy is used to write data out, must make it possible to read that data again later. This requirement would make dynamically determining where to put data very difficult because the mapping routine must have some way of recognizing where data written long ago was placed. As a result, this paper only examines static techniques for determining where to place data. The performance of the disks is determined once at the initial build of the RAID device and then this information is used through-out the life of the RAID device.

2.2 Static Adaptive Implementations

To try and solve the problem of how much of a files data is the right amount of data to write to each disk in the system, three different static techniques were implemented. All of these techniques test each disks’ bandwidth by writing and then reading a large amount of data to/from each disk and timing the operations. The amount of data written to each disk for an individual file is then based on this information. This test is run at the creation of the RAID device and remains the same for the life of the device.



array_size = 10
disk-0_performance = 2
disk-1_performance = 4
disk-2_performance = 1
disk-3_performance = 3

Next Logical File Block to Write Out = 13
virtual_index = 13 % 10 = 3
real_device = virtual_device_array[3] = 1
cycles = 13 / 10 = 1
cycle_start_point = cycles * device_performance
= 1 * 4 = 4
previous_non-same_entries_in_array = 2
offset_for_block = virtual_index - 2
= 3 - 2 = 1
disk_sector = cycle_start_point + offset_for_block
= 4 + 1 = 5

Figure 2: **Example of Mapping Strategy for Each of the Statically Adaptive RAID-0 Techniques**

The performance information is used by all three of the techniques to create an extra level of indirection that is referred to as the *array of virtual devices*. This array consists of one hundred elements where each element contains the identity of a real disk device. Each index of this array can be thought of as a virtual device. The reason for a one hundred element array is the performance of each drive is as it relates to the other drives in the system—with the sum of all performances equaling 100%. This virtual devices array now makes mapping logical blocks to physical devices only slightly more difficult than the original method.

All three implementations perform their mapping strategies in exactly the same way. It is the building of the virtual device array where the three techniques differ. Section 2.2.1 discusses the mapping strategy and Section 2.2.2 describes the way each of the three techniques builds its virtual device array.

2.2.1 Mapping Strategy

When the mapping function is presented with a logical block to place on the RAID device, it modulos the block number by the number of virtual devices (100 for all techniques presented here). The result of this operation is then used to index into the virtual devices array. The value contained at this index is the real device where the block of data should be placed.

Finding the actual sector on the disk to write to is slightly more complicated than the original method of doing integer division between the block number and the number of disks. For the static techniques presented here, the sector is determined by first calculating how many complete cycles have been made through the virtual device array. This is done by taking the integer division of the logical block number and the size of the virtual array (again, 100 in all cases). This number of complete cycles through the virtual array is then multiplied by the relative performance of the actual device to access. The last thing to do is determine how far through the current cycle the logical block number is. This is done by taking the virtual device index and determining how many proceeding entries in the array contain a reference to the same physical device. The number of identical entries is then subtracted from the virtual index. This number is added to the product of the number of complete cycles and the relative performance of the device.

To make the above mapping strategy more concrete, let us examine a simple example. Figure 2 shows a virtual device array for a RAID system with 4 disks. To simplify the example, the array in this example is only 10 elements long instead of the 100 elements used by the techniques described in this paper. Underneath the array is the information about what logical blocks are contained on each disk. The squares in each column represent a physical sector on a disk. The disk number is located at the top of the column. An empty square means no data is yet stored on that sector of the disk, while a number in a square represents which logical block of the file is residing on this physical block of the device. For the sake of simplicity, this diagram assumes logical and physical blocks are the same size (this is not true in the actual system).

2.2.2 Building the Virtual Device Array

The heart of these new software RAID techniques is the virtual devices array. This array is built when the RAID device is created and remains the same for for the life of the device. As the previous section showed, once this array has been built, all three techniques work exactly the same. How each technique builds this array is the topic of this section. Discussion on how each technique performs will be deferred until Section 3.

The first technique, which will be referred to as RAID0-SA1 (RAID0-Statically Adaptive technique 1), is the simplest of the three. It simply looks at the relative performance of each disk in the RAID device and places consecutive entries for it in the virtual devices array. The example virtual device array given in Figure 2 is actually an example of how four devices would fill up an array for the RAID0-SA1 technique.

Next we examine the implementation of RAID0-SA2. In this implementation, consecutive elements in the virtual

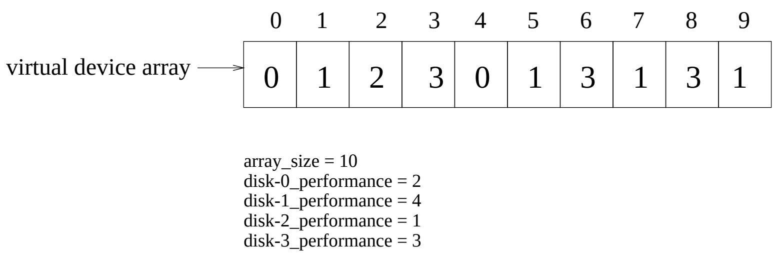


Figure 3: **Example of Virtual Device Array for RAID0-SA2 Technique**

device array alternate between each of the disks in the system. Once a disks performance has been *used up* it is no longer consider in the rotation and will not appear again in the array. One can think of this as having one bucket for each disk. Someone looks at the relative performance for a particular disk and puts that many balls into the disk's corresponding bucket. Then one ball is pulled out of the first bucket and placed on the ground. Then the same is done for buckets 2, 3, and 4. Then the process is repeated. When a particular bucket runs out of balls, the process continues but without that bucket being used in the rotation. Figure 3 shows an example of this as it pertains to the virtual device array.

Finally we come to the last technique developed—RAID0-SA3. This technique is based on the scheduling policy known as striding.[1] This technique starts each disk out with a certain value called a *pass* and a specific *stride*. The pass and the stride are initially both set to one divided by the relative performance of each disk. The first entry in the array is then chosen to be the disk with the lowest pass number. This disk's pass is then incremented by its stride. The entire process is then repeated. For the technique presented here, this process is repeated 100 times—giving each disk a number of entries in the virtual device array that is equal to its relative performance. An example of this technique can be seen by examining Figure 4.

All of these techniques will place the same amount of data from a file on each disk, but it is the order in which they place that data that will be critical to the overall performance of the system. RAID0-SA1 should appear the least productive of the three because it fails to take advantage of the parallelism of the multiple disks. It will place a large amount of consecutive data on one disk, then the next disk, and so on. This should make RAID0-SA1 perform in-line with a single disk—substantially worse than the other two techniques and the RAID0-original.

One would expect RAID0-SA2 to perform considerably better than RAID0-SA1 because it takes advantage of the parallelism of the system by alternating where blocks of file data are placed. At first glance, one might also expect it

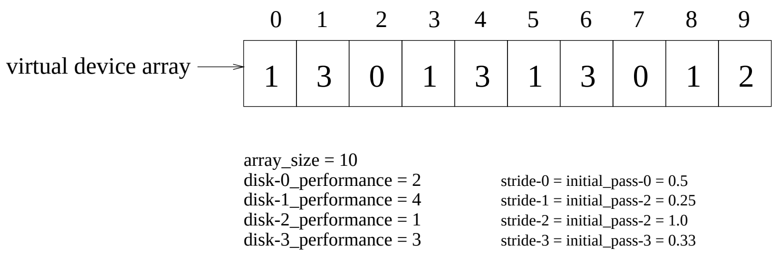


Figure 4: **Example of Virtual Device Array for RAID0-SA3 Technique**

to perform better than RAID0-original because a disk with higher performance will have more entries in the virtual device array. This second technique will insure that a faster disk gets written to more often. However, on closer inspection it should be realized that, with the exception of the points in the array when a disk gets dropped from the rotation, it looks exactly like normal striping. After a disk gets dropped from the rotation, the technique is simply striping across fewer disks. There could potentially be a slight benefit from this technique but it can be expected to be very slight—if it improves performance at all.

The last technique developed, RAID0-SA3, should alleviate the problems of the first technique because it is striping file data across the disks in a much more intelligent manner. RAID0-SA2 is weighted towards the back end. This means that if a disk requires more data it will have many more entries at the end of the virtual device array. In fact, all the ”balls” the last disk has left in its ”bucket” when the others have run out will get put at the end of the array. In RAID0-SA3 however, a disk that requires more data will have extra entries through-out the virtual device array—not just at the end of the array.

3 Results

3.1 System Configuration

The following three static adaptive techniques were implemented on a machine running dual Intel Pentium processors with five IBM 9.1 GB SCSI hard drives. Each hard drive has 16 different zones that produce decreased bandwidth between the disk and the host as the zone gets closer to the center of the disk. Additional information about these disks can be found in [2]. One of the five disk drives is used for the root partition, and the other four were used to conduct the experiments presented in this paper.

Each of these four drives was partitioned into several different 128 MB segments. These segments were scattered

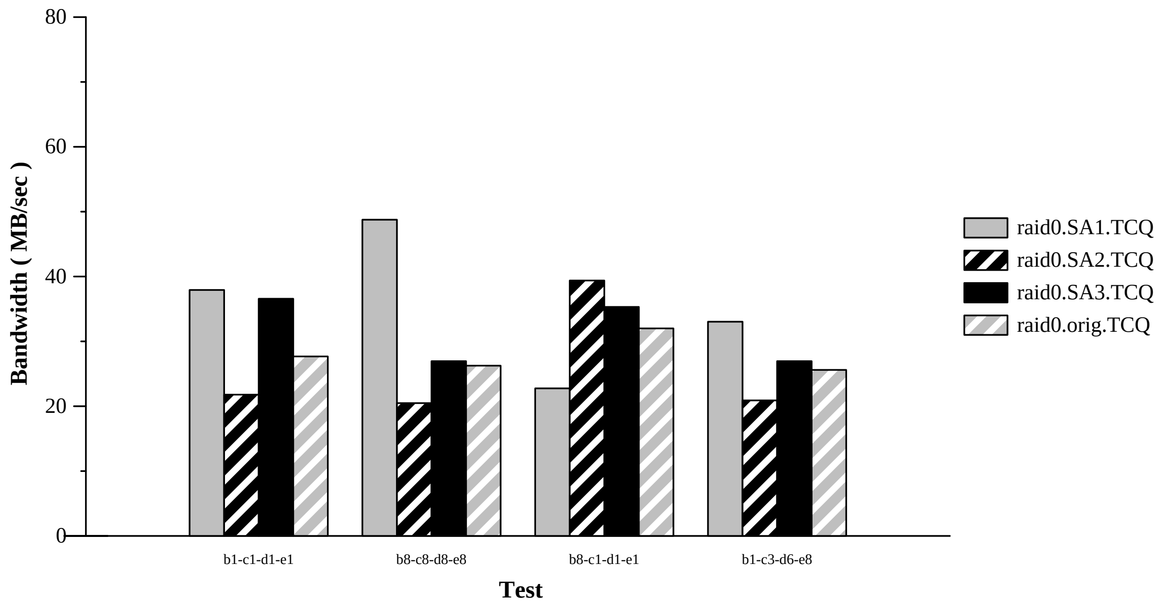


Figure 5: **Read Bandwidth Results for 4 KB block size and 1024 KB file size**

over the different zones in such a way as to simulate multiple disks with differing performance. All the tests presented here utilize a four disk configuration. The notation for each of these configurations works as follows: a letter is used to indicate a specific disk and a number is used to indicate the partition. The higher the number, the lower the performance of the partition. As an example, b8-c1-d1-e1 represents a configuration with disk b utilizing the slowest partition and the remaining disks using the fastest partition.

The first configuration used in the experiments presented here uses the fastest partition from each disk (b1-c1-d1-e1). The second configuration uses the slowest partitions (b8-c8-d8-e8). The third configuration uses the slowest partition on one disk and the fastest partition on the other three disks (b8-c1-d1-e1). The final configuration examined has each disk using a different performance partition from each disk (b1-c3-d6-e8).

Lastly, all of the read and write tests were done by invoking the read and write system calls followed by an fsync system call. The hope here is to force the data out to or in from the disk instead of simply copying it into and reading from system buffers. One thing to note is that the disk itself is capable of buffering up to 4 MB of data.

3.2 Analysis of Results

One of the most seemingly interesting graphs is Figure 5. This shows the read bandwidth for a 1 MB file for each of the different RAID techniques. This graph shows all sorts of strange results. RAID-original performs better with

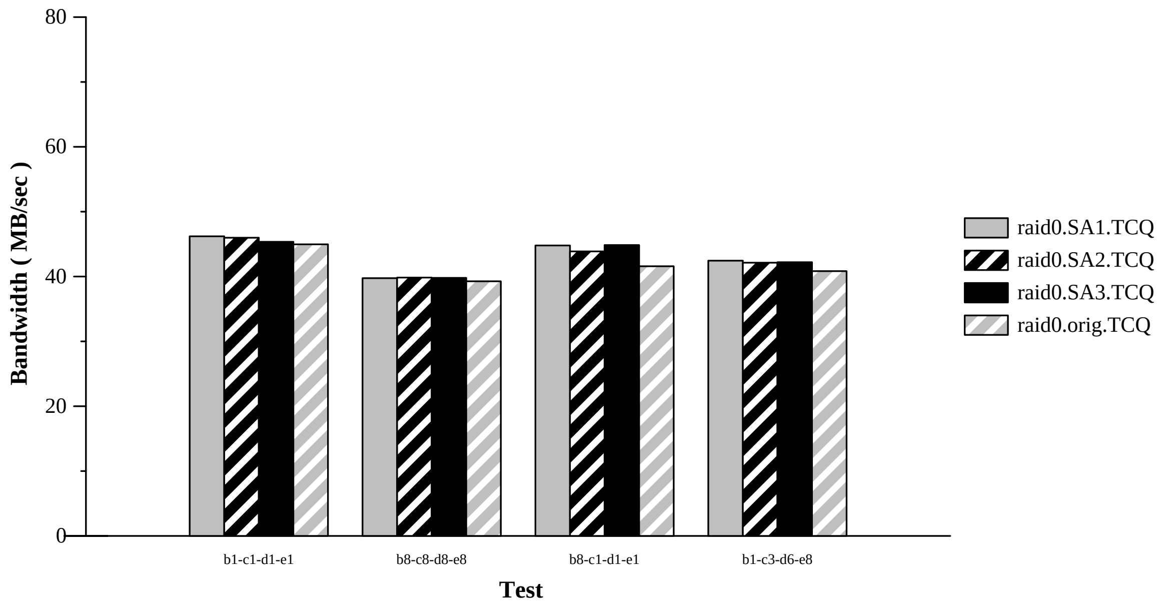


Figure 6: **Write Bandwidth Results for 4 KB block size and 262144 KB file size**

one slow disk than it did with all fast disks, RAID0-SA1 is drastically outperforming all the other techniques for the first two configurations, and RAID0-SA3 outperforms RAID0-original for the all fast configuration even though they should be doing the exact same mapping. The reason for this is the 4 MB buffering at each disk. Depending on how the data is prefetched and what operations were performed in between the writing of the data and the reading of it, this graph—and all of the graphs for small files—are not very useful for examining the performance of the different RAID techniques. As a result, the rest of this section will mostly focus on the read and write tests for a 256 MB file.

So how accurate were the predictions made in Section 2.2.2? Some of them were correct and others were completely wrong. For example, the original intuition stated that RAID0-original and RAID0-SA2 should perform almost identical. Figures 6 and 7 show this is indeed the case. This is due to the fact that RAID0-SA2 is basically doing striping with fewer disks involved in the striping process at the end of the virtual device array.

On the other side of the coin, we can see that the results presented in Figures 6 and 7 show that RAID0-SA1 performs at about the same level as the rest of the techniques—not much worse, as predicted. This is because the large file is being written or read in its entirety. Because the disks cannot process the first request before the subsequent requests arrive, all of the techniques doing adaptive mapping can be expected to perform about the same. This is indeed what we see for most of the tests presented here. One would expect different results if we began to access random data in the file, especially if the size of the data being accessed were larger than the chunk size the RAID

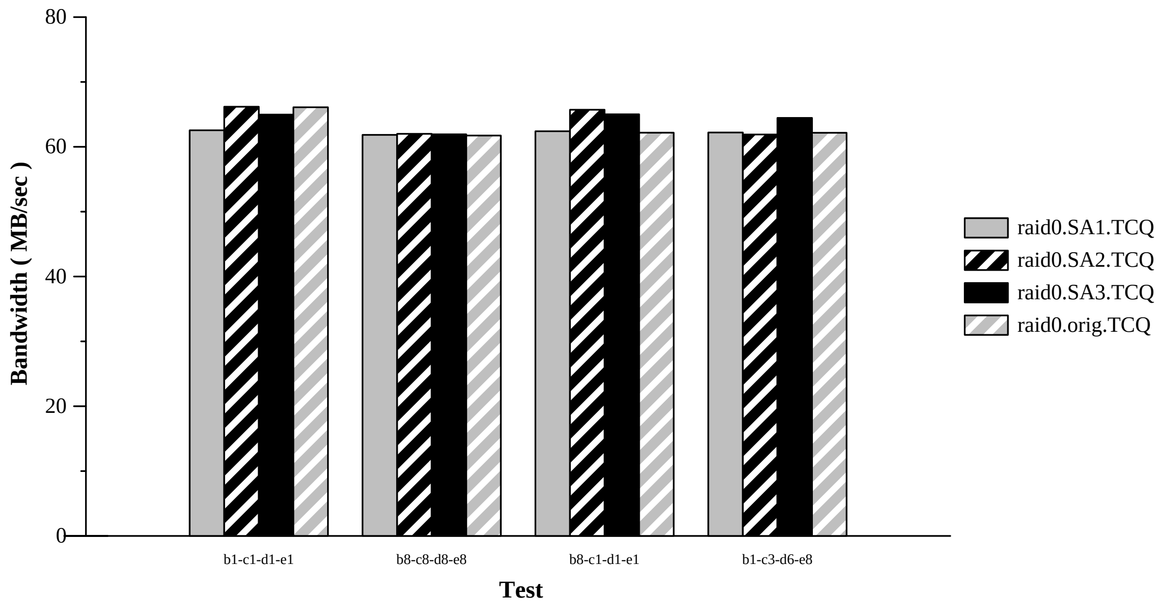


Figure 7: **Read Bandwidth Results for 4 KB block size and 262144 KB file size**

device works with (4 KB for Figures 6 and 7). In this case, the parallel capabilities of the disk should come into play. This is discussed further at the end of the section.

Perhaps a more interesting finding of this research comes from looking at [2]. This is the information on the SCSI disks used in these experiments. It shows us that the sustained bandwidth from the disk buffer to the media is rated at just over 20 MB/s for zone 1 and 15 MB/s for zone 16. We can see in Figure 8 that for a 64 KB file the four disks combine to produce a write bandwidth of around 5 KB/s. For a 1 MB file it increases to approximately 25 MB/s (Figure 9). These bandwidths are substantially less than the 60 MB/s to 80 MB/s that would be achieved if the bandwidth of the disk alone was responsible for the delay. Obviously the overheads of seeking for the data and handling it once it has been transferred to the host play a major role for smaller files.

What is interesting is the results for writing a 256 MB file. We see that the maximum bandwidth is about 46 MB/s for a write using configuration b1-c1-d1-e1 (Figure 6). This is substantially lower than the 80 MB/s that one would expect considering that each device is capable of handling up to 20 MB/s in this configuration. Figure 7 shows that reads perform significantly better but still top out at about 65 MB/s. The Adaptec AIC-7896 Dual Channel Ultra II SCSI controller used by this system is rated at 80 MB/s. [3] So it appears that the culprit for the low bandwidth is not due to the bus but rather due to the overheads involved when reading or writing data. The sources of this overhead when doing a read are the time to enter and leave the OS on the read system call, read the file i-node from disk, seek

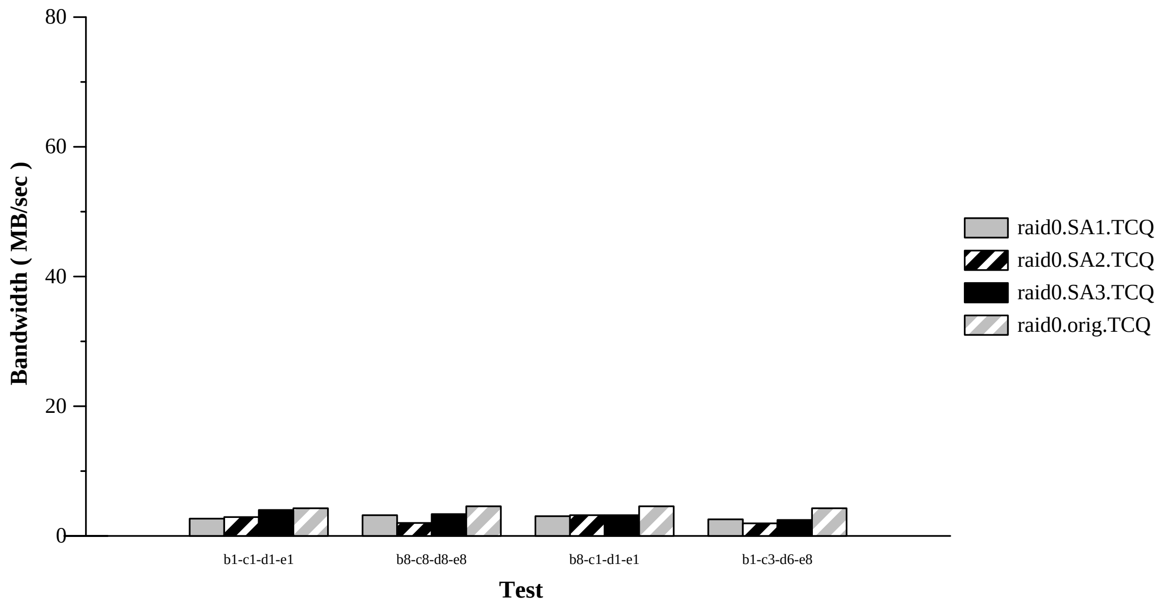


Figure 8: **Write Bandwidth Results for 4 KB block size and 64 KB file size**

to the proper location on the disk, and copy the data into the user space. Of these four, the copying of data should be the only one that adds any significant time to the read operation for a large amount of data.

Writes, on the other hand, appear to incur substantially larger overheads. The additional sources of overhead could come from the time to determine which sectors on disk to add to a file, doing a second access to the i-nodes to modify metadata, and writing the inodes back to the disk. It does not seem reasonable that these factors would be creating a large amount of overhead for large files. This paper proposes the reason for this overhead is actually do to the operating system buffering of data. While this provides great help under normal operation, when doing an fsync (or any other type flush to disk), the system first waits for the data to be copied to the OS buffers, then copies the data out to disk. There is no overlap of these two operations because under normal circumstances the user would not care that the data is not actually being copied to disk.

For a read, the normal case is that data is accessed immediately. As a result, the operating system brings the data directly from the disk into the user's buffers. In a sense, the transfer of data from disk and the copying into the user's buffers are overlapped. Since the bandwidth for the four disk configuration is not reaching 80 MB/s, it appears that this copying is not completely overlapped with the data transfer. But it does appear to be overlapped to a much greater extent than the writes. Further research is needed to determine if this is indeed the source of the low write bandwidth.

Bandwidth tests of the different zones show that partition 1 on each disk is about 20% faster than partition 8. This

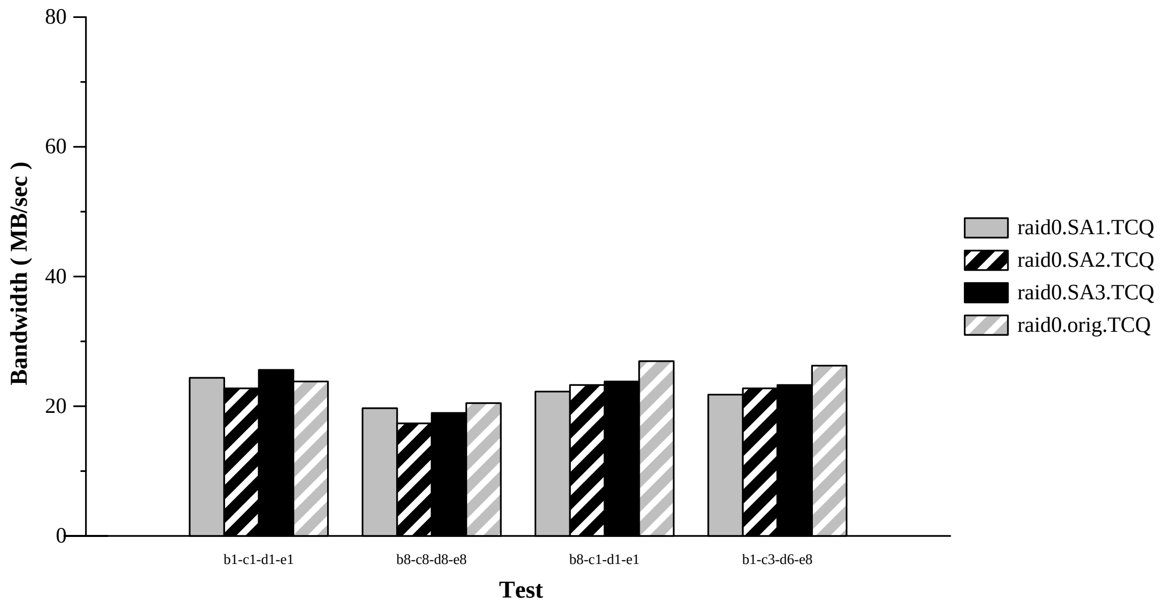


Figure 9: **Write Bandwidth Results for 4 KB block size and 1024 KB file size**

test was done by conducting a write and then read on the raw device and then measuring the time of the operations. Despite this substantial difference in bandwidths, RAID0-SA3 was only able to achieve about a 7% increase in performance over RAID0-original for reads and writes when there were 3 fast disks and one slow disk. The preceding two paragraphs help to explain why this is the case. While the SCSI bandwidth may not be getting completely saturated for either the read or the write case, it appears that the system is being saturated due to a combination of the data transfer and the time to copy data between buffers. If this is the case, then the performance from better mapping of the data onto disk is being limited by the overheads in the system. It should also be clear that if the original RAID system *were* capable of achieving the 80 MB/s that the SCSI bus offers, any improvement in mapping strategy would be wasted for the reading of large files.

An interesting extension to this research would be to examine how the different RAID techniques compare when accessing a smaller amount of data from a file but where buffering of the data in either the operating system or the disk cache does not come into play. An example of this would be accessing random pieces of data from a file. If the file is sufficiently large—say 500 MB or 1 GB—and the number of accesses sufficiently high, we should be able to achieve our goal of accessing data that is not being buffered. By keeping each individual read at a sufficiently small size we could insure that the system resources do not become saturated as in the case of reading a large file. And by making sure the size of the data read is larger than the RAID chunk size (twice the chunk size for example), we can insure that the parallelism of the RAID system is utilized. This experiment is left as future work.

4 Future Work

One of the most important items for future work was discussed in the last paragraph of Section 3.2—running experiments that involve accessing data randomly from a large file. Of equal importance is examining the system to find out exactly where the time for a read or write call is spent. This would be very beneficial in confirming or contradicting the theory put forth in Section 3.2 about why the read and write bandwidths were lower than expected.

It is also important to re-run the experiments presented here with a different number of disks in the configurations. More disks would surely saturate the system while fewer disks would lessen the demand on the bandwidth. Again, this would be of great benefit in determining the validity of the the conclusions reached in Section 3.2.

Designs have been done to implement a statically adaptive RAID system that considers the differing bandwidth between zones for a given disk. This technique has not been implemented at this time but is the logical next step in the adaptive RAID techniques.

5 Conclusion

All of the research presented in this paper focuses on the effects of various RAID-0 techniques for placing large, contiguous amounts of data onto multiple disks. Presented are the results for three new, statically adaptive RAID-0 techniques. The results for the traditional software RAID-0 implementation are also provided for comparison. The most striking result is that the overhead involved in reading in or writing out an entire, large file is quite substantial. The SCSI bus in the system tested is capable of handling 80 MB/s of data transfer, but the highest value obtained in this research was 65 MB/s for a read operation. Write operations sustained a substantially worse overhead penalty. The best performance for any write operation was slightly more than 45 MB/s.

Overheads such as entering and leaving the operating system, reading and writing i-node data, and calculating the location of the data on disk all contribute to this overhead. These overheads can be significant when dealing with small files. However, when working with large files, the major culprit for the reduction in overall bandwidth appears to be memory copy operations as data is copied between user and system space. Reads suffer less because the copying operation is overlapped with the transfer from the disk. Writes, on the other hand, copy all of the data into a system buffer and then send the data to the disk. While this helps to drastically improve performance under normal operations, it hurts the overall performance if the user code is waiting for the data to be flushed to disk.

One other interesting observation is that the order in which data blocks are sent to the disks has little effect on the performance if the total number of blocks being sent is large. This means a technique that sends the first 25% of all blocks to the first disk, the next 25% of the blocks to the second disk and so on (RAID0-SA1); will see roughly the same performance as a technique that alternates blocks to disks at a much more fine grained level (RAID0-SA2 or RAID0-SA3). This is a result of working with large amounts of data, the speed at which requests can be sent out by the CPU, and the rate at which requests can be handled by the disks. If a disk is not capable of finishing one request before the next few arrive, then the amount of data being written to each disk becomes the important factor and not the order in which the data is presented.

Lastly, there is some interesting research left to do. This involves investigating the effects of much smaller accesses to a file, testing systems with different numbers of disks, calculating exactly where the read and write system calls are spending there time, and implementing another statically adaptive technique that accounts for the differences in bandwidth across multiple zones of a single disk.

References

[1] William Weihl Carl Waldspurger. Stride scheduling: Deterministic proportional-share resource management. Technical report, MIT Laboratory for Computer Science, June 1995.

[2] IBM. *ULTRASTAR 9LZX/18ZX Hardware/Functional Specification*, December 1998.

[3] SuperMicro, http://www.supermicro.com/PRODUCT/MotherBoards/440GX/S2DG2.htm. *SUPER S2DG2*.