RAID4S-modthresh: Modifying the Write Selection Algorithm to Classify Medium-Writes as Small-Writes *

Optimizing Medium-Write Performance on RAID4S †

Michael Sevilla‡
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
msevilla@soe.ucsc.edu

Rosie Wacha§
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
rwacha@soe.ucsc.edu

Scott A. Brandt
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
scott@soe.ucsc.edu

ABSTRACT

The goal of this research is to improve the performance of the RAID4S system. For RAID4S, the throughput slope of small-writes, the magnitude of medium sized writes on both sides of the small/large-write selection threshold, and the speedup of small-write performance suggest that medium-writes using the large-write parity calculation will perform better if they use the small-write parity calculation instead. RAID4S-modthresh uses a modified write selection algorithm which only affects medium-writes and keeps the parity calculation and corresponding throughput for small-writes and large-writes intact. RAID4S-modthresh is implemented into the software RAID controller module in the Linux source. Results show a speedup of 2 MB/s for medium-writes without imposing performance or reliability penalties on the rest of the system. This paper also presents other interesting projects for future work in modifying the software RAID controller to improve performance.

1. INTRODUCTION

RAID systems help bridge the gap between processor and cache technology by improving either performance or reliability. Different RAID levels utilize a variety of striping, mirroring, and parity techniques to store data. This allows the user to select the RAID level that best fits the application's cost, performance, complexity, and redundancy requirements [9]. RAID schemes based on parity enhance storage reliability by managing a recovery parity disk but parity calculations degrade performance [1]. This is especially apparent when performing small-writes. A key goal in this research is to improve upon the RAID4S [10] system so that the overhead of parity based RAIDs can be minimized even further in an effort to approach RAID0 schemes [6]. Figure 1 shows the improved throughput measurements of the new write selection algorithm implemented in this project.

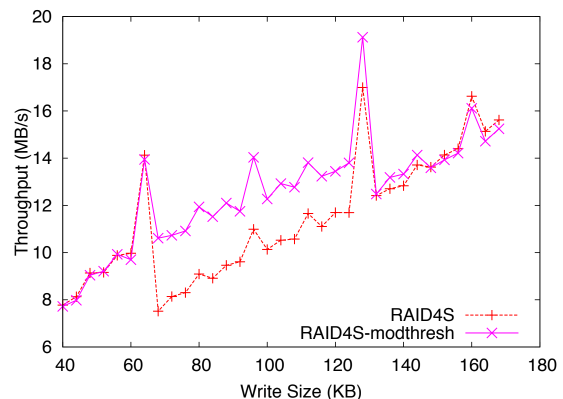

Figure 1: RAID4S-modthresh provides a noticeable speedup over RAID4S when performing medium-sized writes. The RAID4S-modthresh write selection algorithm classifies medium-writes as small-writes, so it uses the small-write parity calculation and enjoys the benefits of RAID4S for medium-sized writes.

When calculating the parity for a parity-based RAID, the RAID controller selects a computation based on the write request size. For maximum efficiency, large-writes, which span at least half the drives, must first compute the parity by XORing the drives to be written with the remaining drives. The large-write parity calculation is:

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

9*(Template taken from ACM_PROC_ARTICLE-SP.CLS. Supported by ACM.

†Inside the `handle_stripe_dirtying5()` function.

‡Enrolled in CMPS 221: Advanced Operating Systems.

§Mentor for project.

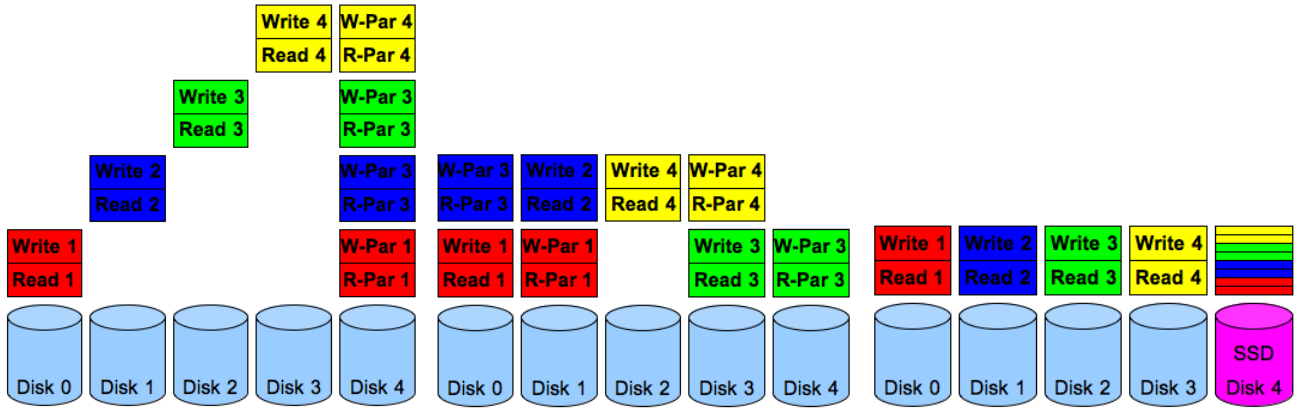

Figure 2: The traditional RAID4 (left image) scheme has a bottleneck at the parity drive. As a result, small-write operations cannot be performed in parallel. RAID5 (middle image) rotates the parity disk, so the rest of the system is not reduced to the I/O bandwidth of 1 disk. Small-writes can be performed in parallel if the 2 small-write requests have different parity disks and write disks. RAID4S (right image) can perform small-writes in parallel if the write disks are unique because the SSD has a much faster I/O bandwidth. If the parity drive has a speedup factor of N, then in the time that it takes one regular disk to read or write the parity can perform N disk read or writes. Diagram idea taken from [10].

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

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

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

where d is the data on the drive(s) to be written and p is the parity drive. After the parity is computed, the new data is written to the drives and the new parity is written to the parity disk. [1, 9, 10]

Small-writes require $2(m+1)$ disk I/Os while large writes always require $N+2$ disk I/Os. The instance where four accesses are needed for each small-write instead of two is known as the small-write problem[8]. This is most apparent with small-writes, whose throughput, compared to the same sized write for mirrored RAIDs (i.e. RAID0), is severely penalized [10] for every small-write. This bottleneck can be seen in Figure 2.

Traditionally, RAID4 have been dumped in favor of RAID5 because in RAID4, as noted in [9], the parity disk is a throughput bottleneck when performing small-writes.

RAID4S is a RAID4 configuration comprised of data HDDs and 1 dedicated parity SSD which aims to overcome the parity bottleneck in an effort to improve small-write performance . With this scheme, small-write performance increased because the SSD parity drive can process small-write reads and writes much faster. This allows small-writes on data disks (the rest of the RAID system) to be performed "in parallel". By simply isolating the parity on the SSD, RAID4S provides excellent performance while still achieving the same reliability of parity-based RAIDs. Figure 3 shows the increased throughput of RAID4S over RAID5 and RAID5S. RAID4S provides nearly 1.75 times the throughput over RAID5 and 3.3 times the throughput for random writes under 128KB (small-writes) [10]. The overall performance speedup is a function of the speed of the SSD relative to the other disks and the percentage of small writes performed. RAID4S alleviates the RAID4 parity bottleneck, offloads work from the slower devices, and fully utilizes the faster device.

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

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

This research improves the performance of RAID4S by introducing a new write selection algorithm to the software RAID controller. The new write selection algorithm modifies the threshold of RAID4S to allow the parity of smaller large-writes (medium-writes) to be calculated using the small-write parity calculation. The new system that uses the new write selection algorithm has been named **RAID4S-modthresh**. This modification to the RAID controller only affects medium-writes and keeps the parity calculation and corresponding throughput for small-writes and large-writes intact. Furthermore, changes to the write selection algo-

rithm do not affect the reliability or cost of the system since the hardware is not being changed.

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

The rest of the paper outlines the steps taken to implement RAID4S-modthresh and analyzes its resulting performance. Section 2 discusses the Background and Methodology of the implementation and focuses on the theoretical performance implications of modifying the threshold. The section also discusses the background for the code and kernel modules that is necessary for understanding the implementation details. Section 3 describes the implementation in detail and shows the exact changes that need to be made to implement RAID4S-modthresh. Section 4 shows how RAID4S-modthresh is tested and provides a brief analysis of the results. Section 6 acknowledges related works with similar goals but different implementations.

2. METHODOLOGY AND BACKGROUND

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

2.1 Motivation

To baseline RAID4S write performance for 4KB block aligned random writes, XDD was configured to make various write sizes, in powers of 2, ranging from the minimum block size to the RAID stripe size (4KB, 8KB, 16KB, ..., 128KB, and 256KB). RAID4S is configured with 4 data drives and 1 parity SSD using 256KB stripes. In this setup, each drive receives 64KB chunks from the data stripe. Figure 3 shows the throughput measurements for the block aligned random writes.

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

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

2.1.1 *Throughput vs. Write Size Slope*

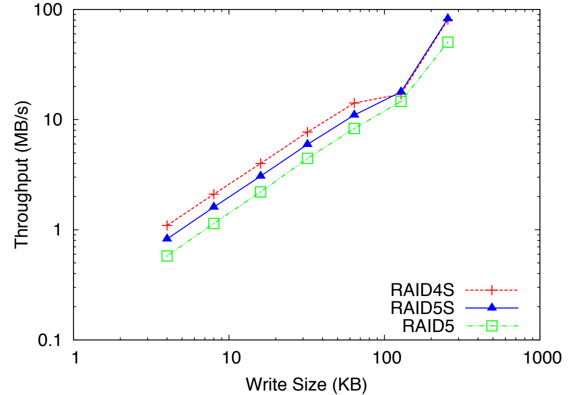

Figure 3: The throughput of RAID4S compared to RAID5 and RAID5S. Using a parity drive significantly improves the performance of small-writes because of the parallel capabilities that the SSD provides. The curve of the small-writes and the magnitudes of the 64KB and 128KB writes suggest that changing the 128KB large-write parity calculation into a small-write parity calculation can improve the performance even more.

The shape of the curves in Figure 3 suggested that for RAID4S, the parity for 128KB writes should be calculated using the small-write parity computation instead of the large-write parity computation. The slope of RAID4S approaching 64KB is different than the slope leaving the 64KB write-size because the write selection algorithm is now using the large-write parity computation. This sudden negative change in slope implies that a medium-sized might perform better if it was calculated as a small-write.

The interpolation of the RAID4S curve in Figure 3 also suggests that small-writes will outperform large-writes for medium-write sizes. The large small-writes have a lower slope but higher performance than smaller large-writes (medium-writes). It should be noted that interpolating the small-write curve much further beyond the halfway point (128KB) would result in lower throughput than the large-write performance despite the higher initial performance.

2.1.2 *Magnitude of Medium-writes*

The magnitude of the throughput of large small-writes and small large-writes is very close. Figure 4 is a subset of the throughput values shown in Figure 3 and is set on a non-log scale. The figure shows that the last small-write of RAID4S, 64KB, has a throughput that is only about 3 MB/s slower than the first large-write, 128KB. Since small-write performance for RAID4S increased dramatically, it is logical to predict that the write after the last small-write may enjoy some speedup over its predecessor, which should improve its throughput over its current large-write parity calculation throughput.

2.1.3 *Extending Parallelism to Medium-Writes*

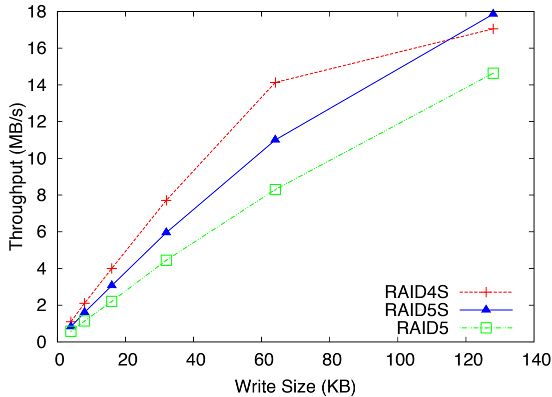

Figure 4: RAID4S shows similar throughput for 64KB and 128KB sized writes. This suggests that the performance improvement of RAID4S will make 128KB sized writes faster using the small-write parity calculation.

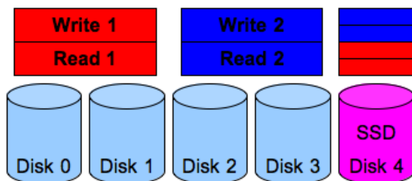

Figure 5: RAID4S classifies medium sized writes as large-writes because a read-modify-write will need to read 3 disks while a reconstruct-write only requires 2 disk reads. RAID4S-modthresh classifies medium sized writes as small-writes because small-writes can be processed in parallel on a small-write RAID4S system.

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

Figure 5 shows how medium-writes can be processed in parallel. RAID4S-modthresh would attempt to take advantage of the improved small-write performance and not the fact that writes can be processed in parallel. Theoretically, the large-write parity calculation would perform equally as well as the situation in Figure 5 but since RAID4S has shown such an extreme throughput enhancement for small-writes, it seemed worth it to try changing the write selection algorithm. Since many of the small-writes can perform faster, it may open up more opportunities for medium-writes to process in parallel alongside small-writes.

2.2 The raid5.c code

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

2.2.1 State Transitions

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

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

2.2.2 The Dirty State - Write Selection / Scheduling

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

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

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

Recall that a r-m-w reads the old data, so checking to see if the device is going to be written to is equivalent to asking if the device in question needs to be read for a r-m-w. Note that the parity drive will always increment `rmw` since the parity drive needs to be read for a read-modify write. Figure 6 shows a model of the typical disk counting for a 128KB and

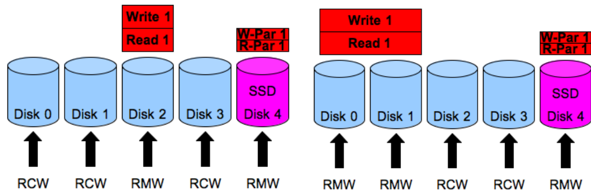

Figure 6: Counting the drives that would be needed for a r-m-w and a r-c-w for a 64KB write (1 drive) and a 128KB write (2 drives). Notice that any drive that is not marked as a drive that needs to be read for r-m-w is automatically marked as a r-c-w read drive. The parity drive is always marked as a r-m-w drive.

64KB writes. The debugging output for the r-m-w and r-c-w calculations is shown in Figure 7. The debugging output first shows how the `handle_stripe_dirtying5()` function first examines the requests of each disk (states) and then it shows how the `rmw` and `rcw` variables are incremented.

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

This threshold is set by two conditionals in `raid5.c`. They have been reproduced in Figure 8

Once the proper drives are read, a reconstruction is scheduled. This has a pipelining effect because new write requests can be considered while older write requests are being passed to the write scheduler. Once the write type has been determined, the function locks the drives that need to be read, which also stops the scheduler from scheduling any other write requests.

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

2.3 Loadable Kernel Module (LKM)

The Ubuntu Linux 2.6.32 kernel is used for the implementation and testing of RAID4S-modthresh. The `raid5.c` compiles into a .ko file and is linked into the kernel through the

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

Figure 8: The raid5.c write selection algorithm threshold.

`raid456.ko` kernel module. After stopping all the RAIDs, the LKM is loaded into the kernel by first removing the old `raid456.ko` and loading the new `raid456.ko` from the offline source tree.

2.3.1 *Debugging*

All debugging is written to `/var/log/kern.log` and can be accessed either by reading this file or through `dmesg`. Scripts are used to extract the relevant debugging output for each run. Typical debugging output is shown in Figure 7 and the output consists of the disk buffer states, the stripes being examined, the `rmw`/`rcw` counts, the stripe operations being performed, the states of various buffer stripe flags, and the type of writes being performed. The debugging is also used to determine which functions are being called when and how the `handle_stripe()` method chooses and schedules stripe operations.

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

3. IMPLEMENTATION

This project modifies the write selection algorithm in the dirty state in order to improve the small-write performance

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

Figure 7: Debugging output for a 64KB write on RAID4S. The write selection algorithm chose a r-m-w parity calculation.

enhancements of RAID4S. This is done by changing the threshold in the `handle_stripe_dirtying5()` function of the `raid5.c` loadable kernel module. The `handle_stripe_dirtying5()` function handles the write selection algorithm and passes the chosen write to the `schedule_reconstruction()` function, as discussed in Section 2.2.2.

Two implementations are made to change the write selection algorithm:

1. RAID4S-modified: schedule only small-writes
2. RAID4S-modthresh: schedule more small-writes

For both implementations, the r-m-w and r-c-w disk counting, discussed in Section 2.2.2, needs to be left intact for the `schedule_reconstruction()` method. The `schedule_reconstruction()` method relies on the `rmw` and `rcw` variables for write selection and will not proceed until the proper disks have been read for the parity calculation. The indication that these reads are in flight occurs when either the `rmw` or `rcw` variable is 0, since the drives have been locked for reading.

Modifications are made to the two conditionals shown in Figure 8, which specify the drives that need to be read.

3.1 RAID4S-modified: Force All Small-Writes

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

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

1. the correct drives are read for the parity calculation
2. the write scheduler always calls for a r-m-w

To satisfy condition (1) changes are made in the second `if` condition (Block 2) of Figure 8 so that in the r-c-w calculation, the drives to be written are read for the parity computation. With this change, the drives to be written are always read for the parity computation, independent of the write type chosen. To satisfy condition (2), the write scheduler parameter is forced to call a r-m-w by changing the `rcw == 0` parameter to 1. The following two lines of code achieve these requirements:

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

// ...

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

// ...

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

 // ...

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

//...
```

If either condition (1) or (2) of the first implementation are skipped, the algorithm behaves incorrectly. Skipping condition (1) causes a crash because when the disks to be written are read, there is no data in the buffer stripe. Skipping condition (2) does not force the algorithm to choose r-m-w

for every write type. Because of these conditions, it is not sufficient to change the `if` statements to always force the program flow into Block 1 of Figure 8.

3.2 RAID4S-modthresh: Modify Threshold to Schedule More Small-Writes

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

The code needs to be modified so that writes that span half the drives are classified as small-writes instead of large-writes. To accomplish this, the two conditionals from Figure8 need to be modified so that Block 1 accepts larger writes and Block 2 accepts the rest (anything that is not a small-write). Changes are made to the following `if` statements :

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

// ...

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

// ...
```

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

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

3.2.1 RAID4S vs. RAID4S-modthresh Example

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

The `rmw` and `rcw` count for a 64KB write is shown in Figure 7. In this debugging log, disk 0 has a write request. The write selection algorithm classifies disk 0 and disk 4 (parity disk) as r-m-w's because they will need to be read if the algorithm were to choose r-m-w. The remaining disks would need to read for a r-c-w. Both the original RAID4S write selection and the RAID4S-modthresh write selection algorithm will select r-m-w because `rmw` < `rcw`.

The `rmw` and `rcw` count for a 128KB write is shown in Figure 9. In this debugging log, disk 0 and disk 1 have a write request. The write selection algorithm classifies disk 0, disk 1, and disk 4 (parity disk) as `rmws`. The remaining disks would need to be read for a r-c-w. The RAID4S write selection algorithm will select a r-c-w parity calculation because `rcw` <= `rmw`. The RAID4S-modthresh write selection algorithm selects a r-m-w parity calculation as is evident by the last three lines of the debugging output. RAID4S-modthresh selects a r-m-w parity calculation because `rmw` <= (`rcw` + 1).

It should now be clear that the `rcw` variable needs to be increased by 1 to account for the parity when half the data drives are being written; the `rmw` and `rcw` values need to be skewed to account for the parity disk. In this implementation, the `schedule_reconstruction()` method is set back to `rcw` == 0 for the write-type, since we still want to execute large-writes if the threshold is overtaken.

3.3 Analysis of RAID4S-modified and RAID4S-modthresh Implementations

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

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

Figure 9: Debugging output for a 128KB write on RAID4S-modthresh. The write selection algorithm chose a r-m-w parity calculation, as is evident from the last three lines.

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

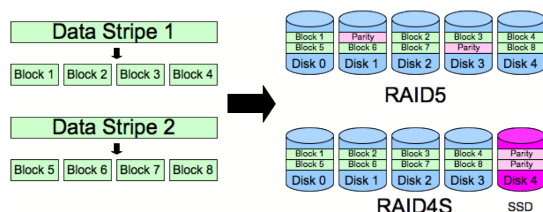

Figure 10: The RAID5 and RAID4S setups. RAID5S has the same topology of RAID5 but with the drives all replaced with SSDs. RAID4S has a dedicated parity SSD while RAID5 schemes rotate the parity amongst the drives.

4. EVALUATION AND RESULTS

To test the performance speedup of RAID4S-modthresh, random write tests are performed on three RAID configurations: RAID4S, RAID5, and RAID5S. These configurations are shown in Figure 10. The XDD [7] tool is a command-line benchmark that measures and characterizes I/O on systems and is used in these experiments to measure performance. Throughput measurements show the improved performance that RAID4S-modthresh has over RAID4S, RAID5, and RAID5S for medium-writes.

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

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

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

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

and

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

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

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

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

4.1 Block Unaligned Writes - Medium Writes

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

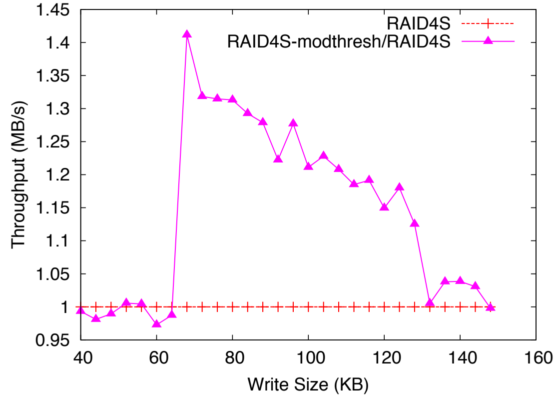

Figure 11: Comparing RAID4S-modthresh to RAID4S. The RAID4S-modthresh values have been divided by RAID4S to get a normalized speedup of RAID4S-modthresh over RAID4S.

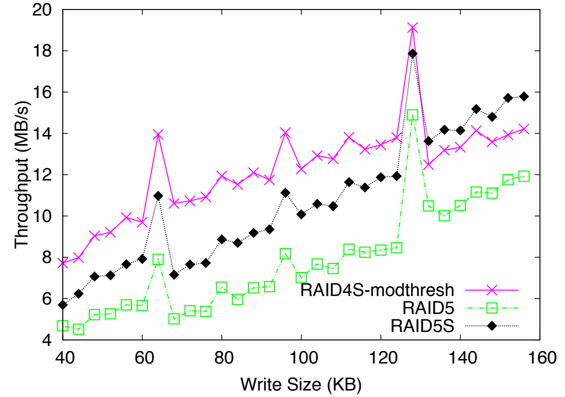

Figure 12: Comparing RAID4S-modthresh to RAID5 and RAID5S. The speedup is noticeably improved for medium-writes.

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

Figure 1 compares the throughput of RAID4S and RAID4S-modthresh for randomly generated writes in the 40KB to 168KB range. From 40KB to 64KB, the write throughputs are identical because both write selection algorithms choose small-write parity computations. Similarly, writes larger than 128KB are also identical in both RAID4S and RAID4S-modthresh because both write selection algorithms choose large-write parity computations. In this case, RAID4S-modthresh calculates $\texttt{rmw} = 4$ and $\texttt{rcw} = 1$, which does not satisfy the threshold condition for small-writes.

The most interesting results are what lies between, in the 64KB to 128KB medium-write range. In the medium-write range, RAID4S-modthresh shows a noticeable difference in the write throughput over RAID4S. The normalized speedup of RAID4S-modthresh over RAID4S can be seen in Figure 11. The throughput increases by 2MB/s for all the unaligned and aligned writes less than or equal to 128KB.

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

Figure 12 compares the throughput of RAID4S-modthresh to RAID5 and RAID5S. The results show significant speedup for RAID4S-modthresh for medium-writes over both RAID5 configurations and is finally overtaken by RAID5S after the 128KB write size. This is expected since RAID5S large-writes are faster than RAID4S because of the extra SSDs used in RAID5S. Even the 128KB write for RAID4S-modthresh compares favorably with RAID5S.

Finally, Figure 13 compares RAID4S-modified to RAID4S-modthresh. RAID4S modified alters the write selection algorithm to always select small-writes. The predicted effect is that small-write computation will perform worse than large-write computation because small-writes need to read more drives than larger write-sizes. The results show that immediately after 128KB, the RAID4S-modified throughput is still better then the RAID4S-modthresh throughput by a small margin. The RAID4S-modthresh eventually achieves higher throughput than RAID4S-modified around 164KB, but the lower performance immediately after 128KB is still peculiar. These results, like Figure 11 suggest that the small/large-write threshold can be pushed even farther to the right to realize even more of a performance gain.

4.2 Block Aligned Writes: Powers of 2

To test the overall performance of the system, XDD is configured to perform block aligned writes, which are write sizes that are powers of 2 (2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB). This experiment includes medium-writes, but unlike the previous experiment, this range also covers very small and very large-write sizes for a 256KB stripe. The block aligned write size experiments do not show as pronounced a speedup as block unaligned writes (Section 4.1) because these experiments focus on the system as a whole. The purpose of these experiments is to show that RAID4S-modthresh will not degrade the performance of RAID4S.

In Figure 14, the left graph shows the performance gain that RAID4S-modthresh experiences for 128KB sized writes compared to RAID4S. All smaller writes are the same since both systems select the same small-write computation for parity calculation. The write-performance for 128KB RAID4S-modthresh was predicted to be higher, judging from the slope of the line but the graph shows a degradation in performance relative to the expected performance. Notice that the

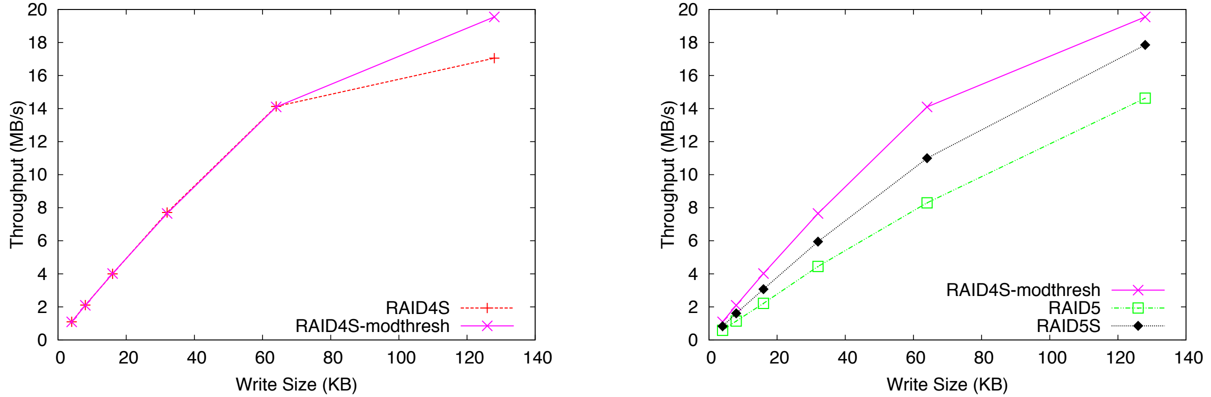

Figure 14: Comparing RAID4S-modthresh to RAID4S , RAID5, and RAID5S on a straight scale without the 256KB write-size. The 256KB write-size skewed the graph because it has such a high performance relative to the medium and small-write throughputs. For medium-writes, RAID4S-modthresh provides a noticeable speedup over the three other RAID configurations.

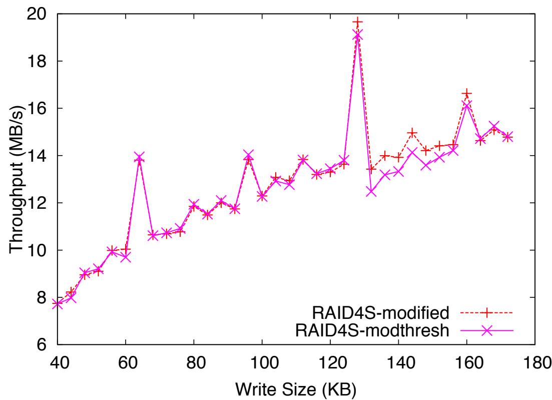

Figure 13: The throughput of RAID4S-modified (all small-writes) and RAID4S-modthresh have interesting results. It was expected that throughput would be the same for all writes up to and including 128KB but performance varies. Even more interesting is the trend immediately after 128KB, where RAID4S-modified has better performance than RAID4S-modthresh in throughput.

the throughput for RAID4S-modthresh steadily increases at a linear rate but veers off to a much lower value than interpolation would suggest. This behavior can probably be attributed to the additional disk read and its unanticipated overhead.

Figure 14 also shows the throughput speedup of RAID4S-modthresh compared to RAID5 and RAID5S. Classifying and computing the 128KB write as a small-write permits RAID4S-modthresh to enjoy more of a speedup than RAID4S when compared to RAID5 and RAID5S. Although the performance is better than RAID4S, it is still far less than expected, as discussed in the observations above.

Finally, Figure 15 shows the minimal improvement that RAID4S-modthresh has on the entire RAID system. The speedups are evident but not overwhelming. The takeway from these graphs is not that RAID4S-modthresh facilitates a large performance increase, but rather that a small performance increase can be achieved by making a small change to the RAID controller without negatively impacting the rest of the system.

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

4.3 Future Work and Implementations

In addition to showing encouraging speedups for RAID4S-modthresh, the experiments show some interesting possibilities for future work. As noted in Section 4.1, peculiarities in the comparison medium-write throughputs of RAID4S, RAID4S-modified, and RAID4S-modthresh show that the

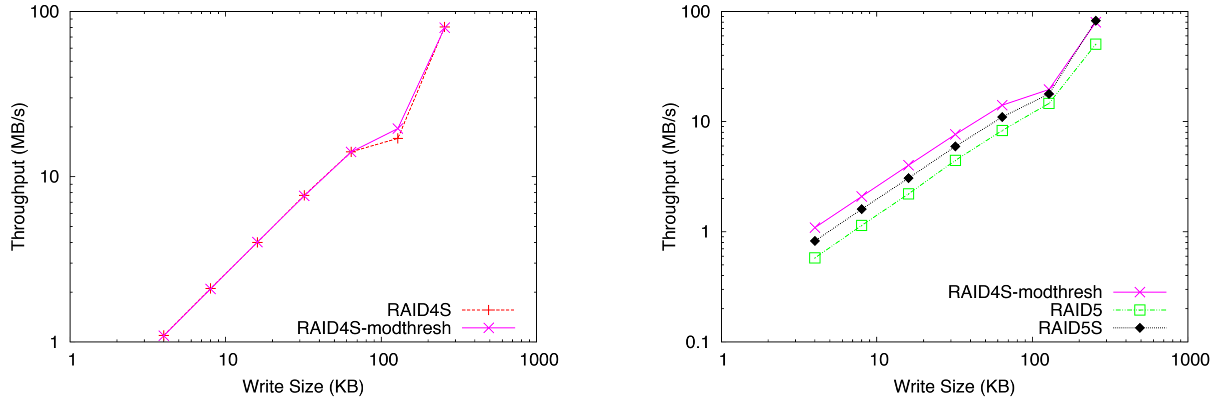

Figure 15: Comparing all the write sizes for block aligned writes (powers of 2). The graph shows that RAID4S-modthresh increases the throughput of medium-sized writes with minimal impact on the rest of the RAID4S write sizes. Overall, the throughput for all write sizes is the same as the measurements in Figure 3 but with a small improvement in the 128KB medium-write.

threshold for small-write computations can be extended to even larger medium-write sizes.

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

Another possibility for future work is to consider the write selection algorithm for RAID4S in choosing a better threshold. It would be beneficial to take a closer look at RAID4S-modthresh and RAID4S-modified, as shown in Figure 11 to figure out why RAID4S-modified performed better for some of the smaller large-writes. This might disprove our preconceived notion that large-writes are always better than small-writes. Debugging the two systems side by side to see what RAID4S-modified does that RAID4S-modthresh doesn't do would lead to an explanation of the speedup that might disprove some of the assumed characteristics of small and large-writes.

5. RELATED WORKS

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

5.1 HDD Solutions

Parity logging [8] is a common solution to the small-write problem. For each small-write, parity logging saves the parity update image, which is the resulting XOR bitwise value of the old data XORed with the new data, called the parity update image, in a fault-tolerant buffer. When the buffer is filled with enough data to constitute an "efficient" disk transfer, the logged parity update image is applied to the in-memory image of the out-of-date parity and the new parity is written sequentially to the RAID setup. Although parity logging improves performance without the added cost of an SSD and can be expanded to RAID5, it also incurs a storage overhead that increases with the striping degree. It also does not address the small-write and large-write relationship and the possibility that making a small transformation to the threshold might not necessitate the need to alter the algorithm or storage implementation of the entire system. Finally, the irregularity of writes and the chance that a write can be unpredictable and may not fit the best sizes, may affect performance in unpredictable ways. Parity logging is great for disks of equal speeds but does not take advantage of SSD technology and the ability to access multiple disks in parallel with many parity log writes.

LFS [6] optimizes writes by destroying locality, using a segment cleaner, and keeping track of data and addresses the overhead of small-writes to data. For small file workloads, the LFS converts the small disk I/Os into large asynchronous sequential transfers. Similarily, Write Anywhere File System (WAFL) [2] is designed to improve small-write performance. This is achieved by letting WAFL write files anywhere on disk. Hence, WAFL improves RAID performance because WAFL minimizes seek time (write to near blocks) and head contention (reading large files). These two methods assume that small-writes are poor and transform large writes by logging the data. By contrast, RAID4S-modthresh takes advantage of the RAID4S layout to improve small-write performance rather than trying to eradicate the small-writes completely.

5.2 SSD Solutions

Heterogenous SSD based RAID4 [5] injects SSDs into a RAID4 setup by replacing all the data drives with SSDs. They measured performance in terms of life span and reliability instead of cost and throughput. They argue that using the SSD as a parity drive is subject to unacceptable wear caused by the frequent updates to the parity. This wear, up to 2% higher than the heterogenous RAID4, can cause an increase in error rates, data corruption, reduced reliability, and a reduction of SSD lifecycles. This RAID4 system values different performance measurements: reliability and life span. They acknowledge the small-write bottleneck but do not address throughput in their evaluation, which is the purpose of RAID4S and, by extension, RAID4S-modthresh. RAID4S and RAID4S-modthresh focus on improving small-write performance without incurring a large cost penalty.

The Hybrid Parity-based Disk Array (HPDA) solution [4] uses SSDs as the data disks and two hard disks drives for the parity drive and a RAID1-style write buffer. Half of the first drive is the traditional parity drive while the remaining half of the first HDD drive and the whole of the second HDD drive is used to log small-write buffers and as a RAID1 backup drive. This solution facilitates high performance for small-writes and avoids the flash wear-out and erase-before-write SSD problems. RAID4S experiences more wear then HPDA, but costs less and requires less storage space because it does not require $n - 1$ SSD drives. RAID4S-modthresh is also simpler, cheaper, and provides similar speedups.

Delayed parity techniques [3] log the parity updates, instead of the small-writes, on a partial parity cache. Partial parity takes advantage of the flash properties to read only when necessary. The process has two states, partial parity creation / updating and partial parity commit. The delayed parity reduces the amount of small-writes. The create/update alters the state of the partial parity cache by inserting the parity if it absent and determining if an existing parity is redundant, i.e. whether it needs to XORed with the old data. This reduces the number of reads, since the full parity is not stored and reduces the resulting parity generation overhead. The problem with this is that it increases the commit speed overhead, introduces an extra calculation overhead (what needs to be read), and costs more because it is a RAID5S system. RAID4S and RAID4S-modthresh introduce a solution to the small-write problem to increase performance without the same storage and complexity overhead.

6. CONCLUSION

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

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

7. ACKNOWLEDGMENTS

I'd like to thank Rosie for her help with this project. Her test bed setup, the prompt responses to my frantic emails, the time she spent paired programming the LKM with me, and the genuine interest in how my project was going did not go unnoticed.

8. REFERENCES

[1] 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*, 26:145–185, 1994.

[2] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance, 1994.

[3] S. Im and D. Shin. Delayed partial parity scheme for reliable and high-performance flash memory SSD. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–6, 2010.

[4] B. Mao, H. Jiang, D. Feng, S. Wu, J. Chen, L. Zeng, and L. Tian. HPDA: A hybrid parity-based disk array for enhanced performance and reliability. In *IPDPS'10*, pages 1–12, 2010.

[5] K. Park, D.-H. Lee, Y. Woo, G. Lee, J.-H. Lee, and D.-H. Kim. Reliability and performance enhancement technique for SSD array storage system using RAID mechanism. *2009 9th International Symposium on Communications and Information Technology*, pages 140–145, 2009.

[6] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[7] T. M. Ruwart. *Xdd User's Guide*. I/O Performance, Inc., 6.5.091706 edition, December 2005.

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

[9] D. Stodolsky, M. Holland, W. V. Courtright II, and G. A. Gibson. A redundant disk array architecture for efficient small writes. Technical report, Transactions on Computer Systems, September 1994.

[10] R. Wacha. RAID4S: Supercharging RAID small writes with SSD. In *To be presented*, October 2010.