A Behind-the-Scenes Story on Applying Cross-Layer Coordination to Disks and RAIDs

Jin Qian and An-I Andy Wang
Florida State University, {qian, awang@cs.fsu.edu}

Abstract

Coordinating storage components across abstraction layers has demonstrated significant performance gains. However, when applied near the physical storage, this approach relies on exposing and exploiting low-level hardware characteristics, perhaps a large number of them, to cope with complex modern disks and RAIDs to apply such an approach.

Through clean-room implementations and validations of prior research on track-aligned accesses and its incorporation in RAIDs, as well as through experiments with our proposed queue coordination in RAIDs, we confirmed that cross-layer coordination can indeed yield high performance gains. On the other hand, the effective use of cross-layer coordination involves overcoming several challenges: (1) developing efficient and automated ways to extract and exploit hardware characteristics due to rapidly evolving disks, (2) fostering a greater understanding of the legacy storage data path, so that we can better predict the benefits of low-level optimizations and their intertwined interactions, and (3) inventing efficient and automated ways to tune the low-level parameters.

1 Introduction

Disk-based storage has been a system-wide performance bottleneck for the past three decades. One major limiting factor is how disks are represented and accessed by the operating system. Disks are generally presented as a sequence of blocks, thus abstracting away their details (e.g., variable number of sectors per track). Disks within a RAID (redundant arrays of independent disks) are presented as a single virtual disk, so that an operating system can access a RAID or a disk with the same mechanisms.

While these abstraction layers ease modular software development, they also hide, and in many cases hinder, opportunities for performance optimizations. For example, disks within a RAID are largely unaware of the existence of other disks, thereby delaying requests that span multiple disks due to the lack of coordination.

An orthogonal way to present disk-based storage to the operating system is to reveal the underlying hardware details, such that a high-level system component can make more informed decisions based on more global details. Applications of this cross-layer coordination have demonstrated significant performance improvements in disk storage [McKusick et al. 1984; Matthews et al. 1997; Carnes et al. 2000; Lumb et al. 2000; Schindler et al. 2002; Schmuck and Haskin 2002; Nugent et al. 2003; Schindler et al. 2004; Schlosser et al. 2005; Sivathanu et al. 2005]. The success of these optimizations also suggests that further exploitation of additional low-level disk details can yield promising performance gains.

To investigate the possibility of more such performance benefits, we explored the use of low-level controls of disks and RAIDs with respect to allocation, access granularity, and scheduling. In particular, we attempted clean-room implementations and validations of the track-aligned extents work [Schindler et al. 2002] (i.e., track-aligned accesses to a disk) and a track-aligned RAID. We also proposed and implemented a way to coordinate disk queues within a RAID.

To our surprise, applying the cross-layer coordination approach involves decisions of greater intricacy than we originally thought, especially considering the quickly evolving physical characteristics of modern disks and associated hardware. Thus, we report our experience to the research community to better understand the low-level decisions required to apply such an approach to modern disks and RAIDs.

2 Background

Software RAIDs: In a software RAID under Linux, a request is first sent to a multi-device driver (e.g., RAID-5), which is responsible for gathering, remapping, and forwarding requests to individual disks within the RAID. The multi-device driver also reorders, merges, and splits requests as needed to improve overall performance. The request queue associated with the RAID device driver can be *plugged* at times, so that the pending requests in the queue wait for additional requests for some time to increase the opportunities for effective reordering. The queue can also be *unplugged* when forwarding requests to underlying per-disk device drivers.

The per-disk software device driver handles vendor-specific details of hard disks, and is associated with a request queue. Therefore, each device driver

independently schedules and optimizes its disk performance, without coordinating with other disks.

Cross-layer coordination: Cross-layer coordination has been increasingly explored in the storage arena, and it has demonstrated significant performance improvements. For example, by exposing the track boundaries of disks, file systems and cache prefetching can effectively allocate and access data in a track-aligned manner [Schindler et al. 2002]. The file system layer can also gain semantic knowledge of specific applications (e.g., database) to optimize disk layout [Sivathanu et al. 2005].

This paper focuses the application of this approach at the levels of disks and RAIDs. In particular, we will examine the decisions involved in coordinating allocation granularity, data access alignment, and IO scheduling policies across storage layers.

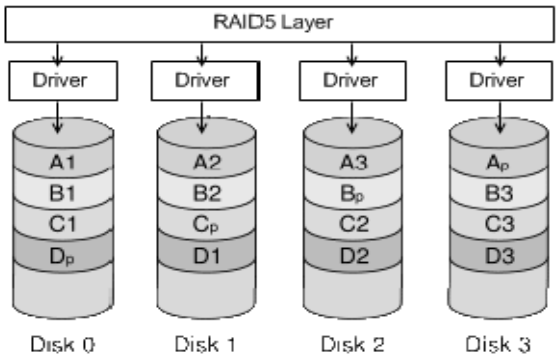

Figure 2.1: A software RAID-5 with four disks. Each request is sent to a RAID-5 multi-device layer, which splits (as needed) and forwards the request(s) to per-disk device drivers. Within the RAID-5, A_p is the parity for A_1, A_2, and A_3. B_p is the parity for B_1, B_2, and B_3, and so on.

3 Goals and Approaches

Our research goals are (1) to unravel behind-the-scenes design decisions involved to exploit low-end details and controls of modern disks and RAIDs, and (2) to understand the various implications of applying such an approach.

Since understanding the cross-layer approach near physical storage involves intimate interactions with specific hardware characteristics, implementations and empirical measurements are required to learn first-hand lessons. To assure positive outcomes, we began with the proven optimization of track-aligned extents (i.e., track-aligned disk accesses) and its derivative RAID for our clean-room implementations. Since track alignment addresses policies regarding storage allocation and access granularity, we also experimented with scheduling policies by exploring our proposed method of coordinating queues within a RAID. The intent is to draw generalized lessons from various ways of exploiting low-level hardware details.

The fundamental observation in track-aligned extents is that an operating system generally accesses disks in blocks, each containing multiple sectors. Therefore, accessing a block can potentially cross a track boundary and incur additional head positioning time to switch tracks. By revealing and exploiting track boundaries above the disk device driver interface, [Schindler et al. 2002] observes that when accessing near a track size of information, aligned accesses according to the track boundary can deliver 50% performance improvement. This range of performance gain also relies on disks that support zero-latency access, which allows the tail-end of a requested track to be accessed before the beginning of the requested track content [Worthington et al. 1995].

The same principle of exploiting low-level hardware details can also be generalized to RAIDs. For instance, when accessing a stripe of information, a RAID needs to wait for the slowest disk in the RAID to complete its service, which can incur the worst-case queuing, seek, and rotational delays. Atropos [Schindler et al. 2004] reduces the worst-case delays by applying track-aligned accesses to disks to reduce the expected worst-case rotational delay for accessing a stripe.

To address the worst-case queuing time among disks in a RAID, we designed and implemented a way to coordinate disk queues, with an aim for a striped request to be sent to disks at approximately the same time. This coordination can also potentially improve the synchrony of disk head locations, ameliorating the worst-case seek time among disks.

4 Recreating Track-aligned Extents

The three main tasks to duplicate the track-aligned extents work are (1) finding the track boundaries and the zero-latency access disk characteristics, (2) making use of such information, and (3) verifying its performance benefits. The hardware and software experimental settings are summarized in Table 4.1.

Hardware/software	Configurations
Processor	Pentium D 830, 3GHz, 16KB L1 cache, 2x1MB L2 cache

Memory	128 MB or 2GB
RAID controller	Adaptec 4805SAS
Disks tested	Maxtor SCSI 10K5 Atlas, 73GB, 10K RPM, 8MB on-disk cache [Maxtor 2004] Seagate CheetahR 15K.4 Ultra320 SCSI, 36GB, 8MB on-disk cache [Seagate 2007] Fujitsu MAP3367NC, 10K RPM, 37GB, with 8MB on-disk cache [Fujitsu 2007]
Operating system	Linux 2.6.16.9
File system	Ext2 [Card et al. 1999]

Table 4.1: Hardware/software experimental specifications.

4.1 Extracting Disk Characteristics

Simple request scanning: Since the reported performance benefits for track alignments are high, conceivably, a user-level program can observe timing variations to identify track boundaries. A program can incrementally issue reads, requesting one more sector than before, staring from the 0th sector. As the request size grows, the disk bandwidth should first increase and then drop as the request size exceeds the size of the first track (due to track switching overhead). The process can then repeat, starting from the first sector of the previously found track. The inefficiency of this algorithm can be reduced via applying binary search.

To reduce disturbances introduced by various hardware and software components of the disk data path, we used `DIRECT_IO` flag to bypass the Linux page cache, and we accessed the disk as a raw device to bypass the file system. We used a modified `aacraid` driver code to bypass the SCSI controller, and we used `sdparm` to disable the read cache (`RCD=1`) and prefetch (`DPTL=0`) of the disk.

As a sanity check, we also attempted to start all reads from an arbitrary position of the 256th sector. Additionally, we attempted to start each read with a random sector between 0 and 512, with each succeeding request size increasing by 1 sector (512 bytes). Figure 4.1.1 shows the resulting graph.

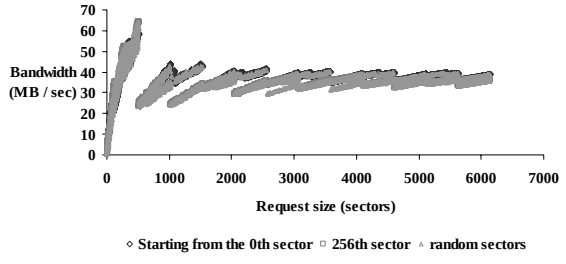

Figure 4.1.1: Bandwidth comparison for different read request sizes from different starting sectors on a Maxtor disk.

Surprisingly, although the graph exhibits bandwidth "cliffs," the characteristic trends are not sensitive to the starting location of requests, suggesting that those cliffs are caused by sources of data misalignments other than tracks. Some possibilities are transfer granularity of the direct memory access (DMA) and the management granularity of IO buffers. The graph also suggests the presence of other optimizations that are not disabled. For example, the high bandwidth before the first cliff far exceeds our expected performance gain. Also, for certain ranges of request sizes (e.g., between 1,000 and 1,500 sectors), the average bandwidth shows multimodal behaviors.

To verify that those cliff locations are not track boundaries, we wrote a program to access random cliff locations with the access size of 512 sectors (256KB) as indicated by the first cliff location. We ran multiple instances of this program concurrently and perceived no noticeable performance difference compared to the cases where the accesses started with random sectors.

SCSI diagnostic commands: Unable to extract track boundaries from a naive user-level program, we resorted to SCSI `SEND/RECEIVE DIAGNOSTIC` commands to map a logical block address (LBA) to a physical track, surface, and sector number.[1] However, this sector-by-sector translation of large model drives is very slow, and it took days to extract an entire 73-GB Maxtor Atlas V drive. We modified the `sg_senddiag` program in the Linux `sg3_utils` package to speed up the extraction process, according to the following pseudocode:

1. Extract from LBA 0 sector-by-sector until we detect a track boundary (e.g., either track number or surface number changes). Record LBA and the physical address of the boundary. Store the track size S.

2. Add track size S to the last known track boundary T and translate S + T and S + T – 1.

   a. If we detect a track change between S + T and S + T – 1, then S + T is a new boundary. Record LBA and the physical address of the boundary. Go to step 2.

   b. If there is no change between S + T and S + T – 1, the track size has changed. Extract sector-by-sector from the last known track boundary until we detect a new track boundary. Record LBA and physical address of the boundary. Update the track size S. Go to step 2.

[1] We did not use Dixtrac [Schindler and Ganger 1999] for the purpose of clean-room implementation and validation.

3. If sector reaches the end of disk in step 2, exit.

Through this mechanism, we extracted information not always documented by vendors' datasheets and manuals [Maxtor 2004; Fujitsu 2007; Seagate 2007] in about 7 minutes.

First, the LBA mapping to the physical track number is not monotonic (Figure 4.1.2). For the Maxtor drive, LBA 0 starts on track 31 of the top surface and increases outward (from the disk spindle) to track 0, and then LBA continues from the bottom surface of track 0 inward to track 31. Next, the LBA jumps to track 63 of the bottom surface growing outward to track 32, and then switches back to the top surface's track 32 and continues inward to track 63. The pattern repeats.

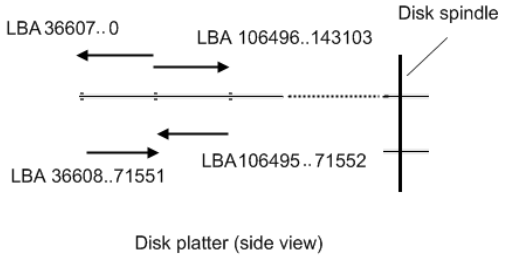

Figure 4.1.2: Non-monotonic mapping between LBA and track numbers.

Variants of this serpentine numbering scheme [Anderson 2003] are observed in Seagate [2007] and Fujitsu [2007] drives as well. One can conjecture this numbering scheme in relation to the elevator and scanning-based IO schedulers. In terms of performance characteristics, one might expect additional timing variations due to the track numbering system, in addition to track boundaries.

Second, the number of sectors contained in each track is different between the top and bottom surfaces, even for the same track number. For example, for a Maxtor drive, a track on the top surface of track 0 may contain 1,144 sectors, and the bottom surface of the track 0 may contain 1,092 sectors. One explanation is that certain sectors are spares. By having spares within each track, bad sectors can be remapped without introducing additional seek delays. In the context of track alignment, this finding implies additional bookkeeping for each disk surface.

Third, the track size differs even for the same disk model from the same vendor. In a batch of 6 Maxtor 10K V drives purchased at the same time, we found 4 different LBA numbering schemes (Table 4.1.1). The implication is that track extraction cannot be performed once per disk model. It potentially needs to be performed on every disk. Track size differs even in the same zone on the same surface, though rarely and only slightly. We saw that some tracks begin on their second sectors, that is, LBA skips the first sector of that track. Due to all these irregular anomalies, we are no longer able to calculate track boundary with zone information but have to extract all tracks.

Serial number	Surface 0, outer most track	Surface 1, outer most track
J20 Q3 CZK	1144 sectors	1092 sectors
J20 Q3 C0K	1092 sectors	1144 sectors
J20 Q3 C9K	1092 sectors	1144 sectors
J20 TK 7GK	1025 sectors	1196 sectors
J20 TF S0K	1060 sectors	1170 sectors
J20 TF MKK	1060 sectors	1170 sectors

Table 4.1.1: Different Track Sizes of Maxtor 10K V Drives.

Track boundary verifications: To verify the track information extracted via the SCSI diagnostic commands, we wrote a program to measure the elapsed time to access 64 sectors of data with shifting offsets from random track boundaries. The use of 64 sectors eases the visual identifications of track boundaries. We measured tracks only from the top surface within the first zone of a Maxtor disk, so we can simplify our experiment by accessing mostly a track size of 1,144 sectors.

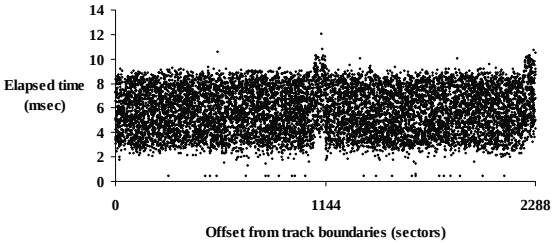

Figure 4.1.3: Elapsed time to access random 64 sectors, starting from different offsets from SCSI-command-extracted track boundaries on a Maxtor drive. The track size is 1,144 sectors.

Figure 4.1.3 confirms our extracted track boundaries. Each data point represents the time to access a 64-sector request starting from a randomly chosen sector offset from a track boundary. The 6-msec range of timing variation reflects the rotation-delay variations for a 10,000 RPM drive. The average elapsed time of accessing 64 sectors across a track boundary is 7.3 msec, compared to 5.7 msec for not crossing the track boundaries. Interestingly, the difference of 1.6 msec is much higher than the track switching time of 0.3 to 0.5 msec [Maxtor 2004]. We also verified this extraction method with other vendor drives. The findings were largely consistent.

Zero-latency feature verification: Since the range of performance gain by track-aligned access depends on whether a disk can access the information within a track out-of-order, we performed the tests suggested in

[Worthington et al. 1995]. Basically, we randomly picked two consecutive sectors, read those sectors in the reverse LBA order, and observed the timing characteristics. This test was performed with various caching options on.

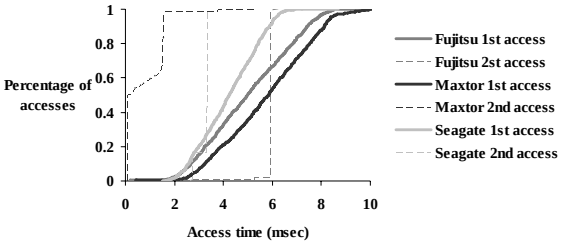

Figure 4.1.4: CDF of disk access times for accessing random sets of two consecutive LBAs in the reverse order.

As shown in Figure 4.1.4, with a Maxtor drive, for 50% of the time the second request is served from the cache, indicating the zero-latency capability. (We did not observe correlations between the chosen sectors and whether the zero-latency feature is triggered.) In contrast, the other two drives always need to wait for a 3- to 6-msec rotational delay before serving the second sector request. For the remaining paper, we will use the Maxtor drives.

4.2 Exploiting Track Boundaries

The track boundary information can be exploited at different levels.

User level: One possibility is to create a user program to make use of this track information. The mechanism is similar to the disk defragmentation process. Instead of moving file blocks to reduce the level of fragmentation, we can move blocks to align with track boundaries. This approach avoids kernel modifications and can make files smaller than a track not crossing track boundaries, and files larger than a track aligned to track boundaries.

Unfortunately, this approach needs to overcome many tricky design points. For example, certain blocks are referenced from many places (e.g., hardlinks). Moving those blocks requires tracking down and updating all references to the block being moved. Such information might not be readily available. Also, we need to consider conditions where a system might crash amid our disk layout reorganization process. Finally, similar to defragmentation, data blocks can become miss-aligned after a period of time (e.g., files being added and deleted). It is necessary to re-align data periodically. The overhead of performing such periodic tasks may outweigh the performance benefits.

File system level: We can mark certain sectors as bad (e.g., modify the bad-block list before running a file system creation program) so a file system cannot allocate blocks that consist of sectors across track boundaries. However, this method alone does not prevent a near-track-size file being allocated across two tracks. This approach also anticipates some bandwidth loss when a single process tries to access multi-track files due to unused sectors. However, when a system is under multiple concurrent streams, the performance benefits of accessing fewer tracks when multiplexing among IO streams can outweigh the performance loss due to unused sectors.

The Linux ext2 file system uses pre-allocation [Card et al. 1999] to reserve a default of 7 blocks adjacent to the block just requested. To allocate files based on tracks, we imported the boundary block list into the kernel so that file system components could use track information. For the ease of validation, we modified the ext2 pre-allocation routine to allocate in tracks (or up to a track boundary, which is marked as a bad block by the file system creation program). One clear disadvantage of this approach is over-allocation, but the unused space can be later returned to the system. However, should the system anticipate mostly track-size accesses, we are less concerned with the wasted space. For instance, database and multimedia applications can adjust their access granularity accordingly.

With the aid of this list, we can also change the read-ahead to perform prefetch on a track basis. Even though files are track-aligned, benefits can hardly show up without a track-based access pattern. Linux readahead not only uses a small prefetch window (default 128KB, compared with around 500KB track size) but also adjusts the window dynamically based on the hit ratio. As a result, several prefetch requests are needed to read the entire track, which can incur additional rotational delay and diminish the benefits of track alignment. Especially, requests to different files can be interleaved and cause both additional rotational delay and seek, further lowering the efficiency.

Implementation: We used the track boundary list extracted by the SCSI diagnostic commands as the bad-block list input for the `mke2fs` program, which marks all these blocks, so that they will not be allocated to files. We also put this list in a kernel module along with two functions. One initializes and reads the list from user space. The other is used by different kernel components to find a track boundary after a given position. For optimization, we implemented binary search in this function.

There are two places in the kernel making use of the search function. First, pre-allocation looks for the first block of a track (the block right after a track boundary)

and then allocates this track to a requesting file. The end of a track (the next boundary) can be identified by a used block marked by `mke2fs` so that the pre-allocation ends properly. One implication is that individual file systems need to be modified to benefit from track alignments. Second, when a readahead starts a new prefetch window, it drops all prefetching requests that exceed the track boundary.

4.3 Verification of the Performance Benefits

Bonnie: We chose a widely used benchmark called Bonnie [Bray 1996], which is unaware of the underlying track-alignment mechanisms. Bonnie consists of many phases, stressing the performance of character and block IOs amidst sequential and random access patterns. The two phases of our interests are the sequential write and read. The sequential write phase creates a 1-GB file, which exceeds our 128-MB memory limit, and reads it sequentially. We enabled SCSI cache, disk caching, and prefetch to better reflect normal usage. Each experiment was repeated 10 times, analyzed at a 90% confidence interval.

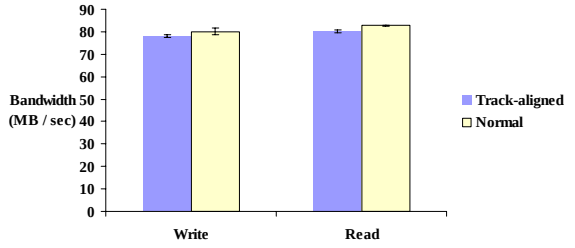

Figure 4.3.1: Bandwidth comparisons between conventional and track-aligned accesses to a single disk, when running the Bonnie benchmark.

Figure 4.3.1 shows the expected 3% slowdown for a single stream of sequential disk accesses, where skipped blocks that cross track boundaries can no longer contribute to the bandwidth.

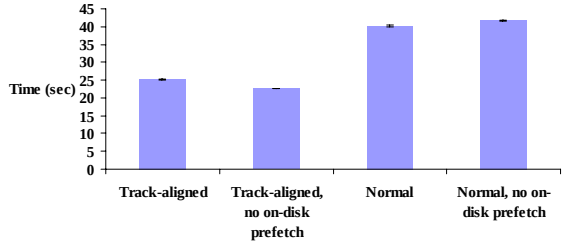

Figure 4.3.2: Speed comparisons between conventional and track-aligned accesses to a single disk, `diff`ing two 512MB files with 128MB of RAM.

We also ran a `diff` program (from GNU `diffutils` 2.8.1) to compare two 512-MB large files via interleaved reads between two files, with the `–speed-large-files` option. Without this option, `diff` will try to read one entire file into the memory and then the other file and compare them if memory permits, which nullifies our intent of testing interleaved reads. We have two settings: the normal and the track-aligned case. Figure 4.3.2 shows that track-aligned accesses are almost twice as fast as the normal case. In addition, we observed that disk firmware prefetch can violate the track-aligned prefetch issued from the file system readahead, as disk firmware prefetch has no regard for track boundaries. Disabling on-disk prefetch further speeds up track-aligned access by another 8%. Therefore, for subsequent experiments, we disabled disk firmware prefetch for track-aligned accesses.

Since track-aligned extents excel in handling concurrent accesses, we conducted an experiment that involves concurrent processes issuing multimedia-like traffic streams at around 500KB/sec. We used 2GB for our memory size. We wrote a script that increases the number of multimedia streams by one after each second, and the script records the startup latency of each new stream. Each emulated multimedia streaming process first randomly selects a disk position and sequentially accesses the subsequent blocks at the specified streaming rate. We assumed that the acceptable startup latency is around 3 seconds, and the program terminates once the latency reaches 3 seconds.

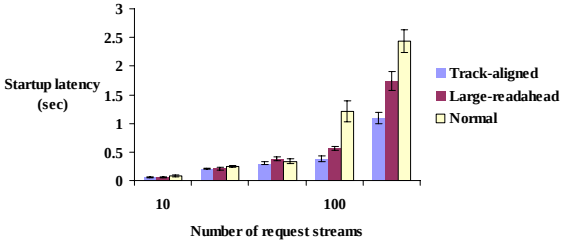

Figure 4.3.3: Startup latency comparisons of conventional I/O requests, requests with a one-track prefetch window, and track-aligned requests on a single disk, with a varying number of multimedia-like request streams.

Figure 4.3.3 shows that the original disk can support up to 130 streams with the startup latency within 3 seconds. A track-size readahead window can reduce the latency at 130 streams by 30%, while the track-aligned access can reduce the latency by 55%.

5 Track-aligned RAIDs

One natural step to generalize the notion of exposing and exploiting the hardware characteristics of a single disk is to construct RAIDs with track-aligned accesses. However, recall in Section 4.1 that the track sizes can differ even for the same disk model from the same vendor. One immediate implication is that each disk in a RAID needs to be scanned one after another on a machine to extract track boundaries.

Another implication is that there are different ways to form stripes. For one, we can just construct stripes with tracks of different sizes. Although this scheme can work with RAID-0, it does not load balance well and work well with other RAID levels. For example, RAID-5 parity is generated via XORing chunk units (units of data striping) of the same size. Suppose we want the chunk unit to be set to the size of a track. Should we use the largest track size as the chunk unit, some disks need to use 1+ tracks to form a chunk. Or, we can use the smallest track size as the chunk unit, leading to unused sectors for disks with larger track sizes.

Intriguingly, we observed that RAID levels that involve parity can interact poorly with prefetching in the following way. Take RAID-5 as an example. At the file system level, prefetching one track from each non-parity disk involves a prefetching window that is the size of a track multiplied by the number of disks that does not contain the parity information. However, as a RAID redirects the contiguous prefetching requests from the file system level, the actual forwarded track-size prefetching requests to individual disks are fragmented, since reads in RAIDs do not need to access the parity information.

Another poor interaction is the Linux plug and unplug mechanisms associated with disk queues and multi-device queues. These mechanisms are designed to increase the opportunities for data reordering by introducing artificial forwarding delays at times (e.g., 3 msec), and do not respect track boundaries. Therefore, by making these mechanisms aware of track boundaries, combined with all prior considerations, we were finally able to make individual disks in a RAID-5 to access in a track-aligned manner.

5.1 Implementation

We modified Linux software RAID-5 to implement the track-aligned accesses. To overcome the heterogeneous track sizes, we used the plug/unplug mechanisms to enforce track-aligned accesses. By doing so, chunk sizes and striping mechanisms become more independent of disks with different track sizes.

We inserted a piece of code in the Linux software RAID-5 `make_request` function. When an IO request arrives, this function translates the RAID virtual disk address into individual disk addresses. We then monitored the translated requests to see if they were requests that cross track boundaries. The unplug functions for individual disk queues were then explicitly invoked to issue track-aligned requests.

To prevent the RAID-5 parity mechanisms from fragmenting track-size prefetching requests to individual disks, we modified RAID-5. Whenever the parity holding disk in a stripe was the only one not requested for that stripe, we filled in the read request for that disk and passed it down with all others. When this dummy request was completed, we simply discarded the data. The data buffer in Linux software RAID-5 is pre-allocated at initialization, so this implementation does not cause additional memory overhead.

5.2 Verification of Performance Benefits

We compared the base case RAID-5 with a track-aligned RAID-5 with 5 disks, and a chunk size of 4KB. For the Bonnie benchmark, we used a 1-GB working set with 128MB of RAM. Figure 5.1 shows that the write bandwidth for the three system settings falls within a similar range due to buffered writes. However, for read bandwidth, the track-aligned RAID-5 outperforms the conventional one by 57%.

The `diff` experiment compared between two 512-MB files with 128MB of RAM. Figure 5.2 shows that the track-aligned RAID-5 can achieve a 3x factor speedup compared to the original RAID-5.

For the multimedia-like workload with 2GB of RAM, the track-aligned RAID-5 demonstrates a 3.3x better scaling in concurrency than the conventional RAID-5, where a RAID-5 with a readahead window comparable to the track-aligned RAID-5 contributes only less than half of the scaling improvement. The latency improvement of track-aligned RAID-5 is particularly impressive considering that the original RAID-5 was expected to degrade in the latency characteristics when compared to the single-disk case, due to the widening timing variance of disks and the need to wait for the slowest disk for striped requests. Track-aligned accesses reduce the worst-case rotational timing variance and can realize more benefits of parallelism.

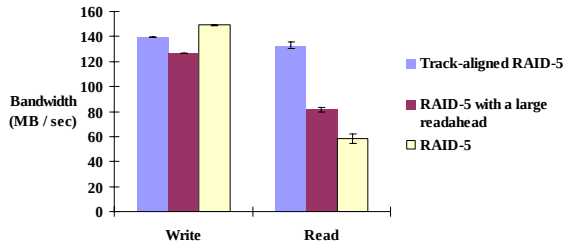

Figure 5.1: Bandwidth comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of 4 tracks, and the original RAID-5, running Bonnie with 1GB working set and 128MB of RAM.

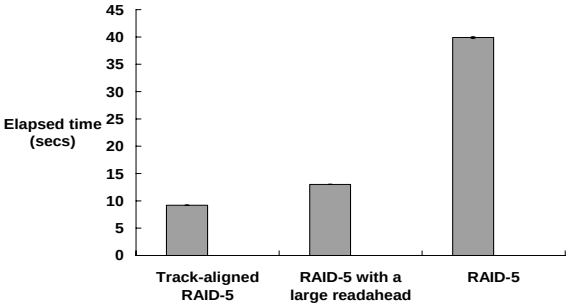

Figure 5.2: Elapsed time comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of 4 tracks, and the original RAID-5, when running `diff` comparing two 512MB files.

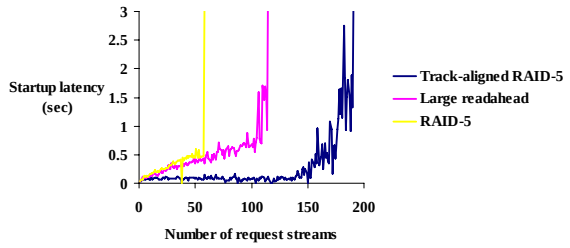

Figure 5.3: Startup latency comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of 4 tracks, and the original RAID-5, with a varying number of multimedia-like request streams.

6 Coordinated I/O Scheduler

Track-alignment mainly addresses the allocation and access granularity aspects. Another aspect of the storage handling concerns IO scheduling.

Linux 2.6 supports a number of IO schedulers, namely, noop, deadline, anticipatory [Iyer and Druschel 2001], and complete fairness queue (CFQ). Their main purposes include (1) gathering and merging small adjacent IO requests into one large contiguous request, and (2) reordering requests so that they can enforce different IO priorities. For example, a deadline-based scheduler maintains one sorted list based on a request's block address and another FIFO list based on the request arrival time. When it is time to send the request to a disk, the scheduler will consider the block distance from the previous block request as well as the expiration time of the current request (i.e., arrival time plus the maximum waiting time) to choose the next request. CFQ additionally considers the IO budgets of each process, similar to a CPU scheduler.

Although per-disk schedulers are effective when operating in isolation, they are largely unaware of other schedulers when a single RAID request involves multiple disks, leading to missed optimization opportunities. As a result, a striped request may wait longer because one of the disks decides to serve other requests before the striped request. Therefore, if we can coordinate different schedulers to serve a striped request at around the same time, we can reduce the worst-case latency by having disk heads and request queues in better synchrony.

6.1 Design Space

One way to coordinate per-disk queues is to make each queue aware of other queues via explicit messaging. Therefore, whenever a queue changes its state, the queue will also inform other queues of its changed status. The downside of this coordination is poor scaling. As the number of disks increases, the number of synchronization messages grows quadratically.

A centralized coordination approach is to associate an IO's request priority with arrival time. By doing so, a striped request will have similar priority values when being processed by the per-disk queue. However, the downside of the time-based coordination is the difficulty of incorporating time into the computation of request priority, knowing that the time value is not bounded by a range. Therefore, as time elapses, the priority computation will eventually be overtaken by the weighting of the time value.

A third approach leverages the length of per-disk queues for synchronization, which the value is bounded. The rationale is that if one of the disks is very busy at the moment, it makes little sense for other disks to serve this striped request early. Therefore, all requests in a stripe can be set with a priority associated with the maximum of all disk queue lengths. Should other requests arrive on disks with shorter queues, they can be served before the striped request. Alternatively, we can set all requests of a stripe to a priority associated

with the minimum of all disk queue lengths, so that striped requests are served at the earliest possible time.

6.2 Implementation

We extended the CFQ scheduler to take various disk queue lengths into consideration when computing the IO priority for various requests. At the RAID layer, before forwarding requests that belong to a stripe, the maximum queue length is computed based on the number of pending requests in per-disk queues. Then, within the per-disk CFQ scheduler, the relative block distance between the current and the previous request (computed in the `cfq_choose_req` function) is adjusted by the maximum queue length left shifted by a weight factor.

$$QLen_{\max} = \max(QLen_{Disk_1} \text{ K } QLen_{Disk_N})$$
$$Dist_{block} = Dist_{block} + (QLen_{\max} << Weight)$$

The maximum relative block distance is determined by the size of the hard drive. In our case, about 19 million blocks for the 73-GB Maxtor drive. The maximum possible queue length is defined by BLKDEV_MAX_RQ in Linux, which is 128 by default. Therefore, for the queue length to have an influence equal to that of the block distance, we need the weight factor to be set to around 16 when the system is under a heavy concurrent load.

6.3 Performance Evaluation

Figure 6.3.1 summarizes the performance of coordinated queues in relation to RAID-5 and track-aligned accesses. Intriguingly, coordinated queues improved the concurrency scaling by a factor of only 1.2x, while the track-aligned RAID improved scaling by a factor of 3.3x. Also, when the track-aligned RAID-5 was combined with coordinated queues, no significant performance differences were observed.

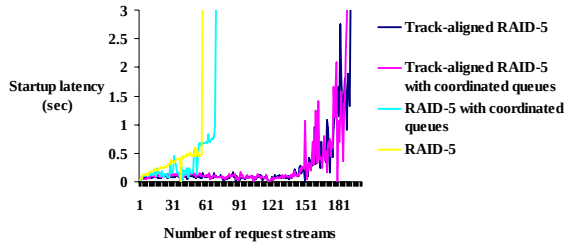

Figure 6.3.1: Startup latency comparisons of the track-aligned RAID-5, the track-aligned RAID-5 with coordinated queues, the original RAID-5 with coordinated queues, and the original RAID-5, with a varying number of multimedia-like request streams.

Since the chosen weight for the coordinated queues can affect the performance, we conducted a sensitivity analysis by varying the weight from 10 to 30 for the same scaling experiment. Figure 6.3.2 shows that the startup latency CDF variation is generally within 10%.

Puzzled by the limited combined benefits between the track-aligned RAID-5 and the coordinated queue, we plotted the moving average of the disk head distance among five disks, for various schemes. Intriguingly, Figure 6.3.3 shows that the track-aligned RAID-5 actually synchronizes disk heads better than the coordinated queues due to two possible reasons. First, the plugging and unplugging mechanisms used to honor track boundaries actually interact with the scheduling of striped requests. Second, our implementation of the track-aligned RAID-5 also requests the parity information on reads, reducing the chances of divergence for disk head locations.

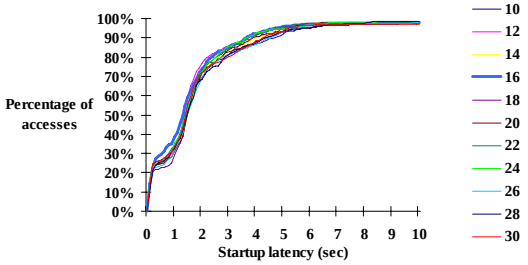

Figure 6.3.2: Startup latency CDF comparisons of different weights used for coordinated queues, with a varying number of multimedia-like request streams.

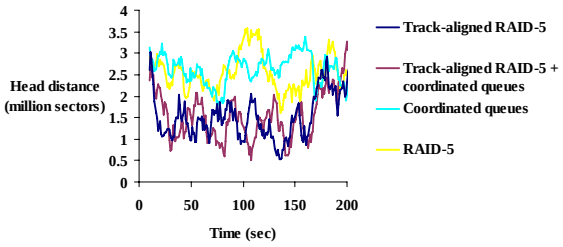

Figure 6.3.3: Disk head location deviation (10 seconds moving average) comparisons of the track-aligned RAID-5, the track-aligned RAID-5 with coordinated queues, the original RAID-5 with coordinated queues, and the original RAID-5, with a varying number of multimedia-like request streams.

7 Lessons Learned

Based on our clean-room implementations of the track-aligned accesses and RAIDs, our duplication of results

confirms the significant performance benefits that can be achieved by exposing and exploiting low-level disk details for high-level optimizations. The validation of research results obtained five years ago also demonstrates the relative resiliency and applicability to several generations of disks. Although the benefits of our proposed coordinated queues seem to be subsumed by the track-aligned accesses, their combined results show our limited understanding of the current data path and how disk allocation granularity, access alignment, and scheduling interact. Additionally, we have experienced the following lessons first-hand.

First, the effectiveness of the cross-layer coordination near the hardware level is only as good as our understanding of modern disks, which is not necessarily made available by vendors. This paper presents only a fraction of graphs that we can explain. We have encountered many more unknowns about modern disks due to their multimodal behaviors, undocumented features, and interactions with the disk controller, DMA, memory manager, IO scheduling, and so on. Since there is no bounding contract between the vendors and the storage system designers other than the device driver interface, cross-layer coordination of storage system needs to be able to extract rapidly evolving disk characteristics and exploit them automatically.

Second, it is difficult to guarantee homogeneous disk behaviors even for the same model that are purchased from the same batch and made by the same vendor. In the case of using many disks, the extraction of hardware characteristics might impose high (one-time) overhead, depending on whether the extraction process can be performed in parallel.

Finally, exploiting hardware characteristics and coordinating parallel components is not a trivial task due to the size of configuration space. Also, optimizations tailored to one aspect (e.g., allocation granularity) of the data-handling characteristics may interact with other aspects (e.g., data alignment and scheduling) in intricate ways. Thus, storage system designers must have a good understanding of how various mechanisms interact to achieve predictable results. Unfortunately, such understanding is largely missing due to the legacy complexity of the storage data path. Unless optimization can bring sufficient benefit to overpower various interactions, the combined performance gains can deviate significantly from those realized in isolation.

8 Related Work

Since the early file systems, storage designers have realized the power of tailoring file system design to the underlying disk characteristics. FFS [McKusick et al. 1984] exploits the spatial locality of accessing consecutive and nearby blocks on disks to improve performance. LFS [Matthews et al. 1997] exploits both spatial locality for sequential writes and temporal locality for nearby disk-block reads for performance improvement.

Within recent years, cross-layer coordination with low-level storage has begun to attract research attention. In addition to the track-alignment, Lumb et al. [2000] exploit the rotational bandwidth that can be extracted during seeks in order to perform low-priority disk requests. Sivathanu et al. [2003] and Sivathanu et al. [2005] made low-level storage aware of the file systems and database applications running above them, so the data location policies could be optimized according to the semantic knowledge of the file system and the database data structures. Atropos [Schindler et al. 2004] stripes data across disks in a track-aligned manner and supports two dimensional data structures via efficient diagonal access to blocks on adjacent tracks. Schlosser et al. [2005] exploit the low seek time for nearby tracks in order to place multidimensional data sets.

Exposing the use of many disks to the file-system level leads to many parallel file system designs. For instance, PVFS [Carnes et al. 2000] modifies the semantics of file system calls to reduce disk contentions. GPFS [Schmuck and Haskin 2002] performs file striping and mirroring at the file system level to achieve good load balancing and reliability.

The Conquest file system [Wang et al. 2002] shows that, by exposing the use of memory storage medium to the file-system level, tailored optimizations achieve a 3.5x speedup compared to conventional memory caching.

In addition to file systems, disk queue performance can be improved further by investigating underlying hardware. For instance, Lo et al. [2005] discovered that shortest-job-first scheduling of independent disks in RAID-0 can interleave striped requests, which leads to longer latency for each stripe to complete. They propose a least-remaining-request-size-first strategy to reduce latency.

Further, cross-layer coordination can be applied at levels higher than the physical hardware to improve performance. For example, Nugent et al. [2003] provided bypass mechanisms for user-level applications to directly control data storage locations on disks to improve the chance of sequential accesses.

9 Conclusion

Through the clean-room duplication of the track-aligned access and its incorporation in RAIDs, and through exploring a proposed method to coordinate per-disk queues in RAIDs, we have validated the performance benefits achievable by applying the cross-layer coordination technique to storage. However, we also need to overcome the diversity of disk behaviors and the size of design space to take advantage of hardware details.

Therefore, for the cross-layer approach to become broadly applicable at these levels, we need to overcome rapid hardware evolution by inventing ways to obtain hardware characteristics efficiently and exploit them automatically. The cross-layer coordination approach also prompts us to either develop a better understanding of the legacy storage data path or simplify the data path enough to make it understandable; otherwise, the benefits of the end-point optimization can be potentially reduced due to unforeseen interactions, or diffused due to the need to explore a vast configuration space.

Acknowledgements

We thank Mark Stanovich and Adaptec for helping us bypass some RAID controller features. We also thank Peter Reiher, Geoff Kuenning, Sarah Diesburg, Christopher Meyers, and Mark Stanovich for reviewing early drafts of this paper. This research is sponsored by NSF CNS-0410896. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, FSU, or the U.S. government.

References

[Anderson 2003] Anderson D. You Don't Know Jack about Disks. *Storage*. 1(4), 2003.

[Bray 1996] Bray T. Bonnie benchmark. http://www.textuality.com/bonnie/download.html, 1996.

[Card et al. 1999] Card R, Ts'o T, Tweedie S. Design and Implementation of the Second Extended Filesystem. *The HyperNews Linux KHG Discussion*. http://www.linuxdoc.org (search for ext2 Card Tweedie design), 1999.

[Carnes et al. 2000] Carns PH, Ligon WB III, Ross RB, Thakur R. PVFS: A Parallel File System For Linux Clusters. *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[Fujitsu 2007] MAP3147NC/NP MAP3735NC/NP MAP3367NC/NP Disk Drives Product/Maintenance Manual. http://www.fujitsu.com/downloads/COMP/fcpa/hdd/discontinued/map-10k-rpm_prod-manual.pdf, 2007.

[Iyer and Druschel 2001] Iyer S, Druschel P. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[Lo et al. 2005] Lo SW, Kuo TW, Lam KY. Multi-disk Scheduling for Time-Constrained Requests in RAID-0 Devices. Journal of Systems and Software, 76(3), pp. 237-250, 2005.

[Lumb et al. 2000] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives. *Symposium on Operating Systems Design and Implementation. USENIX Association, 2000.*

[Matthews et al. 1997] Matthews JN, Roselli D, Costello AM, Wang RY, Anderson TE. Improving the Performance of Log-Structured File Systems with Adaptive Methods. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 238-251, October, 1997.

[Maxtor 2004] Atlas 10K V Ultra320 SCSI Hard Drive. http://www.darklab.rutgers.edu/MERCURY/t15/disk.pdf, 2004.

[McKusick et al. 1984] McKusick MK, Joy WN, Leffler SJ, Fabry RS. A Fast File System for UNIX, Computer Systems, 2(3), pp. 181-197, 1984.

[Nugent et al. 2003] Nugent J, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Controlling Your PLACE in the File System with Gray-box Techniques. *Proceedings of the USENIX Annual Technical Conference*, 2003.

[Patterson et al. 1988] Patterson DA, Gibson G, Katz RH, A Case for Redundant Arrays of Inexpensive Disks (RAID). *ACM SIGMOD International Conference on Management of Data*, 1(3), pp.109-116, 1988.

[Saltzer et al. 1981] Saltzer JH, Reed DP, Clark DD. End-to-End Arguments in System Design. Proceedings of the 2nd International Conference on Distributed Systems, 1981.

[Schindler and Ganger 1999] Schindler J, Ganger GR. Automated Disk Drive Characterization. CMU SCS Technical Report CMU-CS-99-176, December 1999.

[Schindler et al. 2002] Schindler J, Griffin JL, Lumb CR, Ganger GR. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[Schindler et al. 2004] Schindler J, Schlosser SW, Shao M, Ailamaki A, Ganger GR. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[Schlosser et al. 2005] Schlosser SW, Schindler J, Papadomanolakis S, Shao M, Ailamaki A, Faloutsos C, Ganger GR. On Multidimensional Data and Modern Disks. *Proceedings of the 4th USENIX Conference on File and Storage Technology*, 2005.

[Schmuck and Haskin 2002] Schmuck F, Haskin R. GPFS: A Shared-Disk File System for Large

Computing Clusters. *Proceedings of the 1st Conference on File and Storage Technologies*, 2002.

[Seagate 2007] Product Manual: CheetahR 15K.4 SCSI. http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/cheetah/15K.4/SCSI/100220456d.pdf, 2007.

[Sivathanu et al. 2003] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[Sivathanu et al. 2005] Sivathanu M, Bairavasundaram LN, Arpaci-Dusseu AC, Arpaci-Dusseau RH. Database-Aware Semantically-Smart Storage. *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005.

[Wang et al. 2002] Wang AI, Kuenning GH, Reiher P, Popek GJ. *Conquest*: Better Performance Through a Disk/Persistent-RAM Hybrid File System, *Proceedings of the 2002 USENIX Annual Technical Confere*nce, June 2002.

[Worthington et al. 1995] Worthington BL, Ganger GR, Patt YN, Wilkes J. On-line Extraction of SCSI Disk Drive Parameters. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, 1995.