

Avoiding File System Micromanagement with Range Writes

Ashok Anand, Sayandeep Sen, Andrew Krioukov*, Florentina Popovici†
Aditya Akella, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Suman Banerjee
University of Wisconsin, Madison

Abstract

We introduce *range writes*, a simple but powerful change to the disk interface that removes the need for file system micromanagement of block placement. By allowing a file system to specify a list of possible address targets, range writes enable the disk to choose to which address to write the request; the disk improves performance by accessing the “closest” location and subsequently reporting its choice to the file system above. The result is a clean separation of responsibility; the file system (as high-level manager) provides coarse-grained control over placement, while the disk (as low-level worker) makes the final fine-grained placement decision to improve write performance. We show the benefits of range writes through numerous simulations and a prototype implementation, in some cases improving performance by a factor of three, across both synthetic and real workloads.

1 Introduction

File systems are a classic example of micromanagement gone awry. Consider block placement decisions; although the file system has little idea what is going on inside the disk, the file system decides the exact location of each block. The result is poor performance: although the file system only has a coarse-grained intention (*i.e.*, this block should be placed within the block group where its inode resides), it applies fine-grained control (*i.e.*, it specifies a single target address), likely increasing seek and rotational overheads needlessly.

Micromanagement of block placement arose due to the organic evolution of the disk interface. Early file systems such as FFS [20] understood details of disk geometry, including aspects such as cylinders, tracks, and sectors. With these physical characteristics exposed, it is not surprising that file systems evolved to exert control over them.

The interface to storage has become more abstract over time. Currently, a disk presents itself as a linear array of blocks, each of which can be read or written [2, 19]. On the positive side, the interface is simple to use: file systems simply place blocks within the linear array, making it straightforward to utilize the same file system across a broad class of storage devices.

On the negative side, the disk hides critical information from the file system, including the exact logical-to-physical mapping of blocks as well as the current position of the disk head. As a result, the file system does not have an accurate understanding of disk internals. However, the current interface to storage demands such knowledge, particularly when placing a block on disk. The file system must specify a single address to write a block to; the disk must obey this directive, and thus may incur needless seek and rotational overheads. The result: the file system micromanages block placement but without enough information or control to make the decision properly, precisely the case where micromanagement fails to work effectively [6].

Previous work has tried to remedy this dilemma in a number of ways. For example, some have suggested that disks remap blocks on-the-fly, and thus increase performance when writing [7, 9, 30]. Others have suggested a whole-sale move to a higher-level, object-based interface [1, 12]. The former approach is too costly and complex, requiring that the disk track a large amount of persistent metadata; the latter approach requires whole-sale change to existing file systems and disks, and thus inhibits deployment. An ideal approach would instead require little change to existing systems while enabling high performance.

In this paper, we introduce an evolutionary change to the disk interface to address the problem of micromanagement: *range writes*. The basic idea is simple: instead of the file system specifying a single exact address per write, a *list of ranges* is given to the disk. The disk is then free to pick which address to write to based on its internal positioning information, thus minimizing positioning costs. When the request is completed, the disk additionally informs the file system which address it chose, thus allowing for proper bookkeeping. By design, range writes retain the many benefits of the existing interface, and require only small changes to both the file system and disk to be effective.

Range writes make a more successful two-level managerial hierarchy possible. Specifically, the file system (*i.e.*, the manager) can exert coarse-grained control over block placement; by specifying a list of possible targets, the file system gives freedom to the disk without relinquishing all control. In turn, the disk (*i.e.*, the worker)

*Now a Ph.D. student at U.C. Berkeley

†Now at Google

is given the ability to make the best possible fine-grained decision, using all available internal information.

Implementing and utilizing range writes does not come without challenges, however. Specifically, drive scheduling algorithms must change to accommodate ranges efficiently. Thus, we develop two novel approaches to scheduling of range writes. The first, *expand-and-cancel scheduling*, integrates well with current schedulers but induces high computational overhead; the second, *hierarchical range scheduling*, requires a more extensive reworking of the disk scheduling infrastructure but minimizes computational costs as a result. Through simulation, we show that both of these schedulers achieve excellent performance, reducing write latency dramatically as the number of targets increases.

In addition, file systems must evolve to use range writes. We thus explore how range writes could be incorporated into the allocation policies of a typical file system. Specifically, we build a simulation of the Linux ext2 file system and explore how to modify its policies to incorporate range writes. We discuss the core issues, and present results of running a range-aware ext2 in a number of workload scenarios. Overall, we find that range writes can be used to place some block types effectively (*e.g.*, data blocks), whereas other less flexibly-placed data structures (*e.g.*, inodes) will require a more radical redesign to obtain the benefits of using ranges.

Finally, to gain experience with implementation, we develop a software-based prototype that demonstrates how range writes can be used to speed up writes to the log in a journaling file system. Our prototype transforms the Linux ext3 *write-ahead log* into a more flexible *write-ahead region*, and in doing so avoids rotational overheads from log writes. Under a transactional workload, we show how fast journal writes can improve performance by nearly a factor of two.

The rest of this paper is organized as follows. In Section 2, we discuss why previous efforts are either too complex or failure-prone. We then describe range writes in Section 3, and study disk scheduling in Section 4. In Section 5, we study how to modify file system allocation to take advantage of range writes, and then we describe our prototype implementation of fast journal writing in Section 6. Finally, in Section 7, we conclude.

2 Background

We first give a brief tutorial about how modern disk drives behave; more details are available elsewhere [24]. We then review existing approaches for minimizing positioning overheads of a disk.

2.1 Disks

A disk drive contains one or more *platters*, where each platter *surface* has an associated disk head for reading and

writing. Each surface has data stored in a series of concentric circles, or *tracks*. Within each track are the *sectors*, the smallest amount of data that can be read or written on the disk. A single stack of tracks at a common distance from the spindle is called a *cylinder*. Modern disks also contain RAM to perform caching.

The disk appears to its client, such as the file system, as a linear array of logical blocks; thus, each block has an associated logical block number, or LBN. These logical blocks are then mapped to physical sectors on the platters. This indirection has the advantage that the disk can lay out blocks to improve performance, but it has the disadvantage that the client does not know where a particular logical block is located. For example, optimizations such as zoning, skewing, and bad-block remapping all impact the mapping in complex ways.

The service time of reading or writing a request has two basic components: *positioning time*, or the time to move the disk head to the first sector of the current request, and *transfer time*, or the time to read/write all of the sectors in the current request. Positioning time has two dominant components. The first component is *seek time*, moving the disk head over the desired track. The second component is *rotational latency*, waiting for the desired block to rotate under the disk head. The time for the platter to rotate is roughly constant, but it may vary around 0.5% to 1% of the nominal rate. The other mechanical movements (*e.g.*, head and track switch time) have a smaller but non-negligible impact on positioning time [25].

Most disks today also support *tagged-command queuing* [22], in which multiple outstanding requests can be serviced at the same time. The benefit is obvious: with multiple requests to choose from, the disk itself can schedule requests it sees, and, by using detailed knowledge of positioning and layout, improve performance.

2.2 Reducing Write Costs

The idea of minimizing write time is by no means new [18]. However, there is as of yet no consensus on the best approach or the best division of labor between the disk and the file system for achieving this. We briefly describe previous approaches why they are not ideal.

2.2.1 Disk Scheduling

Disk scheduling has long been used to reduce positioning overheads for writes for many years. Early schedulers, built into the OS, tried to reduce seek costs with simple algorithms such as elevator and its many variants.

More recently, schedulers have gone beyond seek optimizations to include rotational delay. The basic idea is to reorganize requests to service the request with the *shortest positioning time first (SPTF)*, instead of simply the request with the shortest seek time (*SSTF*). Performing rotationally-aware scheduling within the disk itself is relatively straightforward since the disk has complete and

accurate information about the current location of the disk head and the location of each requested block. In contrast, performing rotationally-aware scheduling within the OS is much more challenging, since the OS must predict the current head position. As a result, much the scheduling work has been performed through simulation [17, 26] or has been crafted with extreme care [16, 21, 32, 34]. More fundamentally, disk scheduling alone cannot completely eliminate rotational delays. For example, if too few requests exist in the scheduling queue, smart scheduling cannot avoid rotation.

2.2.2 Smart Placement

Many researchers have noted that another way to minimize write delay is to appropriately control the placement of blocks on disk. This work can be divided into two camps: that which has the disk itself decide where blocks should be placed and that which assumes the layer above the disk makes this decision. Most often, when investigating how to optimize write requests, researchers have pushed the decision into the disk; when optimizing read requests, researchers have usually pushed the work into the file system. Each approach has its own strengths and weaknesses, which we now discuss.

Disk Placement: In the first approach, the disk itself controls the layout of logical blocks written by the file system onto the physical blocks in the disk. The basic approach has been to perform *eager writing*, in which the data is written to the free disk block currently closest to the disk head. There are three basic problems with these approaches. First, this approach assumes that an *indirection map* exists to map the logical block address used by the file system to its actual physical location on disk [7, 9, 30]. Unfortunately, updating the indirection map atomically and recovering after crashes can incur a significant performance overhead. Second, these systems need to know which blocks are free versus allocated. Unfortunately, although the file system readily knows the state of each logical block, it is quite challenging for disks to know whether a block is live or dead [27]. Third, this approach forces the file system to completely relinquish any control over placement; given that the file system knows which blocks are related to one another and thus are likely to exhibit temporal locality (*e.g.*, the inode and all data blocks of the same file), the file system would like to ensure that those blocks are placed somewhat near one another to optimize future reads. Thus, pushing full responsibility for block placement into the disk is not the best division of labor.

A related effort allows the disk to control placement but requires a new interface to disk. Originally termed network-attached storage devices (NASD) [12], some now refer to the idea more generally as object-based disks [1]. With this new interface, the disk controls exactly where each object is placed, and thus can make in-

telligent low-level decisions. However, the object-based approach also has its drawbacks. First, and most importantly, it requires more substantial change of both disks and the clients that use them, which is likely a major impediment to widespread acceptance. Second, allowing the disk to manage objects implies that the disk must now be concerned with consistent update; for example, when adding a new block to an object, both the new block and a pointer to it must be allocated inside the disk, and committed to disk in a consistent fashion. Thus, the disk must now also include some kind of journaling machinery (perhaps to NVRAM), duplicating effort and increasing the complexity of the drive.

File System Placement: In the second approach, the file system controls where blocks are positioned on disk. This approach has been often applied to improve read performance; the basic idea is for the file system to *replicate* blocks and to try to read from the closer copy [15, 13, 34]. The basic problem is that the file system does not know the precise location of the disk head at any point in time. Thus, replication at the file system level is better for minimizing seek costs than rotation [15]; alternatively, the file system can try to predict the current location of the disk head, but this approach is fragile [34].

2.3 A Cooperative Approach

In contrast to file system or disk-controlled placement, range writes divide the responsibilities of performing fast writes across both parties; this tandem approach is reminiscent of scheduler activations [3], in which a cooperative approach to thread scheduling was shown to be superior to either a pure user-level or kernel-level approach. Here, the file system does what it does best: makes coarse-grained layout decisions, manages free space, tracks block ownership, and so forth. The disk does what it does best: take advantage of low-level knowledge of disk internals (*e.g.*, head position, actual physical layout) to improve performance. The small change required does not greatly complicate either system while increasing the chances of deployment.

3 Range Writes

In this section, we describe range writes. We describe the basic interface as well as numerous details about the interface and its exact semantics. We conclude with a brief discussion of the natural counterpart of range writes: range reads.

3.1 The Basic Interface

Current disks support the ability to write data of a specified length to a given address on disk. With range writes, the disk supports the ability to write data within a list of *ranges*. Each specified range R is a pair of addresses, B_{begin} , B_{end} , designating that data of a given length can be written to any contiguous range of blocks fitting within

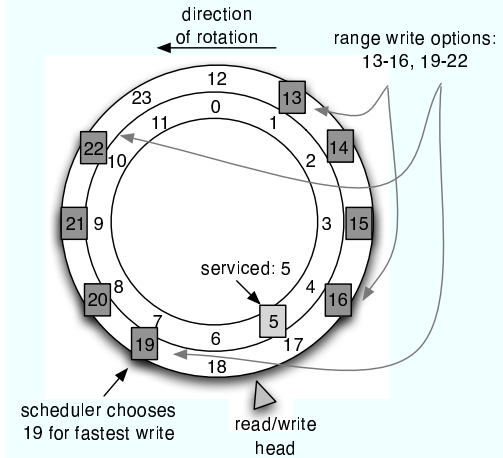


Figure 1: Range Writes. The figure illustrates how a file system might use range writes to perform a write quickly. The disk has just serviced a write to block 5, and is then given a write with two ranges: blocks 13 through 16, and blocks 19 through 22. The disk, given its current position, realizes that 19 will result in the fastest write, and thus chooses it as the write target. When finished, the file system is informed of the decision.

Classic Write
input: address, data, length
output: status
Range Write
input: num ranges, list of ranges, align data, length
output: status, resulting target address

Table 1: Classic vs. Range Writes. The table shows the differences between the classic idealized disk write and a range write.

B_{begin} and B_{end} . A full range write specifies a list of such ranges.

When the operation completes, the disk returns the starting address to which it wrote the data. This information allows the file system to record the address of the data block in whatever structure it is using (e.g., an inode). This interface is shown in Table 1.

One option the client must include is the alignment of writes, which restricts where in the range a write can be placed. For example, if the file system is writing a 4-KB block to a 16-KB range, it would likely specify an alignment of 4 KB, thus restricting the write to one of four locations. Without such an option, the disk could logically choose to start writing at any 512-byte offset within the range.

The interface also guarantees that within a single multi-block write, there is no reordering of blocks. This design decision enables the requester to control the ordering of blocks on disk, which may be important for subsequent (sequential) read performance.

3.1.1 Lists, Ranges, and Lists of Ranges

We initially considered a single range (instead of a list of ranges), but found it was too restrictive. For example, a file system may have a large amount of free space on a given track, but the presence of just a few allocated blocks in said track would greatly reduce the utility of single-range range writes.

We also considered a list of writes approach (without ranges). While this approach delivers equivalent functionality, we felt it added too much overhead to each write command. A file system might wish to specify roughly a large number of choices; with a range, this (in the best case) is just the start and end of a range; in the list approach, it will comprise hundreds or even thousands of target addresses.

Because of these reasons, we decided upon the highly flexible list of ranges. With this approach, the file system can usually specify its exact desires in compact form.

3.1.2 Overlapping Ranges

Modern disks allow multiple outstanding writes. While generally improving performance, having multiple outstanding requests generally complicates range writes. In particular, a file system may wish to issue two requests to the disk, R_1 and R_2 . Assume both requests should end up near one another on disk (e.g., they are to files that live in the same cylinder group). Assume also that the file system has a range of free blocks in that region of the disk, B_1 through B_n .

Thus, the file system would like to issue both requests R_1 and R_2 simultaneously, giving each the range B_1 through B_n . However, the disk is thus posed with a problem: how can it ensure it does not write the two requests to the same location?

The simplest solution would be to disallow overlapped writes, much like many of today’s disks do not allow multiple outstanding writes to the same address (“overlapped commands” in SCSI parlance [31]). In this case, the file system would have two choices. First, it could serialize the two requests, first issuing R_1 , observing which block it was written to (say B_k), and then submitting request R_2 with two ranges (B_1 to B_{k-1} and B_{k+1} to B_n). Alternately, the file system could pick subsets of each range (e.g., B_1, B_3, \dots, B_{k-1} in one range, and B_2, B_4, \dots, B_k in the other), and issue the requests in parallel.

However, we felt the non-overlapped approach was too restrictive; it forces the file system to reduce the number of targets per write request in anticipation of their use and thus reduces performance. Further, non-overlapped ranges complicate use of range writes, as a client must then make decisions on which portions of the range to give to which requests, likely decreasing the disk’s control over low-level placement and thus decreasing performance. For all of these reasons, we chose to allow clients

to issue multiple outstanding range writes with overlapping ranges.

Overlapping writes complicate the implementation of range writes within the disk. Consider our example above, where two writes R_1 and R_2 are issued concurrently, each with the range B_1 through B_n . In this example, assume that the disk schedules R_1 first, and places it on block B_1 . It is now the responsibility of the disk to ensure that R_2 is written to any block except B_1 . Thus, the disk must (temporarily) note that B_1 has been used.

However, this action raises a new question: how long does the disk have to remember the fact that B_1 was written to and thus should not be considered as a possible write target? One might think that the disk could forget this knowledge once it has reported that R_1 has completed (and thus communicated that the target B_1 was chosen). However, because the file system may be concurrently issuing another request R_3 to the same range, a race condition could occur and block B_1 could be (erroneously) overwritten.

Thus, we chose to add a new kind of write barrier to the protocol. A file system would use this feature as follows. First, the file system would issue a number of outstanding writes, potentially to overlapping ranges. The disk would start servicing them, and in doing so would track which blocks in each range get written. The file system at some point would then issue a barrier. What the barrier guarantees to the disk is that all writes following the barrier will take into account the allocation decisions of the disk for the writes before the barrier. Thus, once the disk completes the pre-barrier writes, it can safely forget which blocks it wrote to during that time.

3.2 Beyond Writes: Range Reads

It is of course a natural extension to consider whether range reads should also be supported by a disk. Range reads would be useful in a number of contexts: for example, to pick the rotationally-closest block replica [34, 15], or to implement efficient “dummy reads” in semi-preemptible I/O [8].

However, introducing range reads, in particular for improving rotational costs on reads, requires an expanded interface and implementation from the disk. For example, for a block to be replicated properly to reduce rotational costs, it should be written to opposite sides of a track. Thus, the disk should likely support a replica-creating write which tries to position the blocks properly for later reads. In addition, file systems would have to be substantially modified to support tracking of blocks and their copies, a feature which only a few recent file systems support [28]. Given these and other nuances, we leave range reads for future work.

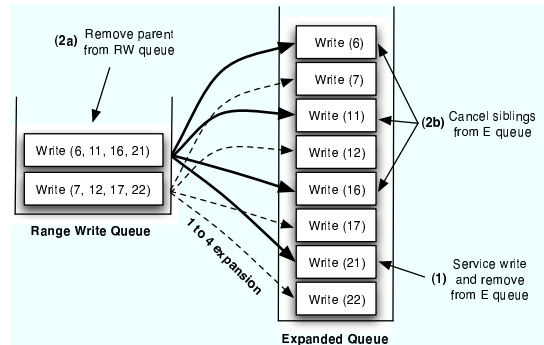


Figure 2: **Expand-and-cancel Scheduling.** The figure depicts how expand-and-cancel scheduling operates. Range writes are placed into the leftmost queue and then expanded into the full set of writes on the right. In step (0) (not shown), two range writes arrive simultaneously and are placed in the range write queue on the left; their expansions are placed in the expanded queue on the right. In step (1), the scheduler (which examines all requests in the expanded queue and greedily chooses the one with minimal latency) decides to service the write to 21. As a result, the range write to (6, 11, 16, 21) is removed from the range write queue (2a), and the expanded requests to 6, 11, and 16 are canceled (2b).

4 Disk Scheduling

In this section, we describe how an in-disk scheduler must evolve to support range writes. We present two approaches. The first we call *expand-and-cancel scheduling*, a technique that is simple, integrates well with existing schedulers, and performs well, but may be too CPU intensive. Because of this weakness, we present a competing approach known as *hierarchical-range scheduling*, which requires a more extensive restructuring of a scheduler to become range aware but thus avoids excessive CPU overheads.

Note that we focus on obtaining the highest performance possible, and thus consider variants of shortest-positioning-time-first schedulers (SPTF). Standard modifications could be made to address fairness issues [26, 17].

4.1 Expand-and-cancel Scheduling

Internally, the in-disk scheduler must be modified to support servicing of requests within lists of ranges. One simple way to implement support for range writes would be through a new scheduling approach we call *expand-and-cancel scheduling (ECS)*, as shown in Figure 2. In the expand phase, a range write request R to block range B_1 through B_n is expanded into n independent requests, R_1 through R_n , each to a single location, B_1 through B_n , respectively. When the first of these requests complete (as dictated by the policy of the scheduler), the other requests would be canceled (*i.e.*, removed from the scheduling queue). Given any scheduler, ECS guarantees that the best scheduling decision over all range writes (and other requests) will be made, to the extent possible given the current scheduling algorithm.

The main advantage of ECS is its excellent integration with existing disk schedulers. The basic scheduling policy is not modified, but simply works with more requests to decide what the best decision to make is. However, this approach can be computationally expensive, requiring extensive queue reshuffling as range writes enter the system, as well as after the completion of each range write. Further, with large ranges, the size of the expanded queue grows quite large; thus the number of options that must be examined to make a scheduling decision may become computationally prohibitive.

Thus, we expect that disk implementations that choose ECS will vary in exactly how much expansion is performed. By choosing a subset of each range request (e.g., 2 or 4 or 8 target destinations, equally spaced around a track, for example), the disk can keep computational overheads low while still reducing rotational overhead substantially. More expensive drives can include additional processing capabilities to enable a greater number of targets, thus allowing for differentiation among drive vendors.

4.2 Hierarchical-Range Scheduling

As an alternative to ECS, we present an approach we call *hierarchical-range scheduling (HRS)*. HRS requires more exact knowledge of drive details, including the current head position, and thus requires a more extensive reworking of the scheduling infrastructure. However, the added complexity comes with a benefit: HRS is much more CPU efficient than ECS, doing much less work to obtain similar performance benefits.

HRS works as follows. Given a set of ranges (assuming for now that each range fits within a track), HRS determines the time it would take to seek and settle on the track of each request and what the resulting head position would be. If the head is within the range on that track, HRS chooses the next closest spot within the range as the target, and thus estimates the total latency of positioning, roughly the cost of the seek and settling time. If the head is outside the range, HRS includes an additional rotational cost to get to the first block of the range. Because HRS knows the time these close-by requests will take, it can avoid considering those requests whose seek times already exceed the current best value. In this manner, HRS can consider many fewer options and still minimize rotational costs.

An example of the type of decision HRS makes is found in Figure 3. In the figure, two ranges are available: 9-11 (on the current track) and 18-20 (on an adjacent, outer track). The disk has just serviced a request for block 5 on the inner track, and thus must choose a target for the current write. HRS observes that range 9-11 is on the same track and thus estimates the time to write to 9-11 is the time to wait until 9 rotates under the head. Then, for the

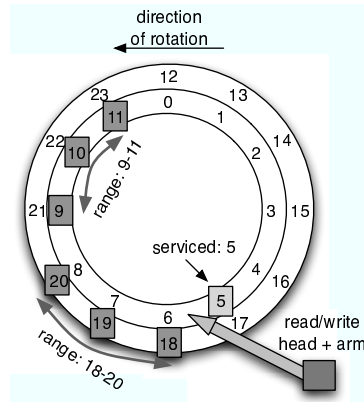


Figure 3: **Hierarchical-range Scheduling.** The figure depicts an example how hierarchical-range scheduling works. The text describes the example in more detail.

18-20 range, HRS first estimates the time to move the arm to the adjacent track; with this time in hand, HRS can estimate where the head will be relative to the range. If the seek to the outer track is fast, HRS determines that the head will be within the range, and thus chooses the next block in the range as a target. If, however, the short seek takes too long, the arm may arrive and be ready to write just after the range has rotated underneath the head (say at block 21). In this case, HRS estimates the cost of writing to 18-20 as the time to rotate 18 back under the head again, and thus would choose to instead write to block 9 in the first range.

A slight complication arises when a range spans multiple tracks. In this case, for each range, HRS splits the request into a series of related requests, each of which fit within a single track. Then, HRS considers each in turn as before. Lists of ranges are similarly handled.

4.3 Methodology

We now wish to evaluate the performance of our range write schedulers. To do so, we utilize a detailed simulation environment built within the DiskSim framework [5].

Specifically, we made numerous small changes throughout the simulator to provide support for range writes. We implemented a small change to the interface to enable lists of ranges to be passed to the disk, and more extensive changes to the SPTF scheduler to implement both EC and HR scheduling. Overall, we changed or added less than one thousand lines of code, perhaps hinting at the relative simplicity of adding range writes to a system.

For all simulated experiments, we use the HP C2249A disk, which has a rotational speed of 5400 RPM, and a relatively small track size (roughly 90 blocks, depending on the zone). Although it is an older model, and thus its absolute positioning costs are high and track size low as compared to modern disks, the relative costs between seek

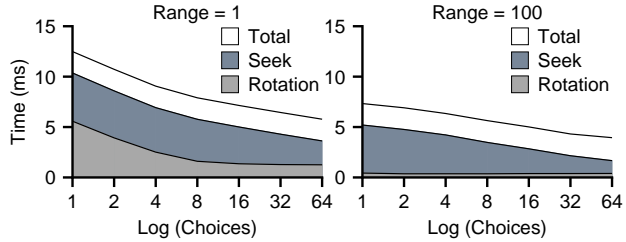


Figure 4: **Scheduler Performance.** The figure shows how performance improves when the scheduler is given more flexibility. Along the x-axis, we increase the number of targets given per write request, and the two graphs show two different range sizes, from 1 (on the left) to 100 (on the right). All requests are made within a small portion of the disk (a block group), and the average time per write is plotted, including breakdowns of seek and rotational costs.

and rotation is reasonable and thus we believe our results generalize quite well to a large variety of modern disks.

4.4 Experiments

4.4.1 What Is The Benefit Of Range Writes?

We first wish to investigate how having more choices (*i.e.*, a larger list) as well as having larger ranges can improve performance. Figure 4 presents the results of a set of experiments which write blocks to random destinations within a small portion of a disk (roughly the size of a small block group, for example). Here, we use the HR scheduler (the EC scheduler performs equivalently when using the full expansion).

On the left, we see a graph that presents the time per write as we increase the number of targets; we fix the range size of each target at 1. The targets are chosen randomly from the entire cylinder group, perhaps emulating that some blocks would be unavailable because they are in use. From the figure, one can observe the rapid decrease in total positioning costs (both seek and rotation) as the number of choices increases. Thus, given a number of potential targets, range writes reduce both rotation and seek times substantially.

On the right, we perform the same experiment, but increase the size of each range to 100 blocks (just greater than the track size of the disk). With large ranges to choose from, performance improves even further, removing virtually all rotational cost while greatly reducing seek costs as well. Overall, with large ranges and a large number of options (64), writes complete in roughly 4 ms (the rightmost point on the right graph), as compared to over 12 ms with a single choice and a range of size 1 (the leftmost point on the left graph). Thus, for random writes to a small portion of the disk, range writes improve performance by over a factor of three.

4.4.2 What If There Are Many Outstanding Writes?

Until now, we have issued only a single write to disk at a time. However, as most disks support tagged queuing, it

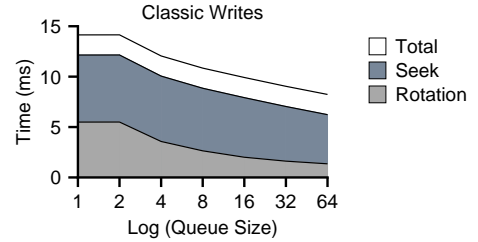


Figure 5: **Varying Queue Depth.** The figure shows how the performance of a traditional SPTF scheduler on a workload of writes to random locations within a block group on disk. The x-axis varies the number of outstanding requests. Note that having two outstanding requests does not improve performance, because in general the scheduler is servicing one request while another is being sent to the disk, thus removing the possibility of choosing the “better” request.

would likely be the case that multiple requests to the disk are outstanding at a given time. This enables a traditional SPTF scheduler to amortize positioning costs over those requests, and thus may lessen the relative benefit of range writes. We now investigate this issue.

In this set of experiments, we vary the number of outstanding requests to the disk. We plot the performance of the in-disk SPTF scheduler without range writes, as seen in Figure 5.

The results show that even with 64 outstanding requests, a traditional system incurs substantial positioning costs (both seek and rotation). With a small set of choices (*i.e.*, within a track), the remaining rotational overhead could be removed. With a larger set of choices (*i.e.*, within a block group), much of the seek overhead can also be removed (as we saw in Figure 4). Thus, even with multiple outstanding requests, range writes can noticeably improve performance.

4.4.3 What Is The Computational Difference Between ECS and HRS?

We next compare the costs of EC scheduling and HR scheduling. Most of the work that is done by either is the estimation of service time for each possible candidate request; thus, we compare the number of such estimations to gain insight on the computational costs of each approach.

Assume that the size of a range is R , and the size of a track on the disk is T . Also assume that the disk supports Q outstanding requests at a given time (*i.e.*, the queue size). We can thus derive the amount of work that needs to be performed by each approach. For simplicity, we assume each request is a write of a single block (generalizing to larger block sizes is straightforward).

For EC scheduling, assuming the full expansion, each single request in the range-write queue expands into R requests in the expanded queue. Thus, the amount of work, W , performed by ECS is:

$$W_{EC} = R \cdot Q \quad (1)$$

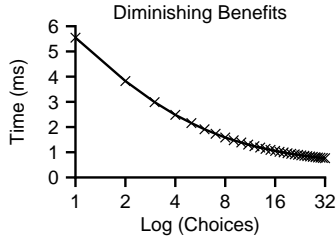


Figure 6: **The Diminishing Benefits of More Choice.** *The figure plots the performance of successive write requests. Along the x-axis, we increase the number of choices available for write targets, and the y-axis plots average write time.*

In contrast, HR scheduling takes each range and divides it into a set of requests, each of which is contained within a track. Thus, the amount of work performed by HRS is:

$$W_{HR} = \lceil \frac{R}{T} \rceil \cdot Q \quad (2)$$

However, HRS need not consider all these possibilities. Specifically, once the seek time to a track is higher than the current best seek plus rotate, HRS can stop considering whether to schedule this and other requests that are on tracks that are further away. The worst case number of tracks that must be considered is thus bounded by the number of tracks one can reach within the time of a revolution plus the time to seek to the nearest track. Thus, the equation above represents an upper bound on the amount of work HRS will do.

Even so, the equations make clear why HR scheduling performs much less work than EC scheduling in most cases. For example, assuming that a file system issues range writes that roughly match track size ($R = T$), the amount of work performed by HRS is roughly Q . In contrast, ECS still performs $R \cdot Q$ work; as track sizes can be in the thousands, ECS will have to execute a thousand times more work to achieve equivalent performance.

4.4.4 How Many Options Does ECS Need?

Finally, given that EC scheduling cannot consider the full range of options, we investigate how many options such a scheduler needs to obtain good performance. To answer this question, we present a simple workload which repeatedly writes to the same track, and vary the number of target options it is given. Figure 6 presents the results.

In this experiment, we assume that if there exists only a single option, it is to the same block of the track; thus, successive writes to the same block incur a full rotational delay. As further options are present, they are equally spaced around the track, maximizing their performance benefit.

From this figure, we can conclude that ECS does not necessarily need to consider all possible options within a range to achieve most of the performance benefit, as

expected. By expanding a full-track range to just eight choices that are properly spaced out across the track, most of the performance benefits can be achieved.

However, this example simplifies that problem quite a bit. For ranges that are larger than a single track, the expansion becomes more challenging; exactly how this should be done remains an open problem.

4.5 Summary

Overall, our study of scheduling has revealed a number of interesting results. First, range writes can have a dramatic effect on performance. In particular, with a large amount of flexibility, range writes can reduce both rotational and seek costs quite dramatically. Thus, to get the best performance, a file system (or other client) should give reasonably large ranges to the disk if possible.

Second, even with a reasonable queue size within the disk, a disk will incur noticeable seek and rotational overheads. Thus, although range writes are clearly useful when there are only a few outstanding writes to the disk, they are still of use when there are many.

Finally, both the EC and HR schedulers perform well, and thus are possible candidates for use within a disk that supports range writes. If one is willing to rewrite the scheduler, HR is the best candidate. However, if one wishes to use the simpler EC approach, one must do so carefully; the full expansion of ranges simply exacts too high of a computational overhead.

5 Integrating Range Writes into Classic File System Allocation

In this section, we explore the issues a file system must address in order to incorporate range writes into its allocation policies. We first discuss the issues in a general setting, and then describe our effort in building a detailed ext2 file system simulator to explore how these issues can be tackled in the context of an existing system.

5.1 File System Issues

There are numerous considerations a file system must take into account when utilizing range writes. Some complicate the file system code, some have performance ramifications, and some aspects of current file systems simply make using range writes difficult or impossible. We discuss these issues here.

5.1.1 Preserving Sequentiality

One problem faced by file systems utilizing range writes is the loss of exact control over placement of files. However, as most file systems only have approximate placement as their main goal (*e.g.*, allocate a file in the same group as its inode), loss of detailed control is acceptable.

Loss of sequentiality, however, would present a larger problem. For example, if a file system freely writes blocks of a file to non-consecutive disk locations, reading back

the file would suffer inordinately poor performance. To avoid this, the file system should present the disk with larger writes (which the disk will guarantee are kept together), or restrict ranges of writes to make it quite likely that the file will end up in sequential or near-sequential order on disk.

5.1.2 Determining Proper Ranges

Another problem that arises for the file system is determining the proper range for a request. How much flexibility is needed by the disk in order to perform well?

Of course, one could extract low-level information from the drive, including track size, using microbenchmarks [33]. However, we believe a simpler approach is for the file system to simply specify the range that naturally fits its policy best. For example, in FFS, the file system could specify that a write should take place to any free block within a cylinder group.

5.1.3 Bookkeeping

One major change required of the file system is how it handles a fundamental problem with range writes which we refer to as *delayed address notification*. Specifically, only as each write completes does the file system know the target address of the write. The file system cares about this result because it is in charge of bookkeeping, and must record the address in a pertinent structure (*e.g.*, an inode).

In general, this may force two alterations in file system allocation. First, the file system must ensure that it does not issue too many writes to a particular range; doing so may result in writes failing if the range fills up. However, this is a small modification.

Second, delayed notification forces an ordering on file systems, in that the block pointed to must be written before the block containing the pointer. Although reminiscent of soft updates [11], this ordering should be easier to implement, because the file system will likely not employ range writes for all structures, as we discuss below.

5.1.4 Inflexible Structures

Finally, while range writes are quite easy to use for certain block types (*e.g.*, data blocks), other fixed structures are more problematic. For example, consider inodes in a standard FFS-like file system. Each inode shares a block with many others (say 64 per 4-KB block). Writing an inode block to a new location would require the file system to give each inode a new inode number; doing so necessitates finding every directory in the system that contains those inode numbers and updating them.

Thus, we believe that range writes will likely be used at first only for the most flexible of file system structures. Over time, as file systems become more flexible in their placement of structures, range writes can more broadly be applied. The good news is that more modern file systems have more flexible structures, *e.g.*, Sun’s ZFS [28],

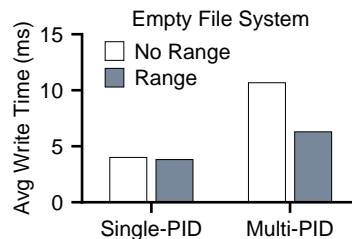


Figure 7: **File Create Time (Empty File System).** *The figure plots the average write time during a file create benchmark. The benchmark creates 1000 4-KB files in the same directory. Range writes are either used or not, and the files are either created by a single process or multiple processes. The y-axis plots the average write time of each write across the 1000 data block writes that occur.*

NetApp’s WAFL [14], and LFS [23] all take a “write-anywhere” approach for most file system structures.

5.2 Incorporating Range Writes into ext2

We now present our experience of incorporating range writes into a simulation of the Linux ext2 file allocation. Allocation in ext2 (and ext3) derives from classic FFS allocation [20] but has a number of nuances included over the years to improve performance. We now describe the basic allocation policies.

When creating a directory, the “Orlov” allocation algorithm is used. In this algorithm, top-level directories are spread out by searching for the block group with the least number of subdirectories and an above-average free block count and free-inode count. Other directories are placed in a block group meeting a minimum threshold of free inodes and data blocks and having a low directory-to-file ratio. In both cases the parent’s block group is preferred given that it meets all criteria.

The allocation of data blocks is done by choosing a goal block and searching for the nearest free block to the goal. For the first data block in the file the goal is found by choosing a block in the same group as the inode. The specific block is chosen by using the PID of calling process to select one of 16 start locations within the block group; we call each of these 16 locations of the disk “mini-groups” within the greater block group. The desire here is to place “functionally related” files closer on disk. All subsequent data block allocations for a given file have the goal set to the next sequential block.

To utilize range writes, our variant of ext2 tries to follow the basic constraints of the existing ext2 policies. For example, if the only constraint is that a file is placed within a block group, than we issue a range write that specifies the free ranges within that group. If the policy wishes to place the file within a mini-group, the range writes issued for that file are similarly constrained. We also make sure to preserve sequentiality of files. Thus, once a file’s first block is written to disk, subsequent

blocks are placed contiguously beyond it (when possible).

5.3 Methodology

To investigate ext2 use of range writes, we built a detailed file system simulator. The simulator implements all of the policies above (as well as a few others not relevant for this section) and is built on top of DiskSim. The simulator presents a file system API, and takes in a trace file which allows one to exercise the API and thus the file system. The simulator also implements a simple caching infrastructure, and writes to disk happen in a completely unordered and asynchronous fashion (akin to ext2 mounted asynchronously). We use the same simulated disk as before (the HP C2249A), set the disk-queue depth to 16, and utilize HR scheduling.

5.4 Results

5.4.1 Small File Creation on Empty File Systems

We first show how flexible data block placement can improve performance. For this set of experiments, we simply create a large number of small files in a single directory. Thus, the file system should create these files in a single block group, when there is space. For this experiment, we assume that the block group is empty to start.

Figure 7 shows the performance of small-file allocation both with and without range writes. When coming from a single process, using range writes does not help much, as all file data are created within the same mini-group and indeed are placed contiguously on disk. However, when coming from different processes, we can see the benefits of using range writes. Because these file allocations get spread across multiple mini-groups within the block group, the flexibility of range writes helps reduce seek and rotation time substantially.

We also wish to ensure that our range-aware file system makes similar placement decisions within the confines of the ext2 allocation policies. Thus, Figure 8 presents the breakdowns of which mini-group each file was placed in.

As one can see from the figure, the placement decisions of range writes, in both the single-process and multi-process experiments, closely follows that of the traditional ext2. Thus, although the fine-grained control of file placement is governed by the disk, the coarse-grained control of file placement is as desired.

5.4.2 Small File Creation on Fuller File Systems

We now move to a case where the block group has data in it to begin. This set of experiments varies the fullness of the block group and runs the same small-file creation benchmark (focusing on the single-PID case). Figure 9 plots the results.

From the figure, we can see that by the time a block group is 50% full, range writes improve performance over classic writes by roughly 20%. This improvement stays

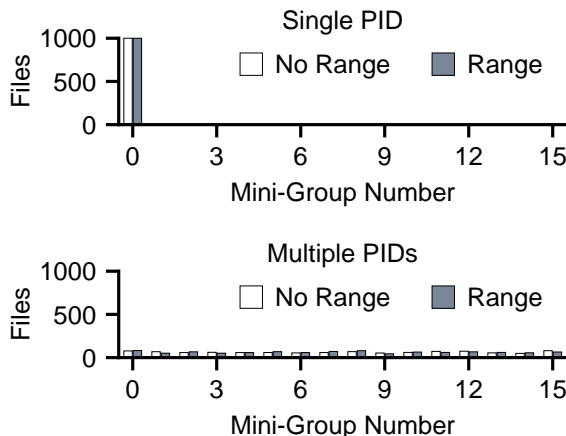


Figure 8: **File Placement.** The figure shows how files were placed per mini-group across two different experiments. In the first, a single process (PID) created 1000 files; in the second, each file was created by a different PID. The x-axis plots the mini-group number, and the y-axis shows the number of files that were placed in the mini-group, for both range writes and traditional writes.

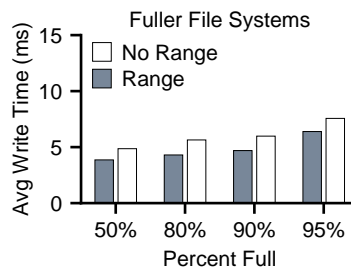


Figure 9: **File Create Time (Fuller File System).** The figure plots the average write time during a file create benchmark. The benchmark creates 1000 4-KB files in the same directory. Range writes are either used or not, and the files are either created by a single process. The x-axis varies the fullness of the block group, and the y-axis plots the average write time across the 1000 data block writes that occur.

roughly constant as the block group fills, even as the average write time of both approaches increases. We can also see the effect of fullness on range writes: with fewer options (as the block group fills), it is roughly 70% slower than it was with an empty block group.

5.4.3 Unpacking A Tarball

The first two synthetic benchmarks focused on file creation in empty or partially-full file systems, demonstrating some of the benefits of range writes. We now simulate the performance of an application-level workload. Specifically, we focus on a simple benchmark that unpacks the Linux source tree. We simply playback the file request stream that such an unpack generates to our simulator and report results.

With range writes, the unpack requires roughly 123 seconds to run, whereas without, it takes 143 seconds (roughly 20% slower). Thus, range writes are also effective

tive under this more realistic workload.

5.5 Summary

Integrating range writes into file system allocation has proven promising. As desired, range writes can improve performance during file creation while following the constraints the higher-level file system policies. As much of write activity is to newly created files [4, 29], we believe our range-write variant of ext2 would be effective in practice. Further, although limited to data blocks, our approach is useful because traffic is often dominated by data (and not metadata) writes.

Of course, there is much left to explore. For example, file overwrite presents an interesting scenario. For best performance, one should issue a range write to a new part of the disk and thus free the old location; however, such a strategy destroys file sequentiality, and thus must be done with care. We plan to explore this and many other workload scenarios in future work.

6 Case Study: Log Skipping

We now present a case study that employs range writes to improve journal update performance. Specifically, we show how a journaling file system (Linux ext3 in this case) can readily use range writes to more flexibly choose where each log update should be placed on disk.

Whereas the previous section employed simulation to study the benefits of range writes, in this section we utilize a prototype implementation. Doing so presents an innate problem: how do we experiment with range writes in a real system, when no disk (yet) supports range writes? To remedy this dilemma, we develop a software layer, Bark, that emulates a disk with range writes for this specific application, building on previous work [21, 10]. Our approach also hints at how range writes could be incrementally incorporated into systems, first via a software prototype to demonstrate real benefits and allow higher-level systems to make use of its features, and later in real hardware.

6.1 Motivation

The primary problem that we address in this section is how to improve the performance of synchronous writes to a log or journal. Thus, it is important to understand the sequence of operations that occur when the log is updated.

A journaling system writes a number of blocks to the log; these writes occur whenever an application explicitly forces the data or after certain timing intervals. First, the system writes a *descriptor block*, containing information about the log entry, and the actual data to the log. After this write, the file system waits for the descriptor blocks and data to reach the disk and then issues a synchronous *commit block* to the log; the file system must wait until the first write completes before issuing the commit block in case a crash occurs.

In an ideal world, since all of the writes to the log are sequential, the writes would achieve sequential bandwidth. Unfortunately, in a traditional journaling system, the writes do not. Because there is a non-zero time elapsed since the previous block was written, and because the disk keeps rotating at a constant speed, the commit block cannot be written immediately. The sectors that need to be written have already passed under the disk head and the disk has to perform an almost full rotation to be able to write the commit block.

Our approach here is to transform the write-ahead log of a journaling file system into a more flexible *write-ahead region*. Instead of issuing a transaction to the journal in the location directly following the previous transaction, we instead allow the transaction to be written to the next rotationally-closest location. This has the effect of spreading transactions throughout the region with small distances between them, but improves performance by minimizing rotation.

We should note that our goals here are similar to the goals in a paper by Gallagher *et al.* [10]. Therein, the authors take a similar approach but for a database log. Thus, our contribution here is to show how to apply the same idea to a file system journal, and to do so with a different (and we believe, more robust) performance model based on the disk mimic approach [21].

We now discuss how we implement write-ahead regions in our prototype system. The biggest challenge to overcome is the lack of range writes in the disk. Thus, we describe our software layer, Bark, which builds a model of the performance contours of the log (hence the name) and uses it to issue writes to the journal so as to reduce rotational overheads. We then describe our experiments with the Linux ext3 journal mounted on top of Bark.

6.2 Log-Performance Modeling

Bark is a layer that sits between the file system and disk and redirects journal writes so as to reduce rotational overhead. To do so, Bark builds a performance model of the log *a priori* and uses it to decide where best to write the next log write.

Our approach builds on previous work of Popovici *et al.* that measures the request time between all possible pairs of disk addresses in order to perform disk scheduling [21]. Our problem here is simpler: Bark must simply predict where to write the next request to in order to reduce rotation.

To make this prediction, Bark performs a series of measurements of the cost of writes to the portion of the disk of interest, varying both the distance between the writes (the “skip” distance) and the think time between requests. This data is stored in a table and made available to Bark at run-time.

For the results reported in this paper, we created a disk

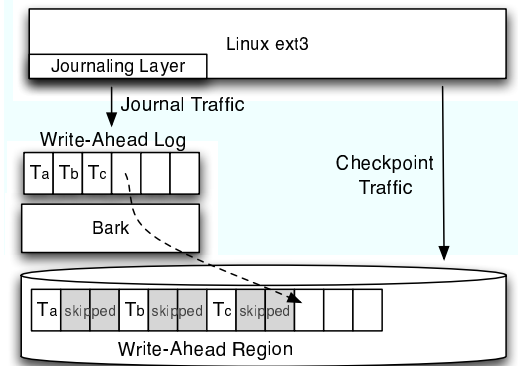


Figure 10: **Bark-itecture.** The figure illustrates how a file system can be mounted upon Bark to improve journal write performance. All journal traffic is directed through Bark, which picks a skip distance based on think time and the position of the last write to disk. Bark performs this optimization transparently, thus improving the performance of journal writes with no change to the file system above. In the specific example shown, the file system has committed three transactions to disk: T_a , T_b , and T_c . Bark, using its performance model, has spread the transactions across the physical disk, leaving empty spaces (denoted as “skipped”) in the write-ahead region.

profile by keeping a fixed write size of 4 KB (size of a block), and varying the think time from 0 ms to 80 ms in intervals of 50 microseconds, and the skip size from 0 KB to 600 KB in intervals of 512 bytes. To gain confidence each experiment was repeated multiple times and the average of the write times was taken.

6.3 From Models to Software

With the performance model in place, we developed Bark as a software pseudo-device that is positioned between the file system journaling code and the disk. Bark thus presents itself to the journaling code as if it were a typical disk of a given size S . Underneath, Bark transparently utilizes more disk space (say $2 \cdot S$) in order to commit journal writes to disk in a rotationally-optimal manner, as dictated by the performance model. Figure 10 depicts this software architecture.

At runtime, Bark receives a write request and must decide exactly where to place it on disk. Given the time elapsed since the last request completed, Bark looks up the required “skip distance” in the prediction table and uses it to decide where to issue the current write.

Two issues arise in the Bark implementation. The first is the management of free space in the log. Bark keeps a data structure to track which blocks are free in the journal and thus candidates for fast writes. The main challenge for Bark is detecting when a previously-used block becomes free. Bark achieves this by monitoring overwrites by the journaling layer; when a block is overwritten in the logical journal, Bark frees the corresponding physical

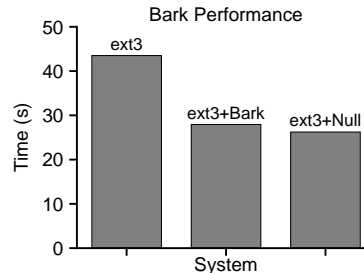


Figure 11: **Overall Performance.** The figure plots the overall performance of TPC-B in three different settings: without Bark, with Bark, and then on top of a “null” log that simply reports success for log writes without performing any actual disk I/O. Experiments were run upon a Sun Ultra20 with 1 GB of memory and two Hitachi Deskstar 7K80 drives attached.

block to which it had been mapped.

The second issue is support for recovery. Journals are not write-only devices. In particular, during recovery, the file system reads pending transactions from the journal in order to replay them to the file system proper and thus recover the file system to a consistent state. To enable this recovery without file system modification, Bark would need to record a small bit of extra information with each set of contiguous writes, specifically the address in the logical address space to which this write was destined. Doing so would enable Bark to scan the write-ahead region during recovery and reconstruct the logical address space, and thus allows recovery to proceed without any change to the file system code. However, we have not yet fully implemented this feature (early experience suggests it will be straightforward).

6.4 Results

We now measure the performance of unmodified Linux ext3 running on top of Bark. For this set of experiments, we mount the ext3 journal on Bark and let all other checkpoint traffic go to disk directly.

For a workload, we wished to find an application that stressed journal write performance. Thus, we chose to run an implementation of the classic transactional benchmark TPC-B. TPC-B performs a series of debits and credits to a simple set of database tables.

Figure 11 plots the performance of TPC-B on Linux ext3 in three separate cases. In the first, the unmodified traditional journaling approach is used; in the second, Bark is used underneath the journal; in the third, we implement an “infinitely fast” journal which simply returns success when given a write without doing any work. This last option serves as an upper-bound on performance.

From the graph, we can see that Bark greatly improves the overall runtime of TPC-B. Figure 12 reveals some insight as to why. In that figure, we plot the cumulative distribution of journal write times across all requests during the run. As one can see, most journal writes using

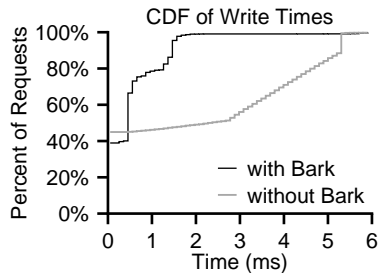


Figure 12: **Write Costs.** The figure plots the cumulative distribution of write request times during TPC-B. Two lines are plotted: the first shows the cost of all writes through Bark, whereas the second shows performance without.

Bark complete quite quickly, as they have been rotationally well placed. In contrast, writes to the journal without bark take much longer on average, and are spread across the rotational spectrum of the disk drive.

6.5 Discussion

From this case study, we learned two important lessons. First, we see that freedom during writes can also be useful for a file system journal. Under certain workloads, journaling can induce a large rotational cost; freedom to place transactions to a free spot in the journal can greatly improve performance.

Second, we see that such the flexibility provided by range writes works in practice, rather than just in simulation. Although we believe our earlier work on scheduling and file system simulation is an accurate predictor of real range-write performance, we felt much more comfortable overall when we were able to show that prototype could derive benefits from our approach.

We should note that we chose here to incorporate flexible writes underneath in the simplest possible way, without changing the file system implementation at all. Thus, if range writes existed within the disk, the Bark layer would do very little: simply issue the range writes to disk instead of performing modeling to find the next fast location to write to. A different approach would be to modify the file system code and change the journaling layer to support range writes more directly, something we plan to do in future work.

7 Conclusions

We have presented a small but important change to the storage interface, known as range writes. By allowing the file system to express flexibility in exact write location, the disk is free to make better decisions for write targets and thus improve performance.

We believe that the key element of range writes is their evolutionary nature; there is a clear path from today's disks without range writes to tomorrow's disks with them. This fact is crucial for established industries, where

change is fraught with many complications, both practical and technical; for example, consider object-based drives, which have taken roughly a decade to begin to come to market [12].

Interestingly, the world of storage may be in the midst of a revolution as solid-state devices become more of a marketplace reality. Fortunately, we believe that range writes are still quite useful in this and other new environments. By letting the disk take responsibility for low-level placement decisions, range writes enable high performance through device-specific optimizations. Further, range writes naturally support functionality such as wear-leveling, and thus may also help increase device lifetime while reducing internal complexity.

Finding the right interface between two systems is always challenging. Too much change, and there will be no adoption; too little change, and there is no significant benefit. We believe range writes present a happy medium; a small interface change with large performance gains.

Acknowledgments

We thank the members of our research group for their insightful comments. We would also like to thank our shepherd Phil Levis and the anonymous reviewers for their excellent feedback and comments, all of which helped to greatly improve this paper.

This work is supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance and Sun Microsystems.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

- [1] Dave Anderson. OSD Drives. www.snia.org/events/past/developer2005/0507_v1_DBA_SNIA_OSD.pdf, 2005.
- [2] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface: SCSI vs. ATA. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.
- [3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991.
- [4] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.
- [5] John S. Bucy and Gregory R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, January 2003.
- [6] Harry Chambers. *My Way or the Highway: The Micromanagement Survival Guide*. Berrett-Koehler Publishers, 2004.
- [7] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

- [8] Zoran Dimitrijevic, Raju Rangaswami, and Edward Chang. Design and Implementation of Semi-preemptible IO. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 145–158, San Francisco, California, April 2003.
- [9] Robert M. English and Alexander A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 237–252, San Francisco, California, January 1992.
- [10] Bill Gallagher, Dean Jacobs, and Anno Langen. A High-performance, Transactional Filestore for Application Servers. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 868–872, Baltimore, Maryland, June 2005.
- [11] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.
- [12] Garth A. Gibson, David Rochberg, Jim Zelenka, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, and Erik Riedel. File server scaling with network-attached secure disks. In *Proceedings of the 1997 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '97)*, pages 272–284, Seattle, Washington, June 1997.
- [13] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherd. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 283–296, Stevenson, Washington, October 2007.
- [14] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.
- [15] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 263–276, Brighton, United Kingdom, October 2005.
- [16] L. Huang and T. Chiueh. Implementation of a Rotation-Latency-Sensitive Disk Scheduler. Technical Report ECSL-TR81, SUNY, Stony Brook, March 2000.
- [17] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.
- [18] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level Storage System. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962.
- [19] Charles M. Kozierok. Overview and History of the SCSI Interface. <http://www.pcguides.com/ref/hdd/ifs/scsi/over-c.html>, 2001.
- [20] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [21] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 297–310, San Antonio, Texas, June 2003.
- [22] Peter M. Ridge and Gary Field. *The Book of SCSI 2/E*. No Starch, June 2000.
- [23] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [24] Chris Rummmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.
- [25] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.
- [26] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '90)*, pages 313–324, Washington, D.C., January 1990.
- [27] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.
- [28] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.
- [29] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.
- [30] Randy Wang, Thomas E. Anderson, and David A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.
- [31] Ralph O. Weber. SCSI Architecture Model - 3 (SAM-3). <http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf>, September 2004.
- [32] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '94)*, pages 241–251, Nashville, Tennessee, May 1994.
- [33] Bruce L. Worthington, Greg R. Ganger, Yale N. Patt, and John Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 146–156, Ottawa, Canada, May 1995.
- [34] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.