# 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 set of possible address targets, range writes enable the disk to choose the final on-disk location of the request; the disk improves performance by writing to 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 micromanage storage. Consider placement decisions: although modern file systems have little understanding of disk geometry or head position, they decide the exact location of each block. The file system has coarse-grained intentions (e.g., that a data block be placed near its inode [22]) and yet it applies fine-grained control, specifying a single target address for the block.

Micromanagement of block placement arose due to the organic evolution of the disk interface. Early file systems such as FFS [22] understood details of disk geometry, including aspects such as cylinders, tracks, and sectors. With these physical characteristics exposed, 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, 21]. 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 [32, 37]. As a result, the file system does not have accurate knowledge of disk internals. However, the current interface to storage demands such knowledge, particularly when writing a block to disk. For each write, the file system must specify a single target address; the disk must obey this directive, and thus may incur unnecessary seek and rotational overheads. The file system micromanages block placement but without enough information or control to make the decision properly, precisely the case where micromanagement fails [6].

Previous work has tried to remedy this dilemma in numerous ways. For example, some have advocated that disks remap blocks on-the-fly to increase write performance [7, 10, 34]. Others have suggested a wholesale move to a higher-level, object-based interface [1, 13]. The former approach is costly and complex, requiring the disk to track a large amount of persistent metadata; the latter approach requires substantial change to existing file systems and disks, and thus inhibits deployment. An ideal approach would instead require little modification 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 specifying a single exact address per write, the file system presents a set of possible targets (i.e., a range) to the disk. The disk is then free to pick where to place the block among this set based on its internal positioning information, thus minimizing positioning costs. When the request completes, the disk informs the file system which address it chose, thereby allowing for proper bookkeeping. By design, range writes retain the benefits of the existing interface, and necessitate only small changes to both 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 set of possible targets, the file system gives freedom to the disk without relinquishing all control. In turn, the disk (i.e., the worker) 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 in-

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

duces 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, we develop and implement 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 during log writes. Under a transactional workload, we show how range writes can improve journaling performance by nearly a factor of two.

The rest of this paper is organized as follows. In Section 2, we discuss previous efforts and why they are not ideal. We then describe range writes in Section 3 and study disk scheduling in Section 4. In Section 5, we show how to modify file system allocation to take advantage of range writes, and then 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 on modern disks; more details are available elsewhere [2, 26]. We then review existing approaches for minimizing positioning overheads.

## 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 [27].

Most disks today also support *tagged-command queueing* [24], 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 improve performance by using detailed knowledge of positioning and layout.

## 2.2 Reducing Write Costs

In this paper, our focus is on what we refer to as the *positioning-time problem* for writes; how do we reduce or eliminate positioning-time overheads for writes to disk? The idea of minimizing write time is by no means new [20]. However, there is as of yet no consensus on the best approach or the best division of labor between disk and file system for achieving this goal. We briefly describe previous approaches and why they are not ideal.

### 2.2.1 Disk Scheduling

Disk scheduling has long been used to reduce positioning overheads for writes. 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 of the scheduling work has been performed through simulation [18, 28] or has been crafted with extreme care [17, 23, 36, 38]

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 File System Structures

Another approach to solving the positioning-time problem for writes is to develop a file system that transforms the traffic stream to avoid small costly writes by design. A prime example is the Log-structured File System (LFS) [25]; many commercial file systems (e.g., WAFL [14], ZFS [31]) have adopted similar approaches.

LFS buffers all writes (including data and metadata) into *segments*; when a segment is full, LFS writes the segment in its entirety to free space at the end of the log. By writing to disk in large chunks (a few megabytes at a time), LFS amortizes positioning costs.

By design, LFS avoids small writes and thus would seem to solve the positioning-time problem. However, LFS is not an ideal solution for two reasons. First, this approach requires the widespread adoption of a new file system; history has shown such adoption is not straightforward. Second, LFS and similar file systems do not perform well for all workloads; in particular, underneath transactional workloads that frequently force data to disk, LFS performance suffers [29].

#### 2.2.3 Write Indirection

Many researchers have noted that another way to minimize write delay is to appropriately control the placement of blocks on disk. This work, which introduces a layer of indirection between the file system and disk, can be divided into two camps: that which assumes the traditional interface to disk (an array of blocks), and that which proposes a new, higher-level interface (usually based on objects or similar abstractions).

**Traditional Disks:** 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, 10, 34]. 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 [30]. 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.

**New Interfaces:** A related set of efforts allows the disk to control placement but requires a new interface; this idea has appeared in different forms in the literature as Logical Disks [8], Network-Attached Storage Devices [13], and Object-based Storage [1]. With this type of new interface, the disk controls exactly where each object is placed, and thus can make intelligent low-level decisions. However, such an 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 (or similar constructs) implies that the disk must now be concerned with consistent update. Consider object-based storage: 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 in a consistent fashion. Thus, the disk must now also include some kind of logging machinery (perhaps to NVRAM), duplicating effort and increasing the complexity of the drive. Logical disks go a step further, adding a new "atomic recovery unit" interface to allow for arbitrary writes to be grouped and committed together [8]. In either approach, complexity within the disk is increased.

### 2.3 A Cooperative Approach

| Previous Approach | Problems |
|---|---|
| Disk scheduling [17, 18, 23, 28, 36, 38] | Needs many requests, hard to implement in OS |
| Write-anywhere file systems [14, 25, 31] | Gaining acceptance, synchronous workloads |
| Eager writing [10, 34] | Drive complexity, lack of FS information |
| Higher-level interfaces [1, 8, 13] | Drive complexity, gaining acceptance |

In contrast to previous approaches, range writes divide the responsibilities of performing fast writes across both file system and disk; 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. The file system does what it does best: make coarse-grained layout decisions, manage free space, track block ownership, and so forth. The disk does what it does best: take advantage of low-level knowledge (e.g., head position, actual physical layout) to improve performance. The small change required does not greatly complicate either system or significantly change the interface between them, thus increasing the chances of deployment.

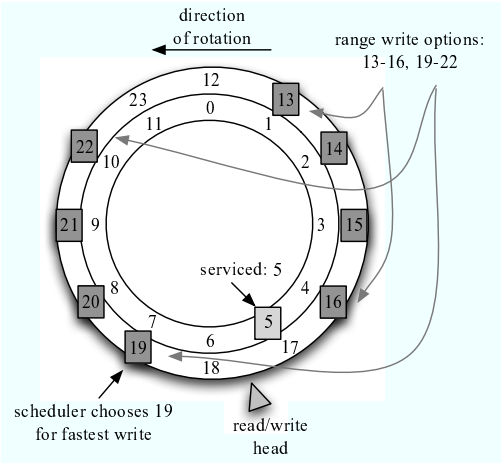


Figure 1: **Range Writes.** *The figure illustrates how to use range writes. The disk has just serviced a write to block 5, and is then given a write with two ranges: 13 through 16, and 19 through 22. The disk, given its current position, realizes that 19 will result in the fastest write, and thus chooses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

# 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 discussion of the 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 to one address out of a set of specified options. The options are specified in a *range descriptor*. The simplest possible range descriptor is comprised of a pair of addresses, $B_{begin}$ and $B_{end}$, designating that data of a given length can be written to any contiguous range of blocks fitting within $B_{begin}$ and $B_{end}$. See Figure 1 for an example.

When the operation completes, the disk returns a *target address*, that is, the starting address of the region 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).

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 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 no reordering of blocks within a single multi-block write. This decision enables the requester to control the ordering of blocks on disk, which may be important for subsequent read performance, and allows a single target address to be returned by the disk. The interface is summarized in Table 1.

### 3.1.1 Range Specification

One challenge is to decide how to specify the set of possible target addresses to the disk. The format of this range description must both be compact as well as flexible, which are often at odds.

We initially considered a single range, 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. In other words, the client may wish to express that a request can be written anywhere within the range $B_{begin}$ to $B_{end}$ *except* for blocks $B_{begin} + 1$ and $B_{begin} + 2$ (because those blocks are already in use). Thus, the interface needs to support such flexibility.

We also considered a list of target addresses. While this approach is quite flexible, we felt it added too much overhead to each write command. A file system might wish to specify a large number of choices; with a range, in the best case, this is just the start and end of a range; in the list approach, it comprises hundreds or even thousands of target addresses.

Due to these reasons, we believe that there are a few sensible possibilities. One is a simple *list of ranges*; the client specifies a list of begin and end addresses, and the disk is free to write the request within any one such range. A similar effect could be achieved with the combination of a single large range and corresponding *bitmap* which indicates which blocks of the range are free. Both the list-of-ranges and bitmap interfaces are equivalent, as they allow full flexibility in compact forms; we leave the exact specification to the future.

### 3.1.2 Overlapping Ranges

Modern disks allow multiple outstanding writes. While improving performance, the presence of multiple outstanding requests 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 free block range in that disk region, $B_1$ 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 [35]). 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

**Classic Write**
*in:* `address, data, length`
*out:* `status`

**Range Write**
*in:* **`range descriptor, alignment,`** `data, length`
*out:* `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. The range descriptor can be specified as a list of free ranges or a (begin, end) pair plus bitmap describing the free blocks within the range.*

$R_2$ 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., $B_1$, $B_3$, ..., $B_{k-1}$ in one range, and $B_2$, $B_4$, ..., $B_k$ in the other), and issue the requests in parallel.

However, 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 the use of range writes, as a client must then make decisions on which portions of the range to give to which requests; this likely decreases the disk's control over low-level placement and thus decreases performance. For 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 indicated that $B_1$ was chosen). However, because the file system may be concurrently issuing another request $R_3$ to the same range, a race condition could ensue 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 uses this feature as follows. First, the file system issues a number of outstanding writes, potentially to overlapping ranges. The disk starts servicing them, and in doing so, tracks which blocks in each range are written. At some point, the file system issues a barrier. The barrier guarantees to the disk that all writes following the barrier 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 [16, 38], or to implement efficient "dummy reads" in semi-preemptible I/O [9].

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 [31]. Given these and other nuances, we leave range reads for future work.

## 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 computationally 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 computational 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 [18, 28].

### 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 are 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 pol-

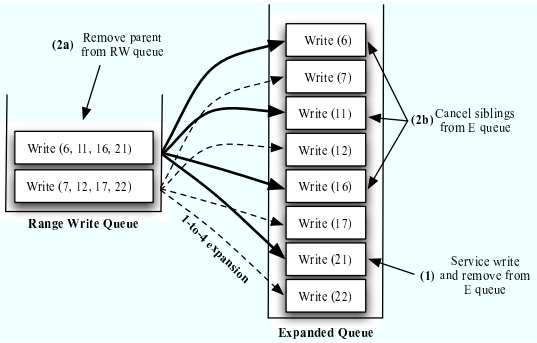


Figure 2: **Expand-and-cancel Scheduling.** *The figure depicts how expand-and-cancel scheduling operates. Range writes are placed into the leftmost queue and then expanded into the full set of writes on the right. In step 0 (not shown), two range writes arrive simultaneously and are placed in the range 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 (step 2a), and the expanded requests to 6, 11, and 16 are canceled (step 2b).*

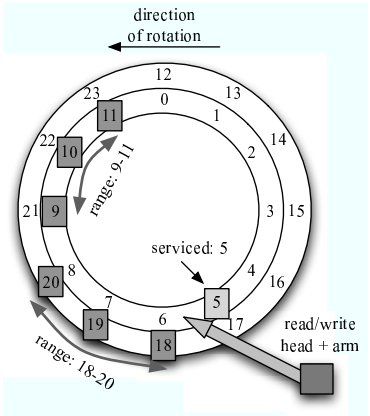


Figure 3: **Hierarchical-range Scheduling.** *The figure depicts how hierarchical-range scheduling works. For the current request, the scheduler must choose which of two possible ranges to write to (18-20 on the adjacent track or 9-11 on the current). The scheduler just serviced a request to 5, and thus must choose whether to stay on the current track and wait for range 9-11 to rotate under the head or switch tracks and write to 18-20. Depending on the relative costs of switching tracks and rotational delay, HRS will decide to which range to write.*

icy is not modified, but simply works with more requests to decide what is the best decision. 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), the disk can keep computational overheads low while still reducing rotational overhead substantially. More expensive drives can include additional processing capabilities to enable more 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 demands a more extensive reworking of the scheduling infrastructure. However, the added complexity comes with a benefit: HRS is much more computationally efficient than ECS, doing 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 takes to seek and settle on the track of each request and the resulting head position. 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 to 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 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

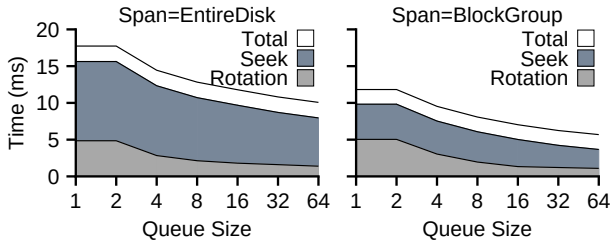


Figure 4: **No Range Writes.** *The figure plots the performance of an in-disk SPTF scheduler on a workload of writes to random locations. The x-axis varies the number of outstanding requests, and the y-axis plots the time per write. The leftmost graph plots performance of writes to random locations over the entire disk; the rightmost graph plots performance of random writes to a 4096-block group.*

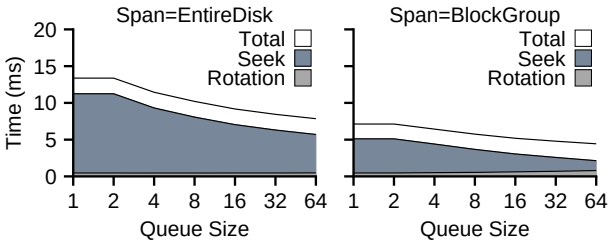


Figure 5: **Track-sized Ranges.** *The graphs plot the performance of range writes under randomized write workloads using the hierarchical range scheduler. The experiments are identical to those described in Figure 4, except that range writes are used instead of traditional writes; the range is set to 100 blocks, just bigger than the track size of the simulated disk (thus eliminating rotation).*

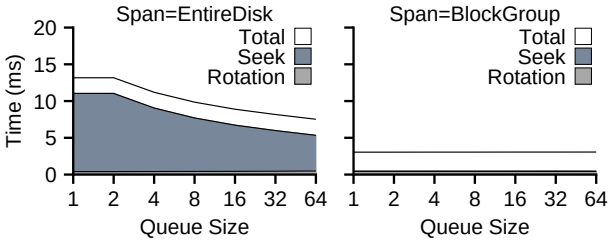


Figure 6: **Group-sized Ranges.** *The graph plots performance of range writes, as described in Figure 5. The small difference: range size is set to 4096 blocks (the size of a block group).*

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 utility of range writes in disk schedulers. To do so, we utilize a detailed simulation environment built within the DiskSim framework [5].

We made numerous small changes throughout DiskSim to provide support for range writes. We implemented a small change to the interface so pass range descriptors to the disk, and more extensive changes to the SPTF scheduler to implement both EC and HR scheduling. Overall, we changed or added roughly one thousand lines of code to the simulator. Unless explicitly investigating EC scheduling, we use the HR scheduler.

For all simulated experiments, we use the HP C2247A disk, which has a rotational speed of 5400 RPM, and a relatively small track size (roughly 60-100 blocks, depending on the zone). It is an older model, and thus, as compared to modern disks, its absolute positioning costs are high whereas track size and data transfer rates are low. However, when writing to a localized portion of disk, the relative balance between seek and rotation is reasonable; thus we believe our results on reductions in positioning time should generalize to modern disks.

## 4.4 Experiments

### 4.4.1 Do multiple outstanding requests solve the positioning-time problem?

Traditional systems attack the positioning-time problem by sending multiple requests to the disk at once; internally, an SPTF scheduler can reorder said requests and reduce positioning costs [28]. Thus, the first question we address is whether the presence of multiple outstanding requests solves the positioning-time problem.

In this set of experiments, we vary the number of outstanding requests to the disk under a randomized write workload and utilize an in-disk SPTF scheduler. Each experiment varies the *span* of the workload; a span of the entire disk implies the target address for the write was chosen at random from the disk; a span of a block group implies the write was chosen from a localized portion of the disk (from 4096 blocks, akin to a FFS cylinder group). Figure 4 presents our results.

From the graphs, we make three observations. First, at the left of each graph (with only 1 or 2 outstanding requests), we see the large amount of time spent in seek and rotation. Range writes will be of particular help here, potentially reducing request times dramatically. Second, from the right side of each graph (with 64 outstanding requests), we observe that positioning times have been substantially reduced, but not removed altogether. Thus, flexibility in exact write location as provided by range writes could help reduce these costs even further. Third, we see that in comparing the graphs, the span greatly impacts seek times; workloads with locality reduce seek costs while still incurring a rotational penalty.

We also note that having a queue depth of two is no better than having a queue depth of one. Two outstanding requests does not improve performance because in the steady state, the scheduler is servicing one request while another is being sent to the disk, thus removing the possibility of choosing the “better” request.

### 4.4.2 What is the benefit of range writes?

We now wish to investigate how range writes can improve performance. Figures 5 and 6 presents the results of a set of experiments that use range writes with a small (track-

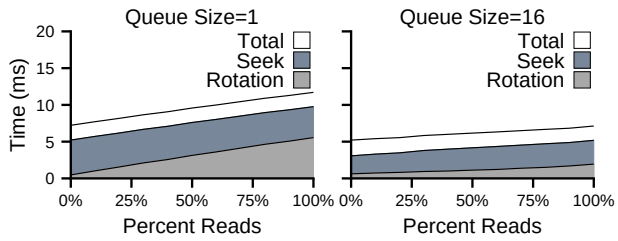


Figure 7: **Mixing in Reads.** *The graphs plot the performance of range writes to random destinations when there is some percentage of reads mixed in. We utilize track-sized ranges and write randomly to locations within a block group. The x-axis varies the percent of reads from 0% to 100%, and the y-axis plots the average time per operation (read or write). Finally, the graph on the left has 1 outstanding request to disk, whereas the graph on the right has 16.*

sized) or large (block-group-sized) amount of flexibility. We again perform random writes to either the entire disk or to a single block group.

Figure 5 shows how a small amount of flexibility can greatly improve performance. By specifying track-sized ranges, all rotational costs are eliminated, leaving only seek overhead and transfer time. We can also see that track-sized range writes are most effective when there are few outstanding requests (the left side of each graph); when the span is set to a block group, for example, positioning costs are halved. Finally, we can also see from this graph that range writes are still of benefit with medium-to-large disk queues, but the benefit is indeed smaller.

Figure 6 plots the results of the same experiment, however with more flexibility: range size is now set to the entire block group. When the span of the experiment is the entire disk (left graph), this makes little difference; rotational delay is eliminated. However, the right graph with a span of a block group shows how range writes can also reduce seek costs. Each write in this experiment can be directed to any free spot in the block group; the result is that there is essentially no positioning overhead, and almost all time spent in transfer.

#### 4.4.3 What if there are reads in the workload?

We have now seen the benefits of range writes in synthetic workloads consisting purely of writes. We now include reads in the workload, and show the diminishing benefit of range writes in read-dominated workloads. Figure 7 plots the results of our experiments.

From the figures, we observe the expected result that with an increasing percentage of reads, the benefit of range writes diminishes. However, we can also see that for many interesting points in the read/write mix, range writes could be useful. With a small number of outstanding requests to the disk and a reasonable percentage of writes, range writes decrease positioning time noticeably.

#### 4.4.4 What is the difference between ECS and HRS?

We next analyze 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 $S$, 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 $S$ requests in the expanded queue. Thus, the amount of work, $W$, performed by ECS is:

$$W_{EC} = S \cdot Q \tag{1}$$

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:

$$W_{HR} = \lceil \frac{S}{T} \rceil \cdot Q \tag{2}$$

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 ($S = T$), the amount of work performed by HRS is roughly $Q$. In contrast, ECS still performs $S \cdot Q$ work; as track sizes can be in the thousands, ECS will have to execute a thousand times more work to achieve equivalent performance.

#### 4.4.5 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 requires 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 8 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

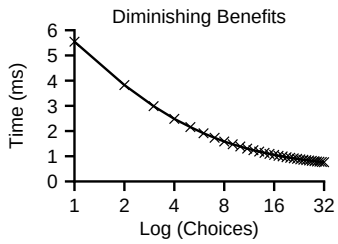


Figure 8: **The Diminishing Benefits of More Choice.** *The figure plots the performance of successive write requests to the same track. Along the x-axis, we increase the number of choices available for write targets, and the y-axis plots average write time.*

delay. As further options are made available to the scheduler, 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 entire-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

Our study of disk scheduling has revealed a number of interesting results. First, the overhead of positioning time cannot be avoided with traditional SPTF scheduling alone; even with multiple outstanding requests to the disk, seek and rotational overheads still exist.

Second, range writes can dramatically improve performance relative to SPTF scheduling, reducing both rotational and seek costs. To achieve the best performance, a file system (or other client) should give reasonably large ranges to the disk: track-sized ranges remove rotational costs, while larger ranges help to noticeably reduce seek time. Although range writes are of greatest utility when there are only a few outstanding writes to the disk, range writes are still useful when there are many.

Third, the presence of reads in a workload obviously reduces the overall effect of range writes. However, range writes can have a noticeable impact even in relatively balanced settings.

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 exacts a high 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 problem, 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?

In general, the larger the range given to the disk, the more positioning time is reduced. The simulation results presented in Section 4 indicate that track-sized ranges effectively remove rotational costs while larger sized ranges (e.g., several thousand blocks) help with seek costs. In the ideal case, positioning time can be almost entirely removed if the size of the target range matches the span of the current workload.

Thus, the file system should specify the largest range that best matches its allocation and layout policy. For example, FFS could specify that a write be performed 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

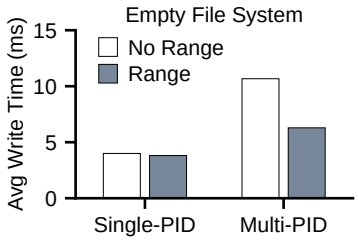


Figure 9: **File Create Time (Empty File System).** *The figure plots the average write time during a file create benchmark. The benchmark creates 1000 4-KB files in the same directory. Range writes are either used or not, and the files are either created by a single process or multiple processes. The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

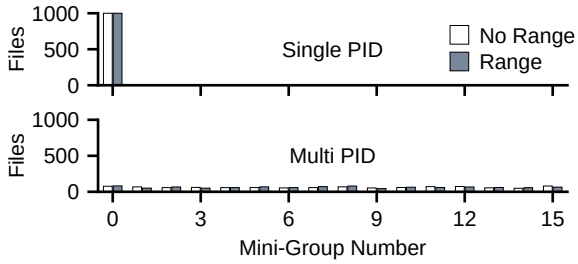


Figure 10: **File Placement.** *The figure shows how files were placed per mini-group across two different experiments. In the first, a single process (PID) created 1000 files; in the second, each file was created by a different PID. The x-axis plots the mini-group number, and the y-axis shows the number of files that were placed in the mini-group, for both range writes and traditional writes.*

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 carefully track outstanding requests to a given region of disk, in order to avoid sending writes to a full region. However, this modification should not be substantial.

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 [12], 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. Fortunately, modern file systems have more flexible structures; for example, Sun's ZFS [31], NetApp's WAFL [14], and LFS [25] all take a "write-anywhere" approach for most on-disk structures.

### 5.2 Incorporating Range Writes into ext2

We now present our experience of incorporating range writes into a simulation we built of Linux ext2. Allocation in ext2 (and ext3) derives from classic FFS allocation [22] 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 the calling process to select one of 16 start locations within the block group; we call each of these 16 locations a *mini-group* 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

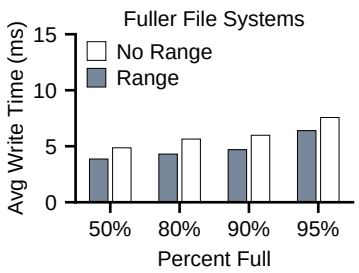


Figure 11: **File Create Time (Fuller File System).** *The figure plots the average write time during a file create benchmark. The benchmark creates 1000 4-KB files in the same directory. Range writes are either used or not, and the files are either created by a single process. The x-axis varies the fullness of the block group.*

| | Traditional ext2 | with Range Writes |
|---|---|---|
| **Untar** | 143.0 | 123.1 |
| **PostMark** | 29.9 | 22.2 |
| **Andrew** | 23.2 | 23.4 |

Table 2: **File System Workloads.** *Each row plots the performance (in seconds) of a simulated workload. In the left column, results represent the time taken to run the workload on our simulated standard ext2, whereas on the right, the time to run the workload on ext2 with range writes is presented. Three workloads are employed: untar, which unpacks the Linux source tree; PostMark, which emulates the workload of an email server (by creating, accessing, and deleting files), using its default settings; and the modified Andrew benchmark, which emulates typical user behavior. The simulations were driven by file-system-level traces of the given workloads which were then played back against our simulated file system.*

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 C2247A), 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 9 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 10 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 follow 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 11 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 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 Real Workloads

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 three workloads: untar, which unpacks the Linux source tree, PostMark [19], which simulates the workload of an email server, and the modified Andrew Benchmark [15], which emulates typical user behavior. Table 2 presents the results.

We make the following two observations. First, for workloads that have significant write components (untar, PostMark), range writes boost performance (a 16% speedup for untar and roughly 35% for PostMark). Second, for workloads that are less I/O intensive (Andrew), range writes do not make much difference.

## 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 of the higher-level file system policies. As much of write activity is to newly created files [4, 33], we believe our range-write variant of ext2 will 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, partial-file overwrites present an interesting scenario. For best performance, one should issue a range write even for previously allocated data; thus, overwritten data may be allocated to a new location on the disk. Unfortunately, this strategy can potentially destroy the sequentiality of later reads and should be performed with care. We leave this and many other workload scenarios to 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. By doing so, a journaling file system can avoid the rotations that occur when performing many synchronous writes to the journal and thus greatly improve performance.

Whereas the previous section employed simulation to study the benefits of range writes, we now 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. Our approach suggests a method to build acceptance of new technology: first via software prototype (to demonstrate potential) and later via actual hardware (to realize the full benefits).

## 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 thus a rotation is incurred to write the commit block.

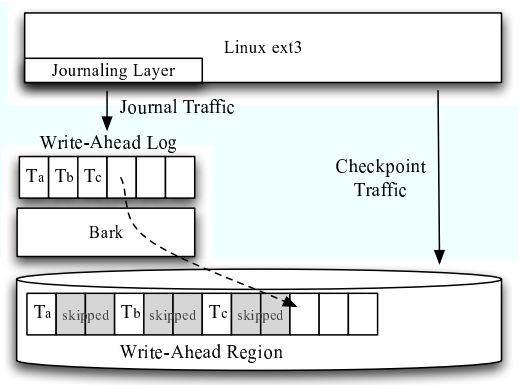


Figure 12: **Bark-itecture.** *The figure illustrates how a file system can be mounted upon Bark to improve journal write performance. All journal traffic is directed through Bark, which picks a skip distance based on think time and the position of the last write to disk. Bark performs this optimization 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: Ta, Tb, and Tc. Bark, using its performance model, has spread the transactions across the physical disk, leaving empty spaces (denoted as "skipped") in the write-ahead region.*

Our approach 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.

Our approach derives from previous work in database management systems by Gallagher et al. [11]. Therein, the authors describe a simple dynamic approach that continually adjusts the distance to skip in a log write to reduce rotation. Perhaps due to the brief description of their algorithm, we found it challenging to successfully reproduce their results. Instead, we decided on a different approach, first building a detailed performance model of the log region of the disk and then using that to decide how to best place writes to reduce rotational costs. The details of our approach, described below, are based on our previous work in building the disk mimic [23].

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. 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.

| | w/o Bark | w/ Bark | Null |
|---|---|---|---|
| **Uncached** | 50.7 | 42.1 | 38.8 |
| **Cached** | 44.2 | 27.3 | 25.4 |

Table 3: **Bark Performance.** *Each row of the table plots the overall performance (in seconds) of TPC-B in three different settings: without Bark, with Bark, and on a "null" journal that reports success for writes without performing disk I/O (the null journal represents the best possible improvement possible by using Bark). The first row reports performance of a cold run, where table reads go to disk. The second row reports performance when the table is in cache (i.e., only writes go to disk). Experiments were run upon a Sun Ultra20 with 1 GB of memory and two Hitachi Deskstar 7K80 drives. The average of three runs is reported; there was little deviation in the results.*

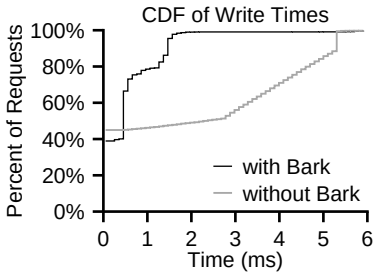


Figure 13: **Write Costs.** *The figure plots the cumulative distribution of write request times during TPC-B. Two lines are plotted: the first shows the cost of writes through Bark, whereas the second shows costs without. The data is taken from a "cached" run as described above.*

## 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 our previous work that measures the request time between all possible pairs of disk addresses in order to perform disk scheduling [23]. Our problem here is simpler: Bark must simply predict where to place the next write in order to reduce rotation.

To make this prediction, Bark performs measurements of the cost of writes to the portion of the disk of interest, varying both the distance between writes (the "skip" size) and think time between requests. The data is stored in a table and made available to Bark at runtime.

For the results reported in this paper, we created a disk profile by keeping a fixed write size of 4 KB (the 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 12 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 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. Because TPC-B forces data to disk frequently, it induces a great deal of synchronous I/O traffic to the ext3 journal.

Table 3 plots the performance of TPC-B on Linux ext3 in three separate scenarios. In the first, the unmodified traditional journaling approach is used; in the second, Bark is used underneath the journal; in the third, we implement

a fast "null" journal which simply returns success when given a write without doing any work. This last option serves as an upper-bound on performance improvements realized through more efficient journaling.

Note also that each row varies whether table reads go to disk (uncached) or are found in memory (cached). In the cached runs, most table reads hit in memory (and thus disk traffic is dominated by writes). By measuring performance in the uncached scenario, we can determine the utility of our approach in scenarios where there are reads present in the workload; the cached workload stresses write performance and thus presents a best-case for Bark under TPC-B.

From the graph, we can see that Bark greatly improves the overall runtime of TPC-B; Bark achieves a 20% speedup in the uncached case and over 61% in the cached run. Both of these approach the optimal time as measured by the "null" case. Thus, beyond the simulation results presented in previous sections, Bark shows that range writes can work well in the real world as well.

Figure 13 sheds some light on this improvement in performance. Therein we plot the cumulative distribution of journal-write times across all requests during the cached run. When using Bark, most journal writes complete quickly, as they have been rotationally well placed through our simple skipping mechanism. 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

We learned a number of important lessons from our implementation of log skipping using range writes. First, we see that range writes are also 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, with read traffic present, the improvement seen by Bark is lessened but still quite noticeable. Thus, even with reads (in the uncached case, they comprise roughly one-third of the traffic to the main file system), flexible writes to the journal improve performance.

Finally, we should note that we chose to incorporate flexible writes underneath the file system in the simplest possible way, without changing the file system implementation at all. If range writes actually existed within the disk, the Bark layer would be much simpler: it would issue the range writes to disk instead of using a model 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 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 the 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 the disk of today without range writes to the disk of tomorrow 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 [13].

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 storage system 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.

We believe there are numerous interesting future paths for range writes, as we have alluded to throughout the paper. The corollary operation, range reads, presents new challenges but may realize new benefits. Integration into RAID systems introduces intriguing problems as well; for example, parity-based schemes often assume a fixed offset placement of blocks within a stripe across drives. An elegant approach to adding range writes into RAIDs may well pave the way for acceptance of this technology into the higher end of the storage system arena.

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] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 15–28, Asheville, North Carolina, December 1993.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[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] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[16] 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.

[17] L. Huang and T. Chiueh. Implementation of a Rotation-Latency-Sensitive Disk Scheduler. Technical Report ECSL-TR81, SUNY, Stony Brook, March 2000.

[18] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[19] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[20] 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.

[21] Charles M. Kozierok. Overview and History of the SCSI Interface. http://www.pcguide.com/ref/hdd/if/scsi/overc.html, 2001.

[22] 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.

[23] 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.

[24] Peter M. Ridge and Gary Field. *The Book of SCSI 2/E*. No Starch, June 2000.

[25] 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.

[26] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[27] 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.

[28] 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.

[29] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In *Proceedings of the USENIX Annual Technical Conference (USENIX '95)*, pages 249–264, New Orleans, Louisiana, January 1995.

[30] 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.

[31] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[32] Nisha Talagala, Remzi H. Arpaci-Dusseau, and Dave Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[33] 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.

[34] 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.

[35] Ralph O. Weber. SCSI Architecture Model - 3 (SAM-3). http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf, September 2004.

[36] 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.

[37] 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.

[38] 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.