Scan-Based Scheduling and Layout in a Reliable Write-Optimized File System

Minwen Ji and Edward W. Felten

Abstract

Disk I/O limits the performance of file systems on modern computers. While the cost of disk reads can be lowered by caching and prefetching, disk writes are harder to optimize. Writes can be delayed and reordered up to a point, but this is limited by the need to maintain file system integrity. The result is frequent use of small, synchronous writes.

This paper presents a scheduling and layout algorithm to maximize disk write bandwidth and improve reliability in a file system with commodity disks. Our algorithm delays and schedules all writes while maintaining system integrity. We do this with a novel combination of segment layout, lazy allocation, request sorting/coalescing, phase-specific recovery.

We have implemented a prototype of our file system on Pentium PCs running Windows NT 4.0 and have compared its performance with NTFS running on the same hardware. Our system outperforms NTFS by a factor of 2 to 10 for writing in I/O intensive benchmarks, while offering comparable read bandwidth.

1 Introduction

Improving the utilization of raw disk bandwidth is an effective way to improve file system performance. In modern computers, disk traffic is dominated by writes because most reads can be satisfied by large in-memory file caches [16]. Although large caches improve read performance, they do not help as much for writes, because data buffered in memory needs to be written to disk eventually in order to survive system crashes.

Previous work shows that small random synchronous writes ("syncs") are the major cause of low utilization of raw disk bandwidth because they introduce a large number of arm seeks, disk rotates and interrupts to host OS per transferred byte [15] [17] [13] [4] [11]. Many scheduling and allocation algorithms, such as SCAN [19], clustering [15] and log-structuring [17], could not achieve their full benefit if the applications generate synchronous requests. Therefore, fully-asynchronous disk write is the key to maximum disk bandwidth.

However, synchronous disk writes exist for a reason: to maintain file system integrity in the face of machine crashes. File system *integrity* refers to the following property of the metadata in the file system:

1. Each disk block in the system is marked either free or in use, but not both. (Counter example: a block is both part of a file and in the free list.)

2. If it is marked in use, a disk block is used for exactly one purpose. (Counter example: a block is allocated to two distinct files.)

3. All blocks in use have been properly initialized. (Counter example: a file contains an uninitialized block.)

There are several existing ways to ensure file system integrity or the recoverability of such a property:

- Synchronous metadata writes: The traditional UNIX file system (UFS) synchronously writes a "pointee" (a block whose address is found in another block) before writing its "pointer" (a block that contains the address of another block). This results in low utilization of disk write bandwidth.

- Recovery scanning: A variation of UFS delays metadata writes, but scans the entire metadata structure (using the tool "fsck") to detect and resolve conflicts [20] after a crash. This process is quite slow for large disks. Besides, it does not prevent metadata from pointing to uninitialized or trashed disk blocks.

- Non-volatile RAM (NVR): NVR can hold "stable" data and survive system crashes or power failures [3]. However, its size is usually limited due to its high cost.

- Journaling metadata writes: Journaling file systems like NTFS guarantee integrity by writing changes to metadata, i.e. pointers, to a journal file on disk before making the actual changes [7]. Writes to the journal and the metadata itself are redundant and have to be synchronized with each other.

- Log structuring: Log-structured file systems always put "pointee" blocks before their "pointers" in the in-memory log, and hence they write these blocks to disk in order. By ordering writes in this way, inconsistencies can be prevented. Log-structured file systems suffer from the cost of maintaining and coalescing free space on the disk.

- Scheduler-enforced ordering: The file system uses asynchronous writes and passes ordering requirements to the disk scheduler [8]. This scheme can reduce the number of disk writes in case of multiple updates, but does not attempt to turn small random writes to large sequential ones. It may require a large amount memory to record fine-grained dependency information and may cause some blocks to be written to disk multiple times in order to solve dependency cycles and aging problem.

In this paper, we present a disk scheduling and layout algorithm for file systems that fully exploits disk write bandwidth. The key to maximizing disk bandwidth is to convert small random synchronous writes into large sequential asynchronous writes, without sacrificing the integrity guarantee that synchronous writes normally provide. Our algorithm allocates disk space in such a way that non-pointers always have lower addresses than pointers. Then it batches, sorts and coalesces disk write requests and issues them in ascending address order. Non-pointers are written before pointers, giving us most of the benefit of write ordering. Pointers that point to other pointers are not guaranteed to be written in order, but can be handled by a combination of logging and recovery scanning at a much lower frequency than that in existing systems.

Our file system scheduling algorithm is similar to the cyclical scan (C-SCAN) algorithm in disk controllers [19], in which the disk arm moves across the entire disk in ascending address order, servicing all write requests in its path. We believe that such an algorithm can be more effective if it is implemented at the file system level, because the file system code can take advantage of the large amount of memory in the host for buffering, sorting and coalescing a large number of requests. Of course, the C-SCAN algorithm faces a new challenge at file system level, i.e. the maintenance of system integrity. We addresses this with a data layout scheme specifically designed for the C-SCAN scheduling.

The rest of the paper is organized as follows: Section 2 describes our scan-based scheduling and layout algorithm in detail, Section 3 briefly discusses the data stability issue, Section 4 compares the performance of the file system built on our disk architecture with that of NTFS on the same hardware, Section 5 references related work, and Section 6 concludes the paper.

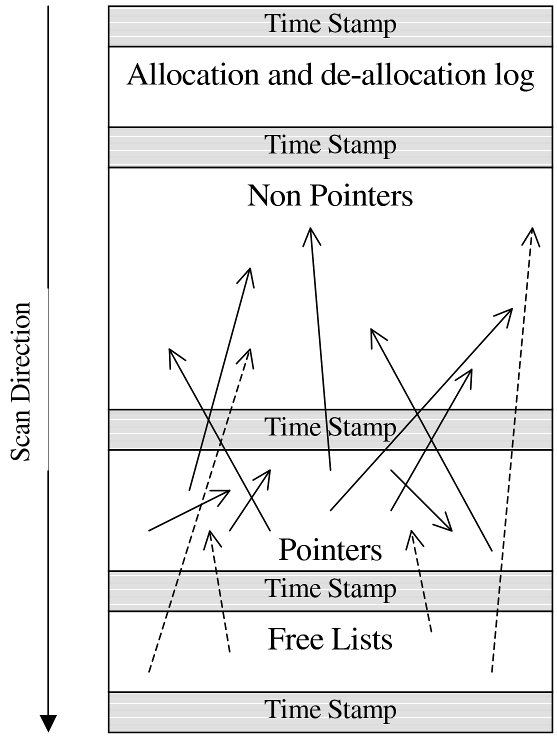

Figure 1: Scan-based segment layout

2 Scan-Based Scheduling and Layout

2.1 Segment layout

We divide the disk into four fixed segments in ascending address order: the allocation and de-allocation log segment, the non-pointer segment, the pointer segment and the free list segment. The allocation and de-allocation log segment is used to temporarily hold a list of changes to the free list. It consists of four sub-logs: a non-pointer allocation log, a pointer allocation log, a non-pointer de-allocation log and a pointer de-allocation log. The file system uses the pointer segment for their metadata that contains addresses of other disk blocks, such as inodes and directories, and the non-pointer segment for other data, such as the contents of files. The free lists keep track of which disk blocks, both non-pointers and pointers, are not currently in use. There is a time stamp following each segment as well as one at the beginning of the disk. After a crash, the time stamps allow us to determine whether a write scan was in progress when the system crashed, and if so, which segment the scan was in when the system crashed. Figure 1 shows the disk layout.

2.2 Lazy allocation and deferred re-allocation

All write requests are batched and handled by a single thread, the disk daemon. The daemon periodically sorts and coalesces batched requests and issues them in a scan across the disk. The daemon writes the five time stamps as well as the four segments on disk in every scan. The daemon is activated whenever the disk is idle.

A "new" data block is a block of data in memory that has not been allocated space on disk. Disk space will not be allocated to new data until the disk daemon needs to write it to disk, a method we call "lazy allocation". After lazy allocation, the daemon sorts all requests (including both new and existing data) in ascending order of addresses and coalesces requests with contiguous addresses into a single request.

Lazy allocation has three performance advantages: spatial locality, coalescing opportunity, and temporal locality. The disk daemon batches the requests for the same file and allocates contiguous disk blocks to them so that future requests for this file can be serviced with few large sequential requests. The daemon also attempts to minimize the number of non-contiguous requests in the scan by placing new data adjacent to existing data as much as possible. If the applications exhibit temporal locality, i.e. data created at the same time will likely be accessed at the same time in the future, then accesses to such data can be done with large sequential requests [17].

De-allocated disk blocks will not be re-allocated to new data until their unused status is permanently stored in the on-disk free list. We call this scheme "deferred re-allocation". This is important for integrity because it helps us guarantee that no block with a pointer can be trashed or overwritten during a crash and no block can belong to more than one file.

A list of the blocks being allocated and de-allocated in the current scan are written in the log segment at the beginning of the scan. The log can be used to undo or redo some operations after a crash at certain phases of the scan.

2.3 Phase-specific crash recovery

We divide a scan into four phases according to the four segments on disk. The disk daemon writes a time stamp consistent within a scan in the beginning and end of each phase. At the end of a successful scan, all five time stamps on disk are identical. Otherwise the first time stamp that differs from the previous one marks the end of the phase in which the system crashed. When the system is rebooted, the disk daemon compares the five time stamps, identifies the phase in which the system crashed (if any) and invokes recovery procedures accordingly. The integrity status after a crash in each phase and the corresponding recovery strategy is the following.

Phase 1 Crashed while writing the log segment. All changes made to the file system after the last scan are lost, but there is no inconsistency in the metadata. No recovery is needed. Except for the log, which will never be used in this case, the system is in the same state as at the end of the last successful scan.

Phase 2 Crashed while writing the non-pointer segment. All changes to pointers are lost, some changes to non-pointers are lost, no free list changes have occurred, but there is no inconsistency in the metadata. No recovery is needed. The pointer segment and the free list are both in the same state as the end of the last successful scan. Changes in the non-pointer segment do not invalidate any integrity guarantees as listed in Section 1.

Phase 3 Crashed while writing the pointer segment. Since some changes to pointers are lost, some blocks might be in a bad state: they might be marked both free and in use, they might be marked neither free nor in use, or some pointers might point to uninitialized pointer blocks. However, a block cannot be in a bad state unless that block appears in the log segment. Recovery in this case takes the following steps: 1) Delete all the pointers to the blocks in the pointer allocation sub log and the pointer and non-pointer de-allocation sub logs. 2) Rebuild the free lists with all the blocks that are not in use. The conflicts between the pointer segment and the free lists are resolved by rebuilding the free lists. Pointers to uninitialized blocks no longer exist because all non-pointer blocks have been successfully written and all pointers to newly allocated pointer blocks have been deleted.

Phase 4 Crashed while writing the free lists segment. The free lists may be inconsistent with the metadata, since some updates to the free lists may have been lost. Recovery in this case takes the following steps: 1) Remove all blocks in the allocation log from the free lists. 2) Add all blocks in the de-allocation log to the free lists, if they are not already in the free lists. The non-pointer and pointer segments are in the same state as if the entire scan was successful. The recovery procedure puts the free lists into the state they would have been in had the scan succeeded.

Our phase-specific recovery strategy provides a stronger integrity guarantee than fsck, since we eliminate pointers to uninitialized or trashed blocks. Fsck resolves the conflicts in block and i-node reference counts, but does not address the problem of uninitialized or trashed blocks.

2.4 Recovery cost

As we can see in Section 2.3, crashes in cases 1 and 2 do not need any recovery procedures and crashes in case 4 need only a trivial recovery procedure, updating the free lists with the allocation and de-allocation log. Crashes in case 3, while writing the pointer segment, need a scan of the file system metadata structure similar to the fsck utility in UFS. The pointer segment, i.e. the area for file system metadata, is typically 20% of the total disk space. Since we need to scan only if the system crashed while writing this small part of the disk, scans will be relatively rare.

Recovery has very little cost other than that of scanning the metadata structure. The repair process only involves deleting pointers and the number of such operations is bounded by the number of blocks in the allocation and de-allocation log, which is a very small portion of the total number of blocks on disk. Recovery is made easier and faster because the underlying scheduling and layout algorithm restricts the range for inconsistency checking to a much smaller one than the entire disk and prevents problems like conflicting pointers in the first place.

2.5 Read requests

Our write algorithm contributes to potentially better read performance in three ways:

- Since writing the same amount of data takes less time, more disk time can be used to service read requests.
- Since dirty blocks stay in memory for a shorter time, more cache space is available to reduce the number of disk reads.
- The lazy allocation scheme takes advantage of both spatial and temporal locality so that data blocks in the same file or created at the same time can be read in large sequential requests if they are accessed at the same time in the future.

There are several ways to schedule read requests. If the disk is busy when they arrive, the read requests can be batched, sorted and coalesced in the same way as write requests and issued in a read scan once the disk becomes idle. Alternatively, the read requests can be sorted into the same list as the write requests and issued in a single scan.

In order to reduce the variance in read latency, the write requests in a scan can be split into chunks so that read requests can be interleaved between the chunks; write performance will still be good as long as the chunks are large enough. The best chunk size is one that is large enough to give good write performance, but not so large as to introduce unacceptable variance in read latency. Our experimental results, presented below, show that there is a "sweet spot" chunk size that works well. Read-ahead or prefetching can also be used to further optimize read performance.

2.6 Implementation

We have implemented our scheduling and layout algorithm in a user-level file system called Scan-FS for proof of concept and for performance measurement. While a real implementation should be in the form of a kernel driver, the user-level implementation significantly eases our debugging efforts. Scan-FS exports the same interface as NTFS. Therefore, applications that were written to run on NTFS can be easily modified to run on Scan-FS.

Scan-FS communicates with the physical disk driver through a handle to the device and bypasses all intermediate buffering and caching in NT. It uses logical block numbers (LBNs) for sorting and coalescing requests. Most drives use a linear mapping of logical blocks to physical sectors, where adjacent logical blocks come from adjacent physical sectors [18]. Previous work shows that issuing requests in order of LBNs, as we do, tends to work well [21] [4].

Coalesced requests are issued to the disk driver in a scatter/gather fashion, i.e. with a list of discrete buffers for a contiguous region of disk space. Most DMA engines support a "chained buffer" bit in their request address register, and NT provides higher level scatter/gather interfaces in its Platform Software Development Kit (SDK) [1]. For such requests, each buffer should be the size of a system memory page (4 KB on our machine) and should be aligned on a page size boundary. Scan-FS employs a standard First In First Out (FIFO) cache replacement policy.

3 Discussion

Besides system integrity, data stability is another reason for synchronous writes. Applications may explicitly request that specific writes be made stable. For example, transactions in database applications rely on the assumption that committed data is stable. Many systems have used approaches other than synchronous disk writes, such as non-volatile memory [3], warm-rebootable memory [5] and data replication [14], to provide equivalent or relaxed guarantees. Our scheduling and layout algorithm does not prevent the file system from working with any of those approaches.

4 Performance

We run a set of benchmarks on both Scan-FS and NTFS on the same hardware to compare their performance. All the benchmarks in our experiments are measured on a Pentium 166 MHz PC running Windows NT Workstation 4.0 with 64 MB physical memory, 4 KB page size and an empty 4.2 GB Quantum ATLAS II SCSI disk [10].

File system performance can be affected by a lot of factors, e.g. cache management, metadata structure and disk performance. Our goal in the performance measurement of Scan-FS and NTFS is to compare our scheduling and layout algorithm to the logging mechanism in NTFS, both of which are designed to maintain system integrity. We use Andrew benchmark, which is known to stress metadata operations and in-memory operations on modern computers, to show that Scan-FS and NTFS have reasonably comparable performance when disk bandwidth is not a bottleneck. Our main goal is to compare the performance of both file systems in disk I/O intensive benchmarks.

4.1 Andrew benchmark

The Andrew benchmark is a traditional benchmark for file systems on UNIX and was written as a UNIX shell script or make file. It has five phases: directory creation, file copying, directory statistics, file scanning and compilation.

We were not aware of any version of the Andrew benchmark that can run on Windows NT, so we rewrote the benchmark in C++ using NTFS's interface. However, we skipped the compilation phase because we do not have access to the source code of a compiler on NT and cannot modify it so that it can run on Scan-FS. The rewritten benchmark makes calls to both file systems through Win32 RPC.

The total amount of the data involved in the benchmark is about 800 KB, so it easily fits in the physical memory of any commodity computer. The operations mostly involve intensive writes and reads of in-memory metadata. The results show that Scan-FS and NTFS have comparable performance in such operations. See Table 1.

Benchmark Phases	NTFS	Scan-FS
Directory Creation	0.05	0.03
File Copying	0.72	0.65
Directory Statistics	0.30	0.20
File Scanning	1.21	0.96

Table 1: Elapsed time (in seconds) of each phase of the Andrew benchmark on NTFS and Scan-FS. Smaller numbers are better.

4.2 I/O intensive benchmarks

We have designed a set of I/O intensive benchmarks to exercise various aspects of Scan-FS and NTFS. In each benchmark, a client first creates a file and then repeatedly writes data from a buffer to the file. The buffer size, varying from 4 KB to 1 MB, determines the size of each write request while the file size, varying from 16 MB to 128 MB, determines the overall memory footprint of the benchmark. When the memory footprint exceeds the physical memory size, the client bandwidth will be bounded by disk write bandwidth. The client can write to the file either sequentially or randomly, and can fork another thread that repeatedly reads from another file.

We compute the client bandwidth in the benchmarks as the number of bytes delivered divided by the elapsed time. We also analyze the output of the NT performance monitor and used it to explain the difference between Scan-FS and NTFS performance. The NT performance monitor is a Windows NT 4.0 administrative tool that uses a set of counters to track data such as the physical disk bandwidth and the number of bytes per disk request [2].

4.2.1 Writes with small memory footprint

In our first benchmark, a client repeatedly writes data from a 4 KB or 1 MB buffer to a 16 MB file, in sequential or random order. Based on their buffer size and write order, there are four different benchmarks: large sequential, large random, small sequential and small random. All write requests hit in the cache. In all tests, there is about 20 MB of memory available for caching and no noticeable page faults. Table 2 shows the client bandwidth and physical disk bandwidth on NTFS and Scan-FS.

	Client BW	Client BW	Disk BW	Disk BW
	NTFS	Scan-FS	NTFS	Scan-FS
Large Seq.	4.6	31.1	4.3	14.3
Large Ran.	4.7	30.0	4.1	13.6
Small Seq.	9.0	29.6	8.3	12.7
Small Ran.	4.5	25.9	2.5	9.6

Table 2: Small memory footprint benchmark results: client write bandwidth (MB/s) and physical disk write bandwidth (MB/s). Larger numbers are better.

From the physical disk write bandwidth shown in the NT performance monitor, we learn that NTFS's client bandwidth is bounded by the disk write bandwidth even with sufficient cache space. The monitor shows that the client's write operations are serialized with certain NT kernel operations. During the waiting time of the client, the active part of the kernel is the lazy writer, a set of threads that periodically flush the dirty data in cache to disk [6]. NTFS's client bandwidth drops when the size of the buffer increases and the monitor shows that with a larger buffer, the lazy writer has fewer chances to flush dirty data, which can be explained by the fact that it takes a longer time for the writing client to complete a request.

Because of the scan-based scheduling and layout algorithm, Scan-FS does not need this strict synchronization between memory writes and disk writes. Synchronization is used only in rare cases to avoid race conditions.

4.2.2 Writes with large memory footprint

In our next benchmark, a client repeatedly writes data from a 4 KB or 10 MB buffer to a 128 MB file, in sequential or random order. There are also four different benchmarks of this type: large sequential, large random, small sequential and small random. The physical memory size of our machine is 64 MB. Scan-FS uses up to 32 MB for caching while NT expands or shrinks its cache size according to the availability of memory. The results show that in all tests, the client bandwidth is bounded by disk write bandwidth.

Table 3 shows the client write bandwidth, physical disk write bandwidth and the number of bytes per disk write shown in the NT performance monitor.

As we expected, the disk write bandwidth increases significantly with the number of bytes per write. NTFS has a fixed limit of the number of bytes per write, 64 KB. Scan-FS always tries to coalesce as many requests as possible. There is little opportunity to coalesce when the requests come in a random and "sparse" pattern; therefore, the number of bytes per write in the small random write tests on Scan-FS is significantly smaller than that in the other tests, and so is the write bandwidth. Fortunately, workloads like this are rare in real-world applications [21].

The workload in an office/engineering environment is similar to the small random write with small memory footprint or the large random write with large memory footprint. Multiple clients access files concurrently and their files are randomly located on disk. However, the requests tend to be sequential within each file or client, so some subsets of the requests can be coalesced. Scan-FS performs well in both benchmarks.

4.2.3 Reads with burst writes

Our next benchmark uses two threads. One thread repeatedly writes a 1 MB buffer to a 128 MB file. The other thread repeatedly reads 4 KB or 1 MB of data from a file. The writer's bandwidth is bounded by disk write bandwidth due to limited cache space. The reader is programmed to access the file without any intermediate buffers; therefore, it always reads directly from disk. We measure the read latency as the elapsed time from when a request is issued by a client until the request is completed.

Because a write scan writes at most one cache full of data to disk, the worst-case read latency is roughly bounded by the time required to write a full cache to disk. With a 32 MB cache and effective write bandwidth of 9 MB/s in our tests, the worst-case read latency is about 3.6 seconds.

Our current policy for scheduling read requests was a simple First Come First Served (FCFS) algorithm. Without any changes to the write scans, both the average and variance of read latency in Scan-FS are much larger than those in NTFS. This result suggests that the write requests in a scan might be split into chunks so that read requests can be interleaved between the chunks, given that each chunk is large enough to utilize a high percentage of raw disk bandwidth. We chose the chunk size to be 140 KB because it was the smallest number of bytes per write in our tests that achieved above 10 MB/s disk write bandwidth. Results show that read latency is significantly improved with chunked writes and there is no noticeable drop in write bandwidth.

Table 4 shows the maximum, average and minimum latency in reads and the bandwidth in reads and writes in the read-write benchmark on three systems, NTFS, Scan-FS and Scan-FS with chunked writes.

5 Related Work

A lot of work has been done to improve disk I/O performance, using methods such as extent-like UFS [15], log-structured file systems [17], scheduler-enforced ordering [8], disk-directed I/O for MIMD multiprocessors [13], journaling file systems [12] [9] [7], application-controlled file caching, prefetching and disk scheduling [4], and disk caching disk [11]. All of this work, like ours, attempts to exploit maximum disk bandwidth in the same way: by converting small random disk accesses into large sequential disk accesses. However, each method has its own way of approaching this goal.

Extent-like UFS [15] combines blocks adjacent to the requested blocks into a larger I/O request. Our lazy allocation differs from their clustering allocation in that we allocate files in blocks to avoid wasted space or secondlevel allocation, but the blocks in our file are likely to be adjacent to each other as a result of spatial and temporal locality. Our coalescing differs from their clustering in that our coalesced requests have variable sizes so that it can take advantage of increased cache size or it can be split into chunks for better read latency. UFS guarantees system integrity by synchronously writing metadata; therefore, its clustering scheduling is subject to interruption by synchronous writes.

Log-structured file systems collect large amounts of data in main memory and write the data to a new disk location in a single large I/O [17]. Our scan-based scheduling and layout algorithm resembles it in the sense that file data is always written before its metadata, thus avoiding pointers to uninitialized blocks. Our lazy allocation resembles log-structuring in the sense that new data is allocated adjacent on disk to other new data or to existing data written at the same time, so that the chance of a large sequential write is increased. Log-structured file systems have to maintain large free areas on disk; this requires a "cleaning" procedure to counteract the fragmentation caused by old copies of blocks becoming obsolete. Instead of "forcing" existing data to be sequential by re-allocating it at every write, we take advantage of spatial locality by sorting and coalescing the existing data. We can achieve maximum disk bandwidth for most practical access patterns without the cost of cleaning.

The file system described in [8] uses asynchronous writes and passes ordering requirements to the disk scheduler. In a naive scheduler-enforced ordering scheme, dependency cycles may be formed among blocks and certain blocks may never be written to disk due to consistent dependency on others. To fix this problem, dependency information is recorded at a finer grain than block and a dirty block can be written to disk at any time, but excluding the part of the block that has dependency on others. This scheme can potentially reduce the number of disk writes in case of multiple updates, but does not attempt to turn small random writes to large sequential ones. Besides, it may require a large amount of memory to record the fine-grained dependency information and may cause some blocks to be written to disk multiple times.

Disk-directed I/O implements a collective-I/O interface for multiprocessors, in which all processors cooperate to make a single large disk request to improve scheduling and coalescing [13]. Scan-based scheduling and layout can create scheduling opportunities regardless of the number of processors or threads that are issuing the requests because most writes can be batched for better performance.

Journaling file systems like Episode [12], Cedar [9] and NTFS [7] keep dirty data in memory and guarantee integrity by writing metadata changes to journals. Writes to the journal and the metadata are redundant and have to be synchronized with each other. We use journaling (or logging) only for allocation and de-allocation, but not for other metadata changes. No additional synchronization is needed for log writing because the log is the first segment to write during each scan. The log does not need to be visited unless a crash occurs in the last two phases of the scan.

Disk caching disk [11] is actually a more general logging mechanism: it can keep the log either on the same disk as the data disk or on a separate disk; and it logs not only metadata but also file contents. It can also be viewed as a log-structured system except that the log structure is temporary and will eventually be replaced by regular data layout. It involves more disk accesses than regular logging systems: all changes to the data will be written to the cache disk, read back from the cache disk and finally written to the data disk. The configuration for their disk cache system in their experiments includes less than 1 megabyte of RAM, tens of megabytes of cache disk and gigabytes of data disk. A commodity computer can easily have tens or even hundreds of megabytes of main memory and memory-to-memory transfer is 5 times faster than memory-to-disk transfer. Scan-FS takes advantage of the large amount of cache space instead of restricting itself to the small buffer in the disk controller or to physical disks.

Application-controlled file caching, prefetching and disk scheduling [4] creates opportunities for sorting large lists of blocks by generating prefetch requests in groups. However, prefetching requests are issued in the order that the data will be consumed by the CPU and therefore can not be arbitrarily reordered. Scan-FS does not yet support prefetching. We will consider prefetching as a candidate approach to optimize read performance in the future.

The main technical contribution in Scan-FS is the novel combination of segment layout, lazy allocation, request sorting/coalescing and phase-specific recovery.

6 Conclusions

We have designed a scan-based scheduling and layout algorithm for file systems to maximize the utilization of raw disk write bandwidth. It strives to convert small random synchronous writes to large sequential asynchronous writes while maintaining system integrity. We have implemented a prototype called Scan-FS and compared its performance with NTFS running on the same Pentium PC with a SCSI disk. Scan-FS outperforms NTFS by a factor of 2 to 10 in write bandwidth in I/O intensive benchmarks, while offering comparable read bandwidth in I/O intensive benchmarks and comparable performance in memory intensive benchmarks.

References

[1] *Platform SDK", in "MSDN library Visual Studio 6.0.* Microsoft Corporation, 1998.

[2] *Windows Resource Kits", in "MSDN library Visual Studio 6.0.* Microsoft Corporation, 1998.

[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1992.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, perfetching and disk scheduling. In *ACM Transactions on Computer Systems*, Nov. 1996.

[5] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: surviving operating system crashes. In *Proceedings of 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[6] H. Custer. *Inside Windows NT, Chapter 6 The virtual memory manager.* Microsoft Press, 1993.

[7] H. Custer. *Inside the Windows NT file system.* Microsoft Press, 1994.

[8] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.

[9] R. B. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th Symposium on Operating Systems Principles*, Nov. 1987.

[10] http://www.quantum.com/products/hdd/atlas_II/specs.html.

[11] Y. Hu and Q. Yang. Dcd - disk caching disk: A new approach for boosting i/o performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, May 1996.

[12] M. L. Kazar, O. T. Anderson, B. W. Leverett, V. A. postolides, B. L. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, and S. Tu amd E. R. Zayas. Decorum file system architectural overview. In *Proceedings of the 1990 USENIX Summer Conference*, June 1990.

[13] Kotz. Disk-directed i/o for mimd multiprocessors. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.

[14] Liskov, S. Chemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proceedings of the 1991 Symposium on Operating System Principles*, Oct. 1991.

[15] J. C. Mogul and S. R. Kleman. Extent-like performance from a unix file system. In *Proceedings of the 1991 Winter USENIX Symposium*, 1991.

[16] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the 10th Symposium on Operating Systems Principles*, 1985.

[17] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems*, Feb. 1992.

[18] F. Schmidt. *The SCSI bus and IDE interface protocols, applications and programming.* Addison-Wesley Publishers Ltd., 1995.

[19] P. H. Seaman, R. A. Lind, and T. L. Wilson. An analysis of auxiliary-storage activity. *IBM System Journal*, Vol. 5, 1966.

[20] A. S. Tanenbaum and A. S. Woodhull. *Operating systems design and implementation (2nd Edition), Chapter 5 File Systems.* Prentice-Hall Inc., 1997.

[21] L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling for modern disk drives. In *Proceedings of ACM Sigmetrics*, 1994.

	Client BW (MB/s)	Client BW (MB/s)	Disk BW (MB/s)	Disk BW (MB/s)	Bytes Per Write	Bytes Per Write
	NTFS	Scan-FS	NTFS	Scan-FS	NTFS	Scan-FS
Large Seq.	4.2	9.3	3.9	11.5	65108	1595108
Large Ran.	4.4	12.4	3.9	10.6	65338	711577
Small Seq.	7.1	8.4	6.3	14.3	65536	2253944
Small Ran.	1.0	2.0	0.9	3.0	4489	4706

Table 3: Large memory footprint benchmark results: client write bandwidth (MB/s), physical disk write bandwidth (MB/s) and number of bytes per write request. Larger numbers are better.

Read Buffer	4 KB	4 KB	4 KB	1 MB	1 MB	1 MB
	NTFS	Scan-FS	Chunked Scan-FS	NTFS	Scan-FS	Chunked Scan-FS
Max. Lat.	30.0	2123.0	90.0	230.0	2243.0	410.0
Avg. Lat.	2.1	56.9	10.3	161.3	777.3	211.5
Min. Lat.	0.0	0.0	0.0	140.0	180.0	140.0
Read BW	0.39	0.32	0.33	6.24	3.02	4.93
Write BW	4.03	8.81	8.86	0.46	9.42	8.44

Table 4: Results of read-write benchmarks with 4 KB and 1 MB read sizes respectively: client read latency (ms), client read bandwidth (MB/s) and client write bandwidth (MB/s). Smaller latency numbers are better. Large bandwidth numbers are better.