Trail: A Track-based Logging Disk Architecture for Zero-Overhead Writes

Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
chiueh@cs.sunysb.edu

Abstract

A novel disk architecture called *Trail* is proposed to optimize the disk write latency without sacrificing the disk read performance. This architecture features a track-based logging technique, which essentially reduces a disk write latency to the transfer delay. In addition, this disk architecture allows concurrent read/write, and implicit write scheduling without compromising data integrity. Through a synthetic-trace simulation study, we have shown that for transaction processing workloads, the write latency improvement of *Trail* over conventional disk devices is at least an order of magnitude. *Trail*'s read latency performance is also better in all cases, sometimes the improvement is also over an order of magnitude. In terms of disk bandwidth utilization, the *Trail* architecture has a close to 100% write bandwidth efficiency and a read bandwidth efficiency at least as good as a conventional disk.

1. Introduction

A disk I/O request's access delay consists of three parts: *seek, rotational,* and *transfer* latency. Only the transfer latency is proportional to the amount of data that is actually accessed. The other two components depend on the position of the disk read/write head when the request arrives. Some researchers advocate the use of large main memory as a buffer cache to reduce the amount of physical disk I/O's, thus bridging the speed gap between processors and I/O devices. The effectiveness of this approach is demonstrated by various file systems caching studies [BAKE92] [WELC90]. A factor of three in disk data traffic reduction can be easily achieved. In addition, it has been found that the I/O read/write ratio has changed from 4 when there is no caching, to less than 2 when a reasonable caching system is in place. Although both read and write requests can be satisfied in the buffer cache, some applications require writes to be synchronous, i.e., only when data is written on the persistent storage can an I/O request be considered completed. For example, in commercial data processing environments, data integrity is usually a much more important criterion than performance. Lower disk read/write ratios and the need for synchronous writes argue for a storage system that is optimized for disk writes *without* significantly sacrificing the disk read performance.

The log-structured file system (LFS) [ROSE92] developed in U.C. Berkeley is based exactly on this observation. There are two central ideas in LFS. First, a set of temporally consecutive disk writes are clustered and performed in one batch. Second, data on the disk is organized as a log. Logically, disk writes are performed as appends to the log. As a result, the write access overhead is amortized among the write requests participating in a batch. Overwritten data segments are garbage-collected and resued by a background cleaner process. It has been shown [ROSE92] that LFS can effectively utilize a substantial percentage (> 60%) of the raw disk transfer bandwidth. This represents a significant improvement compared to the BSD fast file system implementation (FFS) [McKU84], whose bandwidth utilization is typically under 10%.

However, there are several problems associated with LFS. First, disk reads become more complicated because a data item is not placed on a fixed disk location. At least one more level of directory lookup is needed to locate a particular piece of data. The LFS designer claims that the meta-data is small enough to fit in the main memory and therefore this extra work won't incur significant performance overheads. Despite this argument, the disk read performance still suffers because the spatial locality of data is not likely to be preserved by the log-structured storage organization. The only scenario in which LFS can preserve spatial locatity is when the temporal access patterns of disk reads and writes for a file are identical. It is not clear that this is a valid assumption in general. Even if it is the case, multiple disk write streams from different sources, such as in a network file server, can potentially destroy this locality characteristic. In addition, it has been reported [CARS92] that individual read operations may experience longer queuing delays because of the batched write policy.

The second problem with LFS is that it doesn't work for applications that require synchronous writes, e.g., commercial database systems. In these cases, LFS cannot cluster writes and therefore has to pay for the disk access overhead for each disk write. The third problem with LFS is that it requires extra processing/disk bandwidth to reclaim overwritten storage segments. Although it is claimed that this overhead is relatively small [ROSE92], the optimal reclamation policy to achieve an ideal bimodal segment utilization is rather subtle and seems to be workload-dependent. Moreover, a continuous segment cleaner will presumably interfere with normal disk accesses and thus degrades the latency of individual I/O requests.

In this paper, we propose a new disk architecture called *Trail* that incurs almost no disk write overhead, i.e., no seek and rotational latency. The basic idea is to log disk writes to a separate small disk first and to commit the updates to the main disk when it is available. The novelty of our approach is to use a track-based logging disk as a persistent cache and to embed the low-level storage management intelligence inside the disk controller. This disk architecture preserves the same SCSI interface to device drivers so that existing systems can reap the write performance advantage of LFS without modifying the kernel. Moreover, the file system can adopt a read-optimized storage organization such as FFS and lets the hardware optimize the disk write performance. With a little extra cost, one can combine the advantages of FFS and LFS to achieve high performance for both disk reads and writes.

The rest of this paper is organized as follows. The hardware organization and the storage management algorithms used in *Trail* are presented in Section Two. The performance results and analysis of a simulation study for *Trail* is reported in Section Three. Section Four concludes this paper with a summary of major research results and a pointer for future work.

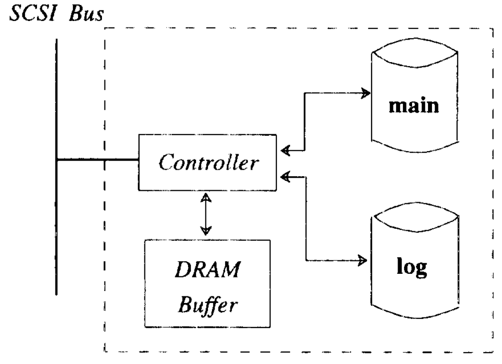

Figure 1 Hardware Organization of *Trail*

2. The Trail Disk Architecture

2.1. Principle of Operation

The design goals of *Trail* are: (1) To improve or at least preserve the disk write performance of LFS without its associated problems; (2) To maintain the same hardware interface of conventional disk devices so that most of the systems software can be left intact. The hardware architecture of *Trail* is shown in Figure 1. It consists of a main disk, a log disk, a DRAM buffer, and an intelligent disk controller. Typically the log disk is much smaller than the main disk. In fact our design only needs the cheapest disk model available in the market, e.g., a 40 MB drive. Therefore, the cost of *Trail* is *not* twice as much as a main disk. Files are permanently stored on the main disk with a data organization determined by the file system. Because *Trail* is optimized for disk write performance, the file-system disk data representation should be oriented towards disk read performance. The log disk serves as a persistent write buffer. The intelligent disk controller interacts with the outside world through a standard interface such as SCSI and coordinates the data transfers between these two disks. A *Trail* disk behaves like a normal disk except with very fast write performance.

A *Trail* disk operates as follows. The read/write head of the log disk is guaranteed to be on a cylinder that has at least one free track. In response to a disk write request, the disk controller simply starts the write to wherever the disk head happens to be. Because there is always a free track under the disk head, the write can be performed immediately. As a result, there is no seek and rotational latency involved in the write process! The only latency is due to data transfer. As soon as data is completely transferred to the log disk, the device can return a completion signal to the software. At this point, there is still a copy of the data kept in the disk controller's DRAM-based buffer. From the software standpoint, a *Trail* disk completes a disk write without any additional delay. Ultimately the data will be put back to the main disk, at a time determined by the intelligent controller. Only when the data copy is safely moved to the main disk can its occupied buffer space be reused. When a read request arrives, the controller first looks up the buffer and returns the requested data if it is there. Otherwise an access to the main disk is scheduled to satisfy the request. Note that the log disk *never* services disk reads. So it is not a cache. It serves as a safety net to fall back on when the contents of the controller buffer are corrupted.

Writing data out to a log before committing it to the "official" data storage is not a new idea. Database systems use this technique mostly for maintaining transactional consistency rather than for performance. Moreover, in transaction processing systems, the software actually has to perform each write twice and is responsible for the management of log space. *Trail* hides all management overheads behind the SCSI interface. This simplifies and improves the performance of application programs. The other difference is that *Trail* uses a track-based logging technique. Conceptually each track only holds one disk write's worth of data. Even if the current track is not full, the log disk's head will move to the next free track in anticipation to service the next write. A track becomes free when the data it contains is transferred to the main disk. There are two cases in which this scheme will still cause additional overheads. First, if no track is free when a write request arrives, the write request will have to wait until a free track becomes available. Second, when the size of a write request is larger than a disk track, , an extra overhead equal to a head switch time if the next free track is in the same cylinder as the current track, or to a track-to-track seek time if the next free track is not in the same cylinder, needs to be paid. Track-based logging eliminates both the rotational and seek latency at the expense of inefficient log disk space usage.

Compared to LFS, *Trail* achieves a better disk write performance because of the reduction in the rotational latency. Moreover, the file system is not required to perform clustered writes in one batch. Consequently the performance of synchronous writes is only limited by the disk's raw transfer bandwidth. Because the main disk assumes a data organization that is determined by the file system, disk read performance is not penalized. Last but not least, there is no need for garbage collection because the hardware assumes the responsibility of managing the log disk. This simplifies the implementation and improves the performance of the storage management software.

2.2. Log Disk Management

The most critical part of the *Trail* architecture is log disk management. Because hardware controls the log disk directly, the management algorithm must be simple. Two issues need to be addressed. First, because the log disk is used as a circular buffer, we need a mechanism to switch quickly to the outermost cylinder when the disk head moves inwards and hits the innermost track and vice versa. Two alternatives are considered. The first one simply assumes that a full seek delay needs to be paid when the disk head reaches the boundary track. If no write request is arriving during the full seek period, this delay is not visible to the applications. Considering a 40-MB drive with a 40-KB track size. This means that such a full seek delay, roughly 20 ms, will *potentially* be visible every 1000 writes, assuming that the size of each disk write is less than 40 KB. The other alternative is to interlace the writes to every other cylinder. Assume cylinders in a disk are numbered from 0 to 999 from the innermost to the outermost. When the disk head is moving outward, the cylinders to be written are 0, 2, 4, ... until it reaches the 998-th cylinder. At this time, the disk head moves to the 999-th cylinder, turns inward, and marches through the cylinders 997, 995, ... until it hits the 1-th cylinder. And then the cycle repeats. The advantage of this scheme is that the full seek delay in the first scheme is distributed over each disk head transition, and therefore every head transition incurs roughly the same overhead. The disadvantage is that the delay of each head transition is a two-track seek time. Since most commercial disks have optimized for the track-to-track seek performance, this technique may not be desirable because it trades the common-case performance for the boundary case. In the simulation study, we choose the first approach.

The other design issue is related to recovery. When a *Trail* disk returns a completion signal to the software, the data is assumed to be persistent although not necessarily in its intended destination disk location. In the case of a power failure, the contents of the controller's DRAM-based buffer are lost. During reconstruction, the controller's buffer is rebuilt from the log disk. To allow crash recovery, each disk write to the log disk must be self-identifying. That is, each disk write must include a special bit pattern, a timestamp, a request size, the destination disk address, and the actual data. The special bit pattern is used as a delimiter to designate the beginning of a write request. The request size allows the reconstruction software to handle disk writes that span multiple tracks. To reconstruct the controller buffer, the system must scan the entire log disk once to eliminate invalid writes and to establish a timestamp order for valid writes. Three rules are used:

[1] Within a track, only the disk write with the most recent timestamp is considered; others are spurious.

[2] At the disk level, the disk writes from each track are ordered according to their timestamps. Those writes that are overwritten by later ones are considered invalid.

[3] Accumulate the request sizes of the valid writes from the most to the least recent until such a write request that when including it, the accumulated sum is greater than the controller buffer size. All writes earlier than this write, including itself, are considered invalid.

The reconstruction process requires a second pass across the log disk, in which the valid writes determined in the first pass via the above rules are brought into the controller buffer. Because the scans involve only sequential access, the reconstruction delay can be significantly reduced. In addition, because the log disk's contents are rebuilt in the controller buffer rather than in the main disk, no random accesses to the main disk are required.

3. Performance Evaluation

3.1. Simulation Methodology

We use a simulation approach to evaluate the performance of the *Trail* architecture. A simulator that accurately models the disk head positions, the seek latency, and the transfer delay is developed. The rotational latency is assumed to be uniformly distributed. Two disk architectures are compared. One is the conventional disk architecture, which services disk I/O requests in their arriving order. There is no buffering and no scheduling. The other is the *Trail* architecture, which features track-based logging, read/write concurrency, and implicit write scheduling. To be fair, disk caching in *Trail* is disabled. All reads must access the main disk. They take a higher priority to the main disk by preempting an on-going write.

The simulation workload is based on synthetic traces, which are characterized by the following parameters: *read/write ratio, minimum request size, maximum request size, mean inter-arrival time,* and *locality characteristic*. Each disk request is either a read or a write. The start location of each request is uniformly and independently distributed among a disk's cylinders. The *locality characteristic* parameter describes the degree of clustering among the start cylinders of neighboring requests. The size of each request is assumed to be uniformly distributed between *minimum request size* and *maximum request size.* The interval between consecutive requests follows an exponential distribution with a mean equal to the *mean inter-arrival time.*

The main disk in the *Trail* architecture is assumed to have the same configuration as the conventional disk. Because their physical configurations don't impact the performance results, we will only focus on their performance-related parameters. The log disk in *Trail* is assumed to have the same performance parameters but a smaller physical configuration. Table I shows the physical configuration and performance parameters of the log disk. For each combination of the workload parameter and disk configuration, 10,000 requests are simulated. All performance results are obtained by taking the average of 10,000 simulation measurements.

Tracks/Cylinder	4
Number of Cylinders	500
Track Size	25 Kbyte
Full Rotation Delay	16.7 ms
Maximum Seek Time	29.4 ms
Track-to-Track Seek Time	2 ms
Head Switch Time	1 ms

Table I Disk Parameters

The metrics of interest in this study are the following: *read latency, write latency, bandwidth efficiency,* and *buffer size.* The *read (write) latency* is the period between the arrival of a read (write) request and its completion. The *bandwidth efficiency* calibrates the effective utilization of the raw disk transfer bandwidth. For a conventional disk, it is calculated by dividing the time when a disk is transferring data by the time when it is either seeking, rotating, or transferring.

Note that the divisor doesn't include the disk idle time. For a *Trail* disk, the *read bandwidth efficiency* is derived from the main disk while the *write bandwidth efficiency* is from the log disk. The *controller buffer size* measures the average space that the controller's buffer needs to provide a reasonable level of performance. Due to space constraints, we didn't include simulation results that depend on workload characteristics, and disk parameters.

3.2. Performance Analysis

We are mainly interested in two kinds of workloads. The first type represents transaction processing workloads. In this environment, the disk request size is small, the I/O rate is high, and the reference locality is low. The second type represents the other end of the spectrum, where the request size is much larger, but the I/O rate is comparatively lower. Let's call the latter the file systems workload. Unless stated otherwise, subsequent performance results are based on the following assumptions:

Minimum request size is fixed at one Kbyte.

Read/write ratio is assumed to be four.

Locality characteristic value is one, i.e., no locality.

Figure 2 shows the write latency ratio of the conventional disk architecture to the *Trail* architecture. The X axis stands for the disk request's inter-arrival time in msec. Three different maximum request sizes, 1 Kbyte, 5 Kbyte, and 10 Kbyte, are used in this study. For high I/O rates (e.g., 200 requests/sec), the improvement of the *Trail* architecture over conventional disk devices is at least three orders of magnitude, a rather significant difference. Even for lower I/O rates, the improvement is still over an order of magnitude. The performance curves of these three workloads exhibit similar structures. In particular, the latency improvement drops significantly when the I/O rate reaches 50 requests/sec (or the inter-arrival time is 20 ms). Except for the rapid transition period, smaller maximum request sizes have better write latency improvement. Figure 3 shows the read latency ratio for the same workloads. Unlike the monotonicity structure in Figure 2, there is a peak in each performance curve. Moreover, the workload conditions in which the peaks in Figure 3 occur roughly correspond to where rapid transitions in Figure 2 take place. Although the *Trail* architecture aims at disk write performance optimization, Figure 3 shows that the read performance is also improved. In some cases, this improvement can be as much as two orders of magnitude. This demonstrates one of the design goals of *Trail*: improving write performance without sacrificing read performance. Like the write case, smaller maximum request sizes have better read latency improvement.

Figure 4 shows the average amount of controller buffer needed in *Trail* to provide the performance improvement shown in Figure 2. The vertical axis stands for the buffer size in Kbyte. The statistics are calculated by assuming an infinite buffer and measuring the active buffer size when a new disk request arrives. The accumulative sum of these measurements is then divided by the number of disk requests. For high I/O rates, the buffer requirement ranges from 1 Mbyte to 5 Mbyte. The buffer requirement reduces dramatically as I/O rates decrease. It is less than 5 Kbyte in the lightest load case. Note that Figure 4 shows a significant drop-off during the same transition period as in Figure 2 and 3.

The above metrics are useful from the application's standpoint. But they don't tell the proposed architecture's effectiveness in utilizing raw disk bandwidth. In Table II, the read and write bandwidth efficiency of the conventional and the *Trail* disk architectures are compared. The bandwidth efficiency is largely independent of the inter-arrival time but is strongly determined by the maximum request size. For the *Trail* architecture, the write bandwidth efficiency calibrates the efficiency of the log disk, which only service writes, while the read bandwidth efficiency calibrates the read/write efficiency of the main disk. *Trail* pushes the write bandwidth efficiency close to 100%, significantly improving over conventional disk architectures. On the other hand, the read bandwidth efficiency of the two architectures are almost identical in all considered workloads.

Maximum Request Size	Conventional		Trail	
	Read	Write	Read	Write
1 Kbyte	3.70%	3.71%	3.76%	93.78%
5 Kbyte	8.78%	8.65%	8.89%	95.94%
10 Kbyte	16.17%	15.84%	16.32%	96.73%

Table II Bandwidth Efficiency Comparison under Transaction Processing Workloads

For the file systems workload, the latency advantage of *Trail* is less significant as shown in Figure 5 In particular the write latency improvement is reduced to 30% when the I/O rate is low. The reason is that large writes are not bypassed. For large writes, the latency improvement from track-based logging is not as significant, but they take twice as much raw disk bandwidth, which eventually reduce the overall improvement. However, the read latency improvement of *Trail* over conventional disks still holds for all workloads. Table III shows the comparison of bandwidth efficiency under file systems workloads.

Maximum Request Size	Conventional		Trail	
	Read	Write	Read	Write
200 Kbyte	80.45%	80.03%	80.44%	99.90%
650 Kbyte	93.08%	92.91%	93.09%	99.92%
1000 Kbyte	95.39%	95.28%	95.42%	99.93%

Table III Bandwidth Efficiency Comparison under File Systems Workloads

4. Conclusion

In this paper, we propose a novel disk architecture called *Trail* that optimizes the write performance without sacrificing the read performance. Using the concept of *track-based logging*, *Trail* achieves almost zero-overhead disk writes. That is, aside from the queuing delay, a disk write latency is only due to its data transfer delay. Consequently the performance impact due to synchronous writes is minimized. Since *Trail* manages logging completely inside a disk device, the software system, either device drivers or file systems, doesn't require any modifications. Because the main disk data organization can be optimized for reads, *Trail* eliminates the potential read performance problem of Log-structured File Systems. In addition, there is no need for garbage collection.

We have shown through simulation that for transaction processing workloads, the write latency improvement over conventional disk devices is at least an order of magnitude. *Trail*'s read latency performance is also better, sometimes the improvement is over an order of magnitude. In terms of disk bandwidth utilization, a *Trail* device has a close to 100% write bandwidth efficiency and a read bandwidth efficiency at least as good as a conventional disk.

REFERENCE

[BAKE92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, M. Seltzer, "Non-Volatile Memory for Fast, Reliable File System," Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 10-22, Boston, MA., October 1992.

[CARS92] S. Carson, S. Seita, "Optimal Write Batch Size in Log-structured File Systems," Proc. of USENIX File Systems Workshop, pp 79-91, Ann Arbor, Michigan, May 1992.

[McKu84] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX," ACM Transactions on Computer Systems, 2(3):181-197, August 1984.

[ROSE92] M. Rosenblum, *The Design and Implementation of a Log-structured File System*, PhD Thesis, UCB/CSD 92/696, June 1992, Computer Science Division, University of California, Berkeley.

[WELC90] B. Welch, "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System," PhD Thesis, 1990, University of California, Berkeley.

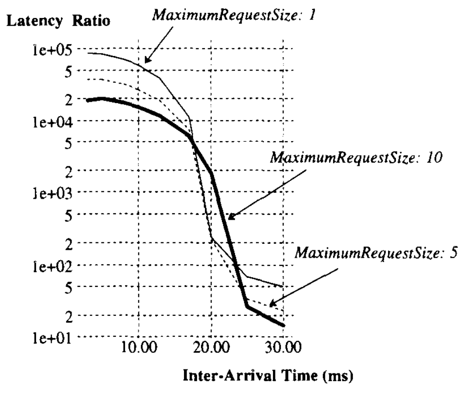

Figure 2 Write Latency Ratio Under Transaction Processing Workloads

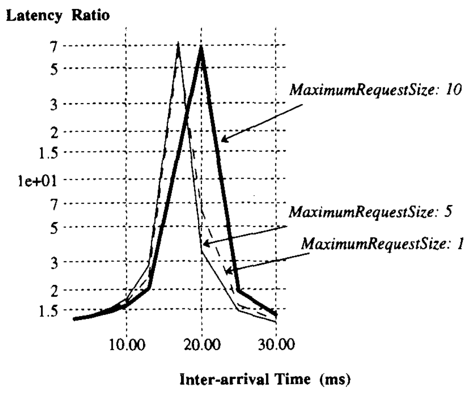

Figure 3 Read Latency Ratio Under Transaction Processing Workloads

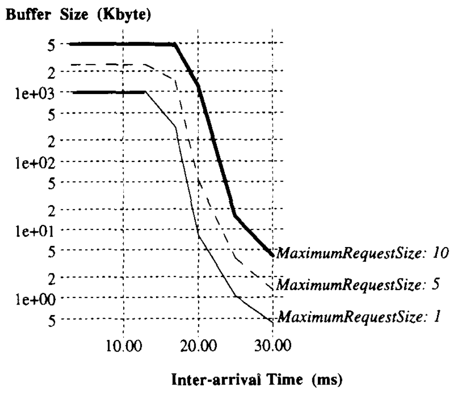

Figure 4 Buffer Size Requirement Under Transaction Processing Workloads

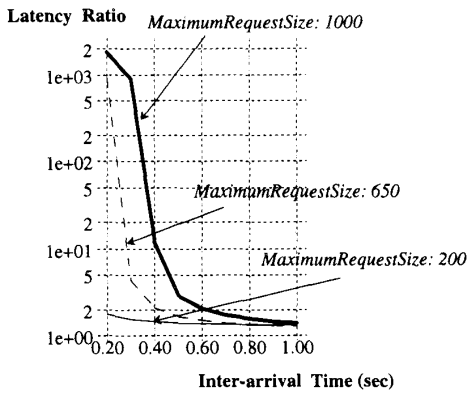

Figure 5 Write Latency Ratio Under File Systems Workloads