Track-Based Disk Logging

Tzi-cker Chiueh Lan Huang

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

Abstract

Disk logging is a fundamental building block for fault-tolerance system design because it captures a persistent snapshot of critical system state for subsequent recovery in the occurrence of failures. Logging typically is required to be synchronous to ensure absolute recoverability. Therefore speeding up synchronous disk write is critical to those fault tolerance systems that are based on disk logging. This paper describes a novel track-based disk logging *technique that is able to reduce the latency of synchronous disk writes to the minimum without compromising data integrity guarantee. As an application of track-based disk logging, we present the design and implementation of a low-write-latency disk subsystem called* Trail*. Through a fully operational* Trail *prototype, we demonstrate that* Trail *achieves the best known disk logging performance record, which is close to* data transfer delay *plus* command processing overhead*. A 4-KByte disk write takes less than 1.5 msec. Based on the TPC-C benchmark, the transaction throughput of a* Trail *-based transaction processing system is on an average 62.9% higher than one based on a standard disk subsystem, and the database logging-related disk I/O overhead is reduced by 42%.*

1. Introduction

Logging system state into persistent storage is a fundamental building block for fault tolerance system design. Disk logging is one of the most popular techniques for keeping a persistent copy of the system state for subsequent failure recovery. To guarantee correctness, disk logging typically requires *synchronous* disk writes, which are relatively expensive especially in view of the growing performance gap between CPU and disk. Techniques have been proposed to improve the logging performance. For example, the idea of *group commit* was proposed and actually deployed in actual relational database system products to reduce the logging overhead at the time of transaction commit. In other cases, expensive non-volatile RAMs are employed to minimize the performance impacts of state logging. In this paper we propose a novel logging mechanism called *track-based disk logging*, which exploits inexpensive disk storage space to reduce the disk logging overhead to the minimum, i.e., approximately the disk data transfer delay. As a demonstration of the track-based disk logging technology, this paper describes the design, implementation, and evaluation of a disk storage subsystem called *Trail* that achieves the lowest synchronous disk write latency ever reported in the literature without compromising data integrity.

The rest of this paper is organized as follows. The next section presents previous research efforts that aim to reducing the performance impacts of synchronous disk writes. The key design issues of track-based disk logging and their solutions are detailed in Section 3. Section 4 describes the design and implementation of *Trail*, a low-latency disk system based on track-based logging. Section 5 presents the results of a detailed performance study on a fully operational Linux-based *Trail* prototype. We conclude this paper with a summary of main research result and a brief outline of on-going work in Section 6.

2. Related Work

The idea of track-based logging was originally proposed by Chiueh [2] to minimize the seek delay and rotational latency associated with synchronous disk write. The author assumed that the disk interface hardware supports a special access primitive that allows a piece of data to be written where the disk head is at that instant. Similar ideas have been implemented in IBM's WADS system [15], which uses fixed-head disks (disk drums) to implement the "write where the disk head is" semantics, and in a paper design [7], which attempts to reduce the rotational latency by using a separate log disk [4] and by intentionally writing to a new track when the current track's utilization reaches a certain threshold.

Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)
0-7695-1597-5/02 $17.00 © 2002 IEEE

Several techniques have been proposed to improve the performance of synchronous writes. Two well known examples are Log-structured File System (LFS) [11] and meta-data logging [6, 3] or journaling file system (JFS). In LFS, the entire file system is organized as a sequential log, and all disk writes are first buffered in main memory and then only written to the disk after a basic write unit, called a segment, has been accumulated. LFS significantly reduces the disk write overhead because a number of small asynchronous writes are batched into a single large write, thus amortizing the fixed disk access overhead over multiple write requests. However, LFS cannot support synchronous writes well because of the inability to batch, and all disk writes still incur rotational latency. Meta-data logging logs all meta-data changes to the file system, so that the need of synchronous file system writes due to file system meta-data updates is eliminated. Meta-data logging is effective in improving overall file system performance and is widely used in many commercial systems. However, this approach does not help the performance of synchronous disk writes of the user applications, e.g., database logging. Ganger [5] designed another approach called "soft update" to solve the meta-data synchronous write problem. Soft update has comparable performance as JFS [12] in many cases. Like JFS, it only improves the write performance of file system meta data.

Yang and Hu proposed disk caching disks [8]. An NVRAM (non-volatile RAM) and a log-based cache-disk together form a two-level cache hierarchy to cache write data. Small write requests are first collected in the small NVRAM buffer and flushed into the cache-disk when the NVRAM buffer is full. Each write to the cache-disk represents a batch of multiple small writes, and thus greatly reduces the overall seek overhead.

Compared to LFS, *Trail* allows the data layout in the file system to be optimized for file reads because *Trail* reduces disk write latency in a way completely transparent to the file system. *Trail* also has a better synchronous write performance than LFS because it eliminates rotational latency. *Trail* incurs less disk access overhead due to garbage collection because pending write requests are written to data disks from main memory rather than from the log disk. In contrast, LFS needs a disk read and a disk write to clean a disk segment. The fundamental reason that *Trail* incurs less garbage collection overhead is that *Trail* allocates and de-allocates the log disk space in a FIFO order. Compared to the meta-data logging approach, *Trail* is more general as it transparently applies the logging technique to all data blocks, rather than just to file system meta-data. Compared to DCD, *Trail* does not require extra hardware (NVRAM), and supports the notion of track-based logging to further reduce the rotational latency. Like DCD, *Trail* is designed and implemented as a self-contained disk device driver with minimum modifications to the file system.

3. Track-Based Logging

There are three challenges in the design of track-based logging. First, to minimize the latency of each logging operation, the disk write in track-based logging needs to be performed at where the disk head is at that instant. However, commodity disk drives do not provide such flexibility. Second, because track-based logging does not maintain a contiguous log, it needs a mechanism to allow recovery software to identify where in each track the log records are. However, this mechanism cannot use any additional persistent data structure. Finally, to maintain the same level of data integrity as synchronous disk logging, after a power failure track-based logging needs to be able to quickly and reliably recover the data that is written to the log disk but not yet to the data disk. The following three subsections describe how these three issues are addressed in more detail.

3.1. Disk Head Position Prediction

Current disk interface standards, such as SCSI and IDE, do not support a command that allows software to write a data block where the disk head currently is. Software has to explicitly specify a disk address for every disk access command. Therefore, the only way to implement the "write where the disk head current is" semantics for the log disk is to have an accurate estimate of the log disk head's position at the time when a disk write request is sent to the *Trail* driver. To derive the disk head position estimate accurately, *Trail* needs to have a detailed knowledge of the log disk's physical geometry as well as the disk controller and disk command processing overhead. Once the disk head position is known, the *Trail* driver writes data to the next closest free sector on the current track. The accuracy of the disk head position prediction algorithm affects the amount of rotational latency log disk writes actually experience, but would not affect the correctness.

When a log disk write is finished, the *Trail* driver checks the space utilization of the current track and positions the disk head to the sector on the next track that is physically the closest to the head's current position, if the current track utilization reaches certain pre-defined threshold [7]. This is an optimization over the scheme in which the log disk head is moved to the next track after each disk write. When a log disk write is finished, the *Trail* driver positions the log disk's head to the sector on the next track that is physically the closest to the head's current position. For example, to move from Track i to $i+1$, knowing the number of sectors in the ith track, *Trail* can calculate the target block address of the physically closest sector on track $i+1$. After the head move, the *Trail* driver takes a timestamp and stores the timestamp in T_0 and the target block address to which the disk head moves, in LBA_0. (T_0, LBA_0) serves as a reference point for disk head prediction. From the disk geometry information, the logical block address LBA_0 can be converted to

a (cylinder, head, sector) address $(C_0,\ H_0,\ S_0)$. The *Trail* driver predicts the disk head position at time T_1, LBA_1 or $(C_1,\ H_1,\ S_1)$, using the following formulas:

$$S_1 = (\frac{(T_1 - T_0)\ mod\ RotateTime}{RotateTime} * SPT + S_0 + \delta)\ mod\ SPT$$

$$H_1 = H_0,\ C_1 = C_0 \quad LBA_1 = LBA_0 - S_0 + S_1$$

where SPT is the number of sectors in the current track, and $RotateTime$ is the disk's rotation cycle time. δ is an empirically derived value to compensate for the command processing overhead and other inherent overhead. To derive δ, we started with (C_0, H_0, S_0), performed a series of single-sector write operations with different target addresses $(C_0, H_0, S_0 + \delta)$ corresponding to different δ values, and measured their latency. In each such write, if the δ value is smaller than desired, the resulting write latency will be close to a full rotation cycle. The smallest δ value that does not incur a full rotation delay is the final δ value used in the disk head position prediction formula. For example, δ value is less than 15 for a Seagate ST41601N drive, which accounts for fixed disk controller-related and on-disk processing overhead. This head prediction mechanism does not require intensive computation. Less than one microsecond is needed to take a timestamp and to compute the prediction formula on a Pentium II 300 MHz machine.

The disk head position prediction algorithm is triggered before each log disk write. If log disk writes are sufficiently frequent, the algorithm's predicted disk head position should be reasonably accurate. However, because of the deviation in the disk rotation speed and periodic internal disk behavior in some drives [10], the predictions will go awry after a long period of disk idle time. Therefore the *Trail* driver needs to periodically reposition the log disk head and updates the reference point accordingly. This periodic repositioning ensures that the disk head position prediction is always accurate even when log disk writes are infrequent. The performance cost of repositioning is minimal since it is activated only when the log disk is idle.

3.2. Self-Describing Log Organization

Trail's log disk contains two types of data: write record for user data and trail disk meta-data (`log_disk_header`). A write record is associated with each log disk write. These write records are self-describing, so that no extra global meta-data is needed. Each write record contains a record header (`record_header`) and the associated data blocks. A global trail disk header (`log_disk_header`) for the entire log disk is stored at the first track on the log disk (or a well-known location). It is also replicated at several other places on the disk to improve the robustness. A disk's physical geometry information is stored right next to the global disk header. The detailed structure of the per-record header and the global log disk header are shown in the following:

```
typedef struct log_disk_header {
   char             signature[MAX_SIG_LEN];
   unsigned long    epoch;
   int              crash_var;
} log_header_t;

typedef struct record_header {
   unsigned char    first_byte_of_header;
   unsigned long    batch_size;
   unsigned char    signature[MAX_SIG_LEN];
   unsigned long    epoch;
   unsigned long    sequence_id;
   unsigned long    prev_sect;
   unsigned char    first_data_byte[MAX_TRAIL_BATCH];
   unsigned long    log_LBA[MAX_TRAIL_BATCH];
   unsigned long    data_LBA[MAX_TRAIL_BATCH];
   unsigned char    data_major[MAX_TRAIL_BATCH];
   unsigned char    data_minor[MAX_TRAIL_BATCH];
   unsigned long    log_head;
} sect_head_t;
```

The `signature` field in the global `log_disk_header` is used to identify a disk as a *Trail* log disk when the *Trail* driver boots up. The `epoch` field is incremented each time when the *Trail* driver is initialized. This counter is used in data recovery. `crash variable` is a flag indicates whether the *Trail* driver needs to invoke a recovery process when it boots up.

The write record for each log disk write operation consists of a data record header, which occupies one sector, and user data blocks, each of which is also a sector. The *Trail* driver uses the `signature` field to identify the beginning of a valid write record during recovery scanning. To guarantee the uniqueness of the signature bit pattern, the *Trail* driver marks the first byte of each write record header as `0xFF` (`first_byte_of_header`) and the first byte of each user data sector as `0x00`. The original first byte of the user data sector is moved to the `first_data_byte[]` field of the data record header and will be put back to user data record during recovery. This arrangement allows the scanning code to distinguish header from payload without resorting to expensive techniques such as bit stuffing as used in communications systems. The `epoch` field has the same meaning as the `epoch` field in `log_disk_header`. The `sequence_id` field is incremented whenever a new write record is written to the disk. `first_byte_of_header`, `epoch` and `sequence_id` together guarantee the uniqueness of a write record. The sector with the largest (`epoch`, `sequence_id`) combination on a track represents the most recent write on that track.

The `prev_sect` field of a write record points to the write record of the previous log disk write, so that the traversal of the disk during recovery can jump from record to record without sequential scanning. The `data_major, data_minor, data_LBA` fields together store the target

address to the data disk of the original disk write request. log_LBA stores the target address of the written data on the log disk. Due to batching optimization, as explained in the next subsection, a single write record could contain multiple synchronous disk write requests, the batch_size field indicates the number of disk writes in the write record. Accordingly, the first_data_byte, log_LBA, data_major, data_minor, data_LBA fields are all provisioned as arrays, one element for each physical write request in the batch. The log_head field of each write record keeps the head of the active portion of the log disk when the write record is being written to the log disk, and thus represents the oldest write record on the log disk that has not been committed to the data disk. The log_head field bounds the amount of backward scanning during the failure recovery process from a given write record and thus significantly improves the recovery speed.

3.3. Failure Recovery

The crash variable in the log disk header is initialized to 0 when the machine boots up and is reset to 1 when the machine shuts down normally. If the system crashes and the log disk is not properly unmounted, the crash variable will remain at 0, which will trigger *Trail*'s data recovery procedure when the machine re-boots after a crash.

Trail's recovery module first scans the entire log disk to locate the youngest active write record, i.e., the one whose epoch value is equal to the epoch value in the log_disk_header and whose sequence_id value is the largest. Once the youngest write record is identified, *Trail* can identify all the write records that potentially have not been committed to the data disks by tracing back through the prev_sect field of the record headers. To ensure logical correctness, pending write records on the log disk are propagated to the data disk in the same temporal order as they were issued to the *Trail* driver, i.e., starting from the track with minimum sequence_id value to the one with the largest sequence_id value.

To speed up the recovery process, *Trail* incorporates two performance optimizations. First, *Trail* samples the tracks of the log disk to locate the youngest active write record using binary search, instead of sequential scan. The youngest pending write record is the record with maximum sequence_id. We implemented a binary search algorithm to locate the youngest active record on trail log disk. It takes O($\lg$ N) track scans to locate the head of the sequence of log records on log disk. For a disk with 100K tracks, the total number of tracks that need to be scanned is less than 30. The second optimization in *Trail* is to reduce the number of write records that need to be propagated back to the data disks during recovery. *Trail* includes in every write record a log_head field, which indicates where the head of the log disk's active portion is when the write record is appended to the log disk's tail. With this field, once *Trail* locates the youngest active write record, it only needs to trace as far back as this record's log_head field indicates, rather than the entire disk.

4. Trail Implementation

4.1. Overview

The *Trail* disk storage subsystem consists of a log disk and one or multiple normal data disks. The idea behind *Trail* is to log each disk write on a separate log disk first and to complete the write request to the corresponding data disk asynchronously at some later time. Data blocks are written back to the normal data disks from main memory rather than from the log disk to ensure that the log disk's head is always on an empty track when a subsequent disk write request arrives. The novelty of *Trail*'s lies in the way it writes to the logging disk.

In *Trail*, 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 *Trail* driver first writes the data block to *wherever* the disk head happens to be at that point in time. Because the disk head of the log disk is always on a free track, it is safe to write the data block to the head's current position without causing data corruption. After a log disk write is completed, the log disk's head immediately moves to the next track to maintain the invariant that the log disk's head is always on a free track. Essentially the entire log disk serves as a circular logging buffer, with tracks as basic logging units, hence the name *track-based logging*. As a result of this write-to-where-the-head-is policy, *Trail* can reduce the synchronous disk write latency to the best known empirical result so far, which is equal to the sum of data transfer delay and disk command processing overhead.

As soon as the data is written to the log disk, the *Trail* driver returns a completion signal to higher-level software and the application process that is blocked on synchronous write can continue after receiving this completion signal. Data logged to the log disk are written to the data disk eventually. A previously occupied track is freed only after the data block logged to the track is written to its corresponding data disk. In the case of power failure, *Trail* reconstructs the contents of the log disk and writes the pending data back to the normal disks after the system reboots. Therefore, *Trail* provides the same level of data integrity guarantee as traditional synchronous disk write implementations.

Figure 1 shows the hardware organization of the *Trail* architecture. Any type of IDE or SCSI hard drive can be used as the *Trail* log disk. There could be one or multiple

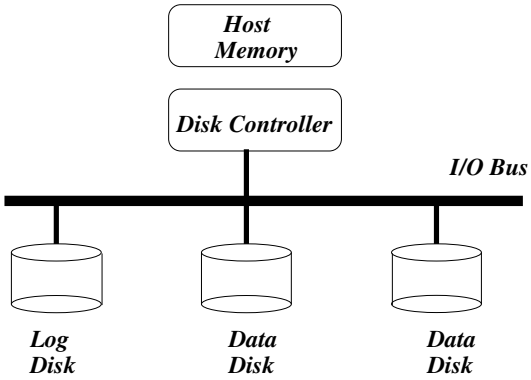


Figure 1. *Hardware Organization of the* Trail *architecture, which consists of a log disk and one or multiple data disks that it serves. The* Trail *driver implements track-based logging using a portion of the host memory and the log disk.*

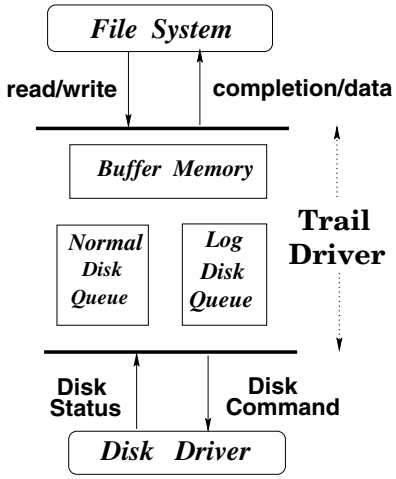


Figure 2. *Software Architecture of the* Trail *, which manages its buffer memory, a normal disk request queue and a log disk request queue.* Trail *is implemented as a thin layer between the file system and the disk device driver and is thus transparent to the rest of the kernel.*

data disks that share a single log disk. The *Trail* software driver services disk read requests in a synchronous fashion, but services disk write requests asynchronously. To reduce the disk read latency, data disk reads are given higher priority than data disk writes. The log disk plays the role of a persistent buffer that achieves similar performance as asynchronous disk writes while maintaining the same level of data integrity as synchronous disk writes. The *Trail* driver uses part of the host memory as its staging buffer.

Figure 2 shows the software architecture of the *Trail* prototype and how it is embedded in the Linux kernel. Linux file system interacts directly with the *Trail* driver using a low-level access interface based on physical disk read/write requests. The interface exposed by the *Trail* driver is exactly the same as those exposed by standard disk device drivers, thus hiding all the operational details of *Trail* from the file system. The only visible difference is that the response time for synchronous write is significantly smaller. With this software architecture, the required modification to the file system is minimal because all storage management intelligence is embedded in the *Trail* driver. At the file system level, the *Trail* prototype includes an additional check to ensure that buffers are not freed prematurely before the data disk writes are finished.

The formatting tool writes the log disk's physical geometry data as well as the `signature` and `crash variable` to the dedicated tracks on log disk, and resets the rest of the disk content to zero. Once a disk is formatted as a *Trail* log disk, every time the *Trail* driver starts up, it reads in the log disk header and initialize the necessary data structures.

4.2. Disk Write

When the file system issues a disk write request, the *Trail* driver writes them to the log disk, and returns control back to the file system. Asynchronously the *Trail* driver writes the disk blocks to the data disks from the buffer memory. Note that the *Trail* driver treats all physical disk writes as synchronous writes.

The *Trail* driver works very much similarly to an IDE or SCSI disk driver. If the log disk request queue is empty at the time when a synchronous disk write request arrives, the request is serviced directly, otherwise it is queued in the log disk queue. When the previous disk write request is completed, the *Trail* driver gains control, *batches all the requests currently in the log disk queue* if possible, and finally writes them into an empty track on the log disk. With this batched write optimization, the *Trail* driver converts multiple small writes requests into one large write.

After each request is serviced, the *Trail* driver moves the disk head to the next track before it starts to service the next request(s) if the space utilization of the current track exceeds a threshold. This threshold is currently set to 30%, which ensures that the next log disk write does not need to incur a long rotation delay to reach the next closest free sectors. This mechanism allows multiple batched writes to be performed on the same track. In the presence of bursty writes, a single batched write typically already exceeds the 30% utilization threshold, and therefore the optimization of multiple batched writes per track is rarely triggered and therefore not very useful in practice.

The *Trail* driver pins down in its buffer memory those disk blocks that have been written to the log disk but have not yet been written back to the data disks. When a write request is finally "committed" to the data disks, its occupied memory as well as the associated log disk blocks are released. Standard file system implementations assume that a buffer page that is being written to the disk cannot be updated by another process. This is typically done through a locking mechanism. In *Trail*, the source buffer page of a disk write request is unlocked immediately after it is written to the log disk. That is, a subsequent write request to the same buffer page can proceed without waiting for the previous write request to be committed to the data disk. A write request for the same buffer may be enqueued to the data disk queue before the previous enqueued I/O request for it has been serviced. In this case, only one request for the buffer is kept in the queue and other writes requests to the same buffer are skipped. The buffers associated with skipped requests are released too. To guarantee correctness, *Trail* cancels the data disk write operation of a write request if its source buffer page is found to be modified since the request's log disk write operation. This cancellation ensures that the write request's associated log disk track remains "used," so that the recovery process can put it back to the data disk if necessary. This also means that when a buffer page is successfully written to the data disk one or multiple log disk tracks that share the same source buffer page may be reclaimed simultaneously. In case of crashes, log records can be replayed and bring the data disk to a consistent state.

4.3. Disk Read

A disk read request is handed to the data disks directly if it cannot be serviced from the *Trail* driver's buffer memory. The log disk itself never services disk read requests. This is unlike write requests, which are serviced through the log disk and the data disks. To reduce the disk read response time, *Trail* gives higher priority to data disk reads than data disk writes when scheduling the requests. Given *Trail* is deigned to optimize for disk writes' performance, file systems that are built on *Trail* potentially can optimize the layout of the data disks for file reads. Consequently, the file read performance for the combination of *Trail* and a read-optimized file system is expected to be better than LFS.

4.4. Other Design Issues

The fast disk write performance advantage of *Trail* cannot sustain when the entire log disk runs out of free track. Such an event is rare because of the large number of tracks in modern disks and huge capacity of them. For example, a 15.3-GByte Western Digital Caviar IDE disk has more than 100,000 physical tracks. Assuming an average number of sectors per track is 550, 512 Bytes per sector, and 30% track utilization, *Trail* can provide more than 8 GBytes of space to buffer synchronous writes.

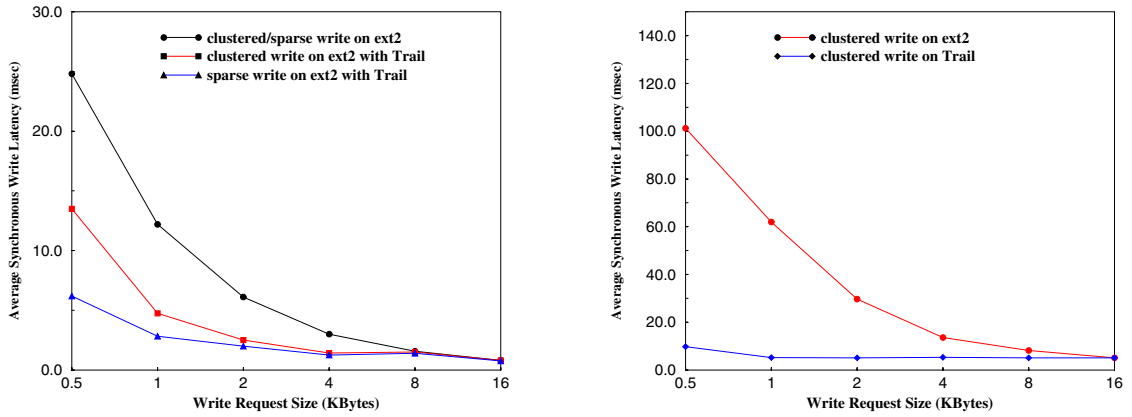
5. Performance Measurements and Analysis

Unless stated otherwise, all the following measurements are performed on a 300-MHz Pentium II machine with 320 MBytes memory, a Seagate 5400-RPM ST41601N SCSI disk (1.37 GBytes, 1.7-msec track-to-track seek time) as the trail log disk, three 5400-RPM Western Digital IDE disks (10 GBytes, 2-msec track-to-track seek time) as the normal data disks, and an NCR53c406a SCSI controller. All disk-level latency measurements are made inside the kernel and immediately above the SCSI driver. Therefore no file system-related processing cost is included. The TPC-C measurements, on the other hand, are made at user level and thus represent end-to-end performance results.

5.1. Synchronous Disk Write Latency

To compare the synchronous write performance of *Trail* and standard disk subsystem, we used a user-level process that sends a sequence of synchronous write requests with random target locations. In the *clustered* mode, a new synchronous write request arrives at the disk request queue immediately after the log disk write for the previous one completes. In the *sparse* mode, a new synchronous write request arrives T after the log disk write step of the previous one completes, where T is larger than the *repositioning overhead*, which is the time taken to move the disk head to the next free track after a log disk write is completed and its typical value is 1.5 msec.

Figure 3 shows the average synchronous write latency of *Trail* and standard disk subsystem for various write request sizes and multi-programming levels in the sparse and clustered mode. *Trail* is up to 11.85 times faster than standard disk subsystem in synchronous write performance. Under *Trail*, synchronous write requests in the clustered mode take longer to complete than those in the sparse mode because the track-to-track switch overhead is more likely to be visible to clustered writes but can be effectively masked in the case of sparse writes. The performance of standard disk subsystem is the same under sparse and clustered writes. When the same sparse-mode and clustered-mode synchronous write workloads are exercised on the Linux's disk subsystem, the synchronous write latency grows drastically, mainly because of the seek and rotational delay. Queuing delay is close to zero in this setup where only one process is sending requests, as shown in Figure 3(a). However, when multiple processes are writing to disks and



(a) (b)

Figure 3. *Average synchronous 1-KByte write latency of* Trail *and Linux's disk subsystem for* sparse *and* clustered *workloads with one process (a) and five processes (b).*

queuing delay is long , as in Figure 3(b), *Trail* can serve writes much faster and thus significantly reduces the queuing delay. Consequently, with multiple processes, the performance difference between *Trail* and standard disk subsystem is actually greater in the clustered mode, which suggests that *Trail* can weather more stressing workloads than standard disk subsystem. As the write size increases, the performance advantage of *Trail* over standard disk subsystem decreases because seek and rotational overhead, which *Trail* aims to eliminate, becomes less significant compared to data transfer delay.

Under *Trail*, the latency of a synchronous disk write contains the log disk write time and the repositioning overhead, if the latter cannot be masked. Measurements from the *Trail* prototype show that each log disk write always experiences fixed disk controller and on-disk processing overhead. For example, the data transfer delay for a single 512-byte sector on a ST41601N disk is about 0.13 msec, and yet the synchronous write latency for a one-sector write request is consistently around 1.40 msec. This extra 1.3 msec overhead is hardware-related and is not under the control of software. When these miscellaneous delays are taken into account, we estimate that *Trail* has reduced the average rotational latency of a synchronous disk write to below 0.5 msec, which is an order of magnitude smaller than the average rotational delay of our testing trail disk, 5.5 msec.

The average reposition or track-to-track switch overhead is about 1.5 msec, which means that *Trail* can complete a one-sector synchronous disk write within 3.0 msec, or 333 writes per second. However, *Trail* supports the *batched write* optimization, which significantly improves the synchronous disk write throughput without adversely affecting latency. Batching aggregates a sequence of logically separate synchronous write requests, which are already in the disk request queue at the time of initiating the next log disk write, into one physical disk write to the log disk. More specifically, after a log disk write operation is completed, the interrupt handler for disk access completion checks if the disk request queue is empty. If yes, *Trail* moves the disk head to the closest sector of the next track by issuing a read request; otherwise, *Trail* attempts to service as many requests in the queue as the free space left on the current track can accommodate. When the current track is full or reaches some utilization threshold, *Trail* also moves the disk head to the next track by issuing a read request explicitly.

Batch Size	1	2	4	8	16	32
Elapsed Time (msec)	129.9	69.6	33.1	17.7	10.9	8.4

Table 1. *The total elapsed time for servicing a sequence of 32 one-sector writes for varying write batch size. As the batch size increases, the number of physical SCSI commands required decreases and so does the total elapsed time.*

Table 1 shows the total elapse time for servicing a sequence of 32 one-sector synchronous write requests when the batch size varies from 1, 2, ... to 32. Because each physical write operation incurs a repositioning delay as well as a write-after-write command delay, as the batch size increases, the number of distinct physical write operations de-

Storage System	EXT2+Trail	EXT2	EXT2+GC
Average Response Time (sec)	0.059	0.097	0.90
Disk I/O Time for Logging (sec)	17.6	30.4	28.8
Throughput (tpmC)	1004	616	663

Table 2. *Performance numbers for performing a sequence of 5000 transactions when the degree of concurrency is 1. Warehouse parameter $w = 1$. Log buffer size is 50 KBytes.*

Log Buffer Size (KBytes)	4	100	400	800	1200
Number of Group Commits	10960	448	113	57	39

Table 3. *Total number of group commits and thus synchronous disk writes in the 10,000-transaction run under different log buffer size. The degree of concurrency is 4. $w = 1$.*

creases and so does the total elapse time required to complete the given sequence. There is a factor of 15 difference between the two extreme cases. In practice, as the size of user-level file write requests increases, it is more likely that low-level disk write requests sent to the *Trail* driver are more clustered, because the file system tends to split a large user-level file access request into multiple consecutive small low-level write requests. Therefore the batched write optimization is triggered more frequently and thus is more useful as user-level file write request size increases. As a final optimization, it is possible to employ multiple log disks to completely hide the disk re-positioning overhead from user applications.

5.2. Database Logging Performance

A major performance claim of *Trail* is that it could the synchronous write latency to only data transfer delay plus a fixed command processing overhead. Consequently, under *Trail* the amount of time required to service multiple disk writes one by one should be comparable to that required for servicing the same set of disk writes in one batch. High-performance database systems typically employ *group commit* to reduce the performance impacts of synchronous writes used to flush log records to disk at transaction commit time. Group commit improves database logging performance by batching the disk writes from multiple transactions, but may compromise data durability also because of delayed commit. *Trail* solves the performance problem associated with group commit, and achieves better performance than synchronous logging on traditional storage system without undermining the durability property of database transactions.

To measure the performance improvements from *Trail* on database applications, we ported on top of *Trail* a transaction processing engine called the Berkeley DB package [1, 14]. The database log file is opened with the `O_SYNC` flag, so that each write to the database log will be a synchronous one. TPC-C benchmark [17], which is representative of high-volume transaction processing workload, is chosen as our studying case. The system under test is set up as following: one IDE disk is dedicated to database logging files, other two disks contain database tables. To bypass the cold-start phase, we first used a set of transactions to warm up the database buffer cache, which is set to 300 MBytes. The number of warm-up transactions is 200,000 in all the following measurements. In all cases the disk I/Os occur in bursts since the CPU time each transaction requires is much smaller than the disk I/O delay due to database logging. The warehouse parameter in the TPC-C benchmark is 1 (i.e. the initial database size is over a half GByte and growing rapidly to over 1 GByte), and the log buffer size is 50 KBytes unless specified otherwise. To simulate group commit behavior, which is not implemented yet in Berkeley DB package, we used a fixed log buffer size as the criterion to decide when to flush database records to disk synchronously. That is, log records in the log buffer are forced to disk once the size of the log records exceeds the chosen log buffer size. The larger the log buffer is, the more aggressive group commit is in batching disk writes.

Table 2 shows the comparison in transaction throughput (tpmC[1]), average response time, and log disk I/O time for servicing a contiguous sequence of 5000 transactions among three design alternatives: running Berkeley DB on *Trail* without group commit, running Berkeley DB on a Linux disk subsystem with group commit, and running Berkeley DB on a Linux disk subsystem without group commit, for various concurrency levels. When concurrency is 1, the effects of lock contention and disk queuing are removed. Running Berkeley DB on *Trail* (`EXT2+Trail`) is 1.51 times faster than running Berkeley DB on standard disk subsystem with group commit (`EXT2+GC`), which in turn is only 1.08 times faster than running Berkeley DB on standard disk subsystem without group commit (`EXT2`). `EXT2+GC` is only slightly better than `EXT2`, because the overall logging I/O time reduction due to group commit is not significant. When group commit forces log records to disk, the associated disk I/O time is exceedingly long and tends to slow down all the transactions that are active at the time, thus further lengthening the disk queuing delay and potentially lock contention delay of active transac-

[1] tpmC is the standard metric used in TPC-C benchmark, meaning transaction per minute.

tions. Therefore, although group commit amortizes fixed disk access overhead over a number of disk writes, it also clusters disk access activities that could have been more evenly spread out over time and thus put less stress on the disk subsystem. This "I/O clustering" effect explains why `EXT2+GC` is worse than `EXT2+Trail` and why it suffers a slightly higher transaction abortion rate. In terms of average response time, `EXT2+GC` is also worse than `EXT2+Trail` as each transaction has to delay its commit time to the point when a batch of transactions complete.

Table 3 gives the total number of group commits (synchronous disk write operations) for each 10,000-transaction run when group commit is enabled. The performance improvement due to write batching in group commit is significant when the request size is relatively small, i.e., less than 10 KBytes, as rotational and seek delay contribute a large portion to the overall disk operation cost. When the log buffer size is larger than 50 KBytes, the disk I/O time for logging and the transaction throughput do not change much with the increase of the group commit batching size (i.e. log buffer size) as rotational and seek delay takes a less significant percentage in I/O cost.

One concern about *Trail*'s write policy is the log disk's space utilization efficiency. Assume in the following that *Trail* performs exactly one batched write to each track in the log disk. For the same TPC-C experiment, when the transaction concurrency is 4, the per-track space utilization of *Trail*'s log disk is 12%. The same per-track space utilization is increased to 21% when the concurrency is 8, and to over 30% when the concurrency is 12. These results show that the batched write optimization alone can achieve reasonably good disk space utilization in the presence of bursty disk writes, and the additional optimization of allowing multiple batched writes per track is rarely needed in practice.

5.3. Data Recovery Overhead

Trail's recovery procedure consists of three steps: locating the youngest active write record on the log disk, rebuilding the active write records at the time of a system crash and writing them back to the data disk. Figure 4(a) shows the breakdown of the recovery overhead among these three steps, when the number of pending disk requests (Q) varies from 32 to 256. We used a binary search algorithm to locate the youngest write record. This step takes $O(\lg N)$ time to scan the tracks, where N is the total number of tracks on the log disk. For example, a total of 35,717 tracks are in our testing disk, the number of tracks that are scanned is about 20 on average. In terms of absolute time, it takes 450 msec to complete the first step for a 5400 RPM disk. In the second step, *Trail* follows each active write record's back pointer `prev_sect` to reach its previous record in the log disk until either all pending records have been found. In the third stage, *Trail* writes these active records back to the data disk and completes the recovery process. Note that to shorten the recovery delay, *Trail* can choose to skip the third stage and resume normal processing immediately after the second stage is completed. This does not compromise the data integrity since the invariant that there is a persistent copy of every written data block still holds even when the third step of the recovery process is bypassed. Figure 4(b) compares the recovery performance when the write-back step is included or not included in the recovery procedure. Because the write-back step involves random accesses to the data disk, bypassing the write-back step can have a significant performance impact. When Q is 256, the recovery process is more than 3.5 times slower if the write-back step is included than if it is bypassed.

6. Conclusion

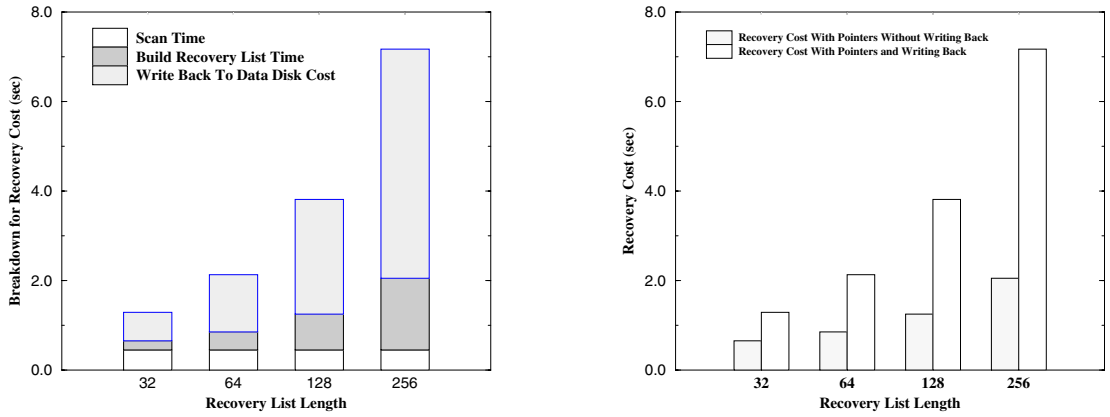
Synchronous disk write performance is crucial to high performance fault tolerance systems in general and transaction processing systems in particular. In this paper, we propose a *track-based logging* architecture and describe the design, implementation, and evaluation of an application of this architecture to a novel disk system called *Trail*, which features the lowest synchronous disk write latency ever reported in the literature, 1.5 msec. In addition, the fact that the *Trail* prototype takes less than 3,500 lines of C code in total demonstrates that the implementation complexity of track-based logging is relatively modest.

The two key enabling technologies for track-based logging are a software-only disk head position prediction algorithm and a novel self-describing logging format that allows recognition of special bit patterns without using requiring exotic bit encoding. The measurements from the prototype demonstrate that disk head position prediction scheme is highly accurate and can be effectively utilized to reduce the rotational latency associated with synchronous disk writes.

Armed with the success of *Trail*, we are currently exploring other applications of track-based logging, such as applying track-based logging directly to database logging rather than indirectly through the file system, and using track-based logging to solve the small write problem in RAID-5 disk arrays.

Acknowledgement

This research is supported by NSF awards MIP-9710622, IRI-9711635, EIA-9818342, ANI-9814934, and ACI-9907485, USENIX student research grants, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., Computer Associates/Cheyenne Inc., Na-



(a) (b)

Figure 4. *(a) The breakdown of data recovery delay in* Trail *when the number of active write records varies. (b) Performance comparison between data recovery procedures that write or do not write the active write records back to the data disk at the time of recovery.*

tional Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

References

[1] Berkeley db package. `Http://www.sleepycat.com`.

[2] CHIUEH, T. Trail: a track-based logging disk architecture for zero-overhead writes. In *Proceedings of 1993 IEEE International Conference on Computer Design ICCD'93*, Cambridge, MA, USA 3-6 Oct. 1993, pp. 339–43.

[3] CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEVERETT, B. W., MASON, W. A., AND SIDEBOTHAM, R. N. The Episode file system. In *Proceedings of the Winter 1992 USENIX Conference* (San Francisco, CA, Winter 1992), pp. 43–60.

[4] ELHARDT, K., BAYER, R. A database cache for high performance and fast restart in database systems. In *ACM Transactions on Database Systems* (Dec 1984), vol. 9, pp. 503–25.

[5] GANGER, G. R., PATT, Y. N. Metadata update performance in file systems. In *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI)* (Monterey, CA, USA Nov. 1994), pp. 49–60.

[6] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (Austin, TX, Nov. 1987), pp. 155–162. In *ACM Operating Systems Review 21*:5.

[7] HAGMANN, R. Low latency logging. Tech. Rep. CSL-91-1, Xerox Corporation. Palo Alto, CA, February 1991.

[8] HU, Y., YANG, Q. DCD–disk caching disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 169–78. Philadelphia, PA, May 1996.

[9] JACOBSON, D. AND WILKES, J. Disk scheduling algorithms based on rotational position. Tech. Rep. HPL-CSP-91-7, Hewlett-Packard Lab., Palo Alto, CA, Feb, 1991.

[10] PC GUIDE. Wear Leveling. `Http://www.pcguide.com/ref/hdd/perf/qual/featuresLeveling-c.html`.

[11] ROSENBLUM, M., OUSTERHOUT, J.K. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems* (Feb 1992), vol. 10, pp. 26–52.

[12] SELTZER, M. I., GANGER, G. R., MCKUSICK, M. K., SMITH, K. A., SOULES, C. A. N., AND STEIN, C. A. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)* (San Diego, CA, June 2000), pp. 71–84.

[13] SELTZER, M., CHEN, P. AND OUSTERHOUT, J. Disk Scheduling Revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pp. 313–24. Washington, DC, USA 22-26 Jan. 1990.

[14] SELTZER, M., OLSON, M. LIBTP: portable, modular transactions for Unix. In *Proceedings of the 1992 USENIX Conference*, pp. 9–25. San Francisco, CA, USA Jan. 1992.

[15] STRICKLAND, J., UHROWCZIK, P., WATTS, V. IMS/VS: an evolving system. In *IBM Systems Journal* (1982), vol. 21, pp. 490–510.

[16] SWARTZ, K. The brave little toaster meets usenet. In *LISA'96* (Chicago, IL, Oct 1996), pp. 161–170.

[17] TRANSACTION PROCESSING PERFORAMNCE COUNCIL. TPC-C Benchmark. `Http://www.tpc.org`.

[18] WANG, R. Y., ANDERSON, T. E., PATTERSON, D. A. Virtual log based file systems for a programmable disk. In *Proc. Third Symposium on Operating Systems Design and Implementation* (February 1999).

[19] WORTHINGTON, B., GANGER, G. R., PATT Y. N., WILKES J. Online extraction of scsi disk drive parameters. In *Performance Evaluation Review* (May 1995), pp. vol.23, no.1, p. 146–56.

[20] YU, X., GUM, B., CHEN, Y., WANG, R. Y., LI, K., KRISHNAMURTHY, A., AND ANDERSON, T. E. Trading Capacity for Performance in a Disk Array. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (October 2000).