Okeanos: Fast and Reliable Stream Storage Through Differential Data Journaling

Andromachi Hatzieleftheriou   Stergios V. Anastasiadis
Department of Computer Science
University of Ioannina, GREECE

Technical Report DCS 2008-08
November 10, 2008

Abstract

Real-time storage of massive stream data is emerging as a critical component in modern computing infrastructures used for continuous monitoring purposes. Traditional file and database systems are not designed for such operation environments and incur excessive resource requirements when handling high-volume streaming traffic. In the present paper, we examine the possibility of employing data journaling in order to combine sequential throughput with low latency during synchronous writes. Experimentally we demonstrate that low-rate streams incur remarkably high data journaling traffic in a commonly-used production file system. In order to alleviate the problem, we designed and implemented differential data journaling in the kernel of the above system. Through extensive experimentation we show substantial reduction in the required disk throughput combined with very low write latency.

Categories and Subject Descriptors D.4.3 [*Operating Systems*]: File Systems Management

General Terms Design, Performance, Experimentation, Reliability

Keywords Journaled File System, Linux, Ext3, Small Write Performance Improvement, Streaming

1. Introduction

Continuous monitoring processes are prevalent today for a wide range of purposes such as network administration, autonomic systems management and physical site safety. Such important applications make stream-oriented functionality highly relevant in modern computing infrastructures. For instance, recently proposed stream management engines demonstrate the feasibility of flexibly applying time-series operators on high-rate streams (Balakrishnan et al. 2004; Iannaccone et al. 2004). Existing stream processing environments store stream data either temporarily before applying real-time operators within time windows (Carney et al. 2002), or permanently in order to support retrospective query processing (Desnoyers and Shenoy 2007).

Prior research has made the case that traditional data management approaches, such as relational databases and general-purpose file systems, are not engineered to efficiently store continuous stream data that are automatically generated from sensors in real time (Carney et al. 2002; Desnoyers and Shenoy 2007). For instance, sensors may generate high-resolution video and audio streams at high rates (Esteve and Palau 2006), or send intermittent variations of environmental conditions at much lower rates (Long et al. 1995). A monitoring system receives messages from high-volume links or large numbers of sensors and stores the received data for a time period that depends on whether the applied processing occurs in real time or retroactively.

Across all types of heterogeneous streams, it would be desirable to store the received data reliably on the same facility without compromising the sequential playback performance required for statistical processing or effective visualization. Thus, a stream storage facility could serve as a building block for a variety of applications in the entire range from network packet processing to urban traffic control or environmental monitoring with the appropriate indexing functionality built separately at a higher level, when support for query processing is required.

Existing general-purpose file systems use journaling in order to synchronously move data or metadata from memory to disk in a sequential manner. Thus they postpone the more costly transfer of data or metadata to the final disk location without penalizing the write latency perceived by the application user. Indeed, previous research has used trace-based emulation to experimentally demonstrate that data journaling can serve random writes with high sequential throughput, but actually makes throughput lower at high data volumes due to the extra disk traffic generated (Prabhakaran et al. 2005). The study made the reasonable conclusion that

data journaling should only be enabled with random writes, but disabled with large sequential writes. Instead, we focus on the efficient and reliable storage of multiple concurrent streams whose aggregate workload demonstrates random-access behavior even though appends corresponding to individual streams may be perfectly sequential. To a large extent, in such environments it remains unclear what is the most appropriate way to handle the incoming data.

In the present paper, we investigate the performance characteristics of data journaling in the context of synchronous writes that would be required among several situations including the reliable storage of incoming streaming data. In order to lower the cost of data journaling we implemented a differential version of it in the default file system of a widely used operating system. We find that depending on the rate of the streams we can reduce up to several factors the required journaling throughput by only journaling the bytes actually written rather than the entire blocks that contain them. As a side-effect of the sequential writes to the journaling device, we also reduce substantially the response time of synchronous writes. Thus, we can use data journaling to reduce the latency of writes at a reduced cost of required disk throughput.

In the rest of the paper, we first describe the existing journaling techniques in Section 2. Then, in Section 3 we present one existing data journaling method that is widely available, and in Section 4 we introduce the differential data journaling technique that we designed and implemented for a commonly used operating system. In Section 5, we explain the experimentation environment that we used in our study, while in Section 6 we present our measurements across different workloads. In Section 7 we summarize previous related work, while in Section 8 we outline our conclusions.

2. Background

In general, file system operations are either data operations that update user data, or metadata operations that modify the structure of the file system itself. Consistent recovery of the metadata after a crash, due to operating system failure or power outage for instance, requires the system metadata to be written on disk in a specific order. The system can achieve consistency simply by updating the system metadata synchronously. Similarly one can recover recently written data after a crash by writing them synchronously to disk. Synchronous data writes are typically applied in database systems that store critical data (Wang et al. 1999; Chen et al. 1996). In the rest of the present section, we describe techniques that have been previously proposed to achieve high performance in file systems during data and metadata updates.

2.1 Log-structured file systems

A log-structured file system writes all data and metadata modifications into a log (Rosenblum and Ousterhout 1992). The log is the only structure on disk and consists of segments that facilitate the removal of deleted areas. This approach takes advantage of sequential writes to improve the update performance. Periodically, the system writes the complete and consistent file structures safely at a fixed position of the log called checkpoint region. After a crash, the file system uses the checkpoint for its initialization and the recent portion of the log to quickly recover recently written data.

A modified version of the log-structured file system has been recently used for the storage of high-volume streams (Desnoyers and Shenoy 2007). StreamFS has incoming stream data written to a frontier that moves in a circular fashion along the disk space and selectively overwrites the expired data. However, StreamFS has been specifically designed for high-rate streams typically generated in network monitoring systems; it is unclear how it would behave in heterogeneous environments where high-rate and low-rate streams co-exist. Additionally, an aggregate high-rate stream typically contains a large volume of information that makes necessary to build an index structure online during data storage and scan entire segments of the stored data during retrospective query processing. Instead, demultiplexing of the incoming data into separate files according to some criterion would possibly facilitate and reduce the load of the subsequent selective retrieval and processing.

2.2 Soft updates

Soft updates is a mechanism that delays writes of metadata and explicitly maintains dependency information to specify the order in which data must be written to disk. Thus, it eliminates the need for a log or most synchronous writes related to metadata. The system maintains for each disk block a list of all the metadata dependencies associated with the block. When the system selects to write a block which requires other blocks to be written first, the system rolls back the affected parts of the selected block to their earlier state. After the write has completed, the system deletes all the fulfilled dependencies and restores the block to its current value. Thus, applications see the most recent version of the metadata blocks and the system keeps disk contents consistent.

The method improves system performance because it aggregates multiple metadata updates into a reduced number of disk writes and postpones time-consuming operations, such as deletes, to a background process. After system crashes the system can be mounted and used immediately, since the only remaining inconsistencies are non-fatal errors that can be corrected in the background during normal operation.

2.3 Journaling file systems

Journaling file systems use an auxiliary log to record all *metadata* operations. Additionally, some implementations also support logging of *data* modifications. The log is maintained as a preallocated file in the same file system or as a standalone separate file system. Write-ahead logging

guarantees that the log is updated on the disk before the pages corresponding to the modified blocks. Thus, the system performs additional disk operations, which are efficient since they are sequential. Batching of log writes originating from different concurrent applications provides additional throughput improvements.

File system journaling allows synchronous writes to complete faster, because they return as soon as the sequential log update completes. Thus, costly disk operations at the final locations of the modified blocks can be deferred and completed periodically and asynchronously. Journaling of file data helps further in that direction, but incurs significant extra throughput on the journaling device. The cost of data journaling can be high for large writes due to the significant volume of data sent to the log. Unfortunately, current implementations incur considerable logging activity even with small writes. In order to simplify the implementation, they log the entire blocks being modified rather than just their modified part. However, journaling reduces write latency in both small and large writes, since it allows the synchronous log updates to be completed sequentially.

2.4 Summary

In summary, current file systems mostly care to maintain their integrity across crashes without compromising their performance. They achieve this goal by flushing *metadata* updates at sequential disk throughput or by avoiding the violation of the dependencies across the block updates. Existing techniques that complete the *data* updates synchronously require significant extra throughput in order to achieve that at relatively low latency. In the rest of the paper, we demonstrate that it is possible to reduce substantially the throughput overhead of synchronous data writes while maintaining low latencies, as well.

3. Journaling in the Ext3 File System

As disk capacities grow faster than disk access speeds over time, modern file systems use journaling to support fast recovery after a crash (Tweedie 1998; Galli 2001; Bovet and Cesati 2005; Prabhakaran et al. 2005). Journaling reduces possible downtime of several hours to a few seconds by considering the most recent disk writes rather than making consistency checks over the entire capacity of the file system. Ext3 implements journaling by performing each high-level change to the file system in two steps. First, it copies the modified blocks into the journal. Second, it transfers the modified blocks into their final disk location. When the file system update completes, ext3 discards the copies of the blocks in the journal.

3.1 Journal

Ext3 handles the journal through a special kernel layer called *journaling block device*. It implements the journal as either a hidden file within the root directory of the file system or a separate disk partition. Each *log record* in the journal corresponds to one low-level operation in the file system that updates one disk block. The journal represents with a log record the entire modified block of the file system rather than the range of block bytes actually modified. Thus, the journal is wasteful in terms of disk throughput and space, but simple in terms of processing complexity because it uses the buffers of the modified blocks directly. Additionally, each log record is associated with auxiliary information that contains the number of the corresponding block in the file system and several status flags.

3.2 Transactions

Each high-level operation of the file system (e.g. a system call) consists of multiple low-level operations that manipulate disk data structures. The *atomic operation handle* refers to a set of low-level operations. When the system recovers from a failure, it ensures that either an atomic operation handle occurs completely or all the low-level operations of the handle are discarded. In order to improve efficiency, the system groups into a *single transaction* the records of multiple atomic operation handles. All the log records of a handle belong to one transaction. After its creation, the transaction accepts log records of new handles for a fixed period of time. The system stores all the log records of a transaction consecutively on the journal. After the log records have been committed to the file system, the system reclaims all the blocks of the transaction.

A transaction is considered *complete* (equivalently in state **T_FINISHED**), if all its log records are fully residing in the journal including the commit block. It is *incomplete*, if at least one log record of the transaction is not in the journal. An incomplete transaction can be in one of the following states

T_RUNNING It still accepts new atomic operation handles.

T_LOCKED It does not accept new handles, but waits for the accepted handles to finish.

T_FLUSH The accepted handles are still in the process of writing log records to the journal.

T_COMMIT All the log records have been written to the journal except for the commit block of the transaction.

When recovering from a failure, the system skips all incomplete transactions and transfers the blocks of the complete transactions to the file system.

If the file system is mounted in *journal* mode, then both data and metadata blocks are copied to the journal, before they update the file system. This mode minimizes the chance of losing file updates, but incurs additional disk accesses. In the rest of the present document, we prefer to use the term *data journaling* when we refer to the journal mode in order to stress out the fact that it journals data in addition to metadata. The *ordered* mode only copies the metadata blocks to the journal. Additionally, metadata blocks update the file

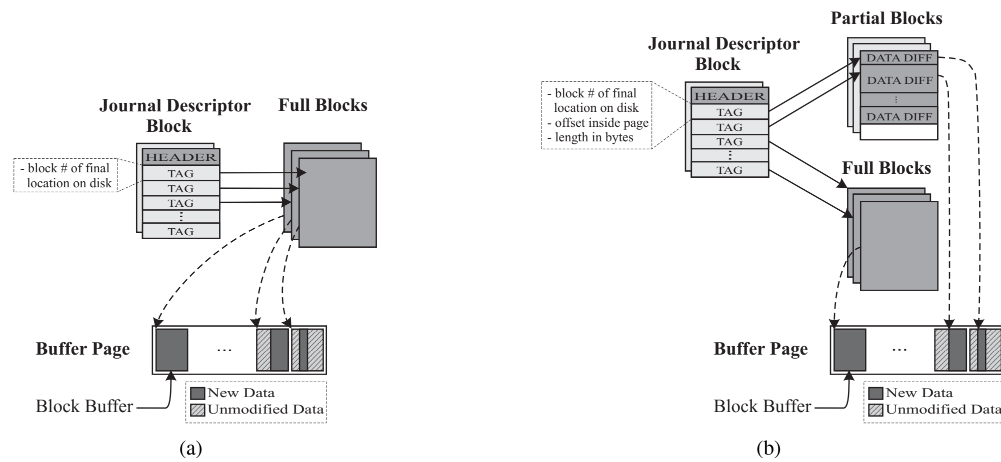

Figure 1. (a). In the original design of the ext3 data journaling, there is a full block for each write operation. (b) In the differential data journaling that we introduce, we use partial blocks to accumulate the data modifications from multiple writes. Thus, we can reduce considerably the traffic to the journaling device when the write operations only modify part of a block.

system after the associated data blocks, which reduces the risk of corrupting data inside a file. Finally, in *writeback* mode, only metadata blocks are copied to the journal, and there are no requirements in the order at which data and metadata blocks update the file system. It is considered the fastest method, but also the weakest in terms of consistency.

Figure 2 depicts the difference in the amount of traffic sent to the journal device across the three mount options of ext3. It is noticeable that data journaling incurs only slightly varying traffic for write requests smaller than 4KB. Essentially, data journaling sends a large amount of traffic to the journal for small writes regardless of the actual size of the write requests.

3.3 Buffers

The Linux kernel uses the *page cache* to temporarily keep page copies from recently accessed disk files in memory. A *block buffer* is the buffer of an individual disk block in memory. A *buffer head* descriptor specifies for each *block buffer* the necessary handling information required by the kernel. Generally, the page cache does not allocate the block buffers individually, but in units of pages called *buffer pages*. The kernel addresses individual blocks using the buffer heads pointed to by the corresponding buffer page. A number of *pdflush* kernel threads flush dirty pages to their final location on disk through two separate mechanisms:

- Systematically scan the page cache every *writeback period*.
- Implement a timeout mechanism on each page according to a configurable *expiration period*.

A user can also use the *fsync* system call to synchronously flush all the data and metadata dirty buffers of the specified file descriptor to disk. Actually, fsync moves the blocks to the journal or the final disk location depending on the mount mode.

3.4 Commit

Each invocation of the *write* system call creates a new atomic operation handle that is added to the current active transaction. When the transaction moves to commit state, the kernel acquires a *journal descriptor block*. This block contains tags that map block buffers to their final location on disk of the file system. When a journal descriptor block fills up with tags, the kernel moves it to the journal together with the corresponding block buffers. The kernel allocates additional journal descriptor blocks as needed for each transaction.

For each block buffer that will be journaled, the kernel allocates a separate buffer head specifically for the I/O needs of journaling (Figure 1(a)). Additionally, the kernel creates an auxiliary structure called *journal head* that associates the block buffer with the respective transaction (Figure 2). In general, the buffer head of a journaled block buffer points to the original copy of the block buffer. However, if this block buffer is going to be used concurrently by another transaction, then the kernel creates in memory a new copy of the block buffer for the journal I/O transfer needs. When all the log records of a transaction have been safely written to the journal, the system allocates and synchronously writes to the journal a final commit block that states the transaction has committed successfully.

3.5 Recovery

The *transaction committing* completes when a transaction has flushed all its records to the journal and has been marked as finished. This is done for each running transaction within a specified time period by the *kjournald* kernel thread. Subsequently, the *transaction checkpointing* completes when all the blocks of a committed transaction have been moved to

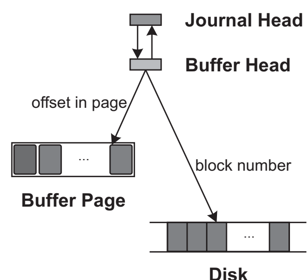

Figure 2. For each journal block buffer there is (i) a buffer head that specifies the respective block number in the journal and, (ii) a journal head that points to the corresponding transaction.

their final location on disk and the corresponding transaction records are removed from the journal.

If the system finds log records in the journal after a crash, it assumes that the unmount was unsuccessful and initiates a recovery procedure in three phases.

PASS_SCAN In the first phase, it finds the last record of the journal. The system only considers committed transactions for replaying.

PASS_REVOKE During the second phase, the kernel builds a hash table from the *revoked* blocks. These are blocks of committed transactions that should not be written to their final disk location, because they are obsoleted by later operations.

PASS_REPLAY In the third phase, the recovery process writes to their final disk location the newest version of all the blocks that occur in committed transactions, and are not present in the hash table of revoked blocks.

If the system crashes again before the recovery finishes, the same journal can be reused to complete the recovery.

4. Differential Data Journaling

Typically, journaled file systems care for the metadata consistency and only log metadata modifications in the journal. Ext3 and ReiserFS are two file systems that additionally support data journaling as an option. Past research investigated the related performance and reported that data journaling improves the throughput of random I/O operations, but incurs much higher disk throughput than metadata journaling (Prabhakaran et al. 2005). Thus, when the journal fills up with log records, data journaling leads to reduced application performance due to the checkpointing delay.

In Figure 3, we verify that data journaling incurs significant journal traffic in comparison to the other two modes of ext3. In the rest of the present section, we describe the approach that we follow in order to keep low the overhead of data journaling and at the same time retain its substantial performance benefits. Even though we consider our approach quite general, in our description we use the previously introduced terminology of ext3, over which we implemented our prototype.

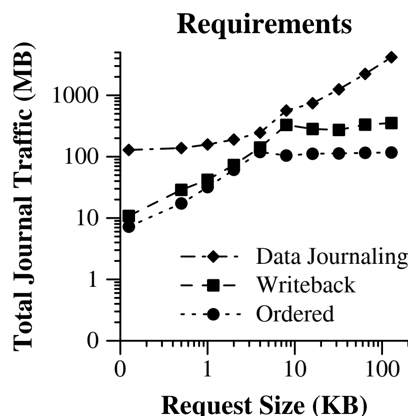

Figure 3. We examine the three mount options of ext3 using periodic writes of varying request sizes. We measure the total write traffic to the journal during a time period of 5 min. The total journal traffic of data journaling is substantially higher in comparison to the other two modes. Additionally, at request sizes lower than 4KB, data journaling incurs traffic that changes sublinearly as a function of the write rate. This is reasonable since data journaling sends to the journal entire blocks rather than only the part that is modified by each write operation.

4.1 Partial blocks

The original journaling process of ext3 transfers a full copy of each modified block buffer from memory to journal. This is true for both data and metadata blocks, when they are journaled according to the mount options of the file system. In the case of small data writes that only modify partially a block buffer, full block journaling can have a multiplier effect in the throughput required by the journal device (Figure 3). The actual waste in journal device throughput depends on the fraction of the block buffer that is left unmodified by each write operation. Practically, we should only send to the journal the modified part of the block. At the uncommon case that a recovery is initiated, we can read the original block from the final location on disk, and write it back after applying the difference that we retrieve from the corresponding journal record.

In order to implement differential data journaling, we introduce a new type of journal block that we use to accumulate the data updates from multiple write operations. We call this block *partial* in order to differentiate it from *full* blocks, which are fully modified by a single write operation (Figure 1). We only use partial blocks to journal *data* rather than *metadata* modifications. Thus, if a write updates part of a data block, we copy the modified part of the block to the current partial block buffer of the transaction. When a partial block has not sufficient space for the current write, we allocate a new partial block and copy there the data of the write. If the write modifies a metadata or a full data block, we use the corresponding full block instead. We might still need to create a copy of the full block in order to freeze the version that we send to the journal, if the block is going to be modified shortly by another transaction.

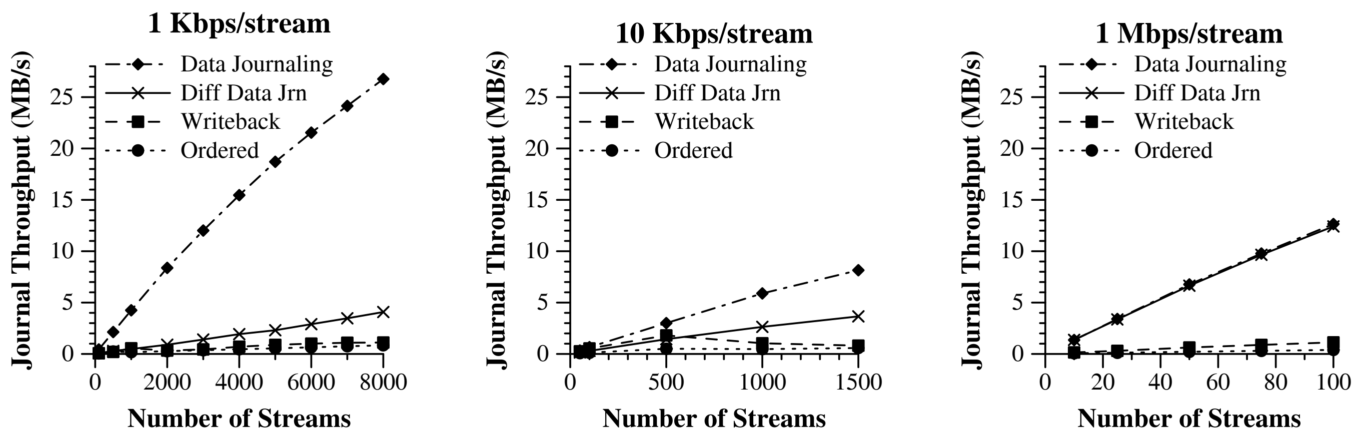

Figure 4. We measure the journal device throughput across different numbers of streams and rates of 1Kbps, 10Kbps and 1Mbps. At low rates, the journal throughput of differential data journaling is close to that of ordered and writeback modes, unlike the default data journaling mode which is several factors higher. Therefore, we can reliably store the data of low-rate streams without excessive journaling cost. However, at high rates, differential data journaling overlaps with the default data journaling mode in terms of journaling throughput.

4.2 Journal heads

As we already explained, for each journal block buffer there is a corresponding journal head that associates the block with a transaction. The journal head points to a buffer head that links the buffer to a buffer page and other information required for the transfer to the journal device. For writes that only modify part of a block, we expanded the journal head with two extra fields, the offset and the length, respectively, of the partial block pointed to by the buffer head. As we see below, we make use of the journal head in order to prepare the blocks that we actually send to the journal.

4.3 Tags

When we start a new transaction, we allocate a buffer for the journal descriptor block. In data journaling, the transaction logs both data and metadata modifications. The journal descriptor block contains a list of fixed-length tags, where each tag corresponds to one write. Originally, each tag contains two fields:

- The final disk location of the modified block.
- Four flags for journal-specific properties of the block.

In our design, we introduce three new fields in each tag:

- A flag to indicate the use of a partial block.
- The length of the write in the partial block.
- The starting offset in the data block of the final disk location.

When the tags fill up a journal descriptor block, we write the descriptor block and all the corresponding data and metadata blocks to the journal. We allocate additional journal descriptor blocks as required by the transaction.

4.4 Recovery

During the recovery process, we retrieve the data modifications from the journal and subsequently apply them to the blocks corresponding to the final disk location. Since the data of consecutive writes are placed next to each other in the partial block, we can deduce their corresponding starting offsets from the length field in the tags. When the length field of a tag exceeds the end of the current partial block, then we read the next block from the journal and treat it as another partial block from the same transaction.

We use the starting offset tag field to read into a kernel buffer the disk block that we will modify. However, if the partial block flag is not set, then we read the next block of the journal and treat it as a metadata or full data block. Obviously, we write a full block directly to the final disk location without reading first the previous version from the disk.

5. Experimentation Environment

We implemented the differential data journaling in the Linux kernel version 2.6.18. As we already described, we modified the transaction commit and recovery procedures of ext3 so that we only journal and replay the data differences from partial block writes. We evaluated our prototype implementation using x86-based server nodes running the Debian Linux distribution. For most of the experiments we used nodes with a quad-core 2.66GHz processor, 2GB RAM, and two SAS 15KRPM disks, each of 300GB storage capacity and 16MB internal buffer. We also did one experiment with 2.33GHz quad-core processor and two SATA 7.5KRPM disks of 250GB and 16MB on-disk cache.

We have the journal and the data partition on the two separate disks except for one case that we explain in our evaluation, and also use the default file system parameters of Linux that set page and block sizes equal to 4KB. In

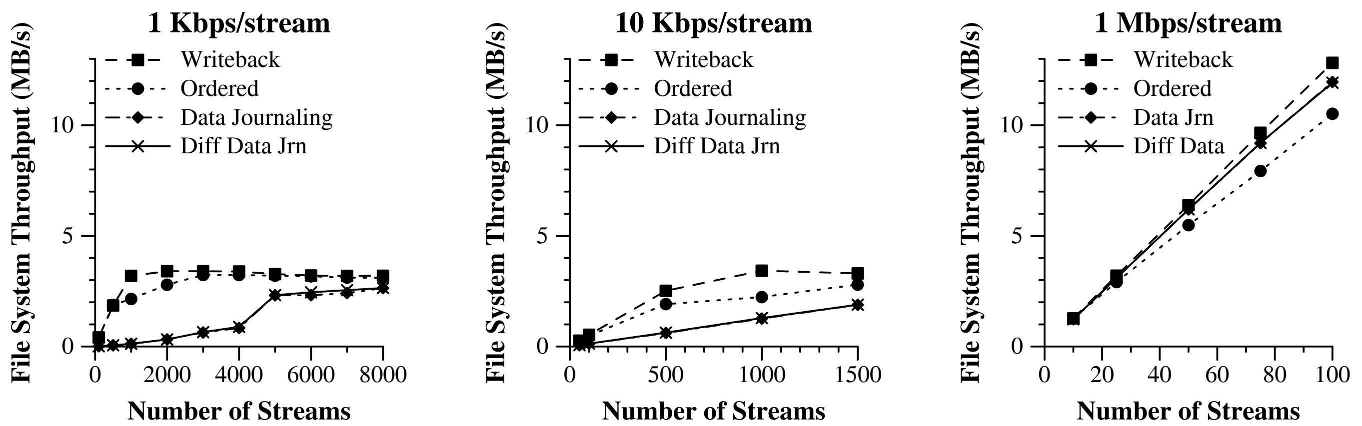

Figure 5. We examine the throughput of the file system device across different numbers of streams and rates. Especially at low rates, the writeback and ordered modes tend to require up to several factors higher throughput than the two data journaling modes. We attribute this benefit of the two data journaling modes to the aggregation of multiple writes that update the same block. Since journaling keeps each update safe on disk, dirty pages can remain for a configurable time period in memory before they are flushed to the file system disk.

our measurements, we assume write operations followed by the fsync call for synchronous completion. We keep the default journal size of 128MB, but manually tune for best performance the writeback period and expiration period of the dirty page flush process.

Past work reports that by default a disk returns a write request when the data reaches the on-disk write cache rather than the media. This behavior makes the system unreliable unless somebody disables the on-disk buffers or uses controllers with battery-backed cache (Nightingale et al. 2006). In most of our experiments, we kept enabled the disk write cache, which essentially emulates devices with battery-backed memory. However, in our evaluation we also experimented with the write caches disabled. As we explain, the disk write cache adds no benefit to streaming workloads but leads to significant performance advantages in traditional applications.

Our prototype implementation of differential data journaling was the working environment of the one coauthor for a period of over two months. The system demonstrated a stable behavior during the entire this period. Additionally, we did extensive performance evaluation to study the characteristics of the system. One benchmark that we use consists of multiple threads periodically writing data at a specific rate. In our evaluation, we examine the disk throughput requirements and the average latency of each write. Additionally, we use the Postmark benchmark to measure performance in an environment of temporary small files that is typical for electronic mail, newsgroups and web-based commerce (Katcher 1997). Thus, we investigate the benefit of data journaling in applications other that streaming.

6. Performance Evaluation

In this section, we study the requirements and performance of our differential data journaling implementation with respect to the ordered, writeback and the default data journaling mode of ext3.

6.1 Streams

In our first set of experiments, we evaluate the benefits and requirements of differential data journaling in a file system, where we store synchronously the incoming data from a large number of concurrent streams. Even though each stream simply appends data to the end of a separate file, the aggregate traffic is random. However, data journaling safely stores data on the journal at sequential throughput and lazily transfers it to the final location at a rate that we can control.

In Figure 4(a), we observe that when the number of streams reaches several thousands, data journaling sends close to 30MB/s of log records to the journal. Instead, differential data journaling keeps the traffic lower than 5MB/s. This behavior is less intense as the stream rate increases from 1Kbps to 10Kbps (Figure 4(b)), and in fact the two data journaling modes overlap for streams of 1Mbps. Thus, differential data journaling accumulates multiple log records into a single block and thus manages to reduce the journaling throughput. Even though a corresponding increase in memcopy activity is likely, this is hardly a problem as we see later in this section.

Additionally, we measure the disk throughput for the update of the final location on the file system (in Figure 5). We notice that the ordered and writeback methods that only journal metadata incur consistently higher throughput to the final disk location, especially at low-rate streams. Instead, the two data journaling modes move the data updates synchronously to the journal, but keep the corresponding data blocks in memory for some time. There, the blocks have the chance to receive the updates from multiple writes, before they are transferred to their final location on disk.

The benefits of the two data journaling modes are even more impressive, when we consider the average latency of

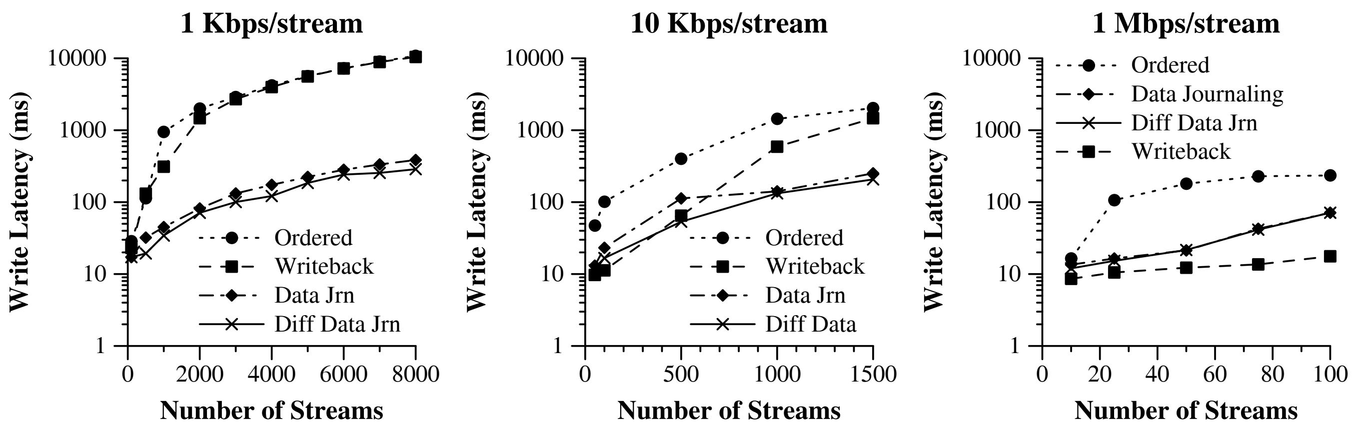

Figure 6. Synchronous writes are usually avoided because they are known to incur high latency in typical file systems. This is true even when the write cache of the disk is enabled. In the above figures, we see the average write latency at different rates and streams. In order to demonstrate the differences across the different modes, we use logarithmic scale at the y axis. As we move from higher to lower rates, the write latency of the ordered and writeback modes appears from several factors up to orders of magnitude higher than those of the two data journaling modes. Thus, the sequential throughput of the journal has a considerable impact to the ability of the system to store safely the incoming data in a short period of time.

the synchronous writes (Figure 6). In Figure 6(a), we see that the ordered and writeback modes incur almost two orders of magnitude higher latency with respect to the other two modes, when serving large numbers of low-rate streams. Thus, a write operation that completes in tens of milliseconds with data journaling, takes as high as 10 seconds with ordered mode. Such high write latency in the ordered mode raises issues about the ability of the system to quickly and safely store incoming measurements. This is crucial, especially at critical time periods before physical catastrophes, when the arriving data matter the most.

Finally, we evaluate the impact of the journaling modes to the total cpu utilization of the system (Figure 7). We see that the system utilization always remains less than 10%. This observation implies that the accumulation of multiple write updates to one block in differential data journaling does not create for memcopy an overhead that is much higher than the other modes. Experimentation with workloads that consist of mixed set of streams with different rates lead to measurements similar to the above. The results of the mixed workload tend to approach respectively the behavior of streams with low or high rate depending on the prevalence of the corresponding type of stream in the workload.

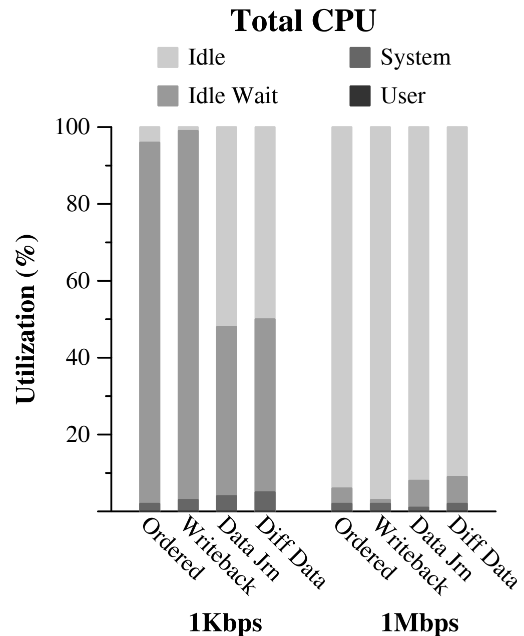

Figure 7. At both low and high rates, we observe that the cpu remains mostly idle, whether doing nothing or waiting for the I/O operations to finish. Therefore, the processing cost of differential data journaling is less than 10%, and comparable to that of the other three mount modes.

6.2 Postmark

Given the very encouraging results that we obtained for workloads with low-rate streams, here we evaluate data journaling with Postmark. This is a benchmark typically used to study the performance of small writes (Hildebrand et al. 2006). We measure the achieved transaction rate with a workload of 10000 transactions over 500 files, and a mix of read, append, create and delete file operations. The actual duration of the experiment varies depending on the efficiency of the requested operations. We run the benchmark in a range of block sizes from 128 bytes to 16KB (Figure 8). Our main observation is that the two data journaling modes perform several factors better from the metadata-only journaling modes. The performance improvement is higher for small blocks. However, even with the block size equal to 16KB, the data journaling modes double the measured transaction rate. Therefore, if somebody uses differential data journaling to keep low the extra journaling throughput, one can improve substantially the performance of applications that need synchronous small writes.

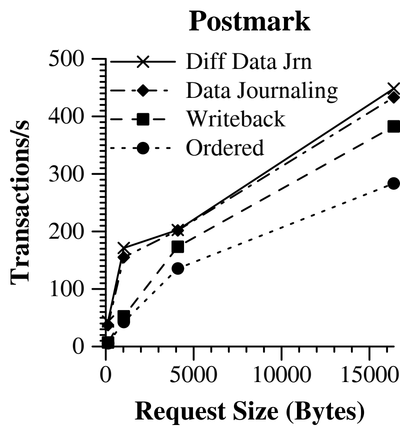

Figure 8. We run Postmark with 100 threads and file ranges from half kilobyte to a hundred kilobyte. The x axis refers to the request size of the read and write operations. We notice that the two data journaling modes almost double the transaction rate with respect to the ordered mode that is commonly used by default.

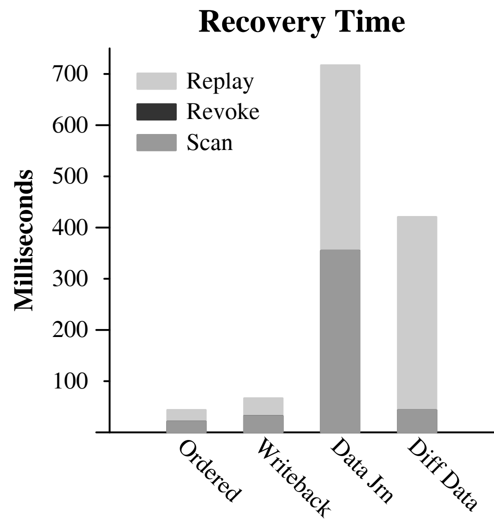

Figure 9. We have 100 threads doing 100 writes each of size 125 bytes with disabled the timer-based flushing of the dirty pages. We cut the power to the system, and during the reboot we measure the recovery time across the four mount modes. We observe that differential data journaling requires much lower time for the scan pass than the default data journaling mode, while the replay pass takes comparable time across the two modes.

6.3 Recovery Time

In a different experiment, we evaluate the ability of the system to recover quickly after a system crash that leads to log records appearing in the journal during the reboot. In this setting, we have 100 threads that call write 100 times with request size 125 bytes. We also disable the writeback and expiration time periods of the pdflush kernel thread in order to ensure that the transactions commit to the journal, but don't checkpoint the updates to the final location on disk. Then we cut the power to the system. During the reboot, we measure within the kernel the time period of the file system recovery. In Figure 9, we breakdown the total recovery across the three passes that scan the transactions, revoke blocks, and replay the committed transactions. We note that the scanning period for differential data journaling is much lower than that of data journaling and actually similar to those of ordered and writeback. Despite the extra block reads involved in the replay of differential data journaling, the time it takes ends up comparable to that of the default data journaling.

6.4 Other Issues

Since the ordered mode does not take full advantage of the separate journal device, we also investigate the case where we use the two SAS disks in RAID0 configuration with hardware controller support. For the differential data journaling, we use as journal a normal file rather than a separate device. From our measurements (not shown) we observe that the write latency drops to half in the ordered mode, when compared to the case where we dedicate one disk to the journal. After the change, the write latency of differential data journaling remains about the same as before. The relative difference between the latencies of the two modes is still high across the different streams rates and in excess of a magnitude order for 1Kbps streams.

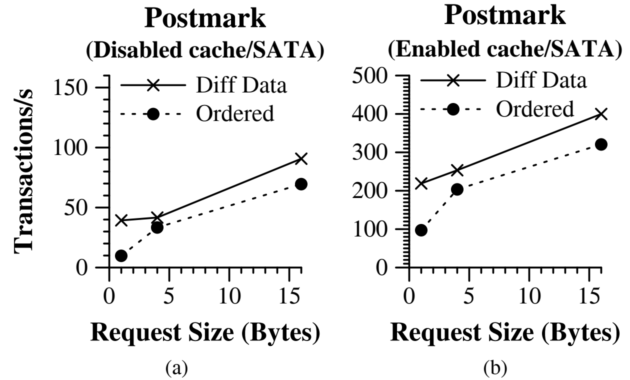

Figure 10. We run Postmark with 5000 transactions over a system with two 250GB SATA disks. (a) We disable the on-disk write cache to ensure that the writes only return after they reach the media. The advantage of differential data journaling is evident especially with small read and write requests. (b) When we enable the on-disk write cache, performance scales similarly for the ordered mode and differential data journaling, while the relative difference remains.

In a different experiment, we examine the effects from disabling the write cache of the disks. For these measurements, we use a server with two 250GB SATA disks. We find that the disabled write cache of the disks makes no difference to the streaming workload measurements in comparison to the case that the cache is enabled. However, in the case of the Postmark benchmark, disabling the write cache scales down the performance of the different mount modes, as shown in Figure 10. Overall, differential data journaling still maintains a significant advantage with respect to the ordered mode, especially at low stream rates.

7. Related Work

Digital streaming infrastructures replace traditional closed-circuit television systems in urban traffic-control applications to store large numbers of video feeds (Esteve and Palau 2006). Previously, environmental, oceanographic and meteorological conditions have been measured and stored over distributed relational databases (Long et al. 1995). Aurora is a stream processing engine that has been developed to support primitives for streaming applications, handle query processing on incoming messages in real time and gracefully deal with spikes in message load (Carney et al. 2002; Balakrishnan et al. 2004). The CoMo is a passive monitoring system that can be used as a building block for a network monitoring infrastructure that processes and shares network traffic statistics over multiple sites (Iannaccone et al. 2004). Como includes a storage process that is data agnostic and treats all data blocks equally. Also, load shedding techniques were developed to maintain the accuracy of traffic queries within acceptable levels at extreme traffic conditions (Barlet-Ros et al. 2007).

The Hyperion Stream File System has been specifically designed to reliably store multiple high-rate streams (Desnoyers and Shenoy 2007). It employs a log-structured write allocation that appends data to the end of each written file. It takes advantage of streaming writes to overwrite old data and avoid data copying for cleaning. It also maintains an index at high speed to efficiently support queries. yFS is a recently proposed file system for general purposes that only uses journal transactions for metadata modifications (Zhang and Ghose 2003). Xsyncfs introduces the idea of externally synchronous I/O that guarantees durability not to the application, but to the external entity that observes application output (Nightingale et al. 2006). However, in the case of applications that do not produce any output, xsyncfs commits data periodically similarly to an asynchronously mounted ext3.

Prabhakaran et al. introduced the semantic block-level analysis technique to trace and analyze file systems, and the semantic trace playback technique to evaluate file system modifications (Prabhakaran et al. 2005). Evaluation of ext3 over Linux showed that data journaling incurs substantial traffic to the journal but with sequential throughput, unlike the ordered mode that mainly writes data to the final location. The authors conclude that sequential workloads should better be served in ordered mode, while random workloads can benefit from data journaling. Using trace-based emulation, the authors show that differential data journaling can reduce substantially the amount of traffic to the journal in database applications.

Hildebrand et al. highlight the prevalance of small and sequential data requests in scientific applications (Hildebrand et al. 2006). They show that it is possible to improve the overall write performance of parallel file systems by using parallel I/O for large write requests and a distributed file system for small write requests. DualFS uses two separates devices for the data and metadata, respectively; it employs a log-structured file system for the metadata and treats data as in typical Unix systems (Piernas et al. 2002). Kulkarni et al. proposed the use of compression, duplicate block suppression and delta encoding to eliminate redundancy of stored data in a scalable and efficient way (Kulkarni et al. 2004).

The log-structured file system tries to solve the synchronous metadata update problem and the small-write problem by coalescing data writes sequentially to a segmented log (Rosenblum and Ousterhout 1992). However, the approach requires a garbage collection process to run in the background that incurs cleaning overheads. The virtual log is another effort to minimize the latency of small synchronous writes by building the log-structured file system over a log with entries that are not necessarily physically contiguous (Wang et al. 1999). Soft updates track and enforce metadata update dependencies so that the file system can safely delay writes for most file operations (Ganger et al. 2000). As a result the file system achieves improved performance with strong integrity and security guarantees at frequent metadata modifications.

The Write Anywhere File Layout improves write performance by writing file system blocks to any location on disk and in any order, while deferring disk space allocation with the help of non-volatile RAM (Hitz et al. 1994). The Google File System handles large files typically mutated by appending new data sequentially rather than overwriting existing data at random file locations (Ghemawat et al. 2003). The data and metadata journaling of the ext2 file system has been documented (Tweedie 1998; Galli 2001). Earlier, Hagmann described metadata update logging in the Cedar File System to improve performance and achieve consistency (Hagmann 1987). Also, the Echo distributed file system used a journal to record disk storage updates thus improving performance and availability (Birrell et al. 1993).

8. Conclusions and Future Work

Motivated from the emerging need to reliably store and handle large numbers of streams for real-time or retrospective processing, we take a fresh look at file systems that support data journaling. We use a commonly used file system with synchronous writes to find out that the journal device throughput is high because the journal log records store entire blocks rather than their modified part. Then, we introduce the differential data journaling mode, where we accumulate the updates from multiple writes into a single journal block. Additionally, we tune the timing of dirty page flushing to complete in the background rather than synchronously with the write operations. Using streaming workloads, we find that differential data journaling reduces the journal traffic substantially in comparison to the default data journaling mode, especially for streams with low rates. The sequential throughput of the journal reduces the write latency up

to orders of magnitude for the data journaling modes with respect to metadata-only journaling. Finally, we experiment with a typical small-write workload and measure substantial improvement in the supported transaction rate. Overall, differential data journaling offers fast storage across streaming and traditional workloads at relatively low disk throughput requirements.

In the future, we plan to investigate the automatic tuning of system parameters related to the timing of dirty page flushes. Additionally, we plan to examine the behavior of differential data journaling in the context of a distributed file system that we are currently building for the needs of streaming data storage.

9. Acknowledgements

In part supported by project INTERSAFE with approval number 303090/YD7631 of the INTERREG IIIA Greece-Albania 2000-20006 neighboring program.

References

Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

Pere Barlet-Ros, Gianluca Iannaccone, Josep Snjuas-Cuxart, Diego Amores-Lopez, and Josep Sole-Pareta. Load shedding in network monitoring applications. In *USENIX Annual Technical Conference*, pages 59–72, Santa Clara, CA, 2007.

Andrew D. Birrell, Andy Hisgen, Chuck Jerian, Timothy Mann, and Garret Swart. The echo distributed file system. Technical Report TR-111, DEC Systems Research Center, Palo Alto, CA, September 1993.

Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Sebastopol, CA, third edition, November 2005.

Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams - a new class of data management applications. In *International Conference on Very Large Data Bases*, pages 215–226, Hong Kong, China, 2002.

Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Interlational Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Cambridge, MA, 1996.

Peter J. Desnoyers and Prashant Shenoy. Hyperion: High volume stream archival for retrospective querying. In *USENIX Annual Technical Conference*, pages 45–58, Santa Clara, CA, June 2007.

Manuel Esteve and Carlos E. Palau. A flexible video streaming system for urban traffic control. *IEEE Multimedia*, 13(1):78–83, January 2006.

Ricardo Galli. Journal file systems in linux. *Upgrade*, 2(6):50–56, December 2001.

Gregory R. Ganger, Marshall K. McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(1):127–153, February 2000.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, Bolton Landing, NY, October 2003.

Robert Hagmann. Reimplementing the cedar file system using logging and group commit. In *ACM Symposium on Operating Systems Principles*, pages 155–162, Austin, TX, 1987.

Dean Hildebrand, Lee Ward, and Peter Honeyman. Large files, small writes, and pnfs. In *ACM International Conference on Supercomputing*, pages 116–124, Cairns, Australia, June 2006.

Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Usenix Winter Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

Gianluca Iannaccone, Christophe Diot, Derek McAuley, Andrew Moore, Ian Pratt, and Luigi Rizzo. The como white paper. Technical Report Technical Report IRC-TR-04-17, Intel Research, 2004.

Jeffrey Katcher. Postmark: A new file system benchmark. Technical Report TR-3022, NetApp, 1997.

Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, pages 59–72, Boston, MA, 2004.

Darrel D. E. Long, Patrick E. Mantey, Craig M. Wittenbrink, Theodore R. Haining, and Bruce R. Montague. Reinas: the real-time environmental information network and analysis system. In *IEEE COMPCON*, pages 482–487, March 1995.

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Usenix Symposium on Operating Systems Design and Implementation*, pages 1–14, Seattle, WA, 2006.

Juan Piernas, Toni Cortes, and Jose M. Garcia. Dualfs: A new journaling file system without meta-data duplication. In *ACM International Conference on Supercomputing*, pages 137–146, New York, NY, 2002.

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference*, pages 105–120, Anaheim, CA, 2005.

Mended Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

Stephen C. Tweedie. Journaling the linux ext2fs filesystem. In *LinuxExpo*, pages 25–29, Durham, NC, 1998.

Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Virtual log based file systems for a programmable disk. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 29–43, New Orleans, LA, 1999.

Zhihui Zhang and Kanad Ghose. yfs: A journaling file system design for handling large data sets with reduced seeking. In *USENIX Conference on File and Storage Technologies*, pages 59–72, San Francisco, CA, 2003.