# Okeanos: Wasteless Journaling for Fast and Reliable Multistream Storage

Andromachi Hatzieleftheriou and Stergios V. Anastasiadis
*Department of Computer Science*
*University of Ioannina, Greece*

## Abstract

Synchronous small writes play a critical role in the reliability and availability of file systems and applications that use them to safely log recent state modifications and quickly recover from failures. However, storage stacks usually enforce page-sized granularity in their data transfers from memory to disk. We experimentally show that subpage writes may lead to storage bandwidth waste and high disk latencies. To address the issue in a journaled file system, we propose *wasteless journaling* as a mount mode that coalesces synchronous concurrent small writes of data into full page-sized blocks before transferring them to the journal. Additionally, we propose *selective journaling* that automatically applies wasteless journaling on data writes whose size lies below a fixed preconfigured threshold. In the Okeanos prototype implementation that we developed, we use microbenchmarks and application-level workloads to show substantial improvements in write latency, transaction throughput and storage bandwidth requirements.

## 1 Introduction

Synchronous small writes lie in the critical path of several contemporary systems that target fast recovery from failures with low performance overhead during normal operation [1, 4, 5]. Typically, synchronous small writes are applied to a sequential file (*write-ahead log*) in order to record updates before the actual modification of the system state. In addition, the system periodically copies its entire state (*checkpoint*) to permanent storage. After a transient failure, recent state can be reconstructed by replaying the logged updates against the latest checkpoint. Write-ahead logging improves system reliability by preserving recent updates from failures; it also increases system availability by substantially reducing the subsequent recovery time. The method is widely applied in general-purpose file systems [6], relational databases [5], distributed key-value stores [4], event processing engines [3], and other mission-critical systems [7]. Furthermore, logging is useful for checkpointing parallel applications to preserve multiple hours or days of processing after an application or system crash [1].

Today, several file systems use a log file (*journal*) in order to temporarily move data or metadata from memory to disk at sequential throughput. Thus, they postpone the more costly writes to the file system without penalizing the corresponding latency perceived by the applications. A basic component across current operating systems is the page cache that temporarily stores recently accessed data and metadata in case they are reused soon. It receives byte-range requests from the applications, and communicates with the disk through page-sized blocks. The page-sized block granularity of disk accesses is prevalent across all data transfers, including data and metadata updates or the corresponding journaling whenever it is used. Asynchronous small writes improve their efficiency, when multiple consecutive requests are batched into page-sized blocks before they are flushed to disk. Instead, each synchronous write is flushed to disk individually causing data and metadata traffic of multiple full pages, even if the bytes actually modified across the pages collectively occupy much less space.

In Figure 1, we measure the amount of data written to the journal across different mount modes. We use a synthetic workload that consists of 100 concurrent threads with periodic synchronous writes of varying request sizes (one req/s). We include the ordered, writeback, and journal –refered to as data journaling from now on for clarity– modes of the ext3 file system (Section 4). As the request size increases up to 4KB, the traffic of data journaling remains almost unchanged at a disproportionately high value. At each write call, the mode appends to the journal the entire modified data and metadata blocks rather than only the corresponding block modifications. Instead, the ordered and writeback modes incur almost linearly increasing traffic, because they only store to the journal the blocks that contain modified metadata.

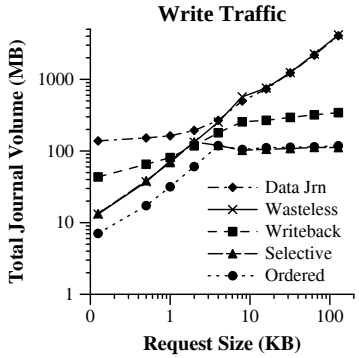



Figure 1: During a 5min interval, we measure the total write traffic to the journal device across different mount modes of the ext3 file system on Linux.

We set as objective to reduce the journal traffic so that we improve the performance of reliable storage at low cost. Thus, we introduce *wasteless journaling* and *selective journaling* as two new mount modes, that we propose, design and fully implement in the Linux ext3 file system. We are specifically concerned about highly concurrent multithreaded workloads that synchronously apply small writes over common storage devices [1, 4, 7]. We target to save the disk bandwidth that is currently wasted due to unnecessary writes of unmodified data, or writes with high positioning overhead. The operations in both these cases occupy valuable disk access time that should be used for useful data transfers instead. To achieve our goal we transform multiple random small writes into a single block append to the journal.

We summarize our contributions as follows: (i) Consider the reduction of journal bandwidth in current systems as a means to improve the performance of reliable storage at low cost; (ii) Design and fully implement wasteless and selective journaling as optional mount modes in a widely-used file system; (iii) Discuss the implications of our journaling optimizations to the consistency semantics; (iv) Apply micro-benchmarks, a storage workload and database logging traces over a single *journal* spindle to demonstrate performance improvements up to an order of magnitude across several metrics; (v) Use a parallel file system to show that wasteless journaling doubles, at reasonable cost, the throughput of parallel application checkpointing over small writes.

In the remaining paper, we summarize the related research in Section 2, present architectural aspects of our design in Section 3, while in Section 4 we describe the implementation of the Okeanos prototype system. In Section 5, we explain our experimentation environment, in Section 6 we present performance measurements across different workloads, and in Section 7 we outline our conclusions and future work.

## 2 Related Work

The log-structured file system addresses the synchronous metadata update problem and the small-write problem by batching data writes sequentially to a segmented log [9]. In transaction processing, group commit is a known database logging optimization that periodically flushes to the log multiple outstanding commit requests [5]. The above approaches gather multiple block writes into a single multi-block request instead of fitting multiple subpage modifications into a single block that we do. Also, subpage journaling of *metadata* updates is already available in commercial file systems, such as IBM JFS and MS NTFS [8]. Adding extra spindles to improve I/O parallelism or non-volatile RAM to absorb small writes could also reduce latency and raise throughput [6]. However, such solutions carry drawbacks that primarily have to do with increased cost and maintenance concerns.

A structured storage system may maintain numerous independent log files to facilitate load balancing in case of failure [4]. However, concurrent sequential writes to the same device create a random-access workload with low disk throughput. To address this issue, the system may store multiple logs into a single file and separate them by record sorting during recovery. Similarly, for the storage needs of parallel applications in high-performance computing, specialized file formats are used to manage as a single file the data streams generated by multiple processes [1]. Instead, we aim to handle the above cases at low cost through the mount modes that we add to a general-purpose file system.

The Echo distributed file system logged subpage updates for improved performance and availability, but bypassed logging for page-sized or larger writes [2]. However, Echo was discontinued in the early nineties partly because its hardware lacked fast enough computation relative to communication. Recent research introduced semantic trace playback to rapidly emulate alternative file system designs [8]. In that context, the authors emulated writing block modifications instead of entire blocks to the journal, but didn't consider the performance and recovery implications. Due to the obsolete hardware platform or the high emulation level at which they were applied, the above studies leave open the general architectural fit and actual performance benefit of journal bandwidth reduction in current systems.

## 3 System Design

We set as objective to safely store recent state updates on disk and ensure their fast recovery in case of failure. We also strive to serve the synchronous small writes and subsequent reads at sequential disk throughput with low bandwidth requirements. We are motivated by the important role that small writes play for reliable storage and the lack of comprehensive studies on subpage

data logging in current systems. In order to reduce the storage bandwidth consumed by data journaling, we designed and implemented a new mount mode that we call *wasteless journaling*. During synchronous writes, we transform partially modified data blocks into descriptor records that we subsequently accumulate into special journal blocks. We synchronously transfer all the data modifications from memory to the journal device. After timeout expiration or due to shortage of journal space, we move the partially or fully modified data blocks from memory to their final location in the file system.

With goal to reduce the journal I/O activity during sequential writes, we further evolved wasteless journaling into *selective journaling*. In that mount mode, the system automatically differentiates the write requests based on a fixed size threshold that we call *write threshold*. Depending on whether the write size is below the write threshold or not, we respectively transfer the synchronous writes to either the journal or directly the final disk location. Thus, we apply data journaling in only those cases that either multiple small writes can be coalesced into a single journal block according to wasteless journaling, or different data blocks that have been fully modified are scattered across multiple locations in the file system. We anticipate that journaling of the modified blocks will reduce the latency of synchronous writes through the sequential throughput offered by the journal device.

For consistency across system failures, each write operation delays metadata updates on disk, until the completion of the corresponding data updates. In wasteless journaling, we log both data and metadata into the journal to consider a write operation effectively completed. Synchronous writes from the same thread are added to the journal sequentially. In case of failure, a prefix of the operation sequence is recovered through the replay of the data modifications that have been successfully logged into the journal. Instead, selective journaling allows a synchronous write sequence to have a subset of the modified data added to the journal, and the rest of the modified data directly transferred to the final location in the file system. Given that a synchronous write from a single thread must be transferred to disk immediately, it only makes sense to accumulate into a journal block the writes from different concurrent threads. As a result, wasteless and selective journaling are mostly beneficial in concurrent environments with multiple writing streams that include frequent small writes.

In selective journaling, we call *update sequence* of a disk block a series of multiple incoming updates applied to the same block buffer. The updates don't have to be back-to-back, but there should be no in-between transfer of the respective buffer to the final disk location. If the first update in such a sequence has subpage size, we log to the journal the entire update sequence of this buffer. Thus, we handle consistency in a relatively clean way, because we eliminate the case that we turn off the journaling of a particular buffer halfway through a transaction. On the other hand, if the first update of the buffer is page-sized, we decide to skip journaling for the entire update sequence of the corresponding block. In our experience, the above two transitions in update sizes along a sequence occur infrequently. Therefore, we anticipate low impact to the journaling activity of selective journaling.

## 4 The Okeanos Prototype Implementation

We implemented wasteless and selective journaling in the Okeanos prototype that we developed based on Linux ext3. Originally, ext3 first copies the modified blocks into the journal, then transfers them to their final disk location. In *data journaling* mount mode, data and metadata blocks are copied to the journal, before they update the file system. To reduce the risk of data corruption, the *ordered* mode only copies the metadata blocks to the journal, after the associated data blocks have updated the file system. The *writeback* mode copies only metadata blocks to the journal, without any constraint in the relative order of data and metadata updates to the file system. It is the weakest mode in terms of consistency and we don't consider it any further in the rest of the paper.

The Linux kernel uses the *page cache* to keep in memory data and metadata of recently accessed disk files. For every disk block cached in memory, a *block buffer* stores its data and a *buffer head* maintains the related bookkeeping information. The page cache manages disk blocks in page-sized groups called *buffer pages*. We use block and page interchangeably, because they typically have the same size. Ext3 implements the journal as either a hidden file in the file system or a separate disk partition. Each *log record* in the journal contains an entire modified block instead of the byte range actually affected. However, the system only needs to log the updated part of each modified block and merge it into the original block to get its latest version during a recovery. To achieve that, we introduce a new type of journal block that we call *multiwrite block*. We only use multiwrite blocks to accumulate the updates from *data* writes that partially modify block buffers. When a block buffer contains metadata or is fully modified by a write operation, we can send it directly to the journal without the need to create an extra copy first in the page cache. We call *regular block* such a journal block.

When a write request of arbitrary size enters the kernel, the request is broken into variable-sized updates of individual block buffers. In wasteless journaling, if the size of a buffer update is less than the block size, we copy the corresponding data modification into a multiwrite block. Otherwise, we point to the entire modified

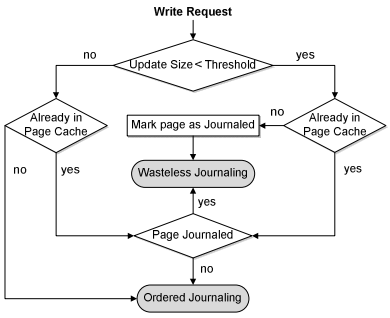
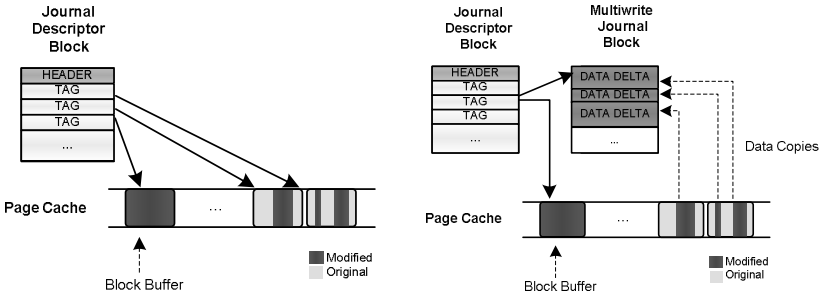



Figure 2: Alternative execution paths of a write request in the selective journaling mode.




(a) (b)

Figure 3: (a). In data journaling, the system sends to the journal the entire blocks modified by write operations. (b) In wasteless journaling, we accumulate multiple data writes into a single multiwrite journal block.

block in the page cache. For selective journaling, we have the write threshold fixed to the page size of 4KB. When a buffer update has size smaller than the write threshold, then we mark the corresponding page as *journaled*. Correspondingly, we copy the modification to the multiwrite block. We clear the journaled flag, after we transfer the corresponding block to its final location on disk. In Figure 2, we use a flowchart to summarize the possible execution paths of a write request through selective journaling.

A system call may consist of multiple low-level operations that atomically manipulate disk data structures of the file system. For improved efficiency, the system groups the records of multiple calls into one *transaction*. Before the transaction moves to the commit state, the kernel allocates a *journal descriptor block* with a list of *tags* that map block buffers to their final disk location. For each block buffer that will be written to the journal, the kernel allocates an extra buffer head specifically for the needs of journaling I/O. Additionally, it creates a *journal head* structure to associate the block buffer with the respective transaction. For writes that only modify part of a block, we expanded the journal head with two extra fields that contain the offset and the length of the multiwrite block pointed to by the buffer head (Figure 3). When we start a new transaction, we allocate a journal descriptor block that contains multiple fixed-length tags, one per write. In our system, we introduce three new fields in each tag: (i) a flag to indicate the use of a multiwrite block, (ii) the length of the write in the multiwrite block, and (iii) the starting offset of the modification in the final data block.

A transaction is *committed*, if it has flushed all its records to the journal; it is *checkpointed*, if all the blocks of a committed transaction have been moved to their final location on disk and the corresponding log records are removed from the journal. If the journal contains log records after a crash, the system initiates a recovery process during which we retrieve the modified blocks from the journal. In the case of multiwrite blocks, we apply the updates to blocks that we read from the corresponding final disk locations. We read into memory and update the appropriate block, as specified by the final disk location and the starting offset in the tag. However, if the multiwrite flag is not set, then we read the next block of the journal and treat it as a regular block. We write every regular block directly to the final disk location without need to read first its older version from the disk.

Both data and wasteless journaling guarantee the atomicity of updates, because they replay the modifications of the committed transactions until they fully reach the file system. Instead, selective journaling makes a decision whether to journal or not an update sequence based on the size of the first write. Journaling of an update sequence implies atomicity of the modification for the corresponding block, while direct transfer of the block to the file system implies consistency similar to that of ordered mode.

## 5 Experimentation Environment

We developed the Okeanos prototype implementation of wasteless and selective journaling by modifying 684 lines of code across 19 files of the original Linux kernel version 2.6.18. Members of our team used the prototype system as working environment for several months. In our experiments we use x86-based servers with one quad-core 2.66GHz processor, 3GB RAM, two Seagate Cheetah SAS 300GB 15KRPM disks and one active gigabit ethernet port. Unless we specify differently, we assume synchronous write operations with the journal and data partition on two separate disks. Our results have half-length of 90% confidence interval within 10% of the reported average. We flushed the page cache between all the repetitions of our experiments.

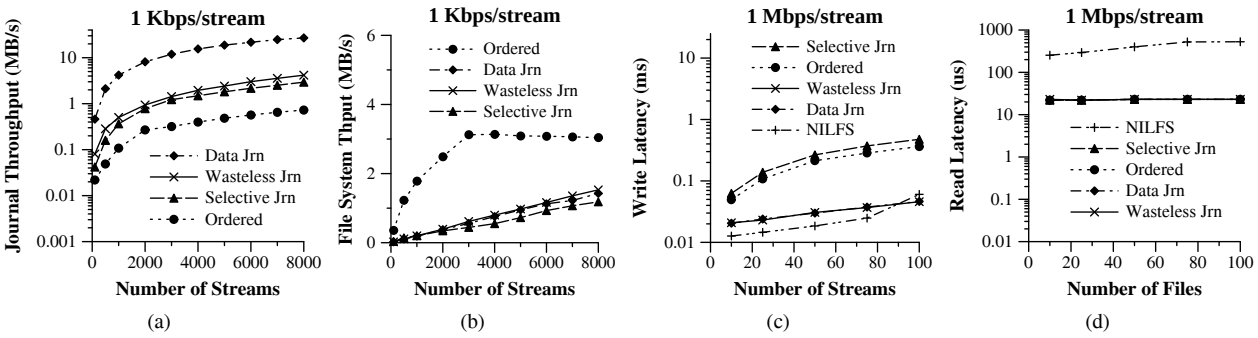



Figure 4: (a) At 1Kbps, the journal throughput (lower is better) of selective and wasteless journaling lies approaches that of ordered, unlike data journaling which is several factors higher. (b) In comparison to ordered at 1Kbps, the remaining three modes reduce file system throughput by several factors (lower is better). (c) At 1Mbps, the selective and ordered modes incur much higher latency in comparison to the other ext3 modes or NILFS. (d) If we create multiple files concurrently, read requests of 4KB with NILFS take an order of magnitude longer with respect to ext3.

## 6 Performance Evaluation

First, we produce a random I/O traffic by running a number of concurrent threads directly on the file server. Each thread appends data to a separate file by calling one synchronous write per second. At increasing number of 1Kbps streams, Figure 4(a) shows that the journal throughput of data journaling is an order of magnitude higher than that of the other modes (up to 27MB/s). On the contrary, selective and wasteless journaling limit the traffic up to about 4MB/s. In Figure 4(b), we measure the write throughput of the file system device. The ordered mode wastes disk bandwidth by sending each write to the final location in units of 4KB. Instead, wasteless, selective and data journaling leave dirty pages temporarily in memory before coalescing them into the file system. With controlled system crashes, we additionally found that selective and wasteless journaling tend to reduce the recovery time of data journaling (by more than 20% in some cases).

Next, we examine the average latency of synchronous writes. In Figure 4(c) with 1Mbps streams, ordered and selective incur orders of magnitude higher latency than the other modes. At 1Kbps (not shown), selective tends to become identical to wasteless journaling. In asynchronous writes that we also tried, we found selective and wasteless journaling to reduce the latency of ordered and data journaling up to two orders of magnitude. In Figure 4(c), we also consider a stable Linux port (NILFS) of the log-structured file system [9]. The write latency of NILFS is comparable to that of wasteless and data journaling. In Figure 4(d), we use a thread to read sequentially one after the other different numbers of files that we previously created concurrently at 1Mbps each, using NILFS or ext3. Then, we measure the average time to read a 4KB block. We observe that NILFS is an order of magnitude slower with respect to ext3. In fact, NILFS interleaves the writes from different files on disk, which may lead to poor storage locality during sequential reads.

We use the Postmark benchmark to examine the performance of small writes as seen in electronic mail, netnews and web-based commerce. We apply version 1.5 with synchronous writes added by FSL of Stony Brook Univ. We assume an initial set of 500 files and use 100 threads for a total workload of 10,000 mixed transactions. We draw the file sizes from the default range, while I/O request sizes lie between 128 bytes and 128KB. In Figure 5(a), we observe that the transaction rate of wasteless journaling gets as high as 738tps. Across different request sizes, wasteless journaling consistently remains faster than the other modes, including data journaling. Instead, selective journaling lies between data journaling and ordered mode, which are slower than wasteless.

We also examine the OLTP performance benchmark TPC-C as implemented in Test 2 of the Database Test Suite. We used the MySQL open-source database system with the default InnoDB storage engine. We tested a configuration with 20 warehouses and 20 connections, 10 terminals per warehouse and 500s duration. InnoDB supports three methods for flushing the database transaction log to disk. In the default method 1 (*Cmt/Disk*), the log is flushed directly to disk at each transaction commit. In method 0 (*Prd/Disk*), the transaction log is written to the page cache and flushed to disk periodically. Finally, in method 2 (*Cmt/Cache*), the transaction log is written to the page cache at each transaction commit and periodically flushed to disk. During an execution of TPC-C, we collect a system-call trace of the MySQL transaction log. Subsequently, we replay a varied number of concurrent instances of the log trace over the ordered and wasteless journaling. In Figure 5(a), we see that wasteless journaling takes up to tens of seconds to complete each log

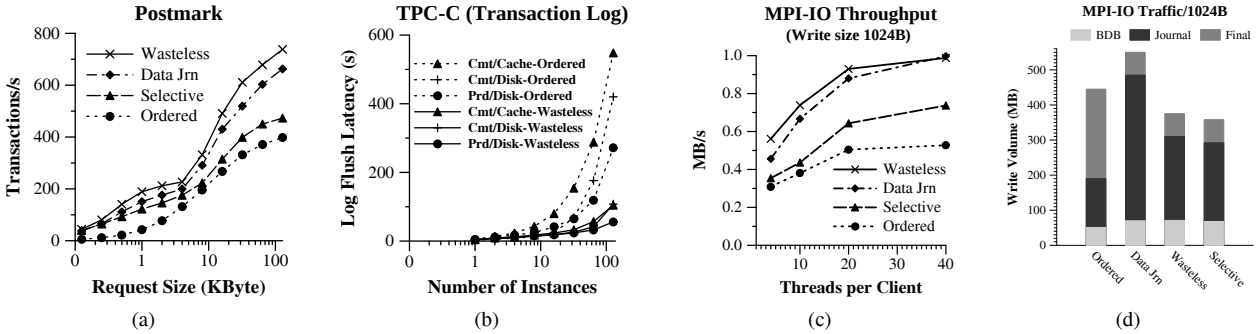



Figure 5: (a) Wasteless journaling consistently achieves the highest transaction rate in Postmark. (b) Across the three flushing methods of MySQL/InnoDB, wasteless journaling substantially reduces the latency to flush the transaction log to disk. (c) Wasteless journaling almost doubles the data throughput (higher is better) of ordered mode. (d) We measure the write traffic (lower is better) to BerkeleyDB (BDB), the journal (Journal) and the file system (Final) at the data server of PVFS2. Selective and wasteless journaling incur less traffic under the MPI-IO benchmark.

flush across the three methods of InnoDB at high load. Instead, ordered mode takes hundreds of seconds, as the number of instances approaches or exceeds 64.

Finally, we use our mount modes in the storage server of a PVFS2 multi-tier configuration. In a networked cluster, we use thirteen machines as clients, one machine as PVFS2 data server and one as PVFS2 metadata server. By default, each server uses a local BerkeleyDB database to maintain local metadata. At the data server we placed the BerkeleyDB on one partition of the root disk, and dedicated the entire second disk to the user data (file system and journal). We fixed the BerkeleyDB partition to ordered mode and tried alternative mount modes at the data disk. We enabled data and metadata synchronization, as suggested to avoid write losses at server failures. We used the LANL MPI-IO Test to generate a synthetic parallel I/O workload on top of PVFS2. In our configuration each process writes to a separate unique file, as suggested for best performance [1]. We varied between 4 and 40 the number of processes on each of the thirteen quad-core clients leading to total processes between 52 and 520. We tried 65000 writes of size 1024 bytes. In Figure 5(c), wasteless journaling almost doubles the throughput of ordered mode, while data journaling and selective lie between the other two modes. In Figure 5(d), wasteless journaling reduces by 42% the journal traffic of data journaling, while selective journaling further reduces the write volume of wasteless journaling.

## 7 Conclusions and Future Work

We rely on journaling of data updates in a file system to ensure their safe transfer to disk at low latency and high throughput without storage bandwidth waste. We design and implement a mount mode that we call wasteless journaling to merge into page-size blocks concurrent subpage writes to the journal. Additionally, we develop the selective journaling mode that only logs updates below a write threshold and transfers the rest directly to the file system. We experimentally demonstrate reduced write latency, improved transaction throughput with low journal bandwidth requirements. Our plans for future work include extension of our journaling methods for virtualization environments and solid-state disks.

## 8 Acknowledgments

We are thankful to our shepherd Junfeng Yang for his valuable guidance. In part supported by project INTERSAFE (No. 303090/YD7631) of the INTERREG IIIA program co-funded by EU and the Greek State.

## References

[1] BENT, J., GIBSON, G., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. Plfs: a checkpoint filesystem for parallel applications. In *ACM/IEEE SC* (2009), pp. 1–12.

[2] BIRRELL, A. D., HISGEN, A., JERIAN, C., MANN, T., AND SWART, G. The echo distributed file system. Tech. Rep. TR-111, DEC Systems Research Center, Palo Alto, CA, Sept. 1993.

[3] BRITO, A., FETZER, C., AND FELBER, P. Minimizing latency in fault-tolerant distributed stream processing systems. In *IEEE ICDCS* (Montreal, QC, 2009), pp. 173–182.

[4] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *USENIX OSDI* (2006), pp. 205–218.

[5] GRAY, J., AND REUTER, A. *Transaction Processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993, ch. 9. Log Manager.

[6] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *USENIX Winter Technical Conference* (San Francisco, CA, Jan. 1994), pp. 235–246.

[7] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *USENIX OSDI* (Seattle, WA, 2006), pp. 1–14.

[8] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *USENIX ATC* (Anaheim, CA, 2005), pp. 105–120.

[9] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26–52.