Reducing Bandwidth Waste in Reliable Multistream Storage

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

Technical Report DCS 2011-02
January 31, 2011

Abstract

Synchronous small writes play a critical role in the reliability and availability of current systems because they are used to safely log recent state modifications and allow fast recovery after failures at the application and system level. In highly demanding environments, it is typical to dedicate separate devices for the logging activity alone. Thus, systems provide adequate performance during normal operation and extra redundancy for state reconstruction after a failure. However, storage stacks usually enforce page-sized granularity in their data transfers from memory to disk. Under various conditions, 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 the selective journaling mode that automatically applies wasteless journaling on data writes whose size lies below a fixed preconfigured threshold. Depending on the request size, written data is either first coalesced to the journal of the file system, or directly transferred to its final location on disk. We developed a prototype implementation of our design based on a widely-used file system. Then, we apply microbenchmarks and application-level workloads to standalone servers and a multi-tier networked configuration to show substantial improvements in handling small write traffic over write latency, transaction throughput, recovery time 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 loss during normal operation [6, 12, 21, 23, 26, 31, 32]. 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 [47]. Write-ahead logging improves system reliability by preserving recent updates from failures; it also increases system availability by substantially reducing the subsequent recovery time. Write-ahead logging is widely applied in general-purpose file systems [2, 22, 26, 40, 44], relational databases [21], distributed key-value stores [12, 32], event processing engines [9, 31], and other mission-critical systems [35]. Furthermore, logging is applied during the checkpointing of parallel applications to avoid losing the processing of multiple hours or days after an application or system crash [6, 37].

Thus, it becomes evident that logging is interpositioned through state modifications and plays a critical role during crash recovery across a broad range of systems. In fact, it is typically suggested to overprovision the logging bandwidth by placing the log file on a device that is different from the device that stores the state data [33]. Thus, the often substantial logging I/O activity will not affect state updates; additionally, the log file shall survive a potential failure of the device that hosts the state data and potentially allow its subsequent reconstruction [21]. Nevertheless, synchronous small writes can create performance bottlenecks due to their relatively high disk-positioning overhead [3, 35, 49]. In more general workload conditions, an I/O-intensive application that writes pages at the rate of flushing makes asynchronous writes appear as synchronous [5].

Furthermore, a reliable distributed service may maintain numerous independent log files [12, 41]. For example, this is the approach used by a structured storage system to facilitate the balanced load distribution in case of failures. However, concurrent sequential accesses of multiple independent files on the same device effectively create a random-access workload that reduces the disk

throughput by an order of magnitude or more. In order to address this issue, it was proposed to store multiple logs in a single file [12]. During recovery, the individual logs can be separated from each other at the cost of extra software complexity –and delay– to sort the corresponding log records. Similarly, for the storage needs of parallel applications in high-performance computing, specialized file formats have been developed to manage as a single file the data streams generated by multiple processes [6, 18, 24, 37].

Today, several file systems use a log file (*journal*) in order to temporarily move data or metadata from memory to disk at sequential throughput [43]. Thus, they postpone the more costly writes to the file system without penalizing the write latency perceived by the applications. A basic component across current operating systems is the page cache that temporarily stores recently accessed data and metadata for the case they are reused soon [8]. 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. If multiple consecutive requests are batched into page-sized blocks before they are flushed to disk, asynchronous small writes can actually improve their efficiency. However, in the case of synchronous small writes, each write is flushed to disk individually causing data and metadata traffic of multiple full pages, even if the bytes actually modified occupy collectively much less space.

For periodic synchronous writes of varying request sizes, Figure 1 shows the amount of data written to the journal across different mount modes. We include the ordered, writeback and journal (we call it data journaling for clarity) modes typically supported by ext3. As the request size increases up to 4KB, the traffic of the data journaling mode remains almost unchanged at a relatively high value. As we explain in Section 3, at each write call the data journaling mode writes to the journal the entire modified data and metadata blocks rather than only the corresponding block modifications. Instead, the ordered and writeback modes incur lower traffic, because they only write to the journal the (entire) blocks of modified metadata.

The problem with the wasteful journal traffic of data journaling is not simply resolved by reducing the granularity of disk writes, e.g., to the size of a single disk sector. In fact, such an approach would have the adverse effect of increasing the relative cost of I/O overhead. However, one promising approach is to accumulate the modifications from multiple writes into a single page-sized block, then pay the I/O overhead for multiple writes only once. This approach cannot be directly applied to writes

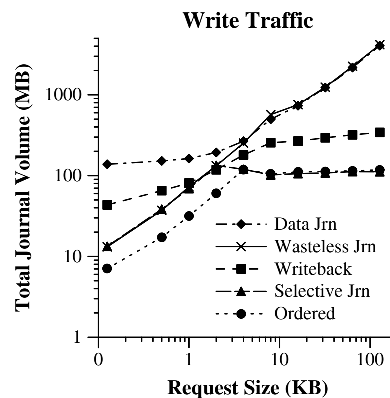

Figure 1: For a duration of 5min, we use the Linux ext3 file system to do periodic (per second) synchronous writes of different request sizes. We measure the total write traffic to the journal across different mount modes (fully explained in Sections 2 and 3). Data journaling (or journal mode of ext3) incurs disproportionately high journal traffic for request sizes below 4KB.

that modify the system state because generally each write corresponds to a different block on disk. However, it is applicable when we journal the updates into a log.

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 and selective journaling as two new mount modes, that we propose, design and fully implement over the Linux ext3 file system. We are specifically concerned about highly concurrent multithreaded workloads that synchronously apply small writes over common storage devices [6, 12, 31, 35]. 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 writes 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 few sequential large ones. We use microbenchmarks and application-level workloads to show that the modes that we introduce can considerably reduce journal traffic when applied to multiple concurrent synchronous small writes. Moreover, they substantially reduce response time across several cases in comparison to alternative mount options and file systems.

The general idea of subpage logging is not new. Two decades ago, researchers at the DEC Systems Research Center prototyped and used the Echo distributed file system [7]. They logged subpage updates for improved performance and availability, but bypassed logging for page-sized or larger writes [25]. Unfortunately, the development of Echo was discontinued in early 1992, partly because the hardware it relied on lacked fast enough com-

putation relative to communication. Recent research introduced semantic trace playback (STP) to rapidly evaluate alternative file system designs without the cost of real system implementation or detailed file system simulation [38]. The authors used STP to emulate a file system that writes modifications of blocks to the journal instead of entire modified blocks. They report dramatic reduction in the amount of data written to the journal, but don't examine the general performance and recovery implications. Due to the obsolete hardware characteristics or the high emulation level at which the above studies were applied, they leave vague the general architectural fit and actual performance benefit of journal bandwidth reduction over current systems.

Today, the journaling of data updates is still considered a costly operation whose use is discouraged or not supported by production-level file systems [3]. After consideration of the benefits that can emerge from reduced journal traffic, we introduce wasteless and selective journaling as two fully-functional mount modes and evaluate their performance in comparison to current representative mount modes and file systems. We summarize our contributions as follows:

1. Consider the reduction of journal bandwidth in current systems as a means to improve the performance of reliable storage at low cost.

2. Design and fully implement wasteless and selective journaling as optional mount modes in a widely-used file system.

3. Discuss the implications of alternative journaling optimizations to the consistency semantics.

4. Apply micro-benchmarks, storage workloads and database logging traces over a single *journal* spindle to demonstrate performance improvements up to an order of magnitude across several metrics.

5. 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 first present architectural aspects of our design in Section 2. Then, in Section 3 we describe the implementation of wasteless and selective journaling. In Section 4, we explain our experimentation environment, in Section 5 we present detailed measurements across different workloads, and in Section 6 we summarize previous related work. In Sections 7 and 8, we discuss our results and present our conclusions.

2 System Design

In the present section, we describe the basic assumptions and objectives of our architecture.

2.1 Data updates

Previous research has already recognized the role of synchronous small writes for the reliable operation of current systems in the context of general storage-based services [2, 35, 44, 49], parallel computations [6, 18, 24, 37], key-value stores [12,32] and event processing engines [9, 31]. Therefore, we aim 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. In the present work, we choose to do subpage journaling of *data* updates, meaning that we store to the journal only the modified part of each data page. 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. The related problem of durable consistency in the *metadata* updates of a file system has been previously studied extensively [20, 23, 44]. Also, subpage journaling of *metadata* updates is already widely available to the end users through popular commercial file systems, such as the IBM JFS and MS NTFS [38]. To the best of our knowledge, the present work is the first to comprehensively investigate the benefits of subpage data journaling using a prototype implementation in a fully operational file system.

2.2 Wasteless Journaling

Historically, journaling was applied to the metadata of a file system with goal to ensure fast structural recovery after a system failure [23, 46]. Today, there are file systems (e.g. ext3, ReiserFS) that optionally support journaling of both data and metadata. Related research reported that data journaling may improve the throughput of random I/O operations. However, it does not recommend data journaling for sequential writes due to the high journaling cost that it incurs in terms of consumed disk bandwidth [3, 38]. In order to reduce journal bandwidth, 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. As expected, we treat normally those data blocks that have been fully modified by write operations. 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.

2.3 Selective Journaling

With goal to reduce the journal I/O activity during sequential writes, we further evolved wasteless journaling into an alternative mount mode that we call *selective*

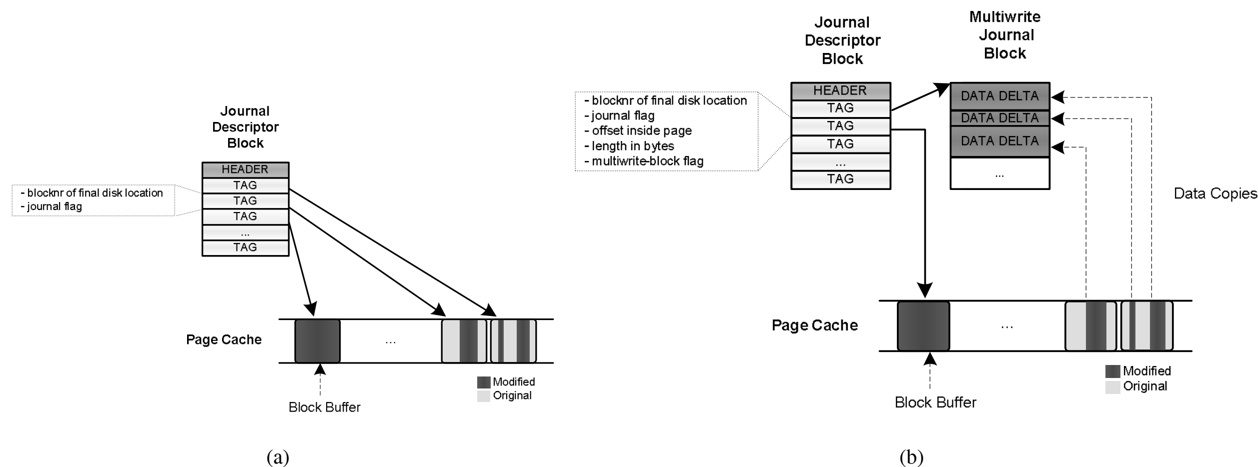

Figure 2: (a). In the original design of data journaling, the system sends to the journal the entire blocks modified by write operations. (b) In wasteless journaling, we use multiwrite journal blocks to accumulate the data modifications from multiple writes.

journaling. In selective journaling 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. The rationale of this approach is to 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. In these cases, we anticipate that journaling of the modified blocks will reduce the latency of synchronous writes through the sequential throughput offered by the journal device.

2.4 Consistency

In order to keep the structure of the file system consistent across system failures, in each write operation we delay metadata updates on disk until the completion of the corresponding data updates. In the case of wasteless journaling, we log both data and metadata into the journal device before we consider effectively completed a write operation. 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. During a recovery after a failure, it is possible that the last operation in a write sequence is fully aborted if the corresponding journal appends were interrupted halfway. However, if the last write operation is large enough to be selected for transfer to the final location, it is possible that it was only partially completed at the instance of the failure. Therefore, the consistency semantics of selective journaling is at least as strict as that of mount modes that journal the metadata after storing the respective data to disk. As the percentage of writes with size below the write threshold increases, selective journaling approaches wasteless journaling in terms of consistency.

We provide additional explanations about the system consistency, when we present our implementation in Section 3. 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. Therefore, we expect wasteless journaling and selective journaling to be mostly beneficial in concurrent environments with multiple writing streams that include frequent small writes.

3 Prototype Implementation

Ext3 implements journaling by performing high-level operations of the file system in two steps. First it copies the modified blocks into the journal; then it transfers the modified blocks into their final disk location and discards the journal blocks. 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 (we refer to this mode as *data journaling*.). The *ordered* mode only copies the metadata blocks to the journal, after the associated data blocks have updated the file system. This reduces the risk of data corruption inside a file. In *write-*

back mode only metadata blocks are copied to the journal, but there are no requirements in the relative order at which data and metadata blocks update the file system. It is considered the fastest, but also the weakest mode in terms of consistency.

3.1 Buffers

The Linux kernel uses the *page cache* to keep in memory data and metadata of recently accessed disk files [8]. For every disk block cached in memory, a *block buffer* stores its data and a *buffer head* maintains the related bookkeeping information. In Linux terms, the page cache manages disk blocks in page-sized groups called *buffer pages*. Typically, block and page have the same size. Therefore, we use the two terms interchangeably in the rest of our presentation. A number of *pdflush* kernel threads flush dirty pages to their final location on disk; they systematically scan the page cache every *writeback period* and implement on each page a timeout mechanism of a configurable *expiration period*. Additionally, a user can synchronously flush to disk the data and metadata dirty buffers of an open file, e.g., through the *fsync* call.

Ext3 uses a special kernel layer called *journaling block device* to implement the journal as either a hidden file in the file system or a separate disk partition. Each *log record* in the journal corresponds to an update of one disk block. The log record contains the entire modified block instead of the byte range actually affected. Thus, journaling in Linux is wasteful in disk bandwidth and space, but straightforward to restore the modified blocks after a crash. In small writes, the wasted journal bandwidth depends on the fraction of the block buffer that is left unmodified by each write operation.

In practice, 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 this end, we introduce a new type of journal block that we call *multiwrite block* (Figure 2(b)). 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 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*. We do that by setting a special flag that we added in the page descriptor of the buffer page. Correspondingly, we copy the modification to the multiwrite block. If the update modifies the entire block, then we prepare the corresponding modified buffer for transfer to the final location without prior journaling. We clear the journaled flag, after we transfer the corresponding block to its final location on disk. Our current prototype also supports write thresholds below the page size in a straightforward way. However, handling write thresholds above the page size would require additional implementation intervention at the Linux path of write requests in order to identify the buffers that correspond to writes of size below threshold.

3.2 Transactions

An *atomic operation handle* (or *handle*) consists of multiple low-level operations that manipulate disk data structures of the file system. When the system recovers from a crash, it ensures atomicity of a handle by having it fully completed or discarding all its low-level operations. For improved efficiency, the system groups the records of multiple handles into one *transaction*. A transaction accepts log records of new handles for a fixed period of time and stores them consecutively on the journal. A transaction is *finished* if all its log records are fully residing in the journal including the commit block, and *incomplete* if at least one log record of the transaction is not in the journal. When recovering from a failure, the system skips all incomplete transactions and transfers the blocks of the finished transactions to the file system.

Each invocation of the *write* system call creates a new handle that is added to the current active transaction. Before the transaction moves to the commit state, the kernel allocates a *journal descriptor block*; this block contains a list of descriptors, called *tags*, that map block buffers to their final disk location. When a journal descriptor block fills up with tags, the kernel moves it to the journal together with the associated block buffers. 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. After all the log records of a transaction have been safely moved to the journal, the system appends to the journal a final commit block.

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 2(b)). When we start a new transaction, we allocate a buffer for the journal descriptor block. The journal descriptor block contains a list of fixed-length tags, where each tag corresponds to one

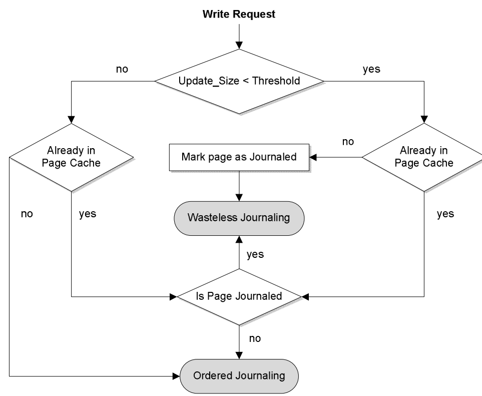

Figure 3: Alternative execution paths of a write request in the selective journaling mode.

write. Originally, each tag contained the final disk location of the modified block, and one flag for journal-specific properties of the block. In our design, 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.

3.3 Recovery

We consider a transaction *committed*, if it 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, we regard the transaction as *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 assumes that the unmount was unsuccessful and initiates a recovery procedure in three phases. In the *scan* phase, it looks for the last record in the journal that corresponds to a committed transaction. During the *revoke* phase, the kernel marks as revoked those blocks that have been obsoleted by later operations. In the *replay* phase, the system writes to their final disk locations the remaining (unrevoked) blocks that occur in committed transactions.

During the recovery process, 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. Since the data of consecutive writes are placed next to each other in the multiwrite block, we can deduce their corresponding starting offsets from the length field in the tags. As soon as the length field of a tag exceeds the end of the current multiwrite block, we read the next block from the journal and treat it as another multiwrite block from the same transaction. 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.

3.4 Update sequences

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 mark the corresponding buffer as journaled. Then, we log to the journal the entire update sequence of this buffer. This approach seems somewhat wasteful in terms of journal bandwidth, if the sequence includes page-sized updates. Thus, however, 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. Consequently, we reduce the possibility of partial block recovery, after journal replay. 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. If we prepare a block for transfer to the final disk location without prior journaling, we keep no journal head to associate the block with a transaction. Adding such an association afterward would complicate considerably our code. 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. In Figure 3, we use a flowchart to summarize the possible execution paths of a write request through selective journaling.

Both data and wasteless journaling guarantee the atomicity of updates, because they can 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. Depending on the percentage of updates that satisfy the journaling criterion, selective journaling follows either the semantics of ordered or data journaling, respectively.

4 Experimentation Environment

We implemented wasteless and selective journaling in the Linux kernel version 2.6.18. Several newer Linux releases have been made available recently, but they still

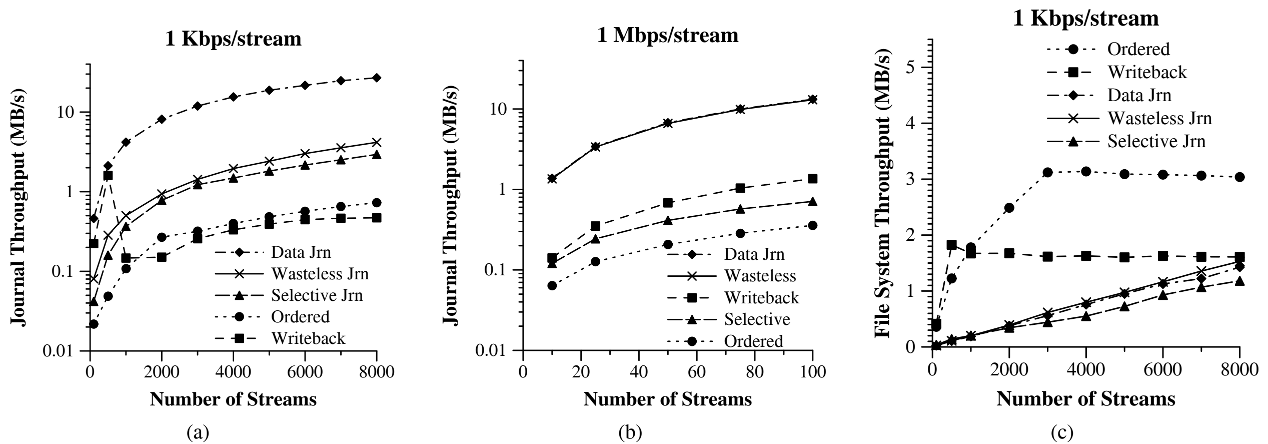

Figure 4: (a) At 1Kbps, the journal throughput (lower is better) of both selective and wasteless journaling approaches that of ordered and writeback modes, unlike data journaling which is several factors higher. (b) At 1Mbps, wasteless and data journaling have the same journal throughput, while selective journaling lies between writeback and ordered. (c) In comparison to ordered and writeback at 1Kbps, the other three modes incur lower file system throughput (lower is better), because they batch multiple writes into fewer page flushes.

do not provide the functionality that we propose. In order to add the proposed functions into ext3, we modified 684 lines of code across 19 files of the original Linux kernel. The design and implementation took us more than two person-years to complete, while members of our team used the modified system as working environment for several months. We evaluated our prototype over a sixteen-node cluster using x86-based servers running the Debian Linux distribution and connected through gigabit ethernet.

In most experiments we use nodes with one quad-core 2.66GHz processor, 3GB RAM, and two Seagate Cheetah SAS 15KRPM disks. Each disk has 300GB storage capacity, 16MB cache, 3.4/3.9ms average read/write seek time and 122-204MB/s sustained transfer rate. We have the journal and the data partition on two separate disks, unless we mention otherwise. In few experiments, we use two SATA 7.2KRPM disks of 250GB and 16 MB cache. We kept the page and block sizes equal to 4KB, while we left the journal size at the default value 128MB. In our measurements, we assume synchronous write operations, unless we specify differently. We keep the default parameters of periodic page flushing: writeback period equal to 5s and expiration period 30s. Between successive repetitions, we flushed the page cache by unmounting the journal device and writing the value 3 to the /proc/sys/vm/drop_caches. With up to fifteen repetitions of our experiments on otherwise idle machines, we ensure that our results have half-length of 90% confidence interval within 10% of the reported average.

The default disk settings increase speed by allowing a synchronous write to return when the data reaches the on-disk cache rather than the storage surface. However this behavior makes the system less reliable, unless somebody disables the on-disk cache or uses controllers with battery-backed cache [35]. In most of our experiments we kept enabled the on-disk write caches, but in Section 5.6 we also report similar comparative results from experiments with the write caches disabled.

5 Performance Evaluation

In this section, we study the performance of microbenchmarks, mailserver benchmarks and traces from database logs directly running on the modified file system. We also evaluate a stable Linux port of the *log-structured file system*, where the entire file system is structured as a log [40]. Additionally, we use a multi-tier configuration based on the PVFS2 distributed file system to examine the impact of the server file system to the parallel workload running across multiple clients. We also measure the recovery time after a crash and examine the sensitivity of our numbers to alternative disk configuration settings.

5.1 Microbenchmarks

First, we put a number of threads running directly on the file server. Each thread appends data to a separate file by calling one synchronous write per second; as a result, they produce random aggregate traffic. With 1Kbps streams in Figure 4(a), we observe that as the load grows from one hundred to several thousand streams, the journal throughput of data journaling remains an order of magnitude higher with respect to the other modes (up to 27MB/s). On the contrary, selective and wasteless

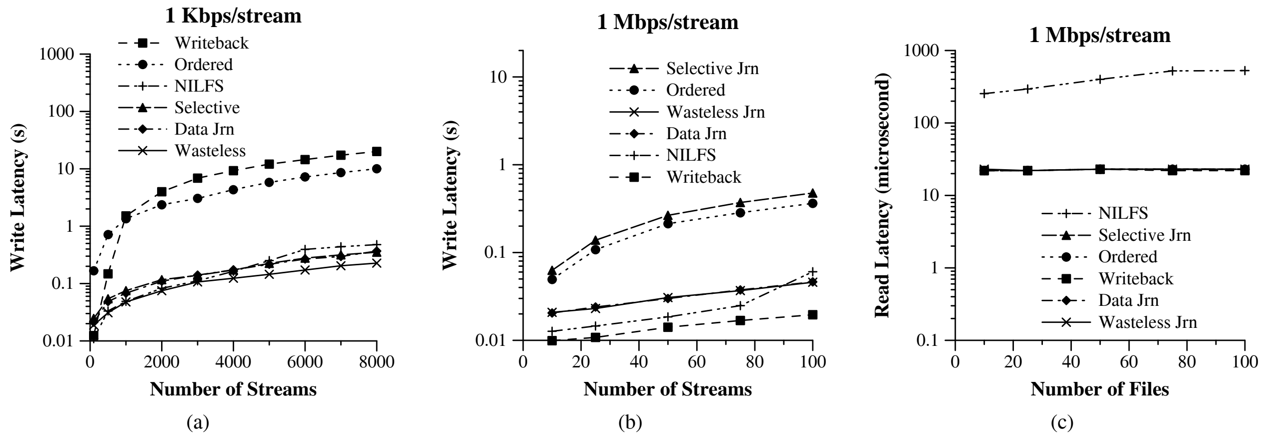

Figure 5: (a) With low rates, the write latency (lower is better) of ordered and writeback appears orders of magnitude higher than the other modes. (b) At higher rates, the selective and ordered modes experience much higher latency. (c) As we read sequentially multiple files that we previously wrote concurrently, read requests of 4KB with NILFS complete in order of magnitude longer time with respect to the different modes of ext3 that we evaluate.

journaling keep the traffic up to about 4MB/s. If we increase the stream rate to 1Mbps, wasteless and data journaling appear identical, while selective generates about twenty times lower journal throughput (Figure 4(b)). We also examined (not shown) mixed workloads consisting of streams with different rates. In that case, the journal throughput of wasteless journaling varies depending on the fraction of requests that are below the write threshold.

In Figure 4(c), we measure the write throughput of the file system device. The ordered mode sends each write to the final location in units of 4KB, thus wasting disk bandwidth with low-rate streams. Instead, wasteless, selective and data journaling leave dirty pages in memory for the expiration period before flushing them to the file system (Section 3.1). Thus, multiple writes to the same data block are automatically coalesced into fewer page flushes leading to lower traffic at the file system. In our microbenchmarks, we also measured the processor utilization and found it higher with low-rate streams over wasteless, selective and data journaling. Nevertheless, processor utilization always remained low, since it only reached 5%.

The benefits of journaling the data is also visible, if we consider the average latency of synchronous writes. In Figure 5(a), we see the ordered and writeback modes to incur orders of magnitude higher latency with respect to the other modes, as they serve multiple streams of 1Kbps. For instance, data journaling completes a write operation in tens of milliseconds, while ordered mode takes several seconds. We repeated the above experiments with asynchronous writes (not shown). Especially at low rates, we found that selective and wasteless journaling reduce the latency of ordered and data journaling up to two orders of magnitude. Thus we validate previous reports that asynchronous workloads may behave as synchronous in several cases [5].

In Figure 5, we also include measurements from a stable port of the log-structured file system (NILFS) to the Linux kernel 2.6 [50]. We find that the write latency of NILFS is comparable to that of wasteless and data journaling at both 1Kbps and 1Mbps streams. As expected, selective follows wasteless journaling at low rates, but ordered mode at high rates. Overall, the sequential throughput of the journal improves significantly the ability of the system to store fast the incoming data.

In Figure 5(c), we use a thread to read sequentially one after the other different numbers of files that were previously written concurrently at 1Mbps each, using NILFS or ext3. In this experiment we measure the average time to read a 4KB block. We observe that NILFS is an order of magnitude slower with respect to ext3. We attribute this behavior to the fact that NILFS interleaves the writes from different files on disk, which may lead to poor storage locality during sequential reads. In supplementary experiments that we did with 1Kbps streams, NILFS along with ordered and writeback also incur much higher read latencies than the other three modes.

5.2 Postmark and Varmail

We use the Postmark benchmark to examine the performance of small writes as seen in electronic mail, netnews and web-based commerce [30]. We apply version 1.5 with the option of synchronous writes added by FSL of Stony Brook Univ. The experiment duration varies depending on the efficiency of the requested operations In order to keep the runtime reasonable, we assume an initial set of 500 files and use 100 threads to apply a

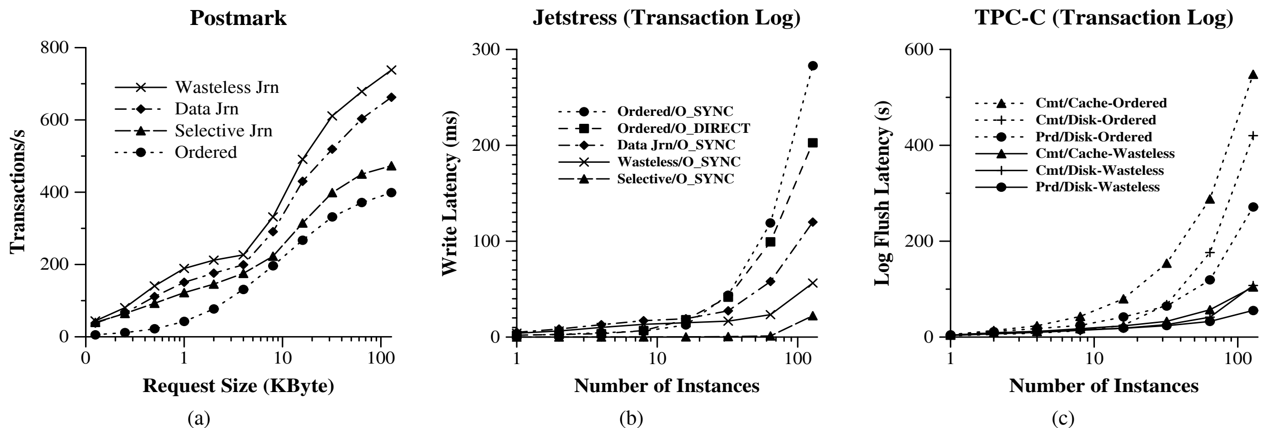

Figure 6: (a) With the Postmark benchmark, wasteless journaling consistently achieves the highest transaction rate. (b) At increasing number of concurrent Jetstress instances, selective journaling maintains the average latency of log writes up to orders of magnitude lower in comparison to the other modes. (c) Across the three different methods of flushing in MySQL/InnoDB, wasteless journaling requires lower latency to flush the transaction log to the disk.

total workload of 10,000 mixed transactions with file read, append, create and delete. We draw the file sizes from the default range between 500 bytes and 97.66KB, while I/O request sizes lie in the range between 128 bytes and 128KB. In Figure 6(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 used the *varmail* benchmark as another multithreaded application workload with small I/O requests. This is a mailserver benchmark provided by the Filebench suite [19] (Linux port by A. Gulati at Rice Univ.). For a duration of 500s, we run 80 threads over 20,000 initial files with 8KB average I/O request size, and 2KB average file size. In Table 1, we see the measured performance of ordered, selective and wasteless modes. We notice that wasteless and selective journaling, respectively, improve both the operation and data throughput of the ordered mode by 53% and 75%. We made similar observations with other small file sizes and request sizes that we tried. Thus, coalescing the small writes into the journal raises the transaction performance of the file system, while keeping at reasonable levels the journal traffic as we confirmed previously.

Varmail (Filebench)				
Mount Mode	**Operations (Total Count)**	**Ops Rate (Ops/s)**	**Data Rate (MB/s)**	**Lat (ms)**
Ordered	1,201,255	2388.0	5.5	108.7
Selective	1,857,699	3693.1	8.4	70.3
Wasteless	2,114,382	4203.7	9.6	62.0

Table 1: We evaluate the performance of the Varmail benchmark from the Filebench suite. With respect to ordered mode, wasteless journaling increases the transaction and data rate (higher is better) by 75%, and reduces the average operation latency by 43%.

5.3 Groupware and Database Logging

System administrators prefer to devote a separate device for the logs of an I/O intensive application to avoid performance bottlenecks [33]. Also, in distributed systems they are likely to place the log files locally at each machine for improved performance and autonomy [21]. Given the high cost of adding extra spindles to a system, we investigate the possibility of serving multiple log files from the same local device with appropriate file system support. For that purpose, we measure the latency to serve the I/O traffic of log traces that we gathered from groupware and database workloads.

Jetstress We consider the Jetstress Tool that emulates the disk I/O load of the Microsoft Exchange messaging and collaboration server [29]. We run Jetstress for two hours in a Windows Server 2003 system with 1GB RAM and two SATA disks in mirrored mode. We used 50 mailboxes with 100MB each and 1 operation per second for each mailbox. We choose these parameters so that we stress the hardware but also keep the reported measurements within acceptable levels to successfully pass the Jetstress test. The tool fixes the database cache to 256MB. Using the MS Process Monitor, we recorded a system-call trace of the Jetstress I/O activity. The I/O traffic of the database log contains appends of size from 512 bytes to tens of KB. The writes are *uncached*, i.e.,

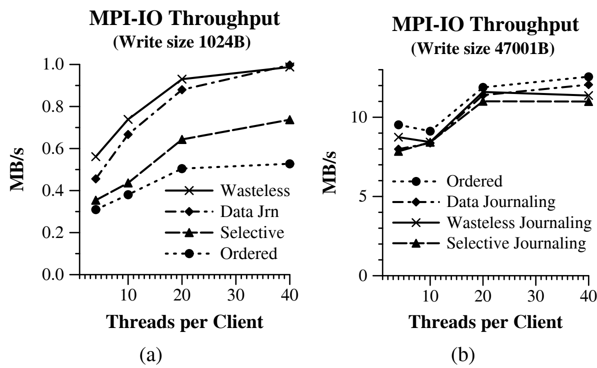

Figure 7: We measure the data throughput (higher is better) of MPI-IO over PVFS2 installed on a networked cluster with different mount modes at the data server. (a) At write size of 1KB, wasteless journaling almost doubles the performance of the default ordered mode. (b) At request size of 47001 bytes, the prevalence of writes above the write threshold keeps similar the behavior of the mount modes.

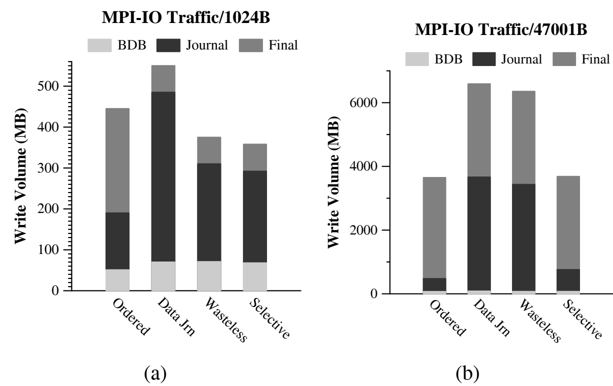

Figure 8: On the data server of PVFS2, we measure the write traffic (lower is better) to BerkeleyDB (BDB), the journal (Journal) and the file system (Final). (a) At write size 1KB, selective and wasteless journaling consume less bandwidth as they cut the journal volume of data journaling and the final volume of ordered mode. (b) At write size 47001 bytes, wasteless is similar to data journaling and selective comparable to ordered mode in terms of disk bandwidth consumption.

configured to bypass the buffer cache and directly reach the disk.

Over Linux, we use the original interarrival times to replay a 15min extract from the middle of the log trace. We consider different ext3 modes with the O_DIRECT option (at file *open*) to bypass the page cache, or the O_SYNC option for synchronous access. In order to study different loads and serve multiple logs from the same device, we varied the number of concurrent replays from 1 to 128. In Figure 6(b), both selective and wasteless journaling keep write latency up to tens of milliseconds even at high load. More specifically, selective is substantially better than wasteless. However, selective journaling is the only mode that distributes across both spindles the incoming appends to the log files. At high load, data journaling and ordered mode incur write latency that reaches hundreds of milliseconds, an order of magnitude longer than our two modes. These results indicate that the default uncached writes of Jetstress can be outperformed with appropriate file system support at comparable consistency.

TPC-C We also examine the OLTP performance benchmark TPC-C [45] as implemented in Test 2 of the Database Test Suite [15]. We used the MySQL open-source database system with the default InnoDB storage engine [34]. After consideration of our hardware capacity, we tested a configuration with 20 warehouses and 20 connections, 10 terminals per warehouse and 500s duration. Running the benchmark over Linux lead to insignificant differences of the transaction throughput among ordered mode, wasteless and selective journaling, mainly because most writes are above the threshold.

The InnoDB storage engine supports three different 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. It is considered the safest to avoid transaction loss in case of database, operating system or hardware failure. In method 0 (*Prd/Disk*), a performance improvement is expected by having the transaction log 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. A transaction loss is probable in case of operating system or hardware failure.

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. We measure the average latency to flush the transaction log to disk. In Figure 6(c), we see that wasteless journaling takes up to tens of seconds to complete each log 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. We also experimented with selective journaling (not shown to avoid clutter) and found it to lie close to wasteless journaling and well below ordered. The illustrated behavior is reasonable because wasteless (and selective) journaling stores the small appends of the database log into the file system journal at sequential disk throughput.

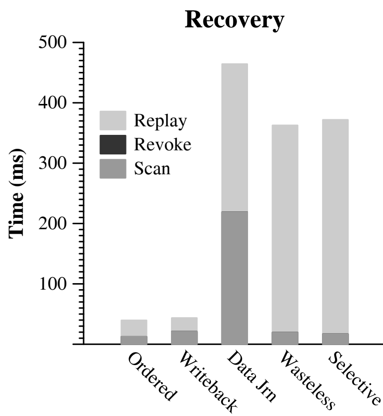

Figure 9: We examine the recovery time across different mount modes for a multithreaded workload. In comparison to data journaling, wasteless and selective journaling reduce scan time by an order of magnitude but increase the replay time by about 40%. Overall, they reduce by 20-22% the recovery time of data journaling.

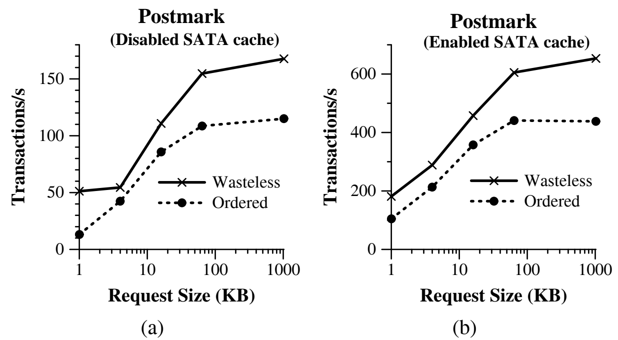

Figure 10: (a) We disable the on-disk caches to ensure that the writes only return after they touch the storage media. The relative advantage of wasteless journaling reaches several factors especially at small requests. (b) The comparative advantage is similar with the on-disk caches enabled.

5.4 MPI-IO over PVFS2

Workload characterization of parallel applications shows the need for improved performance in small I/O requests over small and large files that arise due to normal execution and checkpointing activity [10,24]. Especially small requests of 1KB are known to be problematic because they incur high rotational overhead even after they are transformed into sequential [37]. Also, writes of 47001 bytes appear often in parallel applications and lead to poor performance due to alignment misfit [6]. In the present section we examine the performance gain of a parallel multi-tier configuration, where the storage server runs our mount modes. We chose the PVFS2 as an open-source scalable parallel file system [39]. We configured a networked cluster of fifteen quad-core machines with thirteen clients, one PVFS2 *data server* and one PVFS2 *metadata server*. By default, each server uses a local BerkeleyDB database to maintain local metadata. Through system-call tracing, we observed that the data server uses a single thread for local metadata updates and multiple threads for data updates. To focus our study on multistream workloads, 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 used the default thread-based asynchronous I/O of PVFS2. Also, 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 [1]. In our configuration each process writes to a separate unique file ("N processors to N files"). According to other studies, this is the write pattern suggested to application developers for best performance [6]. 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 with alternative write sizes of 1024 and 47001 bytes. In Figure 7, we compare the data throughput of MPI-IO across different write sizes and loads. With 1KB writes, wasteless journaling almost doubles the throughput of ordered mode, while data journaling and selective lie between the previous two. With writes of 47001 bytes, the write throughput remains about the same across the different modes. In Figure 8, we depict the total volume of write traffic across the BerkeleyDB, the journal and the file system. We observe that wasteless journaling reduces by 42% the journal traffic of data journaling, while selective journaling closely tracks ordered mode in terms of write volume (Figure 8(b)).

In summary, wasteless and selective journaling improve substantially the performance of ordered mode at small writes while they avoid the excessive journal traffic of data journaling. At larger write sizes, performance remains similar across the mount modes, while the volume of journal traffic varies up to several factors.

5.5 Recovery Time

In a different experiment, we evaluate the ability of the system to recover quickly after a system crash that leaves log records in the journal. It was reported that when the free jounal space lies between $\frac{1}{4}$ and $\frac{1}{2}$ of the journal size, the original ext3 system automatically checkpoints the updates to the final location [38]. In order to do a fair comparison across the different modes, we use writes

that are small enough to prevent checkpointing before the crash, but also useful for some application classes, e.g., event stream processing [9]. Thus, we start 100 threads each doing 100 writes of request size 8 bytes. Then we cut the power to the system. At the subsequent reboot, we measure within the kernel the duration of 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. In comparison to data journaling, the *scan* pass of selective and wasteless journaling is an order of magnitude shorter. Respectively, the *replay* pass of selective and wasteless journaling takes about 40% more time due to the extra block reads involved. Overall, selective and wasteless journaling reduce by 20-22% the recovery time of data journaling.

5.6 Device Issues

We examine the performance effects from disabling the on-disk caches on a server with two 7.2KRPM SATA disks. First, we consider the workloads of Figures 5(a) and 5(b) with streams of rates 1Kbps and 1Mbps, respectively. Based on the average write latency across different loads, we find that the disabled on-disk caches do not make any performance difference in comparison to the enabled. This result makes sense because the streaming writes continuously append new data to the caches and don't reuse the blocks that they wrote in the past. Additionally, the number of streams (at least 10) also limits the potential caching benefits. Then, we also measure the transaction rate achieved by the Postmark workload of Figure 6(a). As shown in Figures 10(a) and (b), the disabled write caches across the two mount modes reduce the transaction rate four times in comparison to the enabled caches. Nevertheless, wasteless journaling consistently maintains up to several factors higher transaction rate with respect to the ordered mode across both caching settings.

Arguably, wasteless journaling can take advantage of the two spindles that store the journal and the file system, while the ordered mode uses mostly the spindle of the file system and less the spindle of the journal. To study this asymmetry, we also run our stream microbenchmarks over two SAS disks in RAID0 configuration with hardware controller support. We examine the two modes with the journal instantiated as a hidden file rather than a separate partition. With 1Kbps streams over RAID0, the write latency of ordered mode drops to half, while the write latency of wasteless does not change. Nevertheless, wasteless journaling remains one to two orders of magnitude faster than ordered mode across different numbers of streams. Also, wasteless journaling is up to an order of magnitude faster than ordered mode with 1Mbps streams.

6 Related Work

The log-structured file system addresses the synchronous metadata update problem and the small-write problem by coalescing data writes sequentially to a segmented log [40]. We experimentally observe (Section 5.1) that the log-structured approach may adversely affect the read performance, while previous research reported cleaning overheads and performance limitations under particular workloads [43]. The virtual log is another effort to minimize the latency of small synchronous writes [49]. A modified version of the log-structured file system called StreamFS has been recently used for the storage of high-volume streams [16]. StreamFS writes incoming streams of high rates in a circular fashion along the disk space selectively overwriting older data. Instead, we also handle the storage traffic of multiple streams with low-rate requirements. The hFS file system stores metadata and small files in a separate partition from large files. It treats writes according to the size of the written file rather than the size of the writes themselves that we do [51]. DualFS is a journaled file system that manages the data blocks in groups similarly to other systems, but it stores the metadata separately in a log-structured file system [36].

In real-time processing of huge amounts of data, many applications use multiple operators and require recovery from failures [31]. Recovery is possible through synchronous logging but incurs prohibitive latency cost. Recent research combines software transactional memory with asynchronous logging to optimistically parallelize stream operators [9]. However, such speculative execution is limited to processing operators that do not perform external actions such as I/O [11]. Journaling remains standard recovery feature in the file system of a virtualization product vendor [48]. If we run multiple virtual machines on the same system, the block-based interface of the file system makes small writes from guests appear as full-block updates to the underlying host [4]. In ongoing work, we investigate the applicability of our methods to virtualization environments by relaxing the file system interface between the guest and the host.

Given the importance of small write efficiency, several approaches improve the internal organization of the file system, the storage device, and their in-between communication. High-performance synchronous writes can be supported through specialized hardware, such as battery-backed main memory (NVRAM) [14]. However, previous research also reported that NVRAM creates a single point of failure over disk arrays, while dual-copy NVRAM cache can be costly [27]. Alternatively, disk-specific knowledge can be exploited to align the data accesses on track boundaries, and avoid rotational latency and track-crossing overhead [42]. Knowledge of disk geometry could also be beneficial in our journaling policies, if we transfer recent updates to their final disk

location in the file system. Moreover, our target to reduce storage traffic and prefer sequential writes makes our work compatible with the lifetime and performance limitations of novel devices such as solid state disks [13].

Anand et al. introduce range writes in the disk interface to remove the need for file system micromanagement of block placement [2]. The authors applied range writes to the journal update of Linux ext3 in order to avoid the rotations and improve the response time of synchronous writes. This approach operates at the disk level and could complement our methods, as well. Xsyncfs introduces externally synchronous I/O that guarantees durability to an external observer of application output rather than the application itself [35]; if an application does not produce output, xsyncfs commits data periodically in the way of an asynchronously mounted ext3. WAFL 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 [26]. Instead, we safely delay file system updates at low cost through batching of small writes into the journal.

In early work, Hagmann described metadata update logging in the Cedar File System to improve performance and achieve consistency [23]. Soft updates track and enforce metadata update dependencies so that the file system can safely delay writes for most file operations [20,44]. Both the above systems focus on metadata rather than data updates that we instead investigate. Subpage updates have been previously handled efficiently in the context of distributed shared memory by the Millipage system [28]. In transaction processing, group commit is a known database logging optimization that periodically flushes to the log multiple outstanding commit requests in groups rather than individually [17]. Instead, we introduce wasteless and selective journaling as a general file-system service to support applications with small writes.

7 Discussion

Over a range of operating conditions that are vital for system reliability, existing file systems can be either wasteful or underperforming. We propose and implement several improvements that address these weaknesses without penalizing the general behavior of the file system beyond a reasonable increase in disk traffic. The main theme in our proposed design is to improve performance and reliability at low cost. Thus, adding extra spindles to improve I/O parallelism or a properly-sized NVRAM to absorb small writes, are alternative approaches likely to reduce latency and raise throughput. However, such solutions carry some notable drawbacks that primarily have to do with increased cost and maintenance concerns about additional faulty parts in the system.

8 Conclusions

Journaling is a technique commonly used in current file systems to ensure their fast recovery in case of system failures. In the present work, we rely on journaling of data updates in order to ensure their safe transfer to disk at low latency and high throughput without storage bandwidth waste. We design and implement a method that we call wasteless journaling to merge concurrent subpage writes to the journal into page-size blocks. Additionally, we develop the selective journaling method that only logs updates below a write threshold and transfers the rest directly to the file system. Our experimental results include measurements from streaming microbenchmarks, application-level workloads, database logging traces and multistream I/O over a parallel file system in the local network. Across different cases, we demonstrate reduced write latency and recovery time, improved transaction throughput with low journal bandwidth requirements. Our plans for future work include extension of the above methods for virtualization environments.

9 Acknowledgments

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

References

[1] The los alamos national lab mpi-io test. http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm.

[2] ANAND, A., SEN, S., KRIOUKOV, A., POPOVICI, F. I., AKELLA, A., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND BANERJEE, S. Avoiding file system micromanagement with range writes. In *USENIX OSDI* (San Diego, CA, 2008), pp. 161–176.

[3] APPUSWAMY, R., VAN MOOLENBROEK, D. C., AND TANENBAUM, A. S. Block-level raid is dead. In *Workshop on Hot Topics in Storage in File Systems* (Boston, MA, 2010).

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *ACM SOSP* (2003), pp. 164–177.

[5] BATSAKIS, A., BURNS, R. C., KANEVSKY, A., LENTINI, J., AND TALPEY, T. Awol: An adaptive write optimizations layer. In *USENIX FAST* (Feb. 2008), pp. 67–80.

[6] 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 *SC* (2009), pp. 1–12.

[7] 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.

[8] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, third ed. O'Reilly Media, Sebastopol, CA, Nov. 2005.

[9] BRITO, A., FETZER, C., AND FELBER, P. Minimizing latency in fault-tolerant distributed stream processing systems. In *Intl Conference on Distributed Computing Systems* (Montreal, QC, 2009), pp. 173–182.

[10] CARNS, P., LANG, S., ROSS, R., VILAYANNUR, M., KUNKEL, J., AND LUDWIG, T. Small-file access in parallel file systems. In *IEEE IPDPS* (May 2009), pp. 1–11.

[11] CHANDRASEKARAN, S., AND FRANKLIN, M. Remembrance of streams past: Overload-sensitive management of archived streams. In *VLDB Conference* (Toronto, Canada, Aug. 2004), pp. 348–359.

[12] 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.

[13] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS/Performance* (Seattle, WA, 2009), pp. 181–192.

[14] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The rio file cache: Surviving operating system crashes. In *ACM ASPLOS* (Cambridge, MA, 1996), pp. 74–83.

[15] Database test suite. http://osdldbt.sourceforge.net/.

[16] DESNOYERS, P. J., AND SHENOY, P. Hyperion: High volume stream archival for retrospective querying. In *USENIX ATC* (Santa Clara, CA, June 2007), pp. 45–58.

[17] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation techniques for main memory database systems. In *ACM SIGMOD* (Boston, MA, 1984), pp. 1–8.

[18] ELNOZAHY, E. N., AND PLANK, J. S. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing 1*, 2 (2004), 97–108.

[19] http://www.solarisinternals.com/wiki/index.php/FileBench.

[20] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems 18*, 1 (Feb. 2000), 127–153.

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

[22] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with i/o shepherding. In *ACM SOSP* (Stevenson, WA, Oct. 2007), pp. 293–306.

[23] HAGMANN, R. Reimplementing the cedar file system using logging and group commit. In *ACM SOSP* (Austin, TX, 1987), pp. 155–162.

[24] HILDEBRAND, D., WARD, L., AND HONEYMAN, P. Large files, small writes, and pnfs. In *ACM Intl Conf. on Supercomputing* (Cairns, Australia, June 2006), pp. 116–124.

[25] HISGEN, A., BIRRELL, A., JERIAN, C., MANN, T., AND SWART, G. New-value logging in the echo replicated file system. Tech. rep., Digital Equipment Corporation, Palo Alto, CA, 1993. SRC Research Report 104.

[26] 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.

[27] HU, Y., NIGHTINGALE, T., AND YANG, Q. Rapid-cache–a reliable and inexpensive write cache for high performance storage systems. *IEEE Transactions on Parallel and Distributed Systems 13*, 3 (Mar. 2002), 290–307.

[28] ITZKOVITZ, A., AND SCHUSTER, A. Multiview and millipage - fine-grain sharing in page-based dsms. In *USENIX OSDI* (New Orleans, LA, Feb. 1999), pp. 215–228.

[29] Microsoft exchange server jetstress tool, 2007. http://technet.microsoft.com/en-us/library/bb643093.aspx.

[30] KATCHER, J. Postmark: A new file system benchmark. Tech. Rep. TR-3022, NetApp, 1997.

[31] KWON, Y., BALAZINSKA, M., AND GREENSBERG, A. Fault-tolerant stream processing using a distributed, replicated file system. In *VLDB Conference* (Auckland, New Zeland, Aug. 2008), pp. 574–585.

[32] MAMMARELLA, M., HOVSEPIAN, S., AND KOHLER, E. Modular data storage with anvil. In *ACM SOSP* (Oct. 2009), pp. 147–160.

[33] MULLINS, C. S. *Database Administration: The Complete Guide to Practices and Procedures*. Addison Wesley, 2002, ch. 11. Database Performance (Database Log Placement), p. 308.

[34] http://www.mysql.com/.

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

[36] PIERNAS, J., CORTES, T., AND GARCIA, J. M. The design of new journaling file systems: The dualfs case. *IEEE Transactions on Computers 56*, 2 (Feb. 2007), 267–281.

[37] POLTE, M., SIMSA, J., TANTISIRIROJ, W., GIBSON, G., DAYAL, S., CHAINANI, M., AND UPPUGANDLA, D. K. Fast log-based concurrent writing of checkpoints. In *Petascale Data Storage Workshop* (Nov. 2008).

[38] 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.

[39] Parallel virtual file system, version 2. http://www.pvfs.org.

[40] 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.

[41] SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. Lightweight recoverable virtual memory. In *ACM SIGOPS* (Asheville, NC, Dec. 1993), pp. 146–160.

[42] SCHINDLER, J., GRIFFIN, J. L., LUMB, C. R., AND GANGER, G. R. Track-aligned extents: Matching access patterns to disk drive characteristics. In *USENIX FAST* (Monterey, CA, Jan. 2002), pp. 259–274.

[43] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: a performance comparison. In *USENIX ATC* (1995), pp. 21–21.

[44] 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 *USENIX ATC* (San Diego, CA, 2000), pp. 71–84.

[45] Tpc benchmark c standard specification. Tech. rep., Transaction Processing Council, 1992. Technical Report.

[46] TWEEDIE, S. C. Journaling the linux ext2fs filesystem. In *LinuxExpo* (Durham, NC, 1998), pp. 25–29.

[47] VERISSIMO, P., AND RODRIGUES, L. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[48] Vmware virtual machine file system: Technical overview and best practices. Tech. rep., VMware, Inc, Palo Alto, CA, 2007. White Paper.

[49] WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. Virtual log based file systems for a programmable disk. In *USENIX OSDI* (New Orleans, LA, 1999), pp. 29–43.

[50] YOSHIJI, A., KONISHI, R., SATO, K., HIFUMI, H., TAMURA, Y., KIHARA, S., AND MORIAI, S. Nilfs - continuous snapshotting filesystem for linux, 2009. Nippon Telegraph and Telephone Corporation, http://www.nilfs.org/en/.

[51] ZHANG, Z., AND GHOSE, K. hfs: a hybrid file system prototype for improving small file and metadata performance. In *ACM EuroSys Conference* (Lisboa, Portugal, Mar. 2007), pp. 175–187.