Improving Bandwidth Efficiency for Consistent Multistream Storage

ANDROMACHI HATZIELEFTHERIOU and STERGIOS V. ANASTASIADIS,
University of Ioannina

Synchronous small writes play a critical role in system availability because they safely log recent state modifications for fast recovery from crashes. Demanding systems typically dedicate separate devices to logging for adequate performance during normal operation and redundancy during state reconstruction. However, storage stacks enforce page-sized granularity in data transfers from memory to disk. Thus, they consume excessive storage bandwidth to handle small writes, which hurts performance. The problem becomes worse, as filesystems often handle multiple concurrent streams, which effectively generate random I/O traffic. In a journaled filesystem, we introduce *wasteless journaling* as a mount mode that coalesces synchronous concurrent small writes of data into full page-sized journal blocks. Additionally, we propose *selective journaling* to automatically activate wasteless journaling on data writes with size below a fixed threshold. We implemented a functional prototype of our design over a widely-used filesystem. Our modes are compared against existing methods using microbenchmarks and application-level workloads on stand-alone servers and a multitier networked system. We examine synchronous and asynchronous writes. Coalescing small data updates to the journal sequentially preserves filesystem consistency while it reduces consumed bandwidth up to several factors, decreases recovery time up to 22%, and lowers write latency up to orders of magnitude.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management; D.4.5 [**Operating Systems**]: Reliability; D.4.8 [**Operating Systems**]: Performance

General Terms: Design, Experimentation, Measurement, Performance, Reliability

Additional Key Words and Phrases: Journaling, logging, concurrency, small writes

ACM Reference Format:
Hatzieleftheriou, A. and Anastasiadis, S. V. 2013. Improving bandwidth efficiency for consistent multistream storage. *ACM Trans. Storage* 9, 1, Article 2 (March 2013), 27 pages.
DOI:http://dx.doi.org/10.1145/2435204.2435206

1. INTRODUCTION

Synchronous small writes lie in the critical path of several systems that target fast recovery from failures with low performance loss during normal operation [Anand et al. 2008; Bent et al. 2009; Chang et al. 2006; Fryer et al. 2012; Gray and Reuter 1993; Hagmann 1987; Hitz et al. 1994; Kwon et al. 2008; Mammarella et al. 2009; Shin et al. 2011]. Before modifying the system state, updates are recorded to a sequential file (*write-ahead log*). Periodically the entire system state (*checkpoint*) is copied to permanent storage. After a transient failure, the lost state is reconstructed by replaying recent logged updates against the latest checkpoint [Verissimo and Rodrigues 2001].

The present research was supported in part by project INTERSAFE 303090/YD763 of the INTERREG IIIA program co-funded by the EU and the Greek State.
Authors' address: A. Hatzieleftheriou and S. V. Anastasiadis (corresponding author), Department of Computer Science, University of Ioannina, Ioannina 45110, Greece; email: stergios@cs.uoi.gr.

Write-ahead logging improves system availability by preserving state from failures and substantially reducing recovery time. It is a method widely applied in general-purpose file systems [Hitz et al. 1994; Prabhakaran et al. 2005a; Rosenblum and Ousterhout 1992; Seltzer et al. 2000]; relational databases [Gray and Reuter 1993]; key-value stores [Chang et al. 2006; Mammarella et al. 2009]; event processing engines [Brito et al. 2009; Kwon et al. 2008]; and other mission-critical systems [Borthakur et al. 2011; Calder et al. 2011; Nightingale et al. 2006]. Logging is also one technique applied during the checkpointing of parallel applications to avoid discarding the processing of multiple hours or days after an application or system crash [Bent et al. 2009; Ouyang et al. 2011b; Polte et al. 2008]. Logging incurs synchronous small writes, which are likely to create a performance bottleneck on disk [Appuswamy et al. 2010; Gray and Reuter 1993; Min et al. 2012; Nightingale et al. 2006; Wang et al. 1999]. Thus, the logging bandwidth is typically overprovisioned by placing the log file on a dedicated disk separately from the devices that store the system state (e.g., relational databases [Mullins 2002] and Azure [Calder et al. 2011]). In general, asynchronous writes also behave as synchronous if an I/O-intensive application modifies pages at the flushing rate of the underlying disk [Batsakis et al. 2008].

Furthermore, a distributed service is likely to maintain numerous independent log files at each server (RVM [Satyanarayanan et al. 1993]; Megastore [Baker et al. 2011]; and Azure [Calder et al. 2011]). For instance, multiple logs facilitate the balanced load redistribution after a server failure in a storage system. If the logs are concurrently accessed on the same device, random I/O is effectively generated leading to long queues and respective delays. This inefficiency remains even if the logs are stored over a distributed filesystem across multiple servers. One solution is to manage the multiple logs of each server as a single file (e.g., Bigtable over GFS [Chang et al. 2006] and HBase over HDFS [Borthakur et al. 2011]). Then, individual logs have to be separated from each other at the cost of extra software complexity and processing delay during recovery.

For the needs of high-performance computing, special file formats and interposition software layers have been developed to efficiently store the data streams generated by multiple parallel processes [Bent et al. 2009; Elnozahy and Plank 2004; Hildebrand et al. 2006; Polte et al. 2008]. In structures optimized for multicore key-value storage, the server thread running on each core maintains its own separate log file [Mao et al. 2012]. For higher total log throughput, it is recommended that different logs are stored on different magnetic or solid-state drives. However, fully replacing hard disks with flash-based solid-state drives is currently not considered a cost-effective option for several server workloads [Narayanan et al. 2009]. Also, while the storage density of flash memory continues to improve, important metrics such as reliability, endurance, and performance of flash memory are currently declining [Grupp et al. 2012]. Solutions based on specialized hardware are further discussed in Section 7. In the present work, we address the I/O inefficiency of concurrent stream writing through enhancements in the write path of the underlying filesystem over low-cost hardware.

Journaled filesystems use a log file (*journal*) to copy data and/or metadata from memory to disk [Hisgen et al. 1993; Seltzer et al. 1995; Tweedie 1998; Zhang and Ghose 2007]. Update records are safely appended to the journal at sequential throughput, and costly filesystem modifications are postponed without penalizing the write latency perceived by the user. A page cache temporarily stores recently accessed data and metadata; it receives byte-range requests from applications and forwards them to disk in the form of page-sized requests [Bovet and Cesati 2005]. The page granularity of disk accesses is prevalent across all storage transfers, including data and metadata writes to the filesystem and the journal. In the case of asynchronous small writes, the

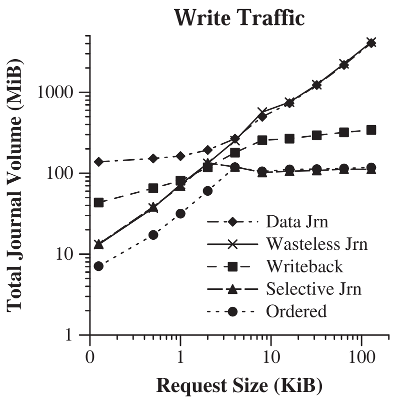

Fig. 1. For a duration of 5min, we use 100 threads over the Linux ext3 filesystem to do periodic synchronous writes at fixed request size (each thread writes 1req/s). We measure the total write traffic to the journal across different mount modes (fully explained in Sections 2 and 3).

disk efficiency is improved, as multiple consecutive requests are coalesced into a page before being flushed to disk. In contrast, synchronous small writes are flushed to disk individually, causing costly random I/O of data and metadata page transfers.

In Figure 1, we measure the amount of data written to the journal of the ext3 filesystem (described in Section 3). We run a synthetic workload of 100 concurrent threads for 5min. Each thread generates periodic synchronous writes of fixed request size at rate 1req/s. We include the ordered and writeback modes along with the data journaling mode. The ordered and writeback modes incur lower traffic because they only write to the journal the blocks that contain modified metadata. Instead, data journaling writes to the journal the entire modified data and metadata blocks. In Figure 1, as the request size drops from 4KiB to 128 bytes (by a factor of 32), the total journal traffic of data journaling only decreases from 267MiB to 138MiB (by about a factor of 2). Thus, data journaling incurs a relatively high amount of journal traffic at subpage requests. By tracing the block transfer activity of ext3 in the Linux kernel, we found that metadata and data modifications are journaled in granularity of entire 4KiB pages regardless of how many bytes are actually modified in a file page.

According to Chidambaram et al. [2012], after a system crash, the data journaling mode correctly associates metadata with data of the matching version (version consistency). This type of crash consistency is stronger than only keeping the metadata structures consistent with each other (metadata consistency), or associating the data blocks with the file they belong to (data consistency). However, data journaling requires each data update to be written twice, first in the journal and later in the filesystem. If it involves a large amount of written data or numerous small writes, double writes lead to inefficient utilization of storage bandwidth that may hurt performance. Consequently, the use of data journaling is explicitly discouraged in several cases, while production systems activate only metadata journaling by default [Anand et al. 2008; Chidambaram et al. 2012; Prabhakaran et al. 2005a; Rajimwale et al. 2011; Zhang and Ghose 2007].

Today, journaling of both data and metadata is applied in a production distributed filesystem to ensure consistency of the on-disk state [Weil et al. 2006]. Additionally, data journaling is desirable for increased consistency across several production environments, including the native filesystem of I/O-intensive high-performance computing systems [Oral et al. 2010], and the host filesystem holding the disk images of virtual machines running write-dominated workloads [Le et al. 2012]. In general, write-optimized filesystems that improve random access performance are increasingly important for networked environments [Leung et al. 2008].

In the present study, we investigate the performance and consistency implications of storage bandwidth consumption in journaled and other filesystems. In the case of data journaling, we find that the excessive disk traffic of synchronous small writes is primarily a result of the page granularity enforced by the storage stack and less a consequence of writes to both the journal and the filesystem. In fact, journaling may actually improve performance because it safely copies updates to disk at sequential throughput. The bandwidth inefficiency of small writes is not trivially overcome by reducing the granularity of disk writes to a single sector because smaller writes would cause higher I/O overhead in the system. Instead, one promising solution is to accumulate the modifications from multiple subpage updates from different threads into a single page, and only pay once the disk I/O cost of the page write. This approach cannot be directly applied to writes that modify the filesystem in-place because each write corresponds to a different block on disk. However, it is applicable to the updates appended into the journal.

We set as an objective achieving filesystem consistency at high I/O performance with efficient bandwidth utilization. We propose, design, and fully implement two new mount modes, *wasteless journaling* and *selective journaling*. We are mainly concerned about highly concurrent multithreaded workloads that synchronously apply small writes over the same storage devices [Anand et al. 2008; Bent et al. 2009; Borthakur et al. 2011; Calder et al. 2011; Chang et al. 2006; Kwon et al. 2008; Nightingale et al. 2006; Shin et al. 2011]. Unnecessary writes of unmodified data and writes of high positioning overhead occupy valuable disk access time. Thus they waste disk bandwidth, which should preferably be spent on useful data transfers. With microbenchmarks and application-level workloads, we show that our two modes can considerably reduce the journal (and filesystem) traffic. More importantly, they improve operation throughput and substantially reduce the response time in comparison to alternative mount options and filesystems.

The general idea of subpage logging is not new. Previously, researchers at DEC prototyped and used the Echo distributed filesystem [Birrell et al. 1993]. For improved performance and availability, Echo logged subpage updates, and bypassed the logging of page-sized or larger writes [Hisgen et al. 1993]. The development of Echo was discontinued in early 1992, partly because it ran on hardware that lacked fast-enough computation relative to communication. Recent research introduced semantic trace playback (STP) to rapidly evaluate alternative filesystem designs without the cost of real system implementation or detailed filesystem simulation [Prabhakaran et al. 2005a]. STP was used to emulate journaling of block modifications instead of entire modified blocks in a filesystem. Although the authors showed a reduced amount of data written to the journal, they did not examine the general performance and recovery implications. Due to the obsolete hardware characteristics or the high emulation level of the above studies, they leave questionable the general architectural fit and actual performance benefit of journal bandwidth reduction in current filesystems.

To the best of our knowledge, the present work is the first to comprehensively investigate the general benefits of subpage data journaling using a prototype implementation in a fully operational filesystem. We summarize our contributions as follows.

(1) We measure bandwidth inefficiencies in journaled filesystems and examine ways to combine filesystem consistency with high performance at moderate cost.
(2) We design and fully implement wasteless and selective journaling as optional mount modes in a widely used filesystem.

(3) We discuss the implications of alternative journaling optimizations to the consistency semantics of the filesystem in the context of different storage configurations.
(4) We apply micro-benchmarks, storage workloads, and database logging traces over a journal spindle to demonstrate performance improvements up to several orders of magnitude across different metrics.
(5) With a parallel filesystem, we show that wasteless journaling doubles the throughput of parallel checkpointing over small writes, while it reduces the total traffic to disk.

In Section 2 we present architectural aspects of our design, while in Section 3 we describe the implementation of wasteless and selective journaling. In Section 4 we explain our experimentation environment, and in Section 5 we present detailed measurements across different workloads. In Section 6 we summarize previous related work, and in Section 7 we consider our results in the context of virtualization and solid-state drives. Finally, in Section 8 we outline our conclusions and plans for future work.

2. SYSTEM DESIGN

In this section we describe the basic assumptions and objectives of our journaling architecture. In a general-purpose filesystem, we aim to safely store recent state updates on disk and ensure their consistent recovery in case of failure. We also strive to serve synchronous small writes and subsequent reads fast, with low bandwidth requirements. The consistency of metadata updates has already been studied previously [Hagmann 1987; Seltzer et al. 2000]. Additionally, subpage journaling of metadata updates is made widely available today through popular commercial filesystems, such as the IBM JFS and MS NTFS [Prabhakaran et al. 2005a]. On the contrary, data journaling is only supported in few filesystems (e.g., ext3/4, ReiserFS) and its use is generally avoided because it is considered harmful for performance [Chidambaram et al. 2012]. Moreover, the subpage journaling of data updates is not supported in current filesystems.

2.1. Wasteless Journaling

Historically, journaling was only applied to the metadata of a filesystem with specific goal to ensure fast structural recovery after a system failure [Hagmann 1987; Tweedie 1998]. Today, support for data journaling is provided in few filesystems to preserve the latest data updates from a system crash and keep them accessible [Chidambaram et al. 2012]. As a side effect of the journal sequential access, data journaling can improve the throughput of random I/O operations. However, this benefit is realized at the cost of excessive bandwidth consumption due to the page granularity of the storage traffic [Appuswamy et al. 2010; Prabhakaran et al. 2005a]. In order to overcome this limitation, we designed and implemented a new mount mode that we call *wasteless journaling*. In synchronous writes, we transform partially modified data blocks into descriptor records, which we subsequently accumulate into special journal blocks (Figure 2(a)). For data blocks that have been fully modified by write operations, we synchronously copy the entire blocks from memory to the journal. After timeout expiration or due to shortage of journal space, we copy the partially or fully modified data blocks from memory to their final location in the filesystem. Subsequently, we clean the respective records from the journal device.

2.2. Selective Journaling

Data journaling adds extra I/O cost because it writes data to both the journal and the filesystem. In the particular case of sequential writes, the benefit from sequential

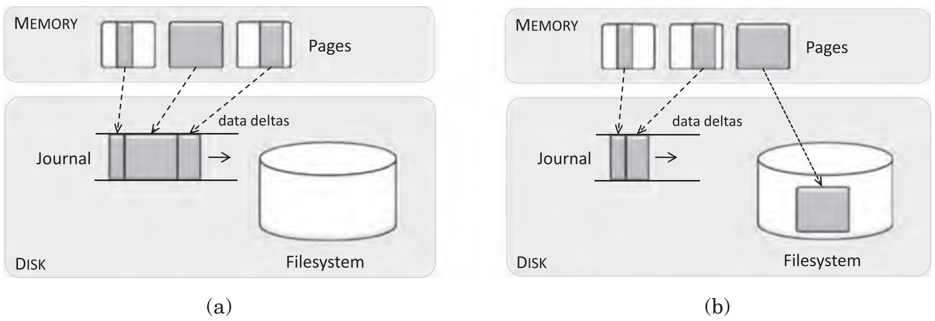

Fig. 2. (a) In wasteless journaling, we journal data updates at subpage granularity. (b) In selective journaling, while we treat small requests approximately as in wasteless mode, we transfer large requests directly to the final location without prior journaling of the data.

appends to the journal device is not that significant, because the data writes to the filesystem are also most likely sequential. In this case, the journal device may actually become a bottleneck and harm performance. With goal to reduce the journal I/O activity of sequential writes, we evolved wasteless journaling into an alternative mount mode that we call *selective journaling*. In this 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 the final disk location directly (Figure 2(b)). The rationale of this approach is to apply data journaling only when multiple small writes can be coalesced into a single journal block, or different data blocks are fully modified and scattered across multiple locations in the filesystem.

2.3. Consistency

Since the synchronous write from a single thread must be transferred to disk immediately, it only makes sense to accumulate into a journal block the writes that originate from different concurrent threads. Therefore, we expect wasteless and selective journaling to be mostly beneficial in environments that consist of multiple writing streams with frequent small writes. In the case of wasteless journaling, we only consider a write operation effectively completed after we log both data and metadata into the journal device. 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. Thus, the structure of the filesystem remains consistent across system failures, and the filesystem metadata refers to the latest data that has been safely stored on disk (version consistency [Chidambaram et al. 2012]).

Selective journaling allows a series of synchronous writes 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 filesystem. During a recovery from crash, a write operation is fully aborted if the corresponding journal appends were interrupted halfway. However, if the write is large enough to be directly transferred to the final location, it may be only partially completed at the instance of the failure. Consequently, selective journaling provides the consistency of mount modes which journal the metadata only after the respective data is saved to disk (data consistency [Chidambaram et al. 2012]). Such modes update the data in place and add metadata modifications to the journal, while selective journaling applies large data updates in place but adds to the journal both metadata modifications and small data updates. If the write traffic is dominated by request sizes below the write threshold, the consistency of selective journaling approaches the version consistency of wasteless journaling.

Arguably, the accumulation of multiple small updates into a single journal block leaves open the possibility of losing multiple updates if the block does not safely reach the journal device. However, the wasteless and selective journaling do not defer in any way the operation of buffer flushing, regardless of whether it is periodically invoked by the filesystem or explicitly requested through synchronous writes. Instead, the two modes merely flush the buffer updates faster because they reduce the amount of I/O involved. This efficiency allows applications to achieve decreased write latency and a shorter vulnerability window during which requested updates remain outstanding. We provide additional explanations about the system consistency, when we describe the atomicity guarantees of our implementation in Section 3.4, while we experimentally demonstrate the reduced flushing latency of our modes in Section 5.

2.4. Update Sequences

In selective journaling, we call *update sequence* a series of multiple incoming updates applied to the buffer of a single data block. The update sequence terminates when the modified data block, along with the respective metadata blocks, are safely transferred from memory to the filesystem. In our definition, the updates are not necessarily back-to-back, but there should be no in-between transfer of the respective data and metadata blocks to the final disk location. For presentation simplicity, but without loss of generality, we assume that the write threshold and the block size are both set equal to the size of a system page.

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 the block as the individual updates are flushed to disk. It would be a straightforward approach to turn off the journaling of the block as soon as the subpage write switched into a full overwrite along an update sequence. As a result, the initial updates of the block would be journaled and the rest would be directly flushed to the filesystem. During journal replay at a subsequent recovery from a failure, the subpage writes would erroneously corrupt the block whose latest update fully overwrote it. We deliberately avoid this situation by journaling the block throughout the update sequence at the cost of paying data journaling I/O for the entire update sequence.

On the other hand, if the first update is page-sized, we skip journaling the entire update sequence of the block. This implies that flushing any subsequent subpage data write to disk causes the entire data block to be flushed to the filesystem and the respective metadata blocks written to the journal. If we trivially did not do that, then only the subpage writes could be recorded in the journal. Our approach in this case sacrifices the sequential I/O of journaling in order to avoid block corruption due to only replaying the subpage writes of the update sequence after a failure.

We prefer to preserve clean recovery semantics in selective journaling at the cost of lower performance gain. In our experience, the two above transitions in write size along an update sequence are not common in practice. Also, an update sequence has limited lifetime due to the periodic flushing of dirty data by the system. According to our experiments, selective journaling maintains significant performance gains across different representative workloads that we examined. In Figure 3, we use a flowchart to summarize the possible execution paths of a write request through selective journaling.

3. PROTOTYPE IMPLEMENTATION

We implemented wasteless and selective journaling in the Okeanos prototype system over Linux ext3 [Bovet and Cesati 2005; Tweedie 1998]. At a high level, the original

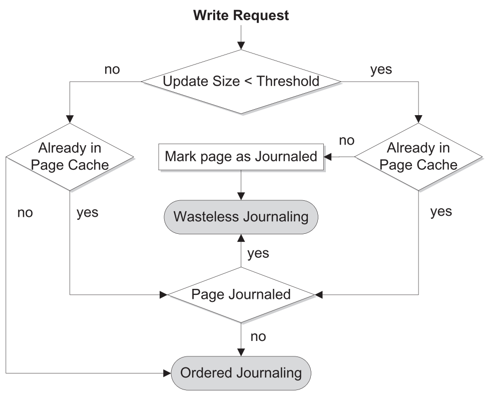

Fig. 3. Alternative execution paths of a write request in the selective journaling mode.

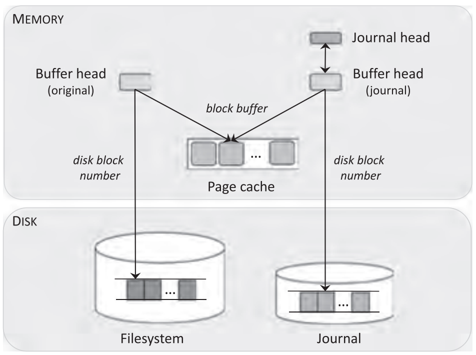

Fig. 4. For each journaled block (i) A dedicated buffer head in memory specifies the respective disk block in the journal device and, (ii) A journal head in memory links the block with the journal transaction to which it belongs.

ext3 filesystem implements journaling of updates in two steps. First, it copies the modified blocks into the journal with a a commit block at the end. Then, it updates the modified blocks in-place at the filesystem and discards the journal blocks. If the filesystem is mounted in *data journaling* mode, both data and metadata blocks are copied to the journal before th filesystem update. The *ordered* mode copies only metadata blocks to the journal after it has stored the associated data blocks at the filesystem. This order ensures that a file structure points to valid data blocks on disk. The *writeback* mode copies only metadata blocks to the journal without any constraints in the relative order at which data and metadata blocks update the filesystem. It is considered the fastest mode, but also the weakest in terms of consistency.

3.1. Buffers

The Linux kernel uses the *page cache* to keep the data and metadata of recently accessed disk files in memory [Bovet and Cesati 2005]. For every cached disk block, a *block buffer* in memory stores the respective data, while a *buffer head* stores the related bookkeeping information (Figure 4). The page cache manages disk blocks in page-sized groups called *buffer pages*. Since the block and page typically have the same size, we use these two terms interchangeably from now on. A number of *pdflush* kernel threads periodically flush dirty pages to their final disk location. The threads systematically scan the page cache every *writeback period*; a dirty page is due for flushing after an *expiration period* has passed since it was last modified. Additionally, applications can synchronously flush the data and metadata blocks of an open file, for instance, through the *fsync* call or after opening the file with the `O_SYNC` option enabled. The *journaling block device* is a special kernel layer used by ext3 to implement the journal as a hidden file in the filesystem, or a separate disk partition. In the journal, each *log record* corresponds to an update of one disk block in the filesystem. The log record contains the entire modified block instead of the byte range actually overwritten. This wastes disk bandwidth and space, but makes straightforward the restoration of modified blocks after a crash. The degree of waste depends on the fraction of the block that is left unmodified by the write operation.

At the minimum, the system only needs to log the modified part of each buffer and merge it into the original block to recover the latest block version. Thus, we introduce a new type of journal block that we call *multiwrite block* (Figure 5(b)). We only use multiwrite blocks to accumulate the updates from data writes that partially modify

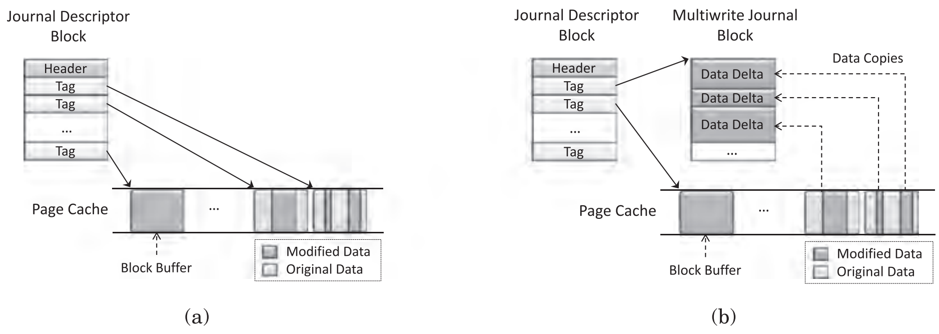

Fig. 5. (a) In the original design of data journaling, the system copies to the journal the entire blocks modified by write operations. (b) In wasteless journaling, we use multiwrite journal blocks to accumulate the data modifications from multiple writes.

block buffers. If a buffer contains metadata or is fully modified by a write operation, we can send it directly to the journal without creation of an extra copy in the page cache. We call such a journal block a *regular 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, for buffer updates smaller than the block size, we copy the corresponding data modification into a multiwrite block. Otherwise, we link the update to the entire modified block in the page cache. In selective journaling, we set the *write threshold* equal to the page size of 4KiB. If a buffer update is smaller than the write threshold, we mark the corresponding block as *journaled* by setting a special flag that we added in the page descriptor of the buffer. Then, we copy the modification to the multiwrite block. If the update modifies the entire block, we prepare the corresponding modified buffer for transfer to the filesystem without prior journaling. We clear the journaled flag after we complete the block transfer to the filesystem.

In a straightforward way, our current prototype can also support arbitrary write thresholds below the page size. In contrast, support for write thresholds above the page size requires additional implementation intervention at the system path of write requests, as described recently in a more general context [Mesnier et al. 2011]. The additional modification is necessary in order to keep track of the write size across the buffers in the page cache and treat them differently based on the write threshold.

3.2. Transactions

A system call may consist of multiple low-level operations that atomically manipulate disk data structures of the filesystem. For improved efficiency, the system assigns to one *transaction* the records of multiple calls. Before the records of a transaction are transferred to the journal, the kernel allocates a *journal descriptor block* with a list of *tags*. A tag maps a buffer to the respective block in the filesystem (Figure 5(a)). 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 (Figure 4). After all the log records of a transaction have been safely transferred 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, the offset and the length of the partially modified block pointed to by the buffer head. 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 block update (Figure 5(b)). Originally, each tag contained the filesystem location of the modified block and one flag for the 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 filesystem data block. These fields are required during recovery to allow the extraction of the update from the multiwrite block and the overwrite of the respective filesystem block at the right offset.

3.3. Recovery

We consider a transaction *committed* if it has flushed all its records to the journal and has been marked as finished. A transaction is automatically committed by the *kjournald* kernel thread after a fixed amount of time has elapsed since the transaction started. Subsequently, we regard the transaction as *checkpointed* if all the blocks of a committed transaction have been moved to their final location in the filesystem and the corresponding log records have been 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 the filesystem 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 filesystem 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 filesystem 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 filesystem without need to read first its older version from the disk.

3.4. Atomicity

Disk drives can guarantee the atomic update of a 512-byte sector through an attached checksum calculated over the sector data [SBC 2005]. For a page that consists of multiple sectors, incomplete page updates can be detected (*torn page detection*) through additional bits calculated over the entire page [Chidambaram et al. 2012; Sears and Brewer 2006]. Accordingly, we assume that the disk supports atomic page updates. One misbehavior not covered by the page atomicity is the case that only a subset of the pages in a transaction actually reaches the filesystem. This is possible because the disk internally uses a write cache to temporarily store incoming data. The on-disk write cache is typically set to operate in write-back mode, which may reorder writes for better performance. Then, the disk is possible to acknowledge a synchronous page write before the data is safely stored.

The integrity of a journaled transaction can be verified with a checksum calculated over the contents of the transaction [Prabhakaran et al. 2005b]. However, a journaled filesystem may silently end up in inconsistent state if the system crashes after a transaction partially updates the filesystem, but before the transaction is safely stored in the journal. Such inconsistency can be avoided if the filesystem explicitly controls the on-disk ordering of journal commits. For that purpose, the Linux ext3 provides the `barrier` mount option while the SCSI specification offers the `SYNCHRONIZE CACHE` command [SBC 2005] (SATA also provides the `FLUSH CACHE` command [SATA 2003]). If the device does not support write barriers, a flush workload can be used to flush the on-disk write cache instead [Rajimwale et al. 2011]. Alternatively, we can disable the write cache and have the disk only acknowledge a write after it really reaches the medium [Shin et al. 2011].

Assuming page atomicity on disk, wasteless journaling provides the consistency of data journaling. If additionally the disk barrier is used or the write cache is disabled, both wasteless and data journaling guarantee the idempotence of write operations. If a transaction replay is interrupted halfway through, from page atomicity it follows that each affected page in the filesystem will carry either the new value or the old value. A safely committed transaction can be repeatedly applied to the filesystem until it completes successfully. At this point, all the affected pages in the filesystem will have the new value.

Selective journaling marks as journaled the buffer of an update sequence based on the size of the first update to the respective data page. If a data page is prepared for direct transfer to the filesystem, there is no journal head to associate this data page with a transaction. It is possible that the system crashes right after a dirty data page is directly transferred to the filesystem. The respective metadata updates will not make it to the journal if we use write barriers or disable the on-disk write cache. After the crash, the above data update can become visible to the user if the update overwrote an existing file page. Essentially, the consistency of selective journaling degenerates to that of ordered mode if the update sequence is not journaled. With journaled update sequence, the respective filesystem page is only modified if it belongs to a safely committed transaction. If all the update sequences of a transaction are journaled, the consistency of selective journaling is that of wasteless journaling. As part of our experimentation, we confirm the comparative benefits of our journaling modes across different settings of the on-disk cache and the write barrier (Section 5.6).

Example. In Figure 6, we examine the potential effect of the updates applied to three different pairs of data and metadata blocks whose buffers are already located in system memory. Assuming that time increases from left to right, we refer to the data and metadata block of updates 1, 2, and 3, respectively, with b_1^d and b_1^m, b_2^d and b_2^m, b_3^d and b_3^m. The squares that contain the character *c* symbolize the commit block of the transaction. We assume that the updates applied to b_1^d and b_2^d are partial, while b_3^d is fully overwritten. With the journal and filesystem on disk, the example indicates that wasteless and selective journaling are likely to require less I/O time to safely flush the updates from memory to disk in comparison to the ordered and data journaling modes.

If the system crashes at instance t_1, then all the updates applied in memory are lost. At the other extreme, if the system crashes at instance t_3, then all the updates can be safely recovered from disk. Although both data and wasteless journaling record all the block updates to the journal, wasteless journaling transfers one less block. The ordered mode transfers all the data blocks to the filesystem, before it appends the three metadata blocks to the journal. Selective journaling only transfers block b_3^d to the filesystem, but it copies to the journal the updates of b_1^d and b_2^d through a multiwrite block.

In this example, if the system crashes at instance t_2, then selective journaling has already modified block b_3^d in the filesystem, while the ordered mode has modified b_1^d, b_2^d, b_3^d in the filesystem. As a result both the ordered and selective journaling modes leave the filesystem in an inconsistent state after the crash. Additionally, given that the commit block has not been safely stored on disk before the crash, all four modes

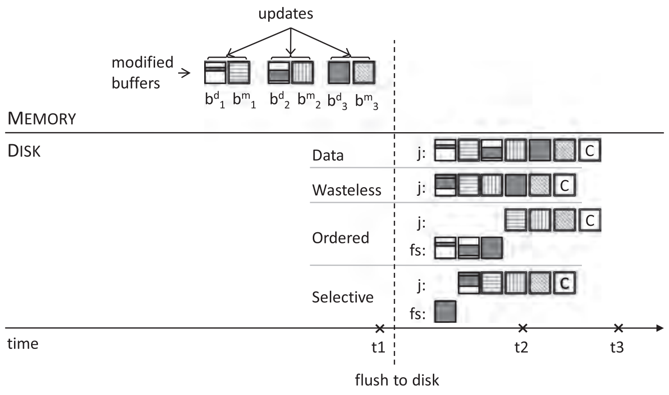

Fig. 6. We consider three different pairs of data and metadata blocks whose respective buffers are updated in memory. From left to right, we show a possible timing of block transfers to the journal (j) and the filesystem (fs) across four different filesystem modes. The superscripts d and m of the blocks refer to data and metadata, respectively, while t_1, t_2, and t_3 refer to three time instances of system crash that we examine. The square containing c refers to the commit block.

fail to recover the three updates. The example indicates that the multiwrite block helps reduce the I/O traffic to the journal, while any in-place updates directly applied to the filesystem may lead to inconsistencies during a crash.

4. EXPERIMENTATION ENVIRONMENT

We implemented wasteless and selective journaling in the Linux kernel version 2.6.18. Newer Linux releases still lack the functionality that we propose (e.g., ext4 [Kumar et al. 2008]). In order to add the proposed functions into ext3, we modified 684 lines of code across 19 files of the original Linux kernel. Members of our team used the modified system as a working environment for several months. We evaluated our prototype over a 16-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, 3GiB RAM, and two SAS 15KRPM disks (Seagate Cheetah ST3300655SS [Cheetah 2007]). Each disk has 300GB storage capacity, multisegmented 16MiB 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. Our conclusions were similar in several experiments that we did (not shown) with two SATA 7.2KRPM disks of 250GB capacity and 16MiB cache. We keep the page and block sizes equal to 4KiB, while we leave the journal size at the default value 128MiB. 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. On otherwise idle machines, with up to fifteen experiment repetitions, we ensure that our results have half-length of 90% confidence interval within 10% of the reported average.

5. PERFORMANCE EVALUATION

We study the performance of microbenchmarks, application-level workloads and traces from database logs directly running on the modified filesystem. We also evaluate a stable Linux port of the Log-structured File System, where the entire filesystem is structured as a log [Rosenblum and Ousterhout 1992]. Additionally, over a multitier

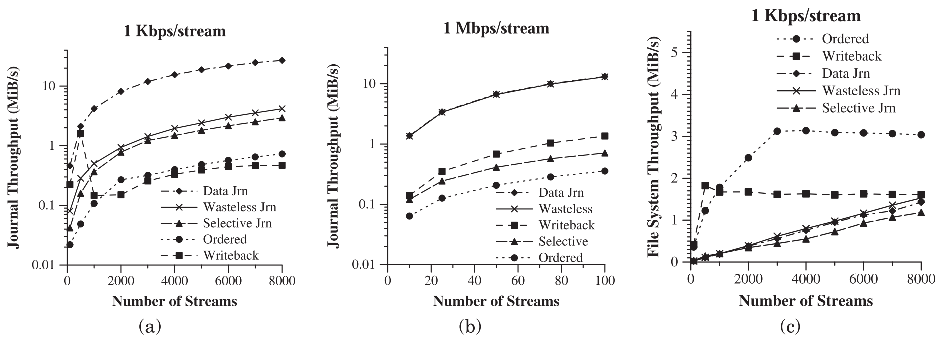

Fig. 7. (a) At 1Kbps, the journal throughput (lower is better) of both selective and wasteless journaling approaches that of ordered and writeback modes, unlike data journaling which is several factors higher. (b) At 1Mbps, wasteless and data journaling have the same journal throughput, while selective journaling lies between writeback and ordered. (c) In comparison to ordered and writeback at 1Kbps, the other three modes incur lower filesystem throughput (lower is better), because they batch multiple writes into fewer page flushes.

configuration based on the PVFS2 distributed filesystem, we examine the impact of the server filesystem to the parallel workload running across multiple clients. Finally, we measure the recovery time after a crash.

The default disk settings typically increase performance by allowing a synchronous write to return when the data reaches the on-disk cache rather than the storage surface. The durability of written data can be improved if one alternatively disables the on-disk cache, applies flush workloads to the cache, or uses controllers with battery-backed cache [Nightingale et al. 2006; Rajimwale et al. 2011; Shin et al. 2011]. In most of our experiments we kept enabled the on-disk caches, but in Section 5.6 we report the sensitivity of our results to alternative cache configurations.

5.1. Microbenchmarks

For a time period of 5min, we ran a number of threads on the local filesystem. Each thread appends data to a separate file by calling one synchronous write per second. The generated aggregate traffic effectively consists of random I/O operations. As a metric of inefficiency, we use the average throughput (the lower the better) on the journal device across the different mount modes of ext3. With 1Kbps streams in Figure 7(a), we observe that as the number of streams increases from one hundred to several thousand, the journal throughput of data journaling reaches 27MiB/s. On the contrary, selective and wasteless journaling limit the journal traffic up to 2.9MiB/s and 4.2MiB/s, respectively. The higher throughput of data journaling is expected because it writes to the journal the entire modified data blocks instead of just the subpage modifications.

At stream rate 1Mbps, wasteless and data journaling are comparable in terms of journal throughput (Figure 7(b)). Instead, the selective and ordered modes transfer data updates directly to the filesystem, which reduces their journal throughput by an order of magnitude or more with respect to wasteless and data journaling. We additionally examined (not shown) streams of 10Kbps and 100Kbps, and mixed workloads with multiple stream rates at different ratios. Not surprisingly, the journal throughput of wasteless journaling varied between the values reported in Figures 7(a–b) according to the fraction of requests that correspond to each stream rate.

In Figure 7(c), we measure the write throughput of the filesystem device for 1Kbps streams. The ordered mode synchronously transfers the data updates directly to the filesystem with costly random I/Os before moving the corresponding metadata to the journal. Instead, wasteless, selective and data journaling synchronously transfer the

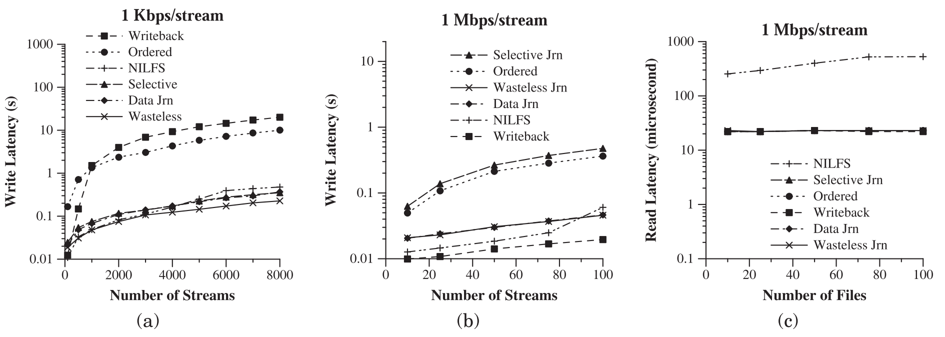

Fig. 8. (a) With low rates, the write latency (lower is better) of ordered and writeback mode appears orders of magnitude longer than the other modes. (b) At higher rates, the selective and ordered modes experience much higher latency. (c) As we read sequentially multiple files that we previously wrote concurrently, read requests of 4KiB size with NILFS complete in order of magnitude longer time than the different modes of ext3.

updates to the journal, and only periodically flush the dirty pages to the filesystem. Thus, multiple writes to the same data block are automatically coalesced into fewer page flushes, leading to lower traffic at the filesystem. Respectively, we also measured the processor utilization (not shown) and found it relatively higher for wasteless, selective and data journaling. Nevertheless, processor utilization in these experiments always remained low, up to 5%.

Ultimately, the journaling of data is expected to reduce the latency of synchronous writes. As they serve multiple streams of 1Kbps, in Figure 8(a), the ordered and writeback modes incur orders of magnitude higher latency with respect to the other modes. Multiple concurrent synchronous requests in ordered mode result in random accesses to the filesystem device. Thus, data journaling completes a write operation in tens of milliseconds, but the ordered mode takes several seconds instead. Selective journaling follows wasteless at low rates, and approaches the ordered mode at high rates (Figures 8(a–b)).

In Figure 8, we also consider a stable Linux port of the Log-structured File System, where all data and metadata updates are written sequentially as a continuous stream (NILFS) [Yoshiji et al. 2009]. We find that the write latency of NILFS is comparable to that of wasteless and data journaling at both 1Kbps and 1Mbps streams. Overall, the sequential throughput of the journal significantly improves the ability of the system to store fast the incoming data. In Figure 8(c), we use a thread to sequentially read one after the other different numbers of files that we previously created concurrently at 1Mbps each, using NILFS or ext3. In this experiment, we measure the average time to read one 4KiB 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. Our results with 1Kbps streams were similar; NILFS along with the ordered and writeback modes incur higher read latencies than the other three modes.

In order to examine the generality of our conclusions, we also considered streams with asynchronous writes. In I/O-intensive workloads, we anticipate that recent updates are flushed to the filesystem as a result of memory pressure, before the page-cleaning daemon is periodically activated over the cache. In Figure 9, with several low-rate streams, we notice that the ordered mode leads to write latency that is considerably longer and highly variable in comparison to selective journaling. Essentially, selective and wasteless journaling move recent updates to the journal device at sequential throughput, which reduces the latency of ordered and data journaling up to several

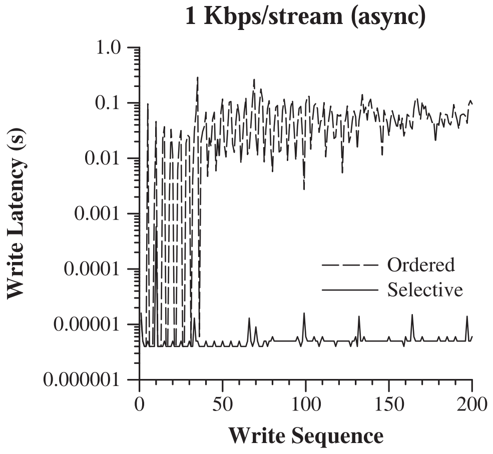

Fig. 9. We depict the average latency of 1000 streams along a sequence of 200 disk writes. Each stream *asynchronously* writes once per second 125 bytes (1Kbps). In comparison to selective journaling, the write latency of ordered mode tends to be highly variable and orders of magnitude longer.

orders of magnitude. Correspondingly, we confirm previous reports that asynchronous workloads may behave as synchronous under conditions of high update rate [Batsakis et al. 2008].

5.2. Postmark and Filebench

We use the Postmark benchmark to examine the performance of small writes as seen in electronic mail, netnews, and web-based commerce [Katcher 1997]. We apply version 1.5 with the option of synchronous writes added by FSL of Stony Brook University. 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 total workload of 10,000 mixed transactions with file read, append, create, and delete operations. We set equal to 5 the ratio of read/append operations and equal to 9 the ratio of create/delete. We draw the file sizes from the default range between 500 bytes and 97.66KiB, while I/O request sizes lie in the range between 128 bytes and 128KiB. In Figure 10(a), we observe that the transaction rate (higher is better) of wasteless journaling gets as high as 738tps. Wasteless journaling combines the sequential throughput of journaling with the reduced amount of written data to the journal and the filesystem during small updates. Across different request sizes between 128 bytes and 128KiB, wasteless journaling consistently remains faster than the other modes, including data journaling (max rate 663tps). It is notable that wasteless journaling improves by 85% the performance of ordered mode (max rate 399tps). Instead, selective journaling with max rate 473tps lies between the data journaling and ordered modes.

As application-level workloads with asynchronous writes, we used the `fileserver` and `oltp` personalities of Filebench v.1.4.9.1 [2011]. Similarly to SPECsfs, the `fileserver` emulates the I/O activity of a simple fileserver using an operation mix of file create, delete, append, read, write, and attribute accesses. By default, the number of threads is set to 50 and the mean size of appends is 16KiB. We let the tool automatically configure the number of files to 250K, based on the memory size of the server. From Table I it follows that the operation throughput (higher is better) of ordered mode is improved by 12.6% with data journaling and 17.5% with wasteless, respectively. Subsequently, we configured the mean append size of `fileserver` to 4KiB. Then,

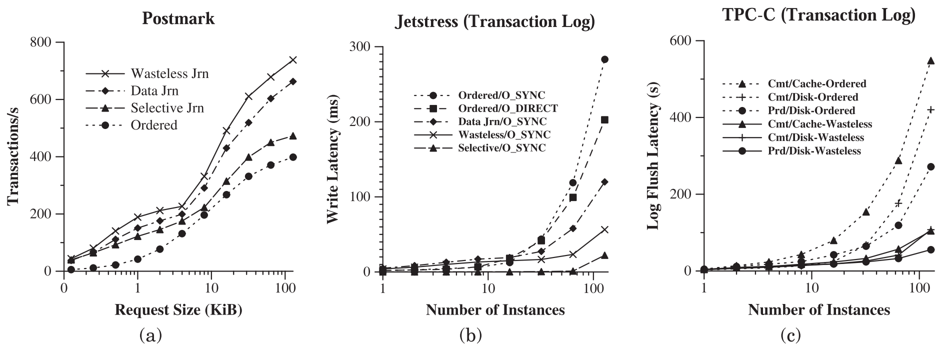

Fig. 10. (a) With the Postmark benchmark, wasteless journaling consistently outperforms the other modes in terms of operation transaction rate (higher is better). (b) We consider up to 128 concurrent Jetstress instances. In comparison to the other modes, selective journaling maintains the latency of log writes lower up to several orders of magnitude. (c) We examine the three flushing methods of MySQL/InnoDB. With respect to the ordered mode, wasteless journaling reduces up to an order of magnitude the latency required to flush the transaction log to the disk.

Table I. Performance of `fileserver` and `oltp` Personalities in Filebench. (In `fileserver` we alternatively examine mean append size equal to 16KiB (default) and 4KiB.)

	Fileserver (16KiB)		**Fileserver (4KiB)**		**OLTP**	
Mount Mode	**Thput (Ops/s)**	**Latency (ms)**	**Thput (Ops/s)**	**Latency (ms)**	**Thput (Ops/s)**	**Latency (ms)**
Ordered	579.2	314.6	576.8	315.5	779.8	182.2
Selective Jrn	493.8	368.9	559.2	326.1	810.2	156.3
Data Jrn	652.2	278.2	700.0	260.1	826.8	146.6
Wasteless Jrn	680.4	266.9	711.0	255.5	825.2	146.7

the respective improvement became 21.4% with data journaling and 23.3% with wasteless. Selective journaling splits the data writes between the journal and the filesystem leading to operation throughput below that of ordered.

In the case of the `oltp` personality, Filebench performs the file system operations of the Oracle 9i I/O model. By default, it uses 200 reader processes, 10 processes for asynchronous writing and a synchronous log writer. The tool automatically configures the file size to 600MiB. The workload involves small random reads and writes, and it is sensitive to the latency of the moderate-sized (128KiB+) writes to the log involved. Data and wasteless journaling achieve a limited throughput improvement (6%) with respect to the ordered mode, while selective journaling lies between the wasteless and ordered. In the following section, we further examine the logging latency of databases by considering multiple concurrent workloads [Calder et al. 2011; Mao et al. 2012].

5.3. Groupware and Database Logging

System administrators prefer to devote a separate device for the logs of I/O-intensive applications for efficiency [Mullins 2002]. Distributed systems place multiple log files locally at each machine for improved performance and autonomy [Calder et al. 2011; Gray and Reuter 1993]. Also, database engines optimized for multicore hardware maintain multiple log files on the same host [Mao et al. 2012]. Given the high cost of maintaining extra spindles in a machine, we investigate the possibility of serving multiple log files efficiently over a single disk with appropriate filesystem support. In the present section, we measure the latency to serve the I/O traffic of log traces that we gathered from groupware and database workloads.

5.3.1. Jetstress. We consider the Jetstress Tool that emulates the disk I/O load of the Microsoft Exchange messaging and collaboration server [Jetstress 2007]. We run Jetstress for two hours in a Windows Server 2003 system with 1GiB RAM and two SATA disks in mirrored mode. We used 50 mailboxes with 100MiB each and 1 operation per second for each mailbox. With these parameter values, 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 256MiB. 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 KiB. The writes are tagged as *uncached*, that is, they are 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_SYNC` option enabled at file open for synchronous access. Additionally, we consider the ordered mode with the `O_DIRECT` option at file open to bypass the page cache. 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 10(b), both selective and wasteless journaling keep write latency up to tens of milliseconds even at high load. Unlike wasteless journaling that writes to the journal all the affected data modifications, selective distributes across both spindles—of the journal and filesystem—the incoming appends. As a result, selective journaling achieves logging latency that is half that of wasteless or less. 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 filesystem support. We assume that the durability of synchronous writes is similar to that of bypassing the page cache.

5.3.2. TPC-C. We also examine the logging activity of the OLTP performance benchmark TPC-C [TPCC 1992] as implemented in Test 2 of the Database Test Suite [DBT]. We used the MySQL open-source database system with the default InnoDB storage engine [MySQL]. 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 led to insignificant differences of the measured transaction throughput among ordered mode, wasteless, and selective journaling. This is reasonable because most updates in the workload have a size above the write threshold; as a result, the disk operations are sequential regardless of whether they update the journal or the filesystem.

The InnoDB storage engine supports three different methods for flushing to disk the transaction log of the database. In 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 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 10(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, at 64 or 128 instances, ordered mode takes hundreds of seconds. We also experimented with selective journaling (not shown) and found it close to wasteless journaling and well below ordered. The reported behavior was anticipated because wasteless and selective journaling sequentially store the small appends of the database log into the system journal.

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 [Carns et al. 2009; Hildebrand et al. 2006]. Especially small requests of 1KiB are known to be problematic because they incur high rotational overhead, even after they are transformed into sequential [Polte et al. 2008]. Writes of 47001 bytes often also appear in parallel applications and lead to poor performance due to alignment misfit [Bent et al. 2009]. In this section we examine the performance gain of a parallel multitier configuration with our mount modes running directly in the kernel-based filesystem of the storage server.

We chose the PVFS2 as an open-source scalable parallel file system [PVFS2]. 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 (filesystem 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 in the system guide 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 [MPI-IO]. In our configuration, each process writes to a separate unique file ("N processors to N files"). According to previous studies, this is the write pattern suggested to application developers for best performance [Bent et al. 2009]. 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 65,000 writes with alternative write sizes of 1024 and 47001 bytes. In Figure 11, we compare the data throughput of MPI-IO across different write sizes and loads. With 1KiB writes, wasteless journaling almost doubles the throughput of ordered mode, while data journaling and selective lie between the other two. With writes of 47001 bytes, the write throughput remains about the same across the different modes.

In Figure 12 we depict the total volume of write traffic across the BerkeleyDB, the journal and the filesystem. At 1KiB requests, data journaling transfers 415MiB to the journal, while wasteless and selective journaling reduce this amount by 42% (Figure 12(a)). The ordered mode writes to the journal 139MiB, but transfers to the filesystem a total of 255MiB. This amount is at least a factor of four higher with respect to the other three modes, which accumulate multiple small writes in memory before transferring them coalesced into the filesystem. At requests of 47001 bytes, selective journaling closely tracks the ordered mode in terms of total write volume. In contrast, data and wasteless journaling almost double the total disk traffic by double-writing the updated data blocks (Figure 12(b)).

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

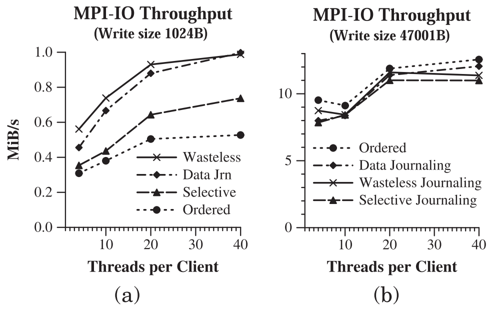

Fig. 11. We measure the data throughput (higher is better) of MPI-IO as client of PVFS2. (a) At 1KiB writes, wasteless journaling almost doubles the performance of the default ordered mode. (b) At request size 47001 bytes, the prevalence of writes above the write threshold keeps similar the relative performance of the mount modes.

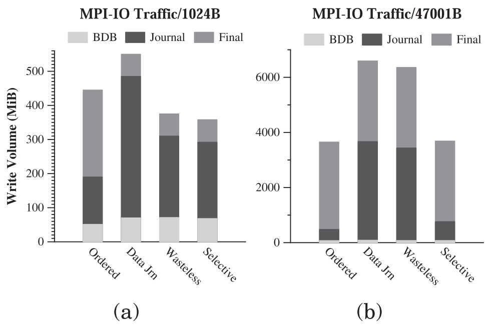

Fig. 12. We measure the disk traffic (lower is better) of BerkeleyDB (BDB), the journal (Journal) and the filesystem (Final) over a PVFS2 data server. (a) At 1KiB writes, selective and wasteless reduce the journal traffic of data journaling and the filesystem traffic of ordered. (b) At 47001 bytes, wasteless is similar to data journaling, and selective comparable to ordered mode, in terms of total disk traffic.

modes, but the journal traffic is higher for the data and wasteless journaling, as they enforce stricter consistently between the data and metadata updates.

5.5. Recovery Time

In a different experiment, we evaluate the ability of the system to recover quickly after a system crash, which leaves the journal with log records before the respective updates are checkpointed to the filesystem. It is known that when the free journal 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 [Prabhakaran et al. 2005a]. In order to make 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 [Brito et al. 2009]). Thus, we start 100 threads, each doing 100 synchronous writes of request size 8 bytes. Then we cut the system power. At the subsequent reboot, we verify that all modes fully and correctly recover the unique written data, while in the kernel we measure the duration of filesystem recovery.

In Figure 13, we break down the total recovery across the three passes that scan the transactions, revoke blocks, and replay the committed transactions, respectively. In comparison to data journaling, the *scan* pass of selective and wasteless journaling is an order of magnitude shorter. This difference arises from journaling entire data blocks by data journaling, which significantly increases the amount of scanned data. The *replay* pass of selective and wasteless journaling takes about 40% more time than the ordered and writeback modes due to the extra block reads involved. Overall, selective and wasteless journaling reduce by 20–22% the recovery time of data journaling. In comparison to these modes, the recovery time of ordered and writeback is an order of magnitude lower, at the cost of weaker consistency guarantees across the stored data and metadata.

5.6. Device Issues

We examine the sensitivity of our performance results to the settings of the on-disk cache and the use of write barriers (Section 3.4). The disk we experimented with (ST3300655SS) organizes the cache into multiple logical segments. It supports the `SYNCHRONIZE CACHE` command to force the transfer of all cached write data to the

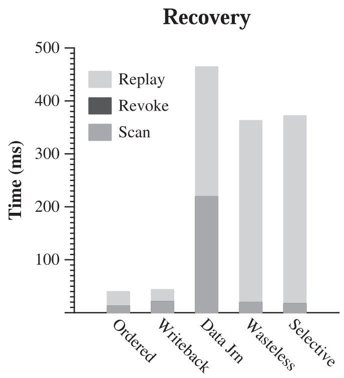

Fig. 13. In comparison to data journaling, wasteless and selective journaling reduce the scan time of recovery by an order of magnitude, but increase the replay time by about 40%. In total, they reduce the recovery time of data journaling by 20–22%.

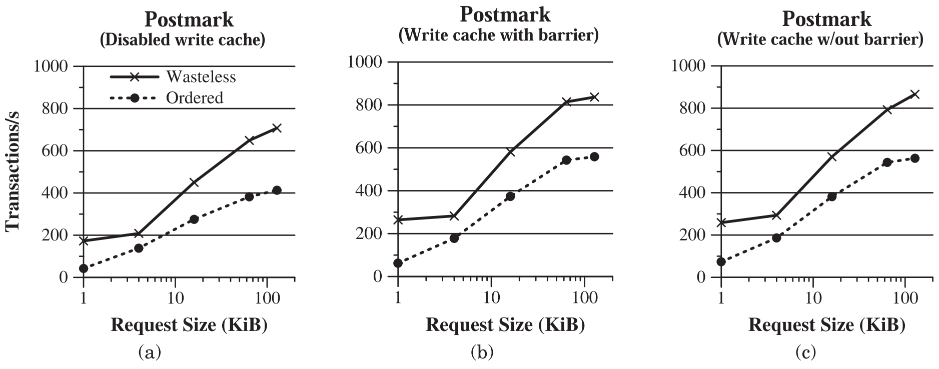

Fig. 14. (a) With disabled the on-disk write caches, wasteless journaling improves the performance of ordered mode by a factor of 4 at 1KiB requests and 73% at 128KiB size. (b) We enable the on-disk write caches and mount the ext3 filesystem with `barrier=1`. Wasteless journaling improves the performance of ordered mode by a factor of 4.3 at 1KiB requests. (c) If we enable the on-disk write caches with mount option `barrier=0` (default ext3), the performance of ordered mode improves up to 18% at 1KiB. However, the relative advantage of wasteless journaling with respect to the ordered mode remains significant (e.g., 3.52 times at 1KiB).

medium, and the `FORCE UNIT ACCESS` bit to enforce medium access on the basis of individual reads and writes [Cheetah 2007; SBC 2005]. We kept the read cache always activated, and used the `sdparm` utility to configure the write cache. In Figure 14(a) we disable the on-disk write caches at both the filesystem and the journal, while we mount the filesystem with the option `barrier=0` (default ext3). We run the Postmark workload with the configuration of Section 5.1. It is not surprising that, for requests of subpage size 1KiB, wasteless journaling maintains a performance advantage of four times in comparison to the ordered mode. The relative improvement drops to 50% at 4KiB requests, and becomes 73% at 128KiB requests.

In Figure 14(b), we enable the write caches of the disks and mount the filesystem with `barrier=1`. Write barriers ensure that the write cache of the journal device is flushed before the commit block is written and also flushed to the medium. With enabled write caches, the two mount modes improve their performance by 21–53% with respect to (a). In comparison to the ordered mode, wasteless journaling maintains a performance advantage up to a factor of 4.25 at 1KiB requests. In Figure 14(c), we enable the on-disk write caches and mount the filesystem with `barrier=0`. In comparison to (b), the performance of ordered mode increases from 62tps to 73tps at 1KiB, and up to 4.6% at larger request sizes. The performance of wasteless journaling without write barriers (c) remains within 3.5% of that achieved with write barriers (b). Also, wasteless journaling improves the performance of ordered mode up to a factor of 3.52 at 1KiB. We conclude that enabling the write caches improves the benchmark performance, while the use of write barriers incurs a relatively low cost, mostly noticeable in the ordered mode. In all the other experiments, we kept the write caches enabled on our disks and used the default ext3 mount option of `barrier=0`.

Arguably, wasteless journaling takes advantage of the two spindles that store the journal and the filesystem, respectively. Instead, the ordered mode mostly uses the spindle of the filesystem and uses the spindle of the journal less. To address 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 the ordered mode with 1Mbps streams.

6. RELATED WORK

The log-structured filesystem addresses the problems of synchronous metadata updates and small writes by coalescing data writes sequentially to a segmented log [Rosenblum and Ousterhout 1992]. Previous research reported cleaning overheads and performance limitations under particular workloads [Seltzer et al. 1995]. Via experiment, we also notice reduced read performance for the log-structured approach in some cases (Section 5.1). Group commit is a known database logging optimization that is used to amortize the I/O cost of inserting transaction commits to the log. It accumulates the log records from multiple transactions and periodically flushes them to the log [DeWitt et al. 1984]. Instead, we emphasize fitting multiple subpage modifications from concurrent synchronous writes into a single block and investigate the related benefits in a general-purpose journaled filesystem.

The virtual log uses a tree to logically link noncontiguous disk blocks and uses free sectors close to the head to minimize the latency of small synchronous writes [Wang et al. 1999]. StreamFS is a modified version of the log-structured filesystem for storing high-volume streams [Desnoyers and Shenoy 2007]. Instead, we also handle the storage traffic of low-rate streams. The hFS filesystem stores metadata and small files in a separate partition from large files. It differentiates updates by file size rather than write size as we do [Zhang and Ghose 2007]. In real-time data processing, application operators can recover from failures through synchronous logging at high latency [Kwon et al. 2008]. Recent research combines software transactional memory with asynchronous logging to optimistically parallelize stream operators [Brito et al. 2009]. However, this approach is limited to operators that do not perform external actions such as I/O [Chandrasekaran and Franklin 2004]. The present work substantially extends our previously published work [Hatzieleftheriou and Anastasiadis 2011b] through motivating references; detailed description of our design and implementation; comprehensive experiments across different workloads and device configurations; comparative presentation of representative related research; and a discussion of our contribution in the context of current technology trends.

In the Ceph distributed filesystem, the storage servers support journaling of both data and metadata similarly to the data journaling mode of ext3 [Weil et al. 2006].

Ceph provides two new journaling modes: (i) In the writeahead mode, a write transaction returns as soon as it reaches the journal. (ii) In the parallel mode, a write transaction is written to both the journal and the filesystem, and returns when either of the two commits. The Ceph designers admit that they write all data twice for safety, and mention the related performance tradeoff between write latency and write throughput. For the efficient storage of the journal, they support several hardware options. In our present work, we extensively examine the resource requirements of data journaling, and propose two new modes to retain high performance at moderate journal traffic.

The I/O characteristics of parallel applications have led to middleware techniques (e.g., data sieving or collective I/O) that handle as contiguous the noncontiguous requests from parallel processes [Thakur et al. 1999]. Additionally, checkpointing has a prominent role in the robust execution of high-performance parallel applications [Elnozahy and Plank 2004]. The Parallel Log-Structured Filesystem (PLFS) introduces an interposition layer that writes transparently the checkpoint data from different processes to different files, instead of having all data written to a single shared file [Bent et al. 2009]. The Checkpoint-Restart File System (CRFS) is a user-level filesystem that aggregates per-file writes in memory [Ouyang et al. 2011b]. When the writes fill up a preconfigured chunk size (e.g., 4MiB), they are asynchronously transferred to disk. The above approaches are complementary to our work because they are specialized for parallel applications or checkpoints, and operate at the middleware or the user level rather than within a general-purpose filesystem.

High-performance synchronous writes can be handled through specialized hardware, such as battery-backed main memory (NVRAM) [Chen et al. 1996]. 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 nonvolatile RAM [Hitz et al. 1994]. Reportedly, NVRAM creates a single point of failure over disk arrays, while dual-copy NVRAM cache can be costly [Hu et al. 2002]. Disk-specific knowledge can be exploited to align the data accesses on track boundaries, and avoid rotational latency and track-crossing overhead [Anand et al. 2008; Schindler et al. 2002]. This approach operates at the disk level and could complement our methods when we update the filesystem. Xsyncfs introduces externally synchronous I/O that guarantees durability to an external observer of application output rather than the application itself [Nightingale et al. 2006]. If an application does not produce output, xsyncfs commits data periodically and asynchronously.

In earlier work, Hagmann described metadata update logging in the Cedar File System to improve performance and achieve consistency [Hagmann 1987]. Soft updates track and enforce metadata update dependencies so that the filesystem can safely delay writes for most file operations [Seltzer et al. 2000]. Unlike our work, both the above systems only focus on metadata rather than data updates. Subpage updates had previously been handled efficiently in the context of distributed shared memory by the Millipage system [Itzkovitz and Schuster 1999]. Instead, we introduce wasteless and selective journaling as a general filesystem service.

7. DISCUSSION

Across a range of consistency conditions, existing filesystems can be wasteful or underperforming. We propose and implement several improvements that address these weaknesses without penalizing the behavior of the filesystem beyond a reasonable increase in disk traffic. The main theme in our proposed design is to improve performance and consistency 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 [Chen et al. 1996; Hitz et al.

1994]. 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.

Our effort to favor sequential writes at moderate storage traffic is compatible with the endurance and performance characteristics of novel devices such as solid-state drives based on flash memory [Chen et al. 2009]. Flash memory exhibits a number of attractive features related to low power consumption and improved access performance, but also several hardware idiosyncrasies that make its behavior workload dependent. Flash memory usually consists of multiple blocks, each of which contains several pages. Data is written in units of pages, and space is erased in units of blocks. Usually a log-structured approach organizes the flash space so that writes incur low cost [Dai et al. 2004; Woodhouse 2001]. A cleaning process periodically merges valid pages into clean blocks and reclaims the invalidated ones. The append-only nature of journaling keeps writes over flash memory relatively cheap [Min et al. 2012]. Simple block remappings of the metadata can transform journaled updates into a permanent state without relocations that lead to duplicate writes [Choi et al. 2009]. Native support of atomic writes at the flash firmware was shown to avoid duplicate writes for the safe update of the database state from logged deltas of data pages [Ouyang et al. 2011a].

In contrast, we focus on a general-purpose filesystem and coalesce concurrent subpage writes to the same storage block of the journal, while we safely delay and batch small writes to the filesystem. Additionally, with selective journaling we avoid duplicate traffic to the device for sequential workloads. Our proposed modes could be directly applied as a journaled filesystem over flash memory to serve two needs: (i) reduce the amount of data sequentially written to a flash-based journal device and the wear it causes [Choi et al. 2009]; (ii) decrease the number of random writes reaching the storage device of a filesystem, since random writes are reported as harmful for the performance and lifespan of flash memory [Min et al. 2012]. In ongoing work, we are developing a flash-optimized filesystem to further explore the above observations [Hatzieleftheriou and Anastasiadis 2011a].

In virtualization environments, the block-based interface of the guest virtual machine makes small writes appear as full-block updates to the underlying filesystem. Recently, the interaction of nested filesystems has been experimentally investigated. Application workloads with reads and writes smaller than 4KiB suffer the most from the full-page I/Os of the guest [Hildebrand et al. 2011]. The data and metadata of the guest disk image are treated as data by the host filesystem. Consequently, write-intensive workloads lead to significant consistency degradation if the filesystem at the host provides metadata-only journaling. Additionally, the journaling of both data and metadata is considered impractical due to the performance degradation [Le et al. 2012]. As one solution to the performance problem of data journaling, it was recently proposed to maintain the journals of multiple virtual machines in the main memory of the host, presuming that the hardware and virtual machine monitor are sufficiently reliable [Huang and Chang 2011].

Instead, our proposed modes could be used either as guest filesystems to reduce the downward write traffic, or as host filesystem to consistently serve the disk images of multiple virtual machines. Application at the host filesystem would make sense under the assumption that the guests communicate with the host through a virtualization-optimized I/O interface that flexibly supports requests of different sizes. Accordingly, we could safely serve the incoming small writes from multiple concurrent threads running across different guests and persistently store both the data and metadata of the guest filesystems. Thus, we anticipate increased consistency in the recovery of virtual images from crashes and improved guest performance during normal operation. In ongoing research, we investigate possible extensions of the present work for virtualization environments.

8. CONCLUSIONS AND FUTURE WORK

Journaling is a technique commonly used in current filesystems 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 operation 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-sized blocks. Additionally, we develop the selective journaling method that only logs updates below a write threshold and transfers the rest directly to the filesystem. Our experimental results include measurements from streaming microbenchmarks, application-level workloads, database logging traces, and multistream I/O over a parallel filesystem in the local network. Across different cases, we demonstrate reduced write latency and recovery time, along with improved transaction throughput with low journal bandwidth requirements. Our plans for future work include extension of the above methods for disk arrays, virtualization environments, and flash memory systems.

ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers for their constructive comments that helped us improve the manuscript.

REFERENCES

Anand, A., Sen, S., Krioukov, A., Popovici, F. I., Akella, A., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Banerjee, S. 2008. Avoiding file system micromanagement with range writes. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 161–176.

Appuswamy, R., van Moolenbroek, D. C., and Tanenbaum, A. S. 2010. Block-level RAID is dead. In *Proceedings of the Workshop on Hot Topics in Storage in File Systems*.

Baker, J., Bondç, C., Corbett, J., Furman, J. J., Khorlin, A., Larson, J., Léon, J., Li, Y., Lloyd, A., and Yushprakh, V. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data Systems Research*. 223–234.

Batsakis, A., Burns, R. C., Kanevsky, A., Lentini, J., and Talpey, T. 2008. AWOL: An adaptive write optimizations layer. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 67–80.

Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. 2009. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*. 1–12.

Birrell, A. D., Hisgen, A., Jerian, C., Mann, T., and Swart, G. 1993. The Echo distributed file system. Tech. rep. TR-111, DEC Systems Research Center, Palo Alto, CA.

Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., and Aiyer, A. 2011. Apache Hadoop goes realtime at facebook. In *Proceedings of the ACM SIGMOD Conference*. 1071–1080.

Bovet, D. P. and Cesati, M. 2005. *Understanding the Linux Kernel* 3rd Ed. O'Reilly Media, Sebastopol, CA.

Brito, A., Fetzer, C., and Felber, P. 2009. Minimizing latency in fault-tolerant distributed stream processing systems. In *Proceedings of the International Conference on Distributed Computing Systems*. 173–182.

Calder, B., Wang, J., Ogus, A., Nilakantan, N., and Skjolsvold, A., et al. 2011. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, New York, 143–157.

Carns, P., Lang, S., Ross, R., Vilayannur, M., Kunkel, J., and Ludwig, T. 2009. Small-file access in parallel file systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. IEEE, Washington, D.C., 1–11.

Chandrasekaran, S. and Franklin, M. 2004. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proceedings of the Conference on Very Large Data Bases*. 348–359.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 205–218.

Cheetah. 2007. Seagate Cheetah 15K.5 SAS (ST3300655SS). Product Manual. http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/cheetah/15K.5/SAS/100384784e .pdf.

Chen, F., Koufaty, D. A., and Zhang, X. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Conference on SIGMETRICS/Performance*. 181–192.

Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. 1996. The Rio file cache: Surviving operating system crashes. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 74–83.

Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2012. Consistency without ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 101–116.

Choi, H. J., Lim, S.-H., and Park, K. H. 2009. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Trans. Storage 4*, 14:1–14:22.

Dai, H., Neufeld, M., and Han, R. 2004. ELF: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems*. 176–187.

DBT. Database test suite. http://osdldbt.sourceforge.net/.

Desnoyers, P. J. and Shenoy, P. 2007. Hyperion: High volume stream archival for retrospective querying. In *Proceedings of the USENIX Annual Technical Conference*. 45–58.

DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A. 1984. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 1–8.

Elnozahy, E. N. and Plank, J. S. 2004. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Trans. Dependable Secure Comput. 1,* 2, 97–108.

Filebench. 2011. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Main_Page.

Fryer, D., Sun, K., Mahmood, R., Cheng, T., Benjamin, S., Goel, A., and Brown, A. D. 2012. Recon: Verifying file system consistency at runtime. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 73–86.

Gray, J. and Reuter, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, Ch. 9.

Grupp, L. M., Davis, J. D., and Swanson, S. 2012. The bleak future of NAND flash memory. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 17–24.

Hagmann, R. 1987. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, New York, 155–162.

Hatzieleftheriou, A. and Anastasiadis, S. V. 2011a. JLFS: Journaling the log-structured filesystem for proactive cleaning in flash storage. In *Proceedings of the USENIX Annual Technical Conference* (poster).

Hatzieleftheriou, A. and Anastasiadis, S. V. 2011b. Okeanos: Wasteless journaling for fast and reliable multistream storage. In *Proceedings of the USENIX Annual Technical Conference*. 235–240.

Hildebrand, D., Ward, L., and Honeyman, P. 2006. Large files, small writes, and pNFS. In *Proceedings of the ACM International Conference on Supercomputing*. 116–124.

Hildebrand, D., Povzner, A., Tewari, R., and Tarasov, V. 2011. Revisiting the storage stack in virtualized nas environments. In *Proceedings of the Workshop on I/O Virtualization* (co-held with *USENIX ATC*).

Hisgen, A., Birrell, A., Jerian, C., Mann, T., and Swart, G. 1993. New-value logging in the Echo replicated file system. Tech. rep. SRC 104, Digital Equipment Corp., Palo Alto, CA.

Hitz, D., Lau, J., and Malcolm, M. 1994. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter Technical Conference*. 235–246.

Hu, Y., Nightingale, T., and Yang, Q. 2002. RAPID-Cache–a reliable and inexpensive write cache for high performance storage systems. *IEEE Trans. Parallel Distrib. Syst. 13,* 3, 290–307.

Huang, T.-C. and Chang, D.-W. 2011. VM aware journaling: Improving journaling file system performance in virtualization environments. *Softw. Pract. Exper. 42*, 3, 303–330.

Itzkovitz, A. and Schuster, A. 1999. MultiView and Millipage - Fine-grain sharing in page-based DSMs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 215–228.

Jetstress. 2007. Microsoft exchange server jetstress tool. http://technet.microsoft.com/en-us/library/bb643093.aspx.

Katcher, J. 1997. PostMark: A new file system benchmark. Tech. rep. TR-3022, NetApp.

Kumar, V. A., Cao, M., Santos, J. R., and Dilger, A. 2008. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*. 263–274.

Kwon, Y., Balazinska, M., and Greensberg, A. 2008. Fault-tolerant stream processing using a distributed, replicated file system. In *Proceedings of the Very Large Data Bases Conference*. 574–585.

Le, D., Hang, H., and Wang, H. 2012. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 87–100.

Leung, A. W., Pasupathy, S., Goodson, G., and Miller, E. L. 2008. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference*. 213–226.

Mammarella, M., Hovsepian, S., and Kohler, E. 2009. Modular data storage with Anvil. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, New York, 147–160.

Mao, Y., Kohler, E., and Morris, R. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the ACM European Conference on Computer Systems*. ACM, New York.

Mesnier, M., Chen, F., Luo, T., and Akers, J. 2011. Differentiated storage services. In *Proceedings of the ACM Symposium on Operating Systems Pinciples*. ACM, New York, 57–70.

Min, C., Kim, K., Cho, H., Lee, S.-W., and Eom, Y. I. 2012. SFS: Random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 139–154.

MPI-IO. The Los Alamos National LabMPI-IO Test. http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm.

Mullins, C. S. 2002. *Database Administration: The Complete Guide to Practices and Procedures*. Addison Wesley, Ch. 11, 308.

MySQL. http://www.mysql.com/.

Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. 2009. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proceedings of the ACM European Conference on Computer Systems*. ACM, New York, 145–158.

Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. 2006. Rethink the sync. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 1–14.

Oral, S., Wang, F., Dillow, D., Shipman, G., Miller, R., and Drokin, O. 2010. Efficient object storage journaling in a distributed parallel file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 143–154.

Ouyang, X., Nellans, D., Wipfel, R., Flynn, D., and Panda, D. K. 2011a. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. IEEE, Los Alamitos, CA, 301–311.

Ouyang, X., Rajachandrasekar, R., Besseron, X., Wang, H., Huang, J., and Panda, D. K. 2011b. CRFS: A lightweight user-level filesystem for generic checkpoint/restart. In *Proceedings of the International Conference Parallel Processing*. 375–384.

Polte, M., Simsa, J., Tantisiriroj, W., Gibson, G., Dayal, S., Chainani, M., and Uppugandla, D. K. 2008. Fast log-based concurrent writing of checkpoints. In *Proceedings of the Petascale Data Storage Workshop*.

Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2005a. Analysis and evolution of journaling file systems. In *Proceedings of the USENIX Annual Technical Conference*. 105–120.

Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2005b. IRON file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, New York, 206–220.

PVFS2. Parallel virtual file system, version 2. http://www.pvfs.org.

Rajimwale, A., Chidambaram, V., Ramamurthi, D., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. 2011. Coerced cache eviction and discreet-mode journaling: Dealing with misbehaving disks. In *Proceedings of the International Conference Dependable Systems and Networks*.

Rosenblum, M. and Ousterhout, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10,* 1, 26–52.

SATA. 2003. Serial ATA: High speed serialized AT attachment. Revision 1.0a, SerialATA Workgroup.

Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. 1993. Lightweight recoverable virtual memory. In *Proceedings of the ACM SIGOPS*. ACM, New York, 146–160.

SBC. 2005. Working draft project American National Standard, SCSI Block Commands-3, Technical Committee T10, INCITS. ftp://ftp.t10.org/t10/document.05/05-369r0.pdf.

Schindler, J., Griffin, J. L., Lumb, C. R., and Ganger, G. R. 2002. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 259–274.

Sears, R. and Brewer, E. 2006. Stasis: Flexible transactional storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 29–44.

Seltzer, M., Smith, K. A., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. 1995. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX Annual Technical Conference*. 21–21.

Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., and Stein, C. A. 2000. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the USENIX Annual Technical Conference*. 71–84.

Shin, D. I., Yu, Y. J., Kim, H. S., Eom, H., and Yeom, H. Y. 2011. Request bridging and interleaving: Improving the performance of small synchronous updates under seek-optimizing disk subsystems. *ACM Trans. Storage 7,* 2, 4:1–4:31.

Thakur, R., Gropp, W., and Lusk, E. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings of the IEEE Symposium Frontiers of Massively Parallel Computation*. 182–189.

TPCC. 1992. TPC benchmark C standard specification. Tech. rep., Transaction Processing Council.

Tweedie, S. C. 1998. Journaling the Linux ext2fs filesystem. In *LinuxExpo*. 25–29.

Verissimo, P. and Rodrigues, L. 2001. *Distributed Systems for System Architects*. Kluwer Academic, Norwell, MA.

Wang, R. Y., Anderson, T. E., and Patterson, D. A. 1999. Virtual log based file systems for a programmable disk. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 29–43.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 307–320. http://ceph.newdream.net/wiki/OSD_journal.

Woodhouse, D. 2001. JFFS: The journaling flash file system. In *Proceedings of the Linux Symposium*.

Yoshiji, A., Konishi, R., Sato, K., Hifumi, H., Tamura, Y., Kihara, S., and Moriai, S. 2009. NILFS: Continuous snapshotting filesystem for Linux. NTT Corp. http://www.nilfs.org/en/.

Zhang, Z. and Ghose, K. 2007. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of the ACM European Conference on Computer Systems*. ACM, New York, 175–187.

Received April 2012; revised July 2012; accepted October 2012