Analysis and Evolution of Journaling File Systems

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{*vijayan, dusseau, remzi*}*@cs.wisc.edu*

Abstract

We develop and apply two new methods for analyzing file system behavior and evaluating file system changes. First, *semantic block-level analysis (SBA)* combines knowledge of on-disk data structures with a trace of disk traffic to infer file system behavior; in contrast to standard benchmarking approaches, SBA enables users to understand *why* the file system behaves as it does. Second, *semantic trace playback (STP)* enables traces of disk traffic to be easily modified to represent changes in the file system implementation; in contrast to directly modifying the file system, STP enables users to rapidly gauge the benefits of new policies. We use SBA to analyze Linux ext3, ReiserFS, JFS, and Windows NTFS; in the process, we uncover many strengths and weaknesses of these journaling file systems. We also apply STP to evaluate several modifications to ext3, demonstrating the benefits of various optimizations without incurring the costs of a real implementation.

1 Introduction

Modern file systems are journaling file systems [4, 22, 29, 32]. By writing information about pending updates to a write-ahead log [12] before committing the updates to disk, journaling enables fast file system recovery after a crash. Although the basic techniques have existed for many years (*e.g.*, in Cedar [13] and Episode [9]), journaling has increased in popularity and importance in recent years; due to ever-increasing disk capacities, scan-based recovery (*e.g.*, via fsck [16]) is prohibitively slow on modern drives and RAID volumes. However, despite the popularity and importance of journaling file systems such as ext3 [32], ReiserFS [22], JFS [4], and NTFS [27] little is known about their internal policies.

Understanding how these file systems behave is important for developers, administrators, and application writers. Therefore, we believe it is time to perform a detailed analysis of journaling file systems. Most previous work has analyzed file systems from above; by writing user-level programs and measuring the time taken for various file system operations, one can elicit some salient aspects of file system performance [6, 8, 19, 26]. However, it is difficult to discover the underlying reasons for the observed performance with this approach.

In this paper we employ a novel benchmarking methodology called *semantic block-level analysis* (SBA) to trace and analyze file systems. With SBA, we induce controlled workload patterns from above the file system, but focus our analysis not only on the time taken for said operations, but also on the resulting stream of read and write requests *below* the file system. This analysis is *semantic* because we leverage information about block type (*e.g.*, whether a block request is to the journal or to an inode); this analysis is *block-level* because it interposes on the block interface to storage. By analyzing the low-level block stream in a semantically meaningful way, one can understand *why* the file system behaves as it does.

Analysis hints at how the file system could be improved, but does not reveal whether the change is worth implementing. Traditionally, for each potential improvement to the file system, one must implement the change and measure performance under various workloads; if the change gives little improvement, the implementation effort is wasted. In this paper, we introduce and apply a complementary technique to SBA called *semantic trace playback* (STP). STP enables us to rapidly suggest and evaluate file system modifications without a large implementation or simulation effort. Using real workloads and traces, we show how STP can be used effectively.

We have applied a detailed analysis to both Linux ext3 and ReiserFS and a preliminary analysis to Linux JFS and Windows NTFS. In each case, we focus on the journaling aspects of each file system. For example, we determine the events that cause data and metadata to be written to the journal or their fixed locations. We also examine how the characteristics of the workload and configuration parameters (*e.g.*, the size of the journal and the values of commit timers) impact this behavior.

Our analysis has uncovered design flaws, performance problems, and even correctness bugs in these file systems. For example, ext3 and ReiserFS make the design decision to group unrelated traffic into the same compound transaction; the result of this *tangled synchrony* is that a single disk-intensive process forces *all* write traffic to disk, particularly affecting the performance of otherwise asynchronous writers. (§3.2.1). Further, we find that both ext3 and ReiserFS artificially *limit parallelism*, by preventing the overlap of pre-commit journal writes and fixed-place updates (§3.2.2). Our analysis also reveals that in ordered and data journaling modes, ext3 exhibits *eager writing*, forcing data blocks to disk much sooner than the typical 30-second delay (§3.2.3). In addition, we find that JFS

has an *infinite write delay*, as it does not utilize commit timers and indefinitely postpones journal writes until another trigger forces writes to occur, such as memory pressure (§5). Finally, we identify four previously unknown bugs in ReiserFS that will be fixed in subsequent releases (§4.3).

The main contributions of this paper are:

• A new methodology, semantic block analysis (SBA), for understanding the internal behavior of file systems.

• A new methodology, semantic trace playback (STP), for rapidly gauging the benefits of file system modifications without a heavy implementation effort.

• A detailed analysis using SBA of two important journaling file systems, ext3 and ReiserFS, and a preliminary analysis of JFS and NTFS.

• An evaluation using STP of different design and implementation alternatives for ext3.

The rest of this paper is organized as follows. In §2 we describe our new techniques for SBA and STP. We apply these techniques to ext3, ReiserFS, JFS, and NTFS in §3, §4, §5, and §6 respectively. We discuss related work in §7 and conclude in §8.

2 Methodology

We introduce two techniques for evaluating file systems. First, semantic block analysis (SBA) enables users to understand the internal behavior and policies of the file system. Second, semantic trace playback (STP) allows users to quantify how changing the file system will impact the performance of real workloads.

2.1 Semantic Block-Level Analysis

File systems have traditionally been evaluated using one of two approaches; either one applies synthetic or real workloads and measures the resulting file system performance [6, 14, 17, 19, 20] or one collects traces to understand how file systems are used [1, 2, 21, 24, 35, 37]. However, performing each in isolation misses an interesting opportunity: by correlating the observed disk traffic with the running workload and with performance, one can often answer *why* a given workload behaves as it does.

Block-level tracing of disk traffic allows one to analyze a number of interesting properties of the file system and workload. At the coarsest granularity, one can record the *quantity* of disk traffic and how it is divided between reads and writes; for example, such information is useful for understanding how file system caching and write buffering affect performance. At a more detailed level, one can track the *block number* of each block that is read or written; by analyzing the block numbers, one can see the extent to which traffic is sequential or random. Finally, one can analyze the *timing* of each block; with timing information, one can understand when the file system initiates a burst of traffic.

By combining block-level analysis with *semantic* information about those blocks, one can infer much more about the behavior of the file system. The main difference between *semantic block analysis* (SBA) and more standard block-level tracing is that SBA analysis understands the on-disk format of the file system under test. SBA enables us to understand new properties of the file system. For example, SBA allows us to distinguish between traffic to the journal versus to in-place data and to even track individual transactions to the journal.

	Ext3	ReiserFS	JFS	NTFS
SBA Generic	1289	1289	1289	1289
SBA FS Specific	181	48	20	15
SBA Total	1470	1337	1309	1304

Table 1: **Code size of SBA drivers.** *The number of C statements (counted as the number of semicolons) needed to implement SBA for ext3 and ReiserFS and a preliminary SBA for JFS and NTFS.*

2.1.1 Implementation

The infrastructure for performing SBA is straightforward. One places a pseudo-device driver in the kernel, associates it with an underlying disk, and mounts the file system of interest (*e.g.*, ext3) on the pseudo device; we refer to this as the SBA driver. One then runs controlled microbenchmarks to generate disk traffic. As the SBA driver passes the traffic to and from the disk, it also efficiently tracks each request and response by storing a small record in a fixed-sized circular buffer. Note that by tracking the ordering of requests and responses, the pseudo-device driver can infer the order in which the requests were scheduled at lower levels of the system.

SBA requires that one interpret the *contents* of the disk block traffic. For example, one must interpret the contents of the journal to infer the type of journal block (*e.g.*, a descriptor or commit block) and one must interpret the journal descriptor block to know which data blocks are journaled. As a result, it is most efficient to semantically interpret block-level traces on-line; performing this analysis off-line would require exporting the contents of blocks, greatly inflating the size of the trace.

An SBA driver is customized to the file system under test. One concern is the amount of information that must be embedded within the SBA driver for each file system. Given that the focus of this paper is on understanding journaling file systems, our SBA drivers are embedded with enough information to interpret the placement and contents of journal blocks, metadata, and data blocks. We now analyze the complexity of the SBA driver for four journaling file systems, ext3, ReiserFS, JFS, and NTFS.

Journaling file systems have both a journal, where transactions are temporarily recorded, and fixed-location data structures, where data permanently reside. Our SBA driver distinguishes between the traffic sent to the journal and to the fixed-location data structures. This traffic is simple to distinguish in ReiserFS, JFS, and NTFS because the journal is a set of contiguous blocks, separate from the rest of the file system. However, to be backward

compatible with ext2, ext3 can treat the journal as a regular file. Thus, to determine which blocks belong to the journal, SBA uses its knowledge of inodes and indirect blocks; given that the journal does not change location after it has been created, this classification remains efficient at run-time. SBA is also able to classify the different types of journal blocks such as the descriptor block, journal data block, and commit block.

To perform useful analysis of journaling file systems, the SBA driver does not have to understand many details of the file system. For example, our driver does not understand the directory blocks or superblock of ext3 or the B+ tree structure of ReiserFS or JFS. However, if one wishes to infer additional file system properties, one may need to embed the SBA driver with more knowledge. Nevertheless, the SBA driver does not know anything about the policies or parameters of the file system; in fact, SBA can be used to infer these policies and parameters.

Table 1 reports the number of C statements required to implement the SBA driver. These numbers show that most of the code in the SBA driver (*i.e.*, 1289 statements) is for general infrastructure; only between approximately 50 and 200 statements are needed to support different journaling file systems. The ext3 specific code is more than that of the other file systems because in ext3, journal is created as a file and can span multiple block groups. In order to find the blocks belonging to the journal file, we parse the journal inode and journal indirect blocks. In Reiserfs, JFS, and NTFS the journal is contiguous and finding its blocks is trivial (even though the journal is a file in NTFS, for small journals they are contiguously allocated).

2.1.2 Workloads

SBA analysis can be used to gather useful information for any workload. However, the focus of this paper is on understanding the internal policies and behavior of the file system. As a result, we wish to construct synthetic workloads that uncover decisions made by the file system. More realistic workloads will be considered only when we apply semantic trace playback.

When constructing synthetic workloads that stress the file system, previous research has revealed a range of parameters that impact performance [8]. We have created synthetic workloads varying these parameters: the amount of data written, sequential versus random accesses, the interval between calls to `fsync`, and the amount of concurrency. We focus exclusively on write-based workloads because reads are directed to their fixed-place location, and thus do not impact the journal. When we analyze each file system, we only report results for those workloads which revealed file system policies and parameters.

2.1.3 Overhead of SBA

The processing and memory overheads of SBA are minimal for the workloads we ran as they did not generate high I/O rates. For every I/O request, the SBA driver performs the following operations to collect detailed traces:

- A gettimeofday() call during the start and end of I/O.
- A block number comparison to see if the block is a journal or fixed-location block.
- A check for a magic number on journal blocks to distinguish journal metadata from journal data.

SBA stores the trace records with details like read or write, block number, block type, time of issue and completion in an internal circular buffer. All these operations are performed only if one needs detailed traces. But for many of our analyses, it is sufficient to have cumulative statistics like the total number of journal writes and fixed-location writes. These numbers are easy to collect and require less processing within the SBA driver.

2.1.4 Alternative Approaches

One might believe that directly instrumenting a file system to obtain timing information and disk traces would be equivalent or superior to performing SBA analysis. We believe this is not the case for several reasons. First, to directly instrument the file system, one needs source code for that file system and one must re-instrument new versions as they are released; in contrast, SBA analysis does not require file system source and much of the SBA driver code can be reused across file systems and versions. Second, when directly instrumenting the file system, one may accidentally miss some of the conditions for which disk blocks are written; however, the SBA driver is guaranteed to see all disk traffic. Finally, instrumenting existing code may accidentally change the behavior of that code [36]; an efficient SBA driver will likely have no impact on file system behavior.

2.2 Semantic Trace Playback

In this section we describe semantic trace playback (STP). STP can be used to rapidly evaluate certain kinds of new file system designs, both without a heavy implementation investment and without a detailed file system simulator.

We now describe how STP functions. STP is built as a user-level process; it takes as input a trace (described further below), parses it, and issues I/O requests to the disk using the raw disk interface. Multiple threads are employed to allow for concurrency.

Ideally, STP would function by only taking a block-level trace as input (generated by the SBA driver), and indeed this is sufficient for some types of file system modifications. For example, it is straightforward to model different layout schemes by simply mapping blocks to different on-disk locations.

However, it was our desire to enable more powerful emulations with STP. For example, one issue we explore later is the effect of using byte differences in the journal, instead of storing entire blocks therein. One complication that arises is that by changing the contents of the journal,

the *timing* of block I/O changes; the thresholds that initiate I/O are triggered at a different time.

To handle emulations that alter the timing of disk I/O, more information is needed than is readily available in the low-level block trace. Specifically, STP needs to observe two high-level activities. First, STP needs to observe any file-system level operations that create dirty buffers in memory. The reason for this requirement is found in §3.2.2; when the number of uncommitted buffers reaches a threshold (in ext3, $\frac{1}{4}$ of the journal size), a commit is enacted. Similarly, when one of the interval timers expires, these blocks may have to be flushed to disk.

Second, STP needs to observe application-level calls to `fsync`; without doing so, STP cannot understand whether an I/O operation in the SBA trace is there due to a `fsync` call or due to normal file system behavior (*e.g.*, thresholds being crossed, timers going off, etc.). Without such differentiation, STP cannot emulate behaviors that are timing sensitive.

Both of these requirements are met by giving a file-system level trace as input to STP, in addition to the SBA-generated block-level trace. We currently use library-level interpositioning to trace the application of interest.

We can now qualitatively compare STP to two other standard approaches for file system evolution. In the first approach, when one has an idea for improving a file system, one simply implements the idea within the file system and measures the performance of the real system. This approach is attractive because it gives a reliable answer as to whether the idea was a real improvement, assuming that the workload applied is relevant. However, it is time consuming, particularly if the modification to the file system is non-trivial.

In the second approach, one builds an accurate simulation of the file system, and evaluates a new idea within the domain of the file system before migrating it to the real system. This approach is attractive because one can often avoid some of the details of building a real implementation and thus more quickly understand whether the idea is a good one. However, it requires a detailed and accurate simulator, the construction and maintenance of which is certainly a challenging endeavor.

STP avoids the difficulties of both of these approaches by using the low-level traces as the "truth" about how the file system behaves, and then modifying file system output (*i.e.*, the block stream) based on its simple internal models of file system behavior; these models are based on our empirical analysis found in §3.2.

Despite its advantages over traditional implementation and simulation, STP is limited in some important ways. For example, STP is best suited for evaluating design alternatives under simpler benchmarks; if the workload exhibits complex virtual memory behavior whose interactions with the file system are not modeled, the results may not be meaningful. Also, STP is limited to evaluating file system changes that are not too radical; the basic operation of the file system should remain intact. Finally, STP does not provide a means to evaluate *how* to implement a given change; rather, it should be used to understand *whether* a certain modification improves performance.

2.3 Environment

All measurements are taken on a machine running Linux 2.4.18 with a 600 MHz Pentium III processor and 1 GB of main memory. The file system under test is created on a single IBM 9LZX disk, which is separate from the root disk. Where appropriate, each data point reports the average of 30 trials; in all cases, variance is quite low.

3 The Ext3 File System

In this section, we analyze the popular Linux filesystem, ext3. We begin by giving a brief overview of ext3, and then apply semantic block-level analysis and semantic trace playback to understand its internal behavior.

3.1 Background

Linux ext3 [33, 34] is a journaling file system, built as an extension to the ext2 file system. In ext3, data and metadata are eventually placed into the standard ext2 structures, which are the fixed-location structures. In this organization (which is loosely based on FFS [15]), the disk is split into a number of *block groups*; within each block group are bitmaps, inode blocks, and data blocks. The ext3 journal (or log) is commonly stored as a file within the file system, although it can be stored on a separate device or partition. Figure 1 depicts the ext3 on-disk layout.

Information about pending file system updates is written to the journal. By forcing journal updates to disk *before* updating complex file system structures, this write-ahead logging technique [12] enables efficient crash recovery; a simple scan of the journal and a redo of any incomplete committed operations bring the file system to a consistent state. During normal operation, the journal is treated as a circular buffer; once the necessary information has been propagated to its fixed location in the ext2 structures, journal space can be reclaimed.

Journaling Modes: Linux ext3 includes three flavors of journaling: *writeback mode*, *ordered mode*, and *data journaling mode*; Figure 2 illustrates the differences between these modes. The choice of mode is made at mount time and can be changed via a remount.

In *writeback mode*, only file system metadata is journaled; data blocks are written directly to their fixed location. This mode does not enforce any ordering between the journal and fixed-location data writes, and because of this, writeback mode has the weakest consistency semantics of the three modes. Although it guarantees consistent file system metadata, it does not provide any guarantee as to the consistency of data blocks.

In *ordered journaling mode*, again only metadata writes are journaled; however, data writes to their fixed location are ordered *before* the journal writes of the metadata. In

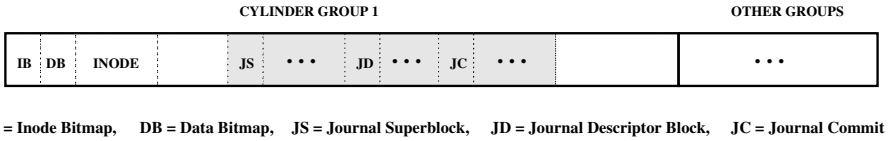

Figure 1: **Ext3 On-Disk Layout.** *The picture shows the layout of an ext3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which has bitmaps to track allocations and regions for inodes and data blocks. The ext3 journal is depicted here as a file within the first block group of the file system; it contains a superblock, various descriptor blocks to describe its contents, and commit blocks to denote the ends of transactions.*

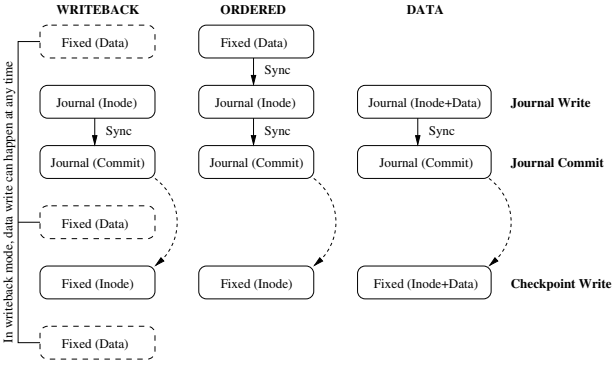

Figure 2: **Ext3 Journaling Modes.** *The diagram depicts the three different journaling modes of ext3: writeback, ordered, and data. In the diagram, time flows downward. Boxes represent updates to the file system, e.g., "Journal (Inode)" implies the write of an inode to the journal; the other destination for writes is labeled "Fixed", which is a write to the fixed in-place ext2 structures. An arrow labeled with a "Sync" implies that the two blocks are written out in immediate succession synchronously, hence guaranteeing the first completes before the second. A curved arrow indicates ordering but not immediate succession; hence, the second write will happen at some later time. Finally, for writeback mode, the dashed box around the "Fixed (Data)" block indicates that it may happen at any time in the sequence. In this example, we consider a data block write and its inode as the updates that need to be propagated to the file system; the diagrams show how the data flow is different for each of the ext3 journaling modes.*

contrast to writeback mode, this mode provides more sensible consistency semantics, where both the data and the metadata are guaranteed to be consistent after recovery.

In full *data journaling mode*, ext3 logs *both* metadata and data to the journal. This decision implies that when a process writes a data block, it will typically be written out to disk *twice*: once to the journal, and then later to its fixed ext2 location. Data journaling mode provides the same strong consistency guarantees as ordered journaling mode; however, it has different performance characteristics, in some cases worse, and surprisingly, in some cases, better. We explore this topic further (§3.2).

Transactions: Instead of considering each file system update as a separate transaction, ext3 groups many updates into a single *compound transaction* that is periodically committed to disk. This approach is relatively simple to implement [33]. Compound transactions may have better performance than more fine-grained transactions when the same structure is frequently updated in a short period of time (*e.g.*, a free space bitmap or an inode of a file that is constantly being extended) [13].

Journal Structure: Ext3 uses additional metadata structures to track the list of journaled blocks. The *journal superblock* tracks summary information for the journal, such as the block size and head and tail pointers. A *journal descriptor block* marks the beginning of a transaction and describes the subsequent journaled blocks, including their final fixed on-disk location. In data journaling mode, the descriptor block is followed by the data and metadata blocks; in ordered and writeback mode, the descriptor block is followed by the metadata blocks. In all modes, ext3 logs full blocks, as opposed to differences from old versions; thus, even a single bit change in a bitmap results in the entire bitmap block being logged. Depending upon the size of the transaction, multiple descriptor blocks each followed by the corresponding data and metadata blocks may be logged. Finally, a *journal commit block* is written to the journal at the end of the transaction; once the commit block is written, the journaled data can be recovered without loss.

Checkpointing: The process of writing journaled metadata and data to their fixed-locations is known as *checkpointing*. Checkpointing is triggered when various thresholds are crossed, *e.g.*, when file system buffer space is low, when there is little free space left in the journal, or when a timer expires.

Crash Recovery: Crash recovery is straightforward in ext3 (as it is in many journaling file systems); a basic form of *redo logging* is used. Because new updates (whether to data or just metadata) are written to the log, the process of restoring in-place file system structures is easy. During recovery, the file system scans the log for committed complete transactions; incomplete transactions are discarded. Each update in a completed transaction is simply replayed into the fixed-place ext2 structures.

3.2 Analysis of ext3 with SBA

We now perform a detailed analysis of ext3 using our SBA framework. Our analysis is divided into three categories. First, we analyze the basic behavior of ext3 as a function of the workload and the three journaling modes. Second, we isolate the factors that control when data is committed to the journal. Third, we isolate the factors that control when data is checkpointed to its fixed-place location.

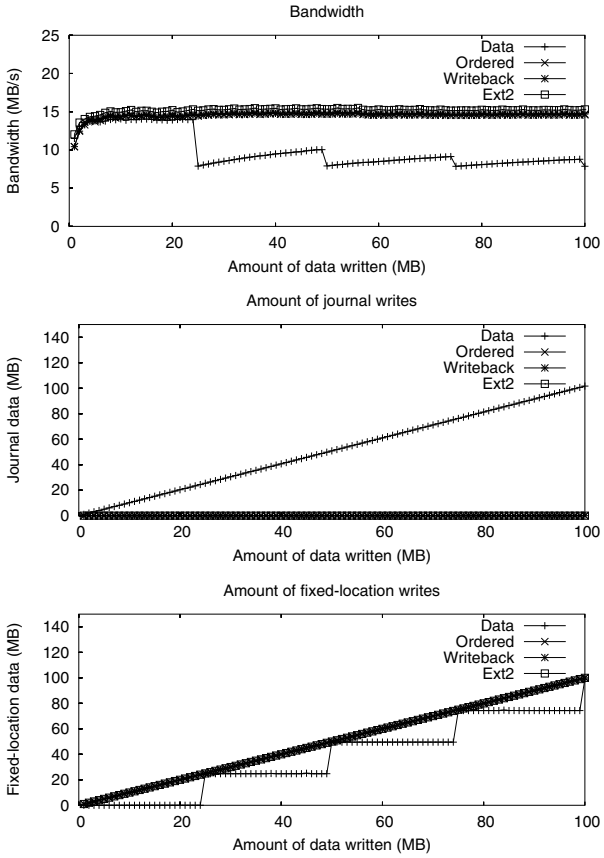

Figure 3: **Basic Behavior for Sequential Workloads in ext3.** *Within each graph, we evaluate ext2 and the three ext3 journaling modes. We increase the size of the written file along the x-axis. The workload writes to a single file sequentially and then performs an* `fsync`*. Each graph examines a different metric: the top graph shows the achieved bandwidth; the middle graph uses SBA to report the amount of journal traffic; the bottom graph uses SBA to report the amount of fixed-location traffic. The journal size is set to 50 MB.*

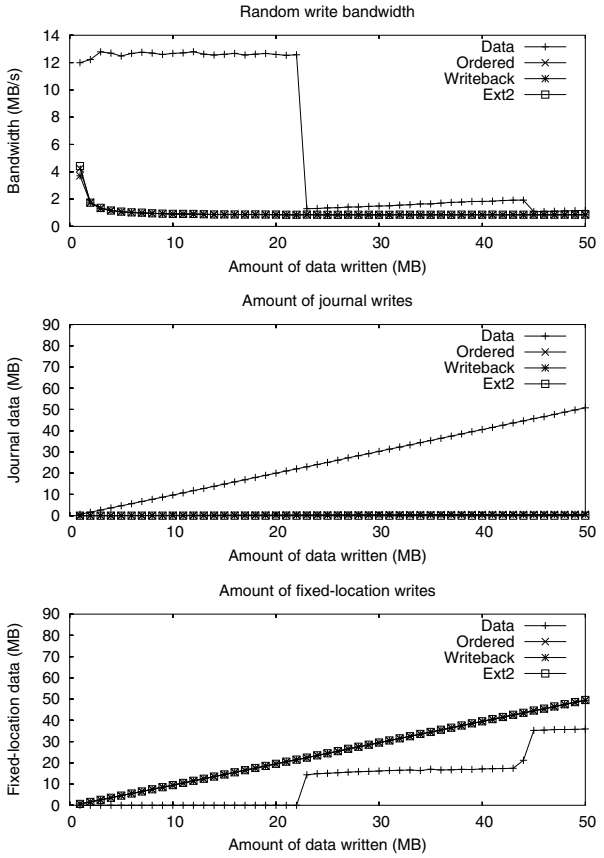

Figure 4: **Basic Behavior for Random Workloads in ext3.** *This figure is similar to Figure 3. The workload issues 4 KB writes to random locations in a single file and calls* `fsync` *once for every 256 writes. Top graph shows the bandwidth, middle graph shows the journal traffic, and the bottom graph reports the fixed-location traffic. The journal size is set to 50 MB.*

3.2.1 Basic Behavior: Modes and Workload

We begin by analyzing the basic behavior of ext3 as a function of the workload and journaling mode (*i.e.*, writeback, ordered, and full data journaling). Our goal is to understand the workload conditions that trigger ext3 to write data and metadata to the journal and to their fixed locations. We explored a range of workloads by varying the amount of data written, the sequentiality of the writes, the synchronization interval between writes, and the number of concurrent writers.

Sequential and Random Workloads: We begin by showing our results for three basic workloads. The first workload writes to a single file sequentially and then performs an `fsync` to flush its data to disk (Figure 3); the second workload issues 4 KB writes to random locations in a single file and calls `fsync` once for every 256 writes (Figure 4); the third workload again issues 4 KB random writes but calls `fsync` for every write (Figure 5). In each workload, we increase the total amount of data that it writes and observe how the behavior of ext3 changes.

The top graphs in Figures 3, 4, and 5 plot the achieved bandwidth for the three workloads; within each graph, we compare the three different journaling modes and ext2. From these bandwidth graphs we make four observations. First, the achieved bandwidth is extremely sensitive to the workload: as expected, a sequential workload achieves much higher throughput than a random workload and calling `fsync` more frequently further reduces throughput for random workloads. Second, for sequential traffic, ext2 performs slightly better than the highest performing ext3 mode: there is a small but noticeable cost to journaling for sequential streams. Third, for all workloads, ordered mode and writeback mode achieve bandwidths that are similar to ext2. Finally, the performance of data journaling is quite irregular, varying in a sawtooth pattern with the amount of data written.

These graphs of file system throughput allow us to compare performance across workloads and journaling modes, but do not enable us to infer the *cause* of the differences. To help us infer the internal behavior of the file system, we apply semantic analysis to the underlying block stream;

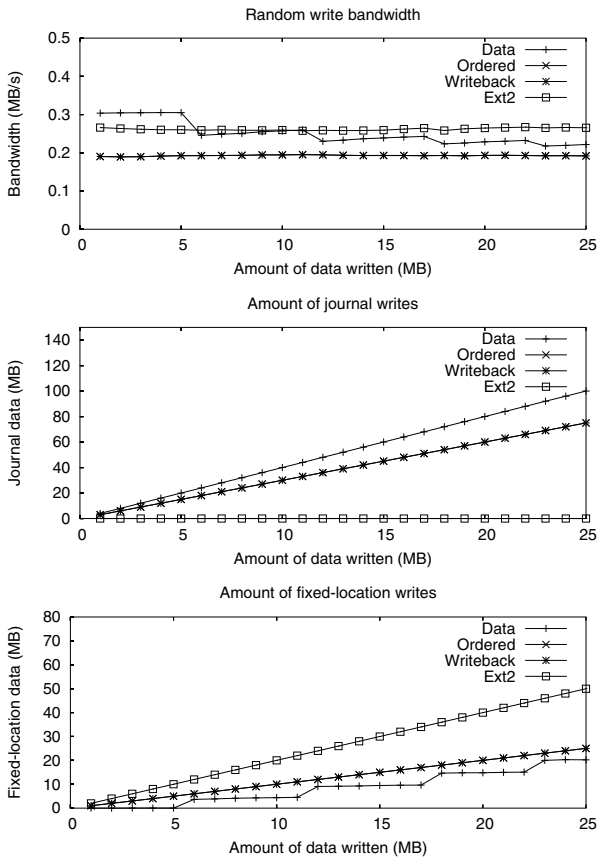

Figure 5: **Basic Behavior for Random Workloads in ext3.** *This figure is similar to Figure 3. The workload issues 4 KB random writes and calls* `fsync` *for every write. Bandwidth is shown in the first graph; journal writes and fixed-location writes are reported in the second and third graph using SBA. The journal size is set to 50 MB.*

in particular, we record the amount of journal and fixed-location traffic. This accounting is shown in the bottom two graphs of Figures 3, 4, and 5.

The second row of graphs in Figures 3, 4, and 5 quantify the amount of traffic flushed to the journal and help us to infer the events which cause this traffic. We see that, in data journaling mode, the total amount of data written to the journal is high, proportional to the amount of data written by the application; this is as expected, since both data and metadata are journaled. In the other two modes, only metadata is journaled; therefore, the amount of traffic to the journal is quite small.

The third row of Figures 3, 4, and 5 shows the traffic to the fixed location. For writeback and ordered mode the amount of traffic written to the fixed location is equal to the amount of data written by the application. However, in data journaling mode, we observe a stair-stepped pattern in the amount of data written to the fixed location. For example, with a file size of 20 MB, even though the process has called `fsync` to force the data to disk, no data is written to the fixed location by the time the application terminates; because all data is logged, the expected consistency semantics are still preserved. However, even though it is not necessary for consistency, when the application writes more data, checkpointing does occur at regular intervals; this extra traffic leads to the sawtooth bandwidth measured in the first graph. In this particular experiment with sequential traffic and a journal size of 50 MB, a checkpoint occurs when 25 MB of data is written; we explore the relationship between checkpoints and journal size more carefully in §3.2.3.

The SBA graphs also reveal why data journaling mode performs better than the other modes for asynchronous random writes. With data journaling mode, all data is written first to the log, and thus even random writes become *logically* sequential and achieve sequential bandwidth. As the journal is filled, checkpointing causes extra disk traffic, which reduces bandwidth; in this particular experiment, the checkpointing occurs near 23 MB. Finally, SBA analysis reveals that synchronous 4 KB writes do not perform well, even in data journaling mode. Forcing each small 4 KB write to the log, even in logical sequence, incurs a delay between sequential writes (not shown) and thus each write incurs a disk rotation.

Concurrency: We now report our results from running workloads containing multiple processes. We construct a workload containing two diverse classes of traffic: an asynchronous foreground process in competition with a background process. The foreground process writes out a 50 MB file without calling `fsync`, while the background process repeatedly writes a 4 KB block to a random location, optionally calls `fsync`, and then sleeps for some period of time (*i.e.*, the "sync interval"). We focus on data journaling mode, but the effect holds for ordered journaling mode too (not shown).

In Figure 6 we show the impact of varying the mean "sync interval" of the background process on the performance of the foreground process. The first graph plots the bandwidth achieved by the foreground asynchronous process, depending upon whether it competes against an asynchronous or synchronous background process. As expected, when the foreground process runs with an asynchronous background process, its bandwidth is uniformly high and matches in-memory speeds. However, when the foreground process competes with a synchronous background process, its bandwidth drops to disk speeds.

The SBA analysis in the second graph reports the amount of journal data, revealing that the more frequently the background process calls `fsync`, the more traffic is sent to the journal. In fact, the amount of journal traffic is equal to the sum of the foreground and background process traffic written in that interval, not that of only the background process. This effect is due to the implementation of compound transactions in ext3: all file system updates add their changes to a global transaction, which is eventually committed to disk.

This workload reveals the potentially disastrous consequences of grouping unrelated updates into the same com-

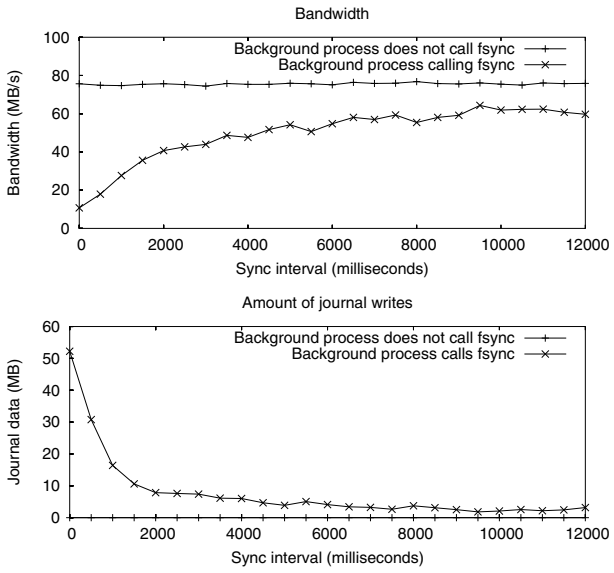

Figure 6: **Basic Behavior for Concurrent Writes in ext3.** *Two processes compete in this workload: a foreground process writing a sequential file of size 50 MB and a background process writing out 4 KB, optionally calling* `fsync`*, sleeping for the "sync interval", and then repeating. Along the x-axis, we increase the sync interval. In the top graph, we plot the bandwidth achieved by the foreground process in two scenarios: with the background process either calling or not calling* `fsync` *after each write. In the bottom graph, the amount of data written to disk during both sets of experiments is shown.*

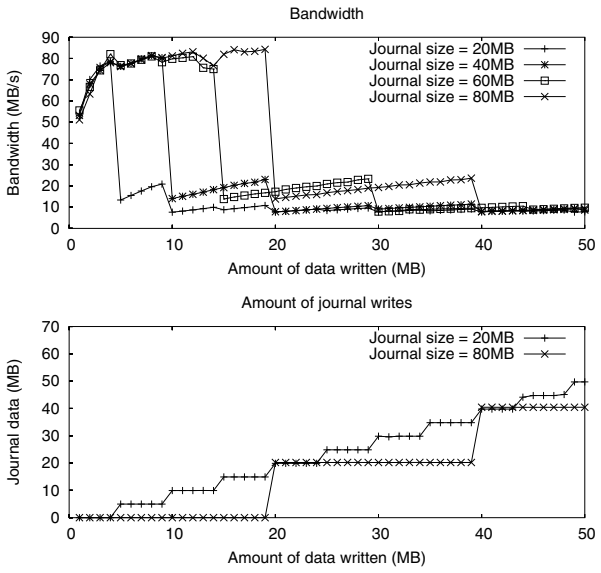

Figure 7: **Impact of Journal Size on Commit Policy in ext3.** *The topmost figure plots the bandwidth of data journaling mode under different-sized file writes. Four lines are plotted representing four different journal sizes. The second graph shows the amount of log traffic generated for each of the experiments (for clarity, only two of the four journal sizes are shown).*

pound transaction: all traffic is committed to disk at the same rate. Thus, even asynchronous traffic must wait for synchronous updates to complete. We refer to this negative effect as *tangled synchrony* and explore the benefits of untangling transactions in §3.3.3 using STP.

3.2.2 Journal Commit Policy

We next explore the conditions under which ext3 commits transactions to its on-disk journal. As we will see, two factors influence this event: the size of the journal and the settings of the commit timers.

In these experiments, we focus on data journaling mode; since this mode writes both metadata and data to the journal, the traffic sent to the journal is most easily seen in this mode. However, writeback and ordered modes commit transactions using the same policies. To exercise log commits, we examine workloads in which data is not explicitly forced to disk by the application (*i.e.*, the process does not call `fsync`); further, to minimize the amount of metadata overhead, we write to a single file.

Impact of Journal Size: The size of the journal is a configurable parameter in ext3 that contributes to when updates are committed. By varying the size of the journal and the amount of data written in the workload, we can infer the amount of data that triggers a log commit. Figure 7 shows the resulting bandwidth and the amount of journal traffic, as a function of file size and journal size. The first graph shows that when the amount of data written by the application (to be precise, the number of dirty uncommitted buffers, which includes both data and metadata) reaches $\frac{1}{4}$ the size of the journal, bandwidth drops considerably. In fact, in the first performance regime, the observed bandwidth is equal to in-memory speeds.

Our semantic analysis, shown in the second graph, reports the amount of traffic to the journal. This graph reveals that metadata and data are forced to the journal when it is equal to $\frac{1}{4}$ the journal size. Inspection of Linux ext3 code confirms this threshold. Note that the threshold is the same for ordered and writeback modes (not shown); however, it is triggered much less frequently since only metadata is logged.

Impact of Timers: In Linux 2.4 ext3, three timers have some control over when data is written: the metadata commit timer and the data commit timer, both managed by the kupdate daemon, and the commit timer managed by the kjournal daemon. The system-wide kupdate daemon is responsible for flushing dirty buffers to disk; the kjournal daemon is specialized for ext3 and is responsible for committing ext3 transactions. The strategy for ext2 is to flush metadata frequently (*e.g.*, every 5 seconds) while delaying data writes for a longer time (*e.g.*, every 30 seconds). Flushing metadata frequently has the advantage that the file system can approach FFS-like consistency without a severe performance penalty; delaying data writes has the advantage that files that are deleted quickly do not tax the disk. Thus, mapping the ext2 goals to the ext3 timers leads to default values of 5 seconds for the kupdate metadata timer, 5 seconds for the kjournal timer,

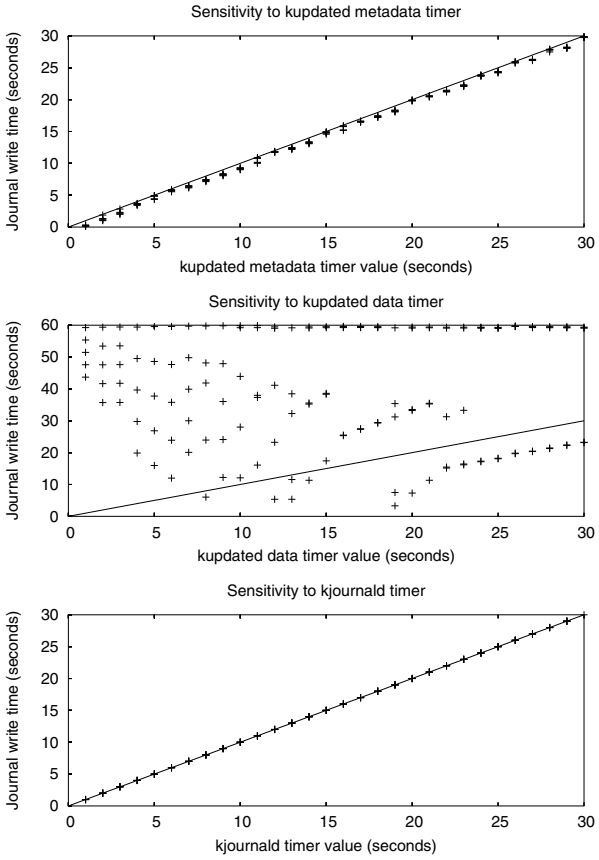

Figure 8: **Impact of Timers on Commit Policy in ext3.** *In each graph, the value of one timer is varied across the x-axis, and the time of the first write to the journal is recorded along the y-axis. When measuring the impact of a particular timer, we set the other timers to 60 seconds and the journal size to 50 MB so that they do not affect the measurements.*

and 30 seconds for the kupdate data timer.

We measure how these timers affect when transactions are committed to the journal. To ensure that a specific timer influences journal commits, we set the journal size to be sufficiently large and set the other timers to a large value (*i.e.*, 60 s). For our analysis, we observe when the first write appears in the journal. Figure 8 plots our results varying one of the timers along the x-axis, and plotting the time that the first log write occurs along the y-axis.

The first graph and the third graph show that the kupdate daemon metadata commit timer and the kjournal daemon commit timer control the timing of log writes: the data points along $y = x$ indicate that the log write occurred precisely when the timer expired. Thus, traffic is sent to the log at the minimum of those two timers. The second graph shows that the kupdate daemon data timer does not influence the timing of log writes: the data points are not correlated with the x-axis. As we will see, this timer influences when data is written to its fixed location.

Interaction of Journal and Fixed-Location Traffic: The timing between writes to the journal and to the fixed-location data must be managed carefully for consistency. In fact, the difference between writeback and ordered mode is in this timing: writeback mode does not enforce any ordering between the two, whereas ordered mode ensures that the data is written to its fixed location before the commit block for that transaction is written to the journal. When we performed our SBA analysis, we found a performance deficiency in how ordered mode is implemented.

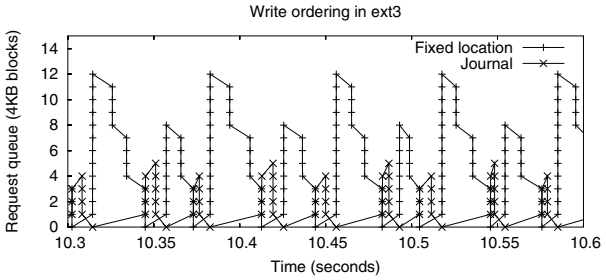

Figure 9: **Interaction of Journal and Fixed-Location Traffic in ext3.** *The figure plots the number of outstanding writes to the journal and fixed-location disks. In this experiment, we run five processes, each of which issues 16 KB random synchronous writes. The file system has a 50 MB journal and is running in ordered mode; the journal is configured to run on a separate disk.*

We consider a workload that synchronously writes a large number of random 16 KB blocks and use the SBA driver to separate journal and fixed-location data. Figure 9 plots the number of concurrent writes to each data type over time. The figure shows that writes to the journal and fixed-place data do *not* overlap. Specifically, ext3 issues the data writes to the fixed location and waits for completion, then issues the journal writes to the journal and again waits for completion, and finally issues the final commit block and waits for completion. We observe this behavior irrespective of whether the journal is on a separate device or on the same device as the file system. Inspection of the ext3 code confirms this observation. However, the first wait is not needed for correctness. In those cases where the journal is configured on a separate device, this extra wait can severely limit concurrency and performance. Thus, ext3 has *falsely limited parallelism*. We will use STP to fix this timing problem in §3.3.4.

3.2.3 Checkpoint Policy

We next turn our attention to checkpointing, the process of writing data to its fixed location within the ext2 structures. We will show that checkpointing in ext3 is again a function of the journal size and the commit timers, as well as the synchronization interval in the workload. We focus on data journaling mode since it is the most sensitive to journal size. To understand when checkpointing occurs, we construct workloads that periodically force data to the journal (*i.e.*, call `fsync`) and we observe when data is subsequently written to its fixed location.

Impact of Journal Size: Figure 10 shows our SBA results

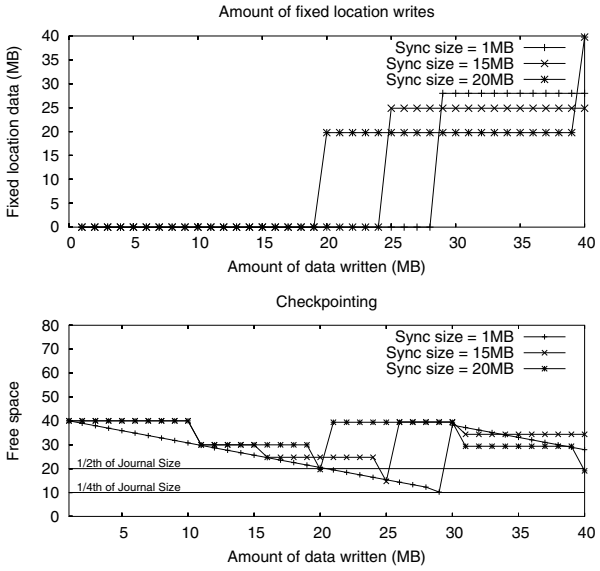

Figure 10: **Impact of Journal Size on Checkpoint Policy in ext3.** *We consider a workload where a certain amount of data (as indicated by the x-axis value) is written sequentially, with a* `fsync` *issued after every 1, 15, or 20 MB. The first graph uses SBA to plot the amount of fixed-location traffic. The second graph uses SBA to plot the amount of free space in the journal.*

as a function of file size and synchronization interval for a single journal size of 40 MB. The first graph shows the amount of data written to its fixed ext2 location at the end of each experiment. We can see that the point at which checkpointing occurs varies across the three sync intervals; for example, with a 1 MB sync interval (*i.e.*, when data is forced to disk after every 1 MB worth of writes), checkpoints occur after approximately 28 MB has been committed to the log, whereas with a 20 MB sync interval, checkpoints occur after 20 MB. To illustrate what triggers a checkpoint, in the second graph, we plot the amount of journal free space immediately preceding the checkpoint. By correlating the two graphs, we see that checkpointing occurs when the amount of free space is between $\frac{1}{4}$-th and $\frac{1}{2}$-th of the journal size. The precise fraction depends upon the synchronization interval, where smaller sync amounts allow checkpointing to be postponed until there is less free space in the journal.[1] We have confirmed this same relationship for other journal sizes (not shown).

Impact of Timers: We examine how the system timers impact the timing of checkpoint writes to the fixed locations using the same workload as above. Here, we vary the kupdate data timer while setting the other timers to five seconds. Figure 11 shows how the kupdate data timer impacts when data is written to its fixed location. First, as seen previously in Figure 8, the log is updated after the five second timers expire. Then, the checkpoint write occurs later by the amount specified by the kupdate data timer, at a five second granularity; further experiments (not shown here) reveal that this granularity is controlled by the kupdate metadata timer.

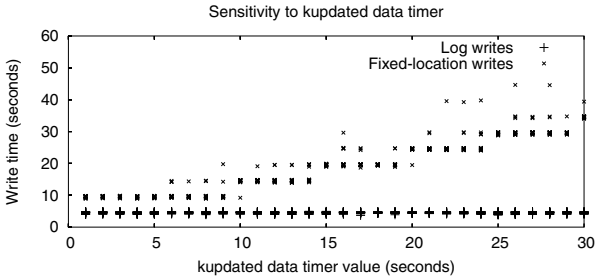

Figure 11: **Impact of Timers on Checkpoint Policy in ext3.** *The figure plots the relationship between the time that data is first written to the log and then checkpointed as dependent on the value of the kupdate data timer. The scatter plot shows the results of multiple (30) runs. The process that is running writes 1 MB of data (no* `fsync`*); data journaling mode is used, with other timers set to 5 seconds and a journal size of 50 MB.*

Our analysis reveals that the ext3 timers do not lead to the same timing of data and metadata traffic as in ext2. Ordered and data journaling modes force data to disk either before or at the time of metadata writes. Thus, *both* data and metadata are flushed to disk frequently. This timing behavior is the largest potential performance differentiator between ordered and writeback modes. Interestingly, this frequent flushing has a potential advantage; by forcing data to disk in a more timely manner, large disk queues can be avoided and overall performance improved [18]. The disadvantage of early flushing, however, is that temporary files may be written to disk before subsequent deletion, increasing the overall load on the I/O system.

3.2.4 Summary of Ext3

Using SBA, we have isolated a number of features within ext3 that can have a strong impact on performance.

- The journaling mode that delivers the best performance depends strongly on the workload. It is well known that random workloads perform better with logging [25]; however, the relationship between the size of the journal and the amount of data written by the application can have an even larger impact on performance.
- Ext3 implements compound transactions in which unrelated concurrent updates are placed into the same transaction. The result of this *tangled synchrony* is that all traffic in a transaction is committed to disk at the same rate, which results in disastrous performance for asynchronous traffic when combined with synchronous traffic.

[1] The exact amount of free space that triggers a checkpoint is not straightforward to derive for two reasons. First, ext3 reserves some amount of journal space for overhead such as descriptor and commit blocks. Second, ext3 reserves space in the journal for the currently committing transaction (*i.e.*, the synchronization interval). Although we have derived the free space function more precisely, we do not feel this very detailed information is particularly enlightening; therefore, we simply say that checkpointing occurs when free space is somewhere between $\frac{1}{4}$-th and $\frac{1}{2}$-th of the journal size.

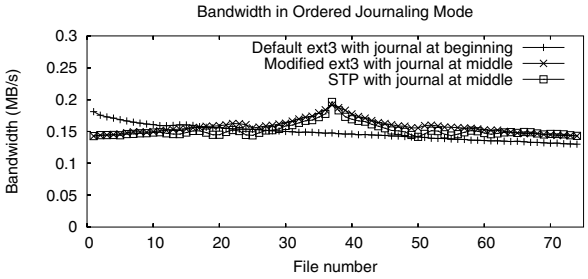

Figure 12: **Improved Journal Placement with STP.** *We compare three placements of the journal: at the beginning of the partition (the ext3 default), modeled in the middle of the file system using STP, and in the middle of the file system. 50 MB files are created across the file system; a file is chosen, as indicated by the number along the x-axis, and the workload issues 4 KB synchronous writes to that file.*

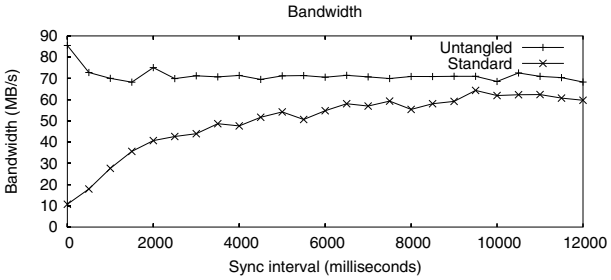

Figure 13: **Untangling Transaction Groups with STP.** *This experiment is identical to that described in Figure 6, with one addition: we show performance of the foreground process with untangled transactions as emulated with STP.*

• In ordered mode, ext3 does not overlap any of the writes to the journal and fixed-place data. Specifically, ext3 issues the data writes to the fixed location and waits for completion, then issues the journal writes to the journal and again waits for completion, and finally issues the final commit block and waits for completion; however, the first wait is not needed for correctness. When the journal is placed on a separate device, this *falsely limited parallelism* can harm performance.

• In ordered and data journaling modes, when a timer flushes meta-data to disk, the corresponding data must be flushed as well. The disadvantage of this *eager writing* is that temporary files may be written to disk, increasing the I/O load.

3.3 Evolving ext3 with STP

In this section, we apply STP and use a wider range of workloads and traces to evaluate various modifications to ext3. To demonstrate the accuracy of the STP approach, we begin with a simple modification that varies the placement of the journal. Our SBA analysis pointed to a number of improvements for ext3, which we can quantify with STP: the value of using different journaling modes depending upon the workload, having separate transactions for each update, and overlapping pre-commit journal writes with data updates in ordered mode. Finally, we use STP to evaluate *differential journaling*, in which block differences are written to the journal.

3.3.1 Journal Location

Our first experiment with STP quantifies the impact of changing a simple policy: the placement of the journal. The default ext3 creates the journal as a regular file at the beginning of the partition. We start with this policy because we are able to validate STP: the results we obtain with STP are quite similar to those when we implement the change within ext3 itself.

We construct a workload that stresses the placement of the journal: a 4 GB partition is filled with 50 MB files and the benchmark process issues random, synchronous 4 KB writes to a chosen file. In Figure 12 we vary which file is chosen along the x-axis. The first line in the graph shows the performance for ordered mode in default ext3: bandwidth drops by nearly 30% when the file is located far from the journal. SBA analysis (not shown) confirms that this performance drop occurs as the seek distance increases between the writes to the file and the journal.

To evaluate the benefit of placing the journal in the middle of the disk, we use STP to remap blocks. For validation, we also coerce ext3 to allocate its journal in the middle of the disk, and compare results. Figure 12 shows that the STP predicted performance is nearly identical to this version of ext3. Furthermore, we see that worst-case behavior is avoided; by placing the journal in the middle of the file system instead of at the beginning, the longest seeks across the entire volume are avoided during synchronous workloads (*i.e.*, workloads that frequently seek between the journal and the ext2 structures).

3.3.2 Journaling Mode

As shown in §3.2.1, different workloads perform better with different journaling modes. For example, random writes perform better in data journaling mode as the random writes are written sequentially into the journal, but large sequential writes perform better in ordered mode as it avoids the extra traffic generated by data journaling mode. However, the journaling mode in ext3 is set at mount time and remains fixed until the next mount.

Using STP, we evaluate a new adaptive journaling mode that chooses the journaling mode for each transaction according to writes that are in the transaction. If a transaction is sequential, it uses ordered journaling; otherwise, it uses data journaling.

To demonstrate the potential performance benefits of adaptive journaling, we run a portion of a trace from HP Labs [23] after removing the inter-arrival times between the I/O calls and compare ordered mode, data journaling mode, and our adaptive approach. The trace completes in 83.39 seconds and 86.67 seconds, in ordered and data journaling modes, respectively; however, with STP adaptive journaling, the trace completes in only 51.75 seconds. Because the trace has both sequential and random write

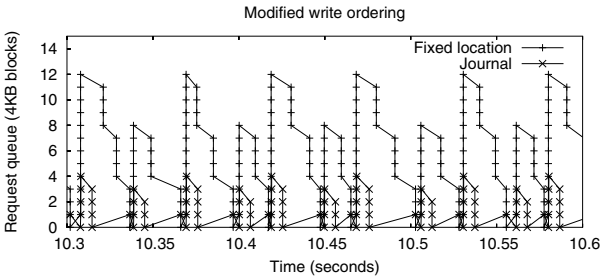

Figure 14: **Changing the Interaction of Journal and Fixed-Location Traffic with STP.** *The same experiment is run as in Figure 9; however, in this run, we use STP to issue the pre-commit journal writes and data writes concurrently. We plot the STP emulated performance, and also made this change to ext3 directly, obtaining the same resultant performance.*

phases, adaptive journaling out performs any single-mode approach.

3.3.3 Transaction Grouping

Linux ext3 groups all updates into system-wide compound transactions and commits them to disk periodically. However, as we have shown in 3.2.1, if just a single update stream is synchronous, it can have a dramatic impact on the performance of other asynchronous streams, by transforming in-memory updates into disk-bound ones.

Using STP, we show the performance of a file system that *untangles* these traffic streams, only forcing the process that issues the `fsync` to commit its data to disk. Figure 13 plots the performance of an asynchronous sequential stream in the presence of a random synchronous stream. Once again, we vary the interval of updates from the synchronous process, and from the graph, we can see that segregated transaction grouping is effective; the asynchronous I/O stream is unaffected by synchronous traffic.

3.3.4 Timing

We show that STP can quantify the cost of falsely limited parallelism, as discovered in 3.2.2, where pre-commit journal writes are not overlapped with data updates in ordered mode. With STP, we modify the timing so that journal and fixed-location writes are all initiated simultaneously; the commit transaction is written only after the previous writes complete. We consider the same workload of five processes issuing 16 KB random synchronous writes and with the journal on a separate disk.

Figure 14 shows that STP can model this implementation change by modifying the timing of the requests. For this workload, STP predicts an improvement of about 18%; this prediction matches what we achieve when ext3 is changed directly. Thus, as expected, increasing the amount of concurrency improves performance when the journal is on a separate device.

3.3.5 Journal Contents

Ext3 uses physical logging and writes new blocks in their entirety to the log. However, if whole blocks are journaled irrespective of how many bytes have changed in the block, journal space fills quickly, increasing both commit and checkpoint frequency.

Using STP, we investigate *differential journaling*, where the file system writes block differences to the journal instead of new blocks in their entirety. This approach can potentially reduce disk traffic noticeably, if dirty blocks are not substantially different from their previous versions. We focus on data journaling mode, as it generates by far the most journal traffic; differential journaling is less useful for the other modes.

To evaluate whether differential journaling matters for real workloads, we analyze SBA traces underneath two database workloads modeled on TPC-B [30] and TPC-C [31]. The former is a simple application-level implementation of a debit-credit benchmark, and the latter a realistic implementation of order-entry built on top of Postgres. With data journaling mode, the amount of data written to the journal is reduced by a factor of 200 for TPC-B and a factor of 6 under TPC-C. In contrast, for ordered and writeback modes, the difference is minimal (less than 1%); in these modes, only metadata is written to the log, and applying differential journaling to said metadata blocks makes little difference in total I/O volume.

4 ReiserFS

We now focus on a second Linux journaling filesystem, ReiserFS. In this section, we focus on the chief differences between ext3 and ReiserFS. Due to time constraints, we do not use STP to explore changes to ReiserFS.

4.1 Background

The general behavior of ReiserFS is similar to ext3. For example, both file systems have the same three journaling modes and both have compound transactions. However, ReiserFS differs from ext3 in three primary ways.

First, the two file systems use different on-disk structures to track their fixed-location data. Ext3 uses the same structures as ext2; for improved scalability, ReiserFS uses a B+ tree, in which data is stored on the leaves of the tree and the metadata is stored on the internal nodes. Since the impact of the fixed-location data structures is not the focus of this paper, this difference is largely irrelevant.

Second, the format of the journal is slightly different. In ext3, the journal can be a file, which may be anywhere in the partition and may not be contiguous. The ReiserFS journal is not a file and is instead a contiguous sequence of blocks at the beginning of the file system; as in ext3, the ReiserFS journal can be put on a different device. Further, ReiserFS limits the journal to a maximum of 32 MB.

Third, ext3 and ReiserFS differ slightly in their journal contents. In ReiserFS, the fixed locations for the blocks in the transaction are stored not only in the descriptor block but also in the commit block. Also, unlike ext3, ReiserFS uses only one descriptor block in every compound

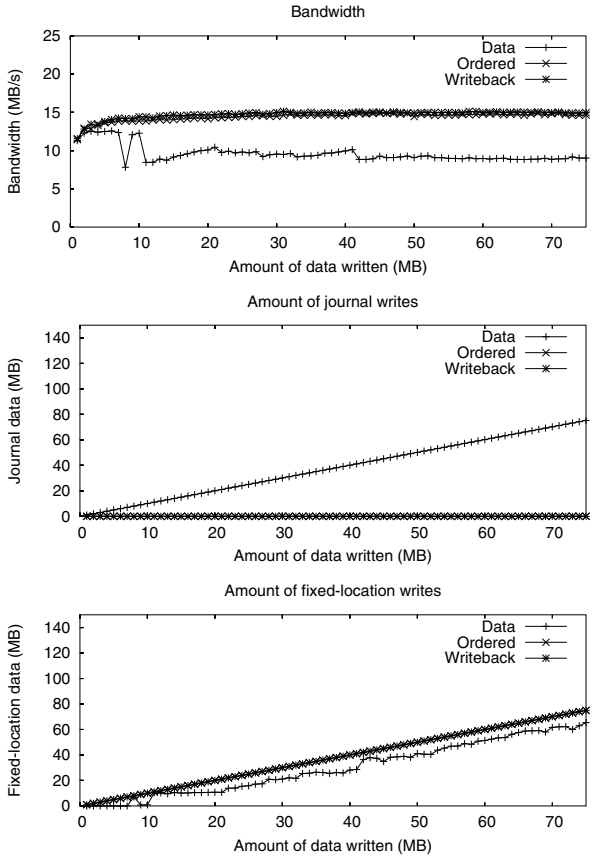

Figure 15: **Basic Behavior for Sequential Workloads in ReiserFS.** *Within each graph, we evaluate the three ReiserFS journaling modes. We consider a single workload in which the size of the sequentially written file is increased along the x-axis. Each graph examines a different metric: the first hows the achieved bandwidth; the second uses SBA to report the amount of journal traffic; the third uses SBA to report the amount of fixed-location traffic. The journal size is set to 32 MB.*

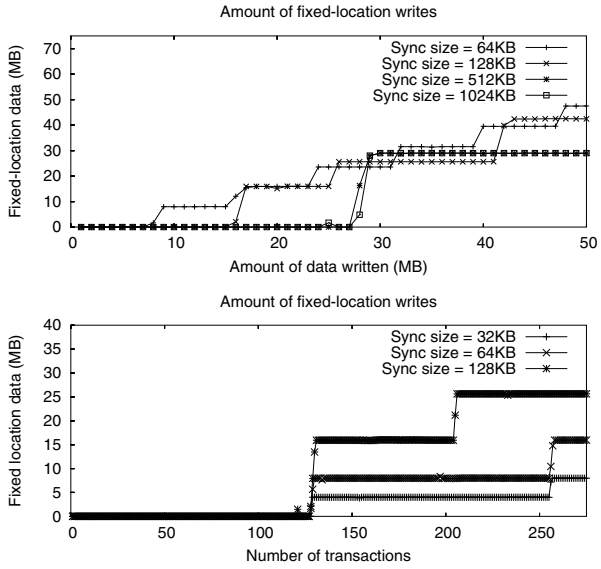

Figure 16: **Impact of Journal Size and Transactions on Checkpoint Policy in ReiserFS.** *We consider workloads where data is sequentially written and an* `fsync` *is issued after a specified amount of data. We use SBA to report the amount of fixed-location traffic. In the first graph, we vary the amount of data written; in the second graph, we vary the number of transactions, defined as the number of calls to* `fsync`*.*

transaction, which limits the number of blocks that can be grouped in a transaction.

4.2 Semantic Analysis of ReiserFS

We have performed identical experiments on ReiserFS as we have on ext3. Due to space constraints, we present only those results which reveal significantly different behavior across the two file systems.

4.2.1 Basic Behavior: Modes and Workload

Qualitatively, the performance of the three journaling modes in ReiserFS is similar to that of ext3: random workloads with infrequent synchronization perform best with data journaling; otherwise, sequential workloads generally perform better than random ones and write-back and ordered modes generally perform better than data journaling. Furthermore, ReiserFS groups concurrent transactions into a single compound transaction, as did ext3. The primary difference between the two file systems occurs for sequential workloads with data journaling. As shown in the first graph of Figure 15, the throughput of data journaling mode in ReiserFS does not follow the sawtooth pattern. An initial reason for this is found through SBA analysis. As seen in the second and third graphs of Figure 15, almost all of the data is written not only to the journal, but is also checkpointed to its in-place location. Thus, ReiserFS appears to checkpoint data much more aggressively than ext3, which we will explore in §4.2.3.

4.2.2 Journal Commit Policy

We explore the factors that impact when ReiserFS commits transactions to the log. Again, we focus on data journaling, since it is the most sensitive. We postpone exploring the impact of the timers until §4.2.3.

We previously saw that ext3 commits data to the log when approximately $\frac{1}{4}$ of the log is filled or when a timer expires. Running the same workload that does not force data to disk (*i.e.*, does not call `fsync`) on ReiserFS and performing SBA analysis, we find that ReiserFS uses a different threshold: depending upon whether the journal size is below or above 8 MB, ReiserFS commits data when about 450 blocks (*i.e.*, 1.7 MB) or 900 blocks (*i.e.*, 3.6 MB) are written. Given that ReiserFS limits journal size to at most 32 MB, these fixed thresholds appear sufficient.

Finally, we note that ReiserFS also has falsely limited parallelism in ordered mode. Like ext3, ReiserFS forces the data to be flushed to its fixed location before it issues any writes to the journal.

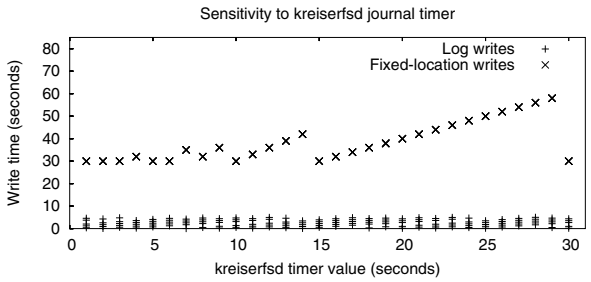

Figure 17: **Impact of Timers in ReiserFS.** *The figure plots the relationship between the time that data is written and the value of the kreiserfs timer. The scatter plot shows the results of multiple (30) runs. The process that is running writes 1 MB of data (no* `fsync`*); data journaling mode is used, with other timers set to 5 seconds and a journal size of 32 MB.*

4.2.3 Checkpoint Policy

We also investigate the conditions which trigger ReiserFS to checkpoint data to its fixed-place location; this policy is more complex in ReiserFS. In ext3, we found that data was checkpointed when the journal was $\frac{1}{4}$ to $\frac{1}{2}$ full. In ReiserFS, the point at which data is checkpointed depends not only on the free space in the journal, but also on the number of concurrent transactions. We again consider workloads that periodically force data to the journal by calling `fsync` at different intervals.

Our results are shown in Figure 16. The first graph shows the amount of data checkpointed as a function of the amount of data written; in all cases, data is checkpointed before $\frac{7}{8}$ of the journal is filled. The second graph shows the amount of data checkpointed as a function of the number of transactions. This graph shows that data is checkpointed at least at intervals of 128 transactions; running a similar workload on ext3 reveals no relationship between the number of transactions and checkpointing. Thus, ReiserFS checkpoints data whenever either journal free space drops below 4 MB or when there are 128 transactions in the journal.

As with ext3, timers control when data is written to the journal and to the fixed locations, but with some differences: in ext3, the kjournal daemon is responsible for committing transactions, whereas in ReiserFS, the kreiserfs daemon has this role. Figure 17 shows the time at which data is written to the journal and to the fixed location as the kreiserfs timer is increased; we make two conclusions. First, log writes always occur within the first five seconds of the data write by the application, regardless of the timer value. Second, the fixed-location writes occur only when the elapsed time is both greater than 30 seconds and a multiple of the kreiserfs timer value. Thus, the ReiserFS timer policy is simpler than that of ext3.

4.3 Finding Bugs

SBA analysis is useful not only for inferring the policies of filesystems, but also for finding cases that have not been implemented correctly. With SBA analysis, we have found a number of problems with the ReiserFS implementation that have not been reported elsewhere. In each case, we identified the problem because the SBA driver did not observe some disk traffic that it expected. To verify these problems, we have also examined the code to find the cause and have suggested corresponding fixes to the ReiserFS developers.

- In the first transaction after a mount, the `fsync` call returns before any of the data is written. We tracked this aberrant behavior to an incorrect initialization.
- When a file block is overwritten in writeback mode, its stat information is not updated. This error occurs due to a failure to update the inode's transaction information.
- When committing old transactions, dirty data is not always flushed. We tracked this to erroneously applying a condition to prevent data flushing during journal replay.
- Irrespective of changing the journal thread's wake up interval, dirty data is not flushed. This problem occurs due to a simple coding error.

5 The IBM Journaled File System

In this section, we describe our experience performing a preliminary SBA analysis of the Journaled File System (JFS). We began with a rudimentary understanding of JFS from what we were able to obtain through documentation [3]; for example, we knew that the journal is located by default at the end of the partition and is treated as a contiguous sequence of blocks and that one cannot specify the journaling mode.

Due to the fact that we knew less about this file system before we began, we found we needed to apply a new analysis technique as well: in some cases we filtered out traffic and then rebooted the system so that we could infer whether the filtered traffic was necessary for consistency or not. For example, we used this technique to understand the journaling mode of JFS. From this basic starting point, and without examining JFS code, we were able to learn a number of interesting properties about JFS.

First, we inferred that JFS uses ordered journaling mode. Due to the small amount of traffic to the journal, it was obvious that it was not employing data journaling. To differentiate between writeback and ordered modes, we observed that the ordering of writes matched that of ordered mode. That is, when a data block is written by the application, JFS orders the write such that the data block is written successfully before the metadata writes are issued.

Second, we determined that JFS does logging at the record level. That is, whenever an inode, index tree, or directory tree structure changes, only that structure is logged instead of the entire block containing the structure. As a result, JFS writes fewer journal blocks than ext3 and ReiserFS for the same operations.

Third, JFS does not by default group concurrent updates into a single compound transaction. Running the same experiment as we performed in Figure 6, we see that

the bandwidth of the asynchronous traffic is very high irrespective of whether there is a synchronous traffic in the background. However, there are circumstances in which transactions are grouped: for example, if the write commit records are on the same log page.

Finally, there are no commit timers in JFS and the fixed-location writes happen whenever the kupdate daemon's timer expires. However, the journal writes are never triggered by the timer: journal writes are indefinitely postponed until there is another trigger such as memory pressure or an unmount operation. This *infinite write delay* limits reliability, as a crash can result in data loss even for data that was written minutes or hours before.

6 Windows NTFS

In this section, we explain our analysis of NTFS. NTFS is a journaling file system that is used as the default file system on Windows operating systems such as XP, 2000, and NT. Although the source code or documentation of NTFS is not publicly available, tools for finding the NTFS file layout exist [28].

We ran the Windows XP operating system on top of VMware on a Linux machine. The pseudo device driver was exported as a SCSI disk to the Windows and a NTFS file system was constructed on top of the pseudo device. We ran simple workloads on NTFS and observed traffic within the SBA driver for our analysis.

Every object in NTFS is a file. Even metadata is stored in terms of files. The journal itself is a file and is located almost at the center of the file system. We used the ntfsprogs tools to discover journal file boundaries. Using the journal boundaries we were able to distinguish journal traffic from fixed-location traffic.

From our analysis, we found that NTFS does not do data journaling. This can be easily verified by the amount of data traffic observed by the SBA driver. We also found that NTFS, similar to JFS, does not do block-level journaling. It journals metadata in terms of records. We verified that whole blocks are not journaled in NTFS by matching the contents of the fixed-location traffic to the contents of the journal traffic.

We also inferred that NTFS performs ordered journaling. On data writes, NTFS waits until the data block writes to the fixed-location complete before writing the metadata blocks to the journal. We confirmed this ordering by using the SBA driver to delay the data block writes upto 10 seconds and found that the following metadata writes to the journal are delayed by the corresponding amount.

7 Related Work

Journaling Studies: Journaling file systems have been studied in detail. Most notably, Seltzer *et al.* [26] compare two variants of a journaling FFS to soft updates [11], a different technique for managing metadata consistency for file systems. Although the authors present no direct observation of low-level traffic, they are familiar enough with the systems (indeed, they are the implementors!) to explain behavior and make "semantic" inferences. For example, to explain why journaling performance drops in a delete benchmark, the authors report that the file system is "forced to read the first indirect block in order to reclaim the disk blocks it references" ([26], Section 8.1). A tool such as SBA makes such expert observations more readily available to all. Another recent study compares a range of Linux file systems, including ext2, ext3, ReiserFS, XFS, and JFS [7]. This work evaluates which file systems are fastest for different benchmarks, but gives little explanation as to *why* one does well for a given workload.

File System Benchmarks: There are many popular file system benchmarks, such as IOzone [19], Bonnie [6], lmbench [17], the modified Andrew benchmark [20], and PostMark [14]. Some of these (IOZone, Bonnie, lmbench) perform synthetic read/write tests to determine throughput; others (Andrew, Postmark) are intended to model "realistic" application workloads. Uniformly, all measure overall throughput or runtime to draw high-level conclusions about the file system. In contrast to SBA, none are intended to yield low-level insights about the internal policies of the file system.

Perhaps the most related to our work is Chen and Patterson's self-scaling benchmark [8]. In this work, the benchmarking framework conducts a search over the space of possible workload parameters (*e.g.*, sequentiality, request size, total workload size, and concurrency), and hones in on interesting parts of the workload space. Interestingly, some conclusions about file system behavior can be drawn from the resultant output, such as the size of the file cache. Our approach is not nearly as automated; instead, we construct benchmarks that exercise certain file system behaviors in a controlled manner.

File System Tracing: Many previous studies have traced file system activity. For example, Zhou *et al.* [37], Ousterhout *et al.* [21], Baker *et al.* [2], and Roselli *et al.* [24] all record various file system operations to later deduce file-level access patterns. Vogels [35] performs a similar study but inside the NT file system driver framework, where more information is available (*e.g.*, mapped I/O is not missed, as it is in most other studies). A recent example of a tracing infrastructure is TraceFS [1], which traces file systems at the VFS layer; however, TraceFS does not enable the low-level tracing that SBA provides. Finally, Blaze [5] and later Ellard *et al.* [10] show how low-level packet tracing can be useful in an NFS environment. By recording network-level protocol activity, network file system behavior can be carefully analyzed. This type of packet analysis is analogous to SBA since they are both positioned at a low level and thus must reconstruct higher-level behaviors to obtain a complete view.

8 Conclusions

As systems grow in complexity, there is a need for techniques and approaches that enable both users and system architects to understand in detail how such systems operate. We have presented semantic block-level analysis (SBA), a new methodology for file system benchmarking that uses block-level tracing to provide insight about the internal behavior of a file system. The block stream annotated with semantic information (*e.g.*, whether a block belongs to the journal or to another data structure) is an excellent source of information.

In this paper, we have focused on how the behavior of journaling file systems can be understood with SBA. In this case, using SBA is very straightforward: the user must know only how the journal is allocated on disk. Using SBA, we have analyzed in detail two Linux journaling file systems: ext3 and ReiserFS. We also have performed a preliminary analysis of Linux JFS and Windows NTFS. In all cases, we have uncovered behaviors that would be difficult to discover using more conventional approaches.

We have also developed and presented semantic trace playback (STP) which enables the rapid evaluation of new ideas for file systems. Using STP, we have demonstrated the potential benefits of numerous modifications to the current ext3 implementation for real workloads and traces. Of these modifications, we believe the transaction grouping mechanism within ext3 should most seriously be reevaluated; an untangled approach enables asynchronous processes to obtain in-memory bandwidth, despite the presence of other synchronous I/O streams in the system.

Acknowledgments

We thank Theodore Ts'o, Jiri Schindler and the members of the ADSL research group for their insightful comments. We also thank Mustafa Uysal for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM and EMC.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *FAST '04*, San Francisco, CA, April 2004.

[2] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *SOSP '91*, pages 198–212, Pacific Grove, CA, October 1991.

[3] S. Best. JFS Log. How the Journaled File System performs logging. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 163–168, Atlanta, 2000.

[4] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.

[5] M. Blaze. NFS tracing by passive network monitoring. In *USENIX Winter '92*, pages 333–344, San Francisco, CA, January 1992.

[6] T. Bray. The Bonnie File System Benchmark. http://www.textuality.com/bonnie/.

[7] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *FREENIX '02*, Monterey, CA, June 2002.

[8] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *SIGMETRICS '93*, pages 1–12, Santa Clara, CA, May 1993.

[9] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *USENIX Winter '92*, pages 43–60, San Francisco, CA, January 1992.

[10] D. Ellard and M. I. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *LISA '03*, pages 73–85, San Diego, California, October 2003.

[11] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Austin, Texas, November 1987.

[14] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[15] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[17] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX 1996*, San Diego, CA, January 1996.

[18] J. C. Mogul. A Better Update Policy. In *USENIX Summer '94*, Boston, MA, June 1994.

[19] W. Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[20] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[21] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *SOSP '85*, pages 15–24, Orcas Island, WA, December 1985.

[22] H. Reiser. ReiserFS. www.namesys.com, 2004.

[23] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, pages 14–29, Monterey, CA, January 2002.

[24] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *USENIX '00*, pages 41–54, San Diego, California, June 2000.

[25] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[26] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX '00*, pages 71–84, San Diego, California, June 2000.

[27] D. A. Solomon. *Inside Windows NT (Microsoft Programming Series)*. Microsoft Press, 1998.

[28] SourceForge. The Linux NTFS Project. http://linux-ntfs.sf.net/, 2004.

[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *USENIX 1996*, San Diego, CA, January 1996.

[30] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[31] Transaction Processing Council. TPC Benchmark C Standard Specification, Revision 5.2. Technical Report, 1992.

[32] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.

[33] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[34] S. C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[35] W. Vogels. File system usage in Windows NT 4.0. In *SOSP '99*, pages 93–109, Kiawah Island Resort, SC, December 1999.

[36] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.

[37] S. Zhou, H. D. Costa, and A. Smith. A File System Tracing Package for Berkeley UNIX. In *USENIX Summer '84*, pages 407–419, Salt Lake City, UT, June 1984.