Analysis and Evolution of a Journaling File System

Vijayan Prabhakaran
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report #1509

June 2004

Analysis and Evolution of a Journaling File System

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison

Abstract

We present measurements and analysis of the Linux ext3 file system. We develop and apply a novel analysis method known as semantic block-level analysis (SBA), which examines the low-level block stream that a file system generates in order to understand its behavior under a series of controlled workloads. We use SBA to evaluate the strengths and weaknesses of the ext3 design and implementation; in comparison to standard benchmarking approaches, SBA enables us to understand *why* the file system behaves in a certain way, knowledge that is usually only available to the file system developers. We then develop and apply a new complementary technique known as semantic trace playback (STP) to evaluate the effect of different possible changes to the file system. In contrast to the cumbersome and time-consuming process of changing the file system implementation directly, STP enables us to rapidly gauge the potential benefits of a given file system modification without the heavy investment in implementation effort. Using STP, we propose and assess various modifications to the base ext3 implementation.

1 Introduction

Modern file systems are journaling file systems. By writing information about pending updates to a write-ahead log [13] before committing the updates to disk, journaling enables fast file system recovery after a crash at some cost in performance. Although the basic techniques have existed for many years (*e.g.*, in Cedar [14] and Episode [10]), journaling has increased in popularity and importance in recent years; due to ever-increasing disk capacities, scan-based recovery (*e.g.*, via fsck [18]) is prohibitively slow on modern drives and RAID volumes.

The Linux ext3 file system [33] represents a modern instance of journaling technology. Ext3 was designed to be compatible with its non-journaling cousin, ext2. Specifically, one can mount an extant ext2 partition as an ext3 file system, and hence easily upgrade to journaling without the difficulties often associated with migrating to a new file system. For this reason, although there are many other journaling file systems for Linux, including ReiserFS [24], IBM's JFS [4], and SGI's XFS [30], we believe ext3 is and will remain an important (if not the most important) Linux file system.

However, little is known about the behavior of ext3. For example, ext3 has three different journaling modes; which of these is best for various workloads? Further, ext3 has many configuration parameters, including such basic choices as the size of the journal; how should such parameters be configured? Finally, ext3 makes many internal design decisions, such as how it stores information in the journal; were appropriate decisions made? All such factors have a strong impact on ext3 behavior.

We believe it is time to perform a detailed benchmark-driven analysis of ext3. Most previous benchmarks analyze 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, 9, 21, 28]. However, it is difficult to discover the underlying reasons for the observed performance with this approach.

1.1 Semantic Block-level Analysis

In this paper we employ a novel benchmarking methodology called *semantic block-level analysis* (SBA) to trace and analyze ext3. 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 reverse-engineer information about block type (*e.g.*, whether a block request is to the journal, to an inode, etc.), and it is *block-level* as it interposes upon 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.

Our analysis is divided into *structural*, *temporal*, and *spatial* components. In studying the structure of ext3, we find that ordered and writeback journaling modes induce only a small overhead as compared to ext2 for most types of workloads; however, for highly random, synchronous workloads, the costs of journaling are noticeable. We also find that the data journaling mode of ext3, which writes both data and metadata to the journal (sequentially) and then to the normal file system structures, has promising performance under certain random-write workloads, reminiscent of log-structured file systems [27]; given enough idle time or free bandwidth [16] to write data to its fixed-place location, application-perceived write performance tracks the sequential rate of the disk system.

In studying the temporal aspects of ext3, we find that journal size and various commit timers play an important role in determining when data moves from memory to disk. Journal size is especially important for data journaling mode, which rapidly consumes resources by journaling both data and metadata; for the other ext3 journaling modes, the size of the journal is less important. The commit timers also play an important role in determining when data goes to disk; in particular, timers that typically force metadata to disk (by default, every 5 seconds in Linux) now also force *data* to disk for both ordered and data journaling modes; in both of those modes, the writing of data to disk is mandated by metadata updates. Finally, we find that the compound transaction mechanism of ext3 ties together the performance of all processes that write to disk; if just a single process writes data synchronously, the perceived performance of other asynchronous processes decreases dramatically.

In exploring the spatial aspects of ext3, we find that placing the journal on a separate partition or device performs somewhat better than the default of placing it within the file system; however, ext3 artificially limits these benefits by preventing overlap of journal and fixed-place updates. We also find that for highly synchronous workloads, update performance to files that are far from the journal perform noticeably more slowly than files in the immediate proximity of the journal.

1.2 Semantic Trace Playback

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. We show how STP can be used effectively as well as discuss its limitations.

We then use STP to evaluate a number of possible modifications of ext3. Specifically, we show how an adaptive journal mode selector tailors the file system mode to the workload at hand, achieving better performance than any single mode can. We also show how a differential journaling approach improves the performance of data journaling mode by reducing the amount of I/O traffic to the journal. We then demonstrate that an untangled transactional grouping mechanism preserves the in-memory performance of asynchronous processes in spite of the presence of a synchronous I/O stream. Finally, we illustrate how improved journal location reduces worst-case performance for synchronous workloads.

1.3 Contributions and Outline

The main contributions of this paper are:

- A new methodology for understanding in detail the behavior of a file system (§3). Semantic block analysis is a simple and powerful approach; we apply it here to ext3 but note that it can be readily be applied to other file systems.
- A detailed analysis of an important file system, ext3 (§4). Linux ext3 is the default file system under RedHat, and understanding how it behaves is important developers, administrators, and application writers.
- A new methodology for rapidly gauging the benefits of file system modifications without a heavy implementation effort (§5). Semantic trace playback enables rapid prototyping of file system modifications.
- An evaluation of different design and implementation alternatives for ext3 (§6). We demonstrate the benefits of adaptive mode selection, differential content journaling, untangled transaction grouping, and improved journal placement and parallelism.

After we present our method, results, and analysis of different design alternatives, we discuss related work (§7) and conclude (§8).

2 Background

In this section, we present an overview of the Linux ext3 file system [34, 35]. Linux ext3 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 we often refer to as fixed or in-place location). In this organization (which is loosely based on the Berkeley FFS [17]), the disk is split into a number of *block groups*; within each block group are bitmaps, inode blocks, data blocks, and some per-group metadata. 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.

In most journaling file systems, the journal is used as a log to record some extra information about pending file system updates. By forcing journal updates to disk *before* updating complex file system structures, this write-ahead logging technique [13] enables simple and efficient crash recovery; a simple scan of the journal and a redo of any incomplete 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.

2.1 Modes of Operation

Linux ext3 is a bit unusual in that it includes not one but three flavors of journaling: *writeback mode*, *ordered mode*, and *data journaling mode*; Figure 2 illustrates the differences between these modes. Although the choice of mode is made at mount time, the mode can be changed by unmounting and then remounting the file system.

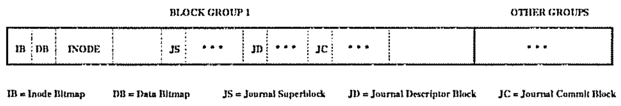

Figure 1: **Ext3 On-Disk Layout.** *The picture shows the layout of an ext3 file system. The disk address space is broken down into a series of block groups (akin to FFS cylinder groups), each of which has bitmaps to track allocations and regions for inodes and data blocks. The ext3 journal is depicted 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.*

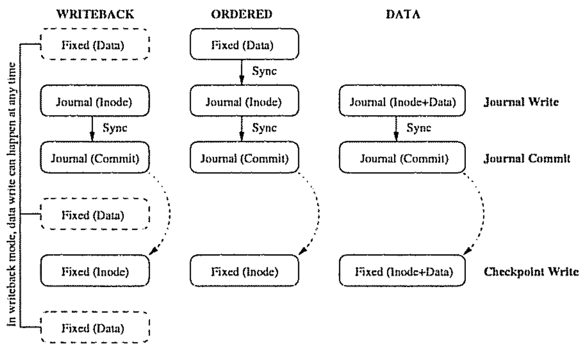

Figure 2: **Ext3 Journaling Modes.** *The diagram depicts the three different journaling modes of ext3: writeback, ordered, and data. In the diagram, time flows downward. Boxes represent updates to the file system, e.g., "Journal (Inode)" implies the write of an inode to the journal; the other destination 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.*

In *writeback mode*, only the 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 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 the data blocks.

For example, assume a process appends a block to a file; internally, the file system allocates a new pointer within the inode I and a new data block at address D to hold the data. In ext3 writeback mode, the file system is free to write D at any time, but is quite careful in how it updates I. Specifically, it will force a journal record to disk containing I, followed by a commit block. Once the commit block is written, ext3 knows it can safely recover from a crash, and at some time later will write I to its fixed location in the ext2 structures. However, if I makes it to the log successfully, and a crash happens *before* D gets to disk, upon recovery I will point to D but the contents of D will not be correct.

As with writeback mode, under *ordered journaling mode*, only the metadata writes are journaled; however, data writes to their fixed location are ordered *before* the journal writes of the metadata. In 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 from a crash. In continuing from our example above, D will be forced to its fixed location in the ext2 structures before I is written to the journal. Thus, even with an untimely crash, the file system will recover in a reasonable manner.

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 (§4).

2.2 Transactions

The transaction model of ext3 is fairly straightforward. 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 [34] and may have better performance than more fine-grained transactions: the compound transaction naturally batches updates to structures that are 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). We explore the impact of transaction grouping in §4.2.3.

2.3 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 new 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/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.

2.4 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. We discuss checkpointing in more detail in §4.2.1.

2.5 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 the new updates (whether to data or just metadata) are written to the log, the process of restoring the in-place file system structures is easy. Upon startup, 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 Semantic Block-Level Analysis

In this section, we describe *semantic block-level analysis* (*SBA*), which enables meaningful low-level tracing of file system behavior. With this benchmarking approach, we are able to not only measure file system performance, but also gain insight as to *why* the system performs as it does.

Our methodology is simple. We place a pseudo-device driver in the kernel, associating it with an underlying disk, and mount the file system of interest (*e.g.*, ext3) on the pseudo device; we refer to this driver as the SBA driver. We then run 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 responses as well, the SBA driver can infer the order in which the requests were scheduled at lower levels of the system.

The main difference between the SBA approach and more standard block-level tracing is that our driver is aware of file system structures and can infer more relevant information; in this way, we draw on previous work that applies such semantic knowledge to building smarter storage arrays [29]. Specifically, we assume the SBA driver has been customized to the file system under test, and understands its basic on-disk format. We do not assume that the driver (or indeed, the analyst) knows policies or configuration parameters of the file system; indeed, SBA can be used to derive those aspects of the file system.

To understand the actions of the file system from this lower-level perspective, semantic analysis must be performed not only on the location of disk block traffic, but on their contents as well. For example, one must interpret the contents of the journal descriptor block to know which data blocks are journaled; likewise, one must interpret the contents of the journal to infer the type of the journal block (*e.g.*, descriptor or commit block). As a result, it is more straightforward and efficient to semantically interpret the block-level trace on-line; performing this analysis offline would require exporting the contents of the blocks as well, greatly inflating the size of trace. Thus, the SBA driver is embedded with enough information to interpret the placement and contents of journal blocks, metadata, and data blocks. Care is taken to ensure that this on-line analysis is performed efficiently.

4 Analyzing ext3

In this section, we perform a detailed analysis of ext3 using our SBA framework. Our analysis is divided into three categories: *structural, temporal, and spatial*. In the structural analysis, we examine the different modes of ext3. In the temporal analysis, we observe how ext3 controls the timing of the data flow, from memory to the journal (*i.e.*, logging) and from memory to the final fixed location (*i.e.*, checkpointing). Finally, in our spatial analysis, we study the sensitivity of ext3 to the location of the journal.

Note that the goal of our analysis is not to characterize the performance of ext3 on a wide range of workloads or even on realistic workloads; our goal is to construct synthetic workloads that uncover the internal structure and decisions made by the file system. Our experiments focus heavily on write-based workloads; reads to files are virtually identical across the ext3 modes as well as ext2, and, therefore, do not expose as many interesting issues.

All measurements are taken on a machine running Linux 2.4.18 with a 600 MHz Pentium processor and 1 GB of main memory. The ext3 file system 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.

4.1 Structural Analysis

In this section we analyze the basic behavior of the three ext3 journaling modes: writeback, ordered, and full data journaling. In particular, we evaluate the overhead of ext3 relative to ext2 as well as the relative performance of the three modes across widely differing workloads. Thus, we consider two extreme workloads: one which performs only sequential writes and another that performs only random writes. Because we want to evaluate behavior of updating the file system data structures on disk, we ensure that all workloads flush their data to disk with `fsync`.

Sequential Writes: The workload in our first experiment consists of creating a single file, writing to it sequentially, and then performing an `fsync` to flush its data to disk; across experiments, we vary the size of the file. Figure 3 shows the performance and our semantic analysis for the three ext3 modes and for ext2.

The topmost graph plots the achieved bandwidth for the different modes. From this graph, we make three observa-

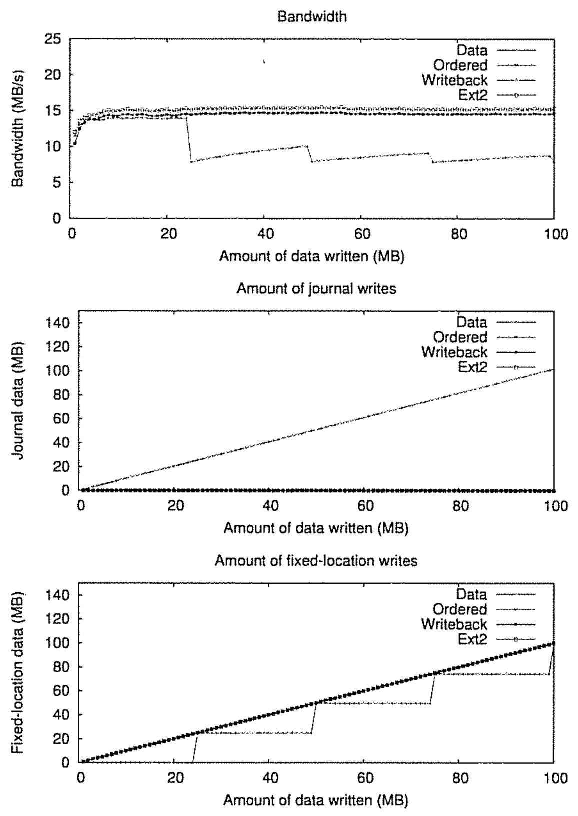

Figure 3: **Journaling Mode Performance under Sequential Workloads.** *The achieved bandwidth to create a file of size x is plotted, where x is varied along the x-axis. The benchmark creates the file of size x by writing it out sequentially and then calling* `fsync` *to flush the data to disk. The journal size is set to 50 MB for these experiments.*

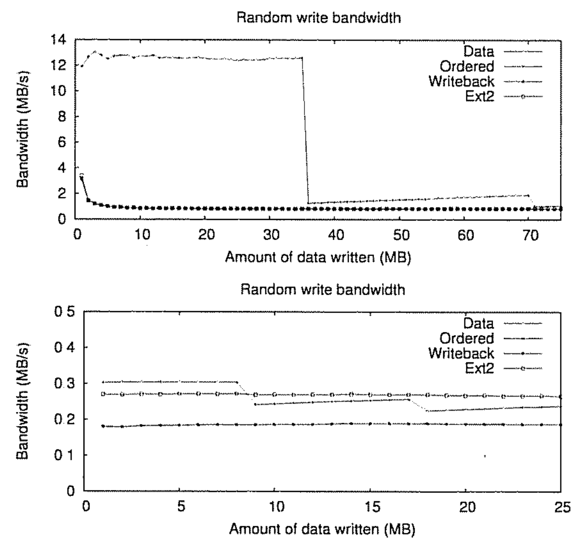

Figure 4: **Journaling Mode Performance under Random Workloads.** *These graphs plot the performance of random writes to a file under the various journaling modes. Along the x-axis, the amount of random I/O is increased, and the y-axis plots achieved throughput (as measured after all data has been flushed to disk). In both sets of experiments, the benchmark process performs 4 KB random updates of a file; what varies across graphs is the amount of outstanding I/O allowed before an* `fsync` *is issued. In the first graph, an* `fsync` *is issued after 256 such random writes (i.e., a group commit), and in the second, a* `fsync` *is issued after each 4 KB write.*

tions. First, ext2 performs slightly better than the highest performing ext3 mode – there is a small but noticeable cost to journaling for sequential streams. Second, both ordered mode and writeback mode perform well, achieving nearly the full sequential write bandwidth of ext2. Finally, full data journaling performs fine with small files, but performance then degrades and fluctuates in a declining sawtooth waveform as the size of the file increases.

We now apply a simple semantic analysis of the underlying block stream to understand why the different modes behave as observed. The next two graphs in Figure 3 plot the amount of data written to the journal and the amount of data written to the fixed ext2 data structures, respectively, for the same set of experiments.

The middle graph shows 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, and therefore the total amount of traffic to the journal is quite small. Note that in ext2 there is no journal and, therefore, no journal traffic.

The bottom graph shows that for ext2, writeback, and ordered mode the amount of traffic written to the file's 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 file's 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 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 §4.2.1.

Random Writes: Our next workload issues 4 KB writes to random locations in a single file, varying the total amount of I/O across experiments. The results are plotted in Figure 4. We examine the impact of synchronization granularity by issuing an `fsync` once for every 256 writes (*i.e.*, a group commit) in the first graph and once per write (*i.e.*, a single-write commit) in the second.

The main observation from the first graph is that data journaling mode performs dramatically better than the other modes for 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 35 MB (this relationship is explored more in §4.2.1).

The second graph illustrates that synchronous 4 KB writes do not perform well, even in data journaling mode; forcing each small 4 KB write to the log, although in logical sequence, incurs a disk rotation for each write and thus does not achieve sequential performance. We also can observe that the costs of journaling (in ordered and writeback modes) are most severe here, as compared to ext2; the extra seeks between the journal and the fixed-location writes reduce performance by roughly one-third.

Summary: From our exploration of ext3 journaling modes, for most workloads, ext3 ordered and writeback mode do not induce significant overheads as compared to ext2. However, for certain highly synchronous, random write workloads, the performance impact is noticeable.

The behavior of data journaling mode is more complex. Applications using data journaling mode can commit data to disk sequentially and achieve high performance, regardless of their access pattern, as even random writes are logged in LFS-like fashion [27]. Note that "random" writes arise for many reasons, including workloads that are metadata intensive or are database-like and update individual or groups of records in large tables. The main obstacle to good performance in data journaling mode is controlling the checkpoint which moves data from the journal to its final fixed location. If the checkpoint occurs in the foreground when application traffic is competing for disk bandwidth, application-perceived performance drops considerably; however, if checkpointing traffic can be moved to the background (*e.g.*, if workloads are bursty), or scheduled for "free" [16], data journaling performance will be superior to the other modes.

4.2 Temporal Analysis

We have seen that the flow of traffic to the journal and to the fixed ext2 structures can have a strong impact on overall performance, particularly for data journaling mode. Hence, we now analyze the timing of data flow through the system. A number of factors influence the timing, including journal size, system timers, and the ext3 transaction grouping machinery.

4.2.1 Journal Size

We begin by examining the effect of journal size on ext3 behavior, specifically how journal size affects when transactions are committed to the log and when data is checkpointed to its ext2 fixed-place location.

Log Commits: We wish to understand how journal size affects the timing of updates to the log. We focus on data journaling mode; this mode is the most sensitive to the size of the journal, since both metadata and data are sent

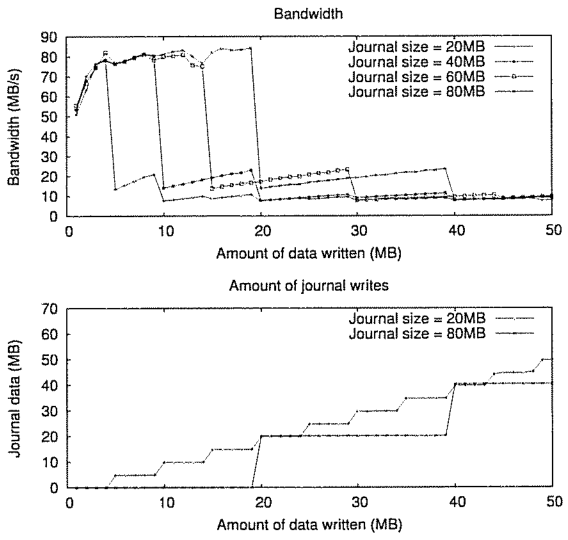

Figure 5: **Effect of Journal Size on Log Updates.** *The topmost figure plots the bandwidth of data journaling mode under different-sized file writes. Four separate lines are plotted, which represent four different journal sizes. The second graph shows the amount of log traffic generated for each of the experiments (only two of the four journal sizes are shown for clarity).*

to the log. To exercise log commits as a function of journal size, we construct a workload in which data is not explicitly forced to disk by the application (*i.e.*, the process does not call `fsync`). We write to a single file to minimize the amount of metadata overhead. By varying the amount of data written in the workload, we can observe the amount of data needed to trigger a log commit.

Figure 5 shows the resulting bandwidth and the amount of journal traffic for data journaling mode, 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 $\frac{1}{4}$-th 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, reveals that a log commit does occur at $\frac{1}{4}$-th the journal size, forcing metadata and data to the journal. Inspection of the 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 as only metadata is logged.

Fixed-Location Checkpoints: We turn our attention to checkpointing, the process of writing data to its fixed location within the ext2 structures. We again 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.

Figure 6 shows our SBA results as a function of file

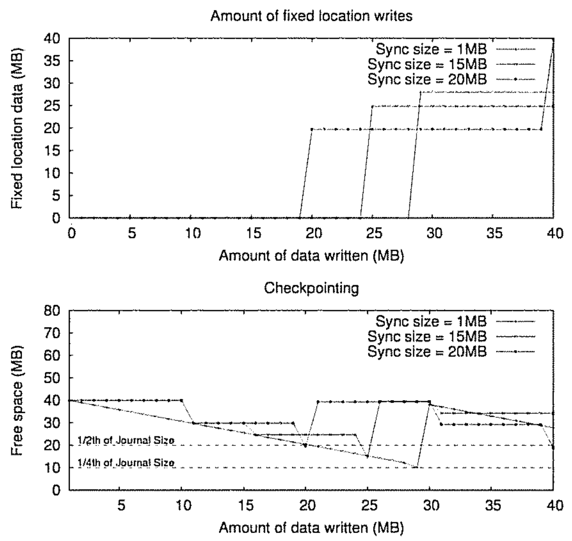

Figure 6: **Effect of Journal Size on Checkpointing.** *The figures plot the result of a sequential write of varying size, with different* `fsync` *intervals shown per line. Specifically, a certain amount of data (as indicated by the x-axis value) is written out, with a* `fsync` *issued after every 1, 15, or 20 MB of writes. The first graph uses SBA to plot the amount of fixed-location traffic, and the second uses SBA to plot the amount of checkpoint data written.*

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 30 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 also confirmed this same relationship for other journal sizes (not shown).

Summary: Journal size plays an important role in determining when updates are forced to disk, both to the log and to their final fixed location; a larger journal allows applications to write more data before seeing the effects of log commits and checkpoints, thus potentially transferring the work of checkpointing to the background. Journal size is especially important for data journaling mode, as this mode increases memory pressure and quickly forces data to the log and then to its fixed location; both types of traffic noticeably reduce perceived application bandwidth. For the other modes (*i.e.*, ordered and writeback), journal size has less impact; as long as the journal is large enough to handle any metadata-intensive portions of a workload, the size of the journal will not impact performance.

4.2.2 Commit Intervals

In this section, we explore the impact of *commit interval timers* on ext3 file system behavior. In Linux 2.4, there are three timer values of interest: 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. We now measure how these timers affect the timing of disk I/O, focusing first on writes to the journal and then checkpoints to the fixed locations.

Log Writes: We construct a workload and environment such that commits to the journal are triggered by a specific timer. Thus, we set the journal size to be sufficiently large, do not issue a `fsync` from the application, and set the other timers to a large value (*i.e.*, 60 s). The actual workload is very simple: a process writes 1 MB of data sequentially to a file and we observe when the first write appears in the journal. Figure 7 plots our results for data journaling mode, varying one of the timers along the x-axis, and plotting the time that the first log write occurs along the y-axis. Results for the other two modes are quite similar and hence not shown.

The first and last graphs show that the kupdate daemon metadata commit timer and the kjournal daemon commit timer directly 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 time 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 below, this timer influences when data is written to its fixed location.

Checkpoint Writes: We examine how the system timers impact the timing of checkpoint writes to the fixed locations using the same workload as above. We focus on the data journaling; writeback and ordered checkpoint timing are identical and therefore not shown.

We now examine data journaling mode. Here, we vary the kupdate data timer while setting the other timers to 5 seconds. Figure 8 shows how the kupdated data timer

[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.

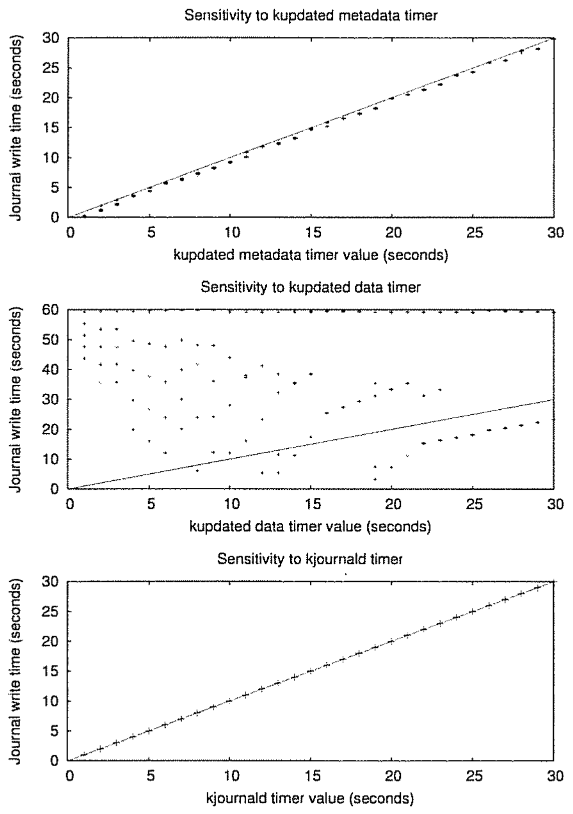

Figure 7: **Commit Timers.** *We present semantic analyses of the impact of the various commit timers. In each graph, the value of one timer is varied across the x-axis, and the time of the first write to the journal is recorded along the y-axis. When measuring the impact of a particular timer, we set the other timers to 60 seconds and the journal size to 50 MB so that they do not affect the measurements.*

impacts when data is written to its fixed location. First, as also seen in the previous experiment, the log is updated after the 5 second timers expire. Then, the checkpoint write occurs later by the amount specified by the kupdated data timer, at a 5 second granularity; further experiments (not shown here) reveal that this granularity is controlled by the kupdated metadata timer.

Summary: 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.

Mapping this strategy to ext3 leads to default timer values of 5 seconds for the kupdate metadata timer, 5 seconds for the kjournal timer, and 30 seconds for the kupdate data timer; however, it does not lead to the same timing of data and metadata traffic as 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. Note that this timing behavior

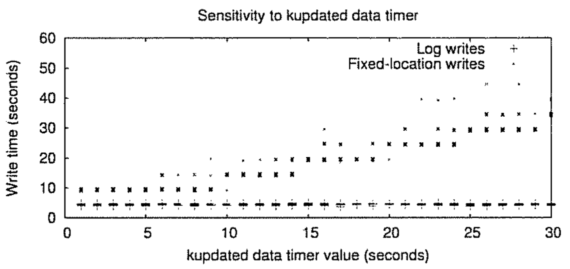

Figure 8: **Checkpoint Timers.** *The figure plots the relationship between the time that data is first written to the log and then checkpointed as dependent on the value of the kupdated data timer. The scatter plot shows the results of multiple (30) runs. The process that is running writes 1 MB of data (no `fsync`); data journaling mode is used, with other timers set to 5 seconds and a journal size of 50 MB.*

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 [20]. The disadvantage of early flush, however, is that temporary files may be written to disk before subsequent deletion, increasing the overall load on the I/O system.

4.2.3 Transaction Grouping

We now investigate the ext3 transaction grouping machinery. Ext3 groups individual file system updates into a single large transaction; all file system updates add their changes to that global transaction, which is eventually committed to disk. Committing the entire transaction as a group has known performance benefits [14], but can also introduce complexities.

To quantify the effect of placing independent transactions into the same group, we construct a workload containing two diverse classes of traffic: an asynchronous foreground process in competition with a background process. Specifically, we consider a foreground process that writes out a 50 MB file without calling `fsync` and a background process that 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 across all three ext3 modes (not shown).

Because the foreground process is asynchronous, it should be able to achieve in-memory performance on writes (assuming no thresholds are triggered); however, as we will see, with the presence of a synchronous background process, the performance of the asynchronous process drops to disk speeds.

In Figure 9 we show the impact of varying the mean "sync interval" of the background process on the performance of the foreground process. The topmost graph plots the bandwidth achieved by the foreground asynchronous process, in two different scenarios. In the first, the background process issues writes periodically but does

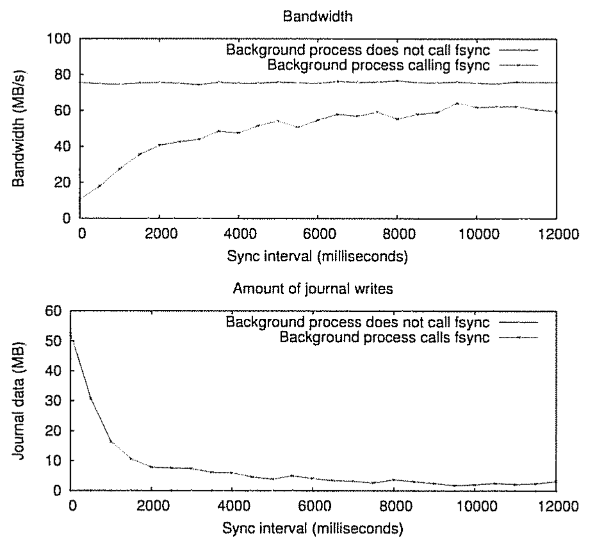

Figure 9: **Grouping Synchronous and Asynchronous Writes.** *In this experiment, an asynchronous foreground process writes a sequential file of size 50 MB. A synchronous background process writes out 4 KB, optionally calls* `fsync`*, sleeps for the "sync interval", and repeats. Along the x-axis, we increase the sync interval. In the top graph, the bandwidth achieved by the asynchronous process is plotted in two scenarios: with the background process running but not calling* `fsync` *after each write, and with a* `fsync` *after each write. In the bottom graph, the amount of data written to disk during both sets of experiments is shown.*

not call `fsync` after each write; hence bandwidth of the foreground process is uniformly high and matches in-memory speeds. In the second, the background process calls `fsync` with each write, with a varying sync interval. From this line in the graph, we can observe that when the sync interval of the background process is small, it issues many `fsync` calls and forces all data in the compound transaction to disk; the performance of the foreground process drops to disk speeds. As the sync interval increases, the asynchronous process is more likely to complete before the background process calls `fsync`, and thus the average performance of the foreground process improves. This explanation is confirmed by the second graph, which reports the SBA analysis of journal traffic. As expected, the more frequently the background process calls `fsync`, the more traffic is sent to the journal.

Summary: Linux ext3 uses a single compound transaction to commit updates to disk. Although this strategy is more straightforward to implement within the file system, it has the downside of mixing all types of I/O together. Although the experiment above focused on data journaling mode, the result holds for the default ordered mode as well. Group transactions tie the performance of one process to another: the perceived performance of one process thus depends on the behavior of the "worst" (most random, synchronous) process in the system.

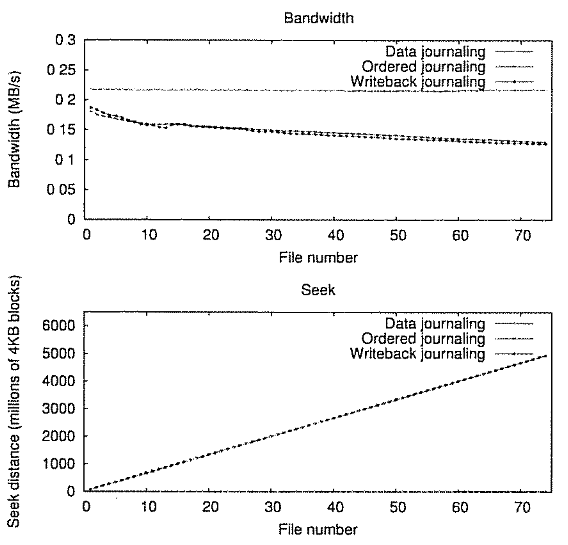

Figure 10: **Journal Within the File System.** *In this experiment for ordered mode, the journal is placed in the (default) location at the beginning of the partition. 50 MB files are created across the file system. A file is chosen, as indicated by the number along the x-axis, and the workload issues 4 KB synchronous writes to that file. In the topmost graph, the y-axis plots achieved bandwidth. The bottom graph shows the logical seek distance for the experiment, calculated from the SBA trace.*

4.3 Spatial Analysis

Finally, we study the spatial aspects of ext3. Specifically, we study how the placement of the journal, whether within the file system as a file or on a separate device, affects overall performance.

Within the File System: The default configuration of ext3 creates the journal as a regular file near the beginning of the partition. In this experiment, we show how this typical placement can affect the performance of the different modes. We construct a workload that stresses the relative 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. We vary which file is chosen along the x-axis, and plot achieved bandwidth on the y-axis, as shown in Figure 10.

The first graph shows that, for ordered mode, bandwidth drops by nearly 30% when the file is located far from the journal. The second graph shows that this performance drop is due to an increase in seek time. In both ordered and writeback modes, a synchronous update to a file causes a write to the file and then a write to the journal; these *coupled* writes are far from one another and hence incur a high seek cost.

As also shown in the figure, data journaling mode is not sensitive to the separation between a file and the journal, because updates are sent to the log; only later (as the journal is filled or a timer expires) is the data moved to its fixed ext2 location. Thus, the write to the log and the fixed location are *decoupled*, and performance remains constant.

Outside the File System: Given that ext3 performance is

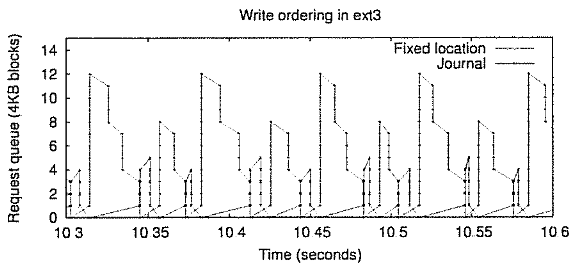

Figure 11: **Limited Parallelism.** *The figure plots the number of outstanding writes to the fixed-location and journal disks. In this experiment, we run five processes, each of which issues 16 KB random synchronous writes. The file system has a 50 MB journal size and is running in ordered mode, and the journal is configured to run on a separate disk.*

sensitive to journal location, we explore this issue more broadly, comparing four different configurations: when the journal is within the file system, is on the same disk in a close-by partition, is on the same disk but in a far-away partition, or is on a separate device entirely. We only report results for ordered mode, since it is the most sensitive. We consider a workload similar to the one used above: the process synchronously writes a large number of random 4 KB blocks. With this workload, placing the journal on a different device unsurprisingly gives the best performance (0.25 MB/s). Locating the journal within the same file system performs almost identically to locating it on a different partition close to the file system (0.17 MB/s), since both setups incur roughly the same seek costs. Finally, placing journal on a distant partition performs the worst (0.12 MB/s) because the seek distance increases.

Given that performance with a device dedicated to journaling did not double performance, we decided to investigate this configuration in more detail. In Figure 11, we plot the disk queue size over time for both the fixed-location and journaling disks (note the workload is slightly different than above for ease of presentation; however, the results are general across many workloads).

From the figure, we observe that writes to the journal disk and fixed-place disk do *not* overlap. Inspection of the code reveals the cause: ext3 ordered mode strictly (and unnecessarily) sequences updates to the fixed-place region and the journal. More 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. The first wait is not needed for correctness (it was probably included to improve scheduling order to a single disk), and in this case limits the potential performance benefit. Hence, although using the extra disk improves performance (by reducing the amount of seeking within each disk), the benefit is limited by the current implementation.

Finally, we have found that data journaling mode is sensitive to whether the journal is placed within the file system or not. This performance difference occurs even when no checkpointing occurs, that is, when only the journal is being written and there are no extraneous seeks. Writing to the journal when it is stored as a regular file in the file system incurs extra overhead than writing to a raw log outside of the file system; specifically, ext3 must perform extra work to calculate the next physical block of the log file and does not deal as efficiently with indirect blocks. This extra work leads to a small but noticeable performance difference for sequential workloads: one can achieve over 14 MB/s for a raw journal, but only about 12 MB/s when the journal is a regular file.

Summary: We find that the location of the journal within the file system, for ordered and writeback modes under synchronous workloads, can make a noticeable (roughly 30%) performance difference in the worst case. We also find that using a different device for the journal improves performance, although the current implementation limits potential benefits. Finally, we discover that in some cases (*i.e.*, data journaling mode), using a separate partition on the same disk is preferred over a journaling file within the file system. Whether these performance benefits overcome the simple convenience of using the default configuration is likely domain dependent.

5 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 in the next section 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

§4.2.1; 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 to 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 §4.

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.

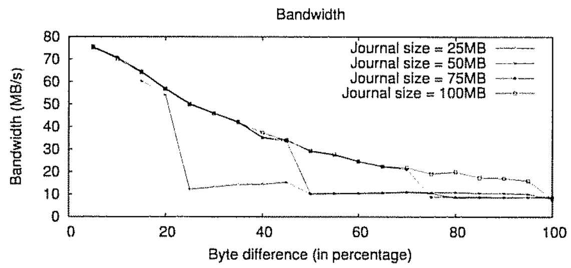

Figure 12: **Differential Journaling.** *This figure plots the emulated performance (using STP) of a 50 MB sequential write in data journaling mode with a single* `fsync` *at the end. Along the x-axis, we vary the percentage of each written block that has changed; the rightmost point shows the behavior of standard ext3 (which is equivalent to differential journaling with 100% of every block changing). Each line varies the size of the journal used for the experiment. The SBA trace was collected from underneath a 50 MB journal; other journal sizes are emulated.*

6 Evolving ext3

In this section, we apply STP to evaluate modifications to ext3. From our analysis, we decided to focus on four aspects of ext3: journaling mode, journal contents, transaction grouping strategies, and journal location. Where possible, we also change ext3 directly; by doing so, we demonstrate the validity of the STP approach (the figure captions contain details on the validated modifications).

6.1 Journaling Mode

The journaling mode in ext3 is set at mount time and remains fixed until the next mount. We consider two modes, data journaling and ordered, each of which is good for certain types of workload. Random writes perform better in data journaling mode as the random writes are written sequentially into the journal (assuming no checkpointing occurs during the writes). Large sequential writes perform better in ordered mode as it avoids the extra traffic generated by data journaling mode.

We implement a new adaptive journaling mode using STP that chooses the journaling mode for each transaction according to the nature of writes that are in the transaction. If a transaction is sequential, then it is journaled in ordered mode. Otherwise, data journaling mode is used. Clearly, a more sophisticated mode detector is likely required; here we simply demonstrate its potential utility.

To demonstrate the potential performance benefits, we run a portion of a trace from HP Labs [25] and compare ordered mode, data journaling mode, and our adaptive approach. In ordered mode, the trace takes 83.39 seconds to complete; in data journaling mode, 86.67 seconds; finally, in adaptive mode, the workload completes in only 51.75 seconds. Because the trace has both sequential and random write phases, no single mode will achieve the best possible performance.

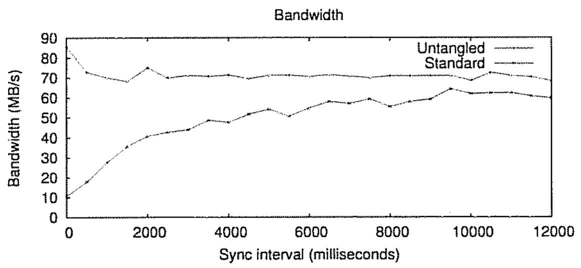

Figure 13: **Untangled Transaction Grouping.** *The experiment in this figure is identical to that described in Figure 9, with one addition: a line showing the performance of the asynchronous sequential process with untangled transaction grouping as emulated via STP.*

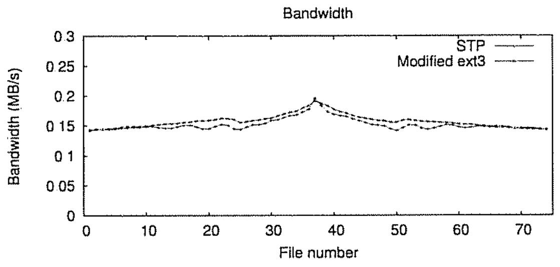

Figure 14: **Improved Journal Placement.** *This figure presents the same experiment as described in Figure 10, except that the journal is placed in the middle of the file system using STP. We also plot the performance of the real system running on a journal that we have coaxed into the middle of the disk by allocating the journal after the (ext2) file system was half full, and then enabling journaling.*

6.2 Journal Contents

Ext3 uses physical logging and writes new blocks in their entirety to the log, as discussed earlier. 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.

We thus 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.

Figure 12 plots the performance of differential journaling under a synthetic workload that varies the percent of each block that changes (and hence must be logged to disk). As we can see from the figure, as the percent of block change increases, performance of data journaling decreases; not only is more data logged, but various thresholds (in this case, the checkpoint threshold) are reached earlier, thus forcing data to disk. Note that the standard non-differential approach is equivalent in performance to differential mode with a 100% byte difference.

To better understand whether differential journaling matters for real workloads, we also analyzed SBA traces underneath two database workloads modeled on TPC-B [31] and TPC-C [32]. 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. Our analysis reveals that with data journaling mode, the amount of data written to the journal would be 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.

6.3 Transaction Grouping

Linux ext3 groups all updates into system-wide compound transactions and commits them to disk periodically. However, as we have shown in §4.2.3, 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.

We now 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.

6.4 Journal Location

Finally, we evaluate the benefit of more careful journal placement within an ext3 partition. Figure 14 plots the performance of a synchronous workload when the journal file has been moved to the center of the volume.

From the figure, we see that the worst-case behavior that can occur in ordered and writeback modes 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). For data journaling mode, we have found that the journal should be placed on the outermost tracks (not shown); as writes to the journal and fixed-location structures are decoupled, placing the journal at the beginning of the volume takes advantage of disk zoning to improve performance.

We also remedy the problem discovered in §4.3 where pre-commit journal writes to a separate disk are not overlapped with data updates in ordered journaling mode. Figure 15 plots the performance when such writes are issued concurrently. The performance when the journal is placed upon a separate device is improved as a result.

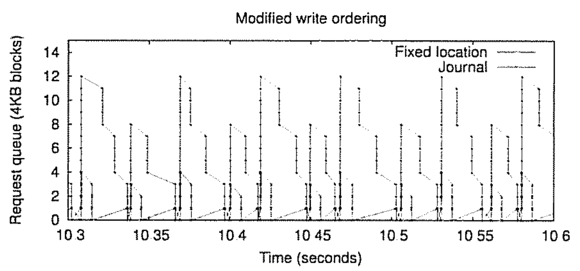

Figure 15: **Unlimiting Parallelism.** *The same experiment is shown as in Figure 11; however, in this run, we use STP to issue the pre-commit journal writes and data writes concurrently. We plot the STP emulated performance, and also made this change to ext3 directly, obtaining the same resultant performance.*

6.5 Summary

We have presented and evaluated numerous modifications to ext3. Of all of our 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. We have also seen that adaptive journaling mode can lead to substantial benefits, outperforming any one single static mode for workloads that exhibit both random and sequential streams. Journal contents can also be important for performance, particularly for data journaling mode; in that case, it may be prudent to consider differential journaling techniques for recording disk updates. Finally, we evaluated the merits of more careful journal placement within the file system. We find that placing the journal in the middle of the file system reduces worst-case seek costs under ordered and writeback modes, and that removing an extraneous wait improves data/metadata write overlap.

7 Related Work

Journaling Studies: Journaling file systems have been studied in detail before, most notably by Seltzer *et al.* [28]. In that work, the authors compare two variants of a journaling FFS to soft updates [12], a different technique for managing metadata consistency for file systems. The authors use both microbenchmarks and applications to compare the two types of system. Although the authors present no direct observation of low-level traffic streams, they are clearly familiar enough with the systems (indeed, they are the implementors!) to make "semantic" inferences to explain file system behavior. For example, in explaining why journaling performance drops in a delete benchmark, the authors state the drop in performance is because the file system is "forced to read the first indirect block in order to reclaim the disk blocks it references" ([28], Section 8.1). A tool such as SBA makes such expert observations more readily available to all.

Another recent study compares a large range of Linux file systems, including ext2, ext3, ReiserFS, XFS, and JFS [7]. In that work, many benchmarks are run to determine which file systems are fastest, but little explanation is given as to *why* any one particular file system does well for a given workload.

File System Benchmarks: There have been numerous previous file system microbenchmarks [6, 9, 19, 21]. Uniformly, all run some kind of synthetic workload and measure overall throughput or runtime in order to draw conclusions about the file system.

Perhaps the most related to our work is Chen and Patterson's self-scaling benchmark [9]. 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, including (for example) how large the file cache is. Our approach differs in that we perform a detailed and very specific study of how a file system is implemented, which we believe is only possible by examining how the file system interacts with the disk. However, our approach is not nearly as automated; instead, we carefully craft benchmarks to bring out certain file system behaviors in a controlled manner.

Other popular file system benchmarks include IOzone [21], Bonnie [6], lmbench [19], the modified Andrew benchmark [22], and PostMark [15]. Some of these (IOZone, Bonnie, lmbench) perform synthetic read/write tests to determine throughput; others (Andrew, Postmark) are intended to model "realistic" application workloads. In contrast to SBA, none are intended to yield low-level insights about the underlying file system.

File System Tracing: Many previous studies have traced file system activity. For example, Zhou *et al.* [37], Ousterhout *et al.* [23], Baker *et al.* [3], and Roselli *et al.* [26] all record various file system operations to later deduce file-level access patterns. Vogels [36] 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). Finally, a recent example of a tracing infrastructure is found in TraceFS [2], which traces file systems at the VFS layer; while many interesting pieces of information can be garnered from this vantage point, TraceFS does not enable the type of meaningful low-level tracing that SBA provides.

Perhaps more relevantly, Blaze [5] and later Ellard *et al.* [11] 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 as they both are positioned at a relatively low-level in the system, and thus must reconstruct certain higher-level behaviors to obtain a full view of file system activity.

Distributed System Tracing And Analysis: Finally, in many ways, our work is similar to recent efforts in debugging the structure of large-scale distributed systems [1, 8]. For example, Aguilera *et al.* take low-level packet traces and use those to infer the structure of distributed applications [1]. Our approach is similar, but informed with more knowledge of the application (*i.e.*, SBA requires a built-in understanding of file system structures).

8 Conclusions

"Much fruit of sense beneath is rarely found."
Alexander Pope

In this paper, we have presented a new methodology for file system benchmarking that uses block-level tracing to provide insight into the design and implementation of a modern journaling file system. Semantic block-level analysis (SBA) is useful for garnering detailed insights into file system behavior.

By using SBA, we have uncovered various detailed behaviors of ext3 that would be difficult to discover using more conventional approaches. Note that while we have focused on ext3, the SBA approach is quite general, simply requiring an understanding of file system structures; such information is available for virtually all modern file systems, including NTFS.

To move beyond the confines of analysis, we have also developed and presented semantic trace playback. STP enables rapid evaluation of new ideas for file system improvement without heavy investment in implementation or simulation. Using STP, we have demonstrated the potential benefits of numerous modifications to the current ext3 implementation.

As systems grow in complexity, there is a need for techniques and approaches that enable system architects to understand in detail how such systems operate. For file systems, the block stream is an excellent source of information, when annotated with semantic information. SBA and STP exploit this block stream to enable system designers to perform a level of analysis beyond existing tools and approaches.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP '03*, Bolton Landing, NY, October 2003.

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

[3] 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.

[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] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *NSDI '04*, San Francisco, CA, March 2004.

[9] 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.

[10] 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.

[11] D. Ellard and M. I. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *FAST '03*, San Francisco, CA, April 2003.

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

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

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

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

[16] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *OSDI '00*, pages 87–102, San Diego, CA, October 2000.

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, March 2003.

[30] 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.

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

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

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

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

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

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

[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.