Enabling Transactional File Access via Lightweight Kernel Extensions

Appears in the Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)

Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, and Erez Zadok
Stony Brook University

Charles P. Wright
IBM T.J. Watson Research Center

Abstract

Transactions offer a powerful data-access method used in many databases today trough a specialized query API. User applications, however, use a different file-access API (POSIX) which does not offer transactional guarantees. Applications using transactions can become simpler, smaller, easier to develop and maintain, more reliable, and more secure. We explored several techniques how to provide transactional file access with minimal impact on existing programs. Our first prototype was a standalone kernel component within the Linux kernel, but it complicated the kernel considerably and duplicated some of Linux's existing facilities. Our second prototype was all in user level, and while it was easier to develop, it suffered from high overheads. In this paper we describe our latest prototype and the evolution that led to it. We implemented a transactional file API inside the Linux kernel which integrates easily and seamlessly with existing kernel facilities. This design is easier to maintain, simpler to integrate into existing OSs, and efficient. We evaluated our prototype and other systems under a variety of workloads. We demonstrate that our prototype's performance is better than comparable systems and comes close to the theoretical lower bound for a log-based transaction manager.

1 Introduction

In the past, providing a transactional interface to files typically required developers to choose from two undesirable options: (1) modify complex file system code in the kernel or (2) provide a user-level solution which incurs unnecessary overheads. Previous in-kernel designs either had the luxury of designing around transactions from the beginning [33] or limited themselves to supporting only one primary file system [43]. Previous user-level approaches were implemented as libraries (e.g., Berkeley DB [39], and Stasis [34]) and did not support interaction through the VFS [15] with other non-transactional processes. These libraries also introduced a redundant page cache and provided no support to non-transactional processes. This paper presents the design and evaluation of a transactional file interface that requires modifications to neither existing file systems nor applications, yet guarantees atomicity and isolation for standard file accesses using the kernel's own page cache.

Transactions require satisfaction of the four ACID properties: Atomicity, Consistency, Isolation, and Durability. Enforcing these properties appears to require many OS changes, including a unified cache manager [12] and support for logging and recovery. Despite the complexity of supporting ACID semantics on file operations [30], Microsoft [43] and others [4, 44] have shown significant interest in transactional file systems. Their interest is not surprising: developers are constantly reimplementing file cleanup and ad-hoc locking mechanisms which are unnecessary in a transactional file system. A transactional file system does not eliminate the need for locking and recovery, but by exposing an interface to specify transactional properties allows application programmers to reuse locking, logging, and recovery code. Defending against TOCTTOU (time of check till time of use) security attacks also becomes easier [28, 29] because sensitive operations are easily isolated from an intruder's operations. Security and quality guarantees for control files, such as configuration files, are becoming more important. The number of programs running on a standard system continues to grow along with the cost of administration. In Linux, the CUPS printing service, the Gnome desktop environment, and other services all store their configurations in files that can become corrupted when multiple writers access them or if the system crashes unexpectedly. Despite the existence of database interfaces, many programs still use configuration files for their simplicity, generality, and because a large collection of existing tools can access these simple configuration files. For example, Gnome stores over 400 control files in a user's home directory. A transactional file interface is useful to all such applications.

To provide ACID guarantees, a file interface must be able to mediate all access to the transactional file system. This forces the designer of a transactional file system to put a large database-like runtime environment either in the kernel or in a kernel-like interceptor, since the kernel typically services file-system system calls. This environment must employ abortable logging and recovery mechanisms that are linked into the kernel code. VFS-cache rollback is also required to revert an aborted transaction [44], its stale inodes, dentries, and other in-kernel data structures. The situation can be simplified drastically if one abandons the requirement that the backing store for file operations must be able to interact with other transaction-oblivious processes (e.g., `grep`), and by duplicating the functionality of the page cache in user space. This concession is often made by transactional libraries such as Berkeley DB [39] and Stasis [34]: they provide a transactional interface only to a single file and they do not solve the complex problems of rewinding the page cache and stale in-memory structures after a process aborts. Systems such as QuickSilver [33] and TxF [43] address this trade-off between the completeness and implementation size by redesigning a specific file system around proper support for transactional file operations. In this paper we show that such a redesign is unnecessary, and that every file system can provide a transactional interface without requiring specialized modifications. We describe our system which uses a seamless approach to provide transactional semantics using a new dynamically loaded kernel module, and only minor modifications to existing kernel code. Our technique keeps kernel complexity low yet still offers a full-fledged transactional file interface without introducing unnecessary overheads for non-transactional processes.

We call our file interface *Valor*. Valor relies on improved locking and write ordering semantics that we added to the kernel. Through a kernel module, it also provides a simple in-kernel logging subsystem optimized for writing data. Valor's kernel modifications are small and easily separable from other kernel components; thus introducing negligible kernel complexity. Processes can use Valor's logging and locking interfaces to provide ACID transactions using seven new system calls. Because Valor enforces locking in the kernel, it can protect operations that a transactional process performs from any other process in the system. Valor aborts a process's transaction if the process crashes. Valor supports large and long-living transactions. This is not possible for `ext3`, `XFS`, or any other journaling file system: these systems can only abort the entire file system journal, and only if there is a hardware I/O error or the entire system crashes. These systems' transactions must always remain in RAM until they commit (see Section 2).

Another advantage of our design is that it is implemented on top of an unmodified file system. This results in negligible overheads for processes not using transactions: they simply access the underlying file system, only using the Valor kernel modifications to acquire necessary locks. Using tried-and-true file systems also provides good performance compared to systems that completely replace the file system with a database. Valor runs with a statistically indistinguishable overhead on top of `ext3` under typical loads when providing a transactional interface to a number of sensitive configuration files. Valor is designed from the beginning to run well without durability. File system semantics accept this as the default, offering `fsync(2)` [9] as the accepted means to block until data is safely written to disk. Valor has an analogous function to provide durable commits. This makes sense in a file-system setting as most operations are easily repeatable. For non-durable transactions, Valor's overhead on top of an idealized mock logging implementation is only 35% (see Section 4).

The rest of this paper is organized as follows. In Section 2 we describe previous experiences with designing transactional systems and related work that have led us to Valor. We detail Valor's design in Section 3 and evaluate its performance in Section 4. We conclude and propose future work in Section 5.

2 Background

The most common approach for transactions on stable storage is using a relational database, such as an SQL server (e.g., MySQL [22]) or an embedded database library (e.g., Berkeley DB [39]); but they have also long been a desired programming paradigm for file systems. By providing a layer of abstraction for concurrency, error handling, and recovery, transactions enable simpler, more robust programs. Valor's design was informed by two previous file systems we developed using Berkeley DB: KBDBFS and Amino [44]. Next we discuss journaling file systems' relationship to our work, and we follow with discussions on database file systems and APIs.

2.1 Beyond Journaling File Systems

Journaling file systems suffer from two draw-backs: (1) they must store all data modified by a transaction in RAM until the transaction commits and (2) their journals are not designed to be accessed by user processes [16, 31, 42]. Journaling file systems store only enough information to commit a transaction already stored in the log (redo-only record). This results in journaling file systems being forced to contain all data for all in-flight transactions in RAM [6, 7, 42]. For metadata transactions, which are finite in size and duration, journaling file systems are a convenient optimization. However, we wanted to provide user processes with transactions that could be megabytes large and run for long periods of time. The RAM restriction of a journaling file system is too limiting to support versatile file-based transactions.

Two primary approaches were used to provide file-system transactions to user processes. (1) *Database file systems* provide transactions to user processes by making fundamental changes to the design of a standard file system to support better logging and rollback of inodes, dentries, and cached pages [33, 36, 43]. (2) *Database access APIs* provide transactions to user processes by offering a user library that exposes a transactional page file. Processes can store application data in the page file by using library-specific API routines rather than storing their data on the file system [34, 39]. Valor represents an alternative to the above two approaches. Valor's design was settled after designing KBDBFS and Amino [44]. We discuss KBDBFS and Amino in their proper contexts in Sections 2.2 and 2.3, respectively.

2.2 Database File Systems

KBDBFS was an in-kernel file system built on a port of the Berkeley Database [39] to the Linux kernel. It was part of a larger project that explored uses of a relational database within the kernel. KBDBFS utilized transactions to provide file-system–level consistency, but did not export these same semantics to user-level programs. It became clear to us that unlocking the potential value of a file system built on a database required exporting these transactional semantics to user-level applications. KBDBFS could not easily export these semantics to user-level applications, because as a standard kernel file system in Linux it was bound by the VFS to cache various objects (e.g., inodes and directory entries), all of which ran the risk of being rolled back by the transaction. To export transactions to user space, KBDBFS would therefore be required to either bypass the VFS layers that require these cached objects, or alternatively track each transaction's modifications to these objects. The first approach would require major kernel modifications and the second approach would duplicate much of the logging that BDB was already providing, losing many of the benefits provided by the database.

Our design of KBDBFS was motivated in part by a desire to modify the existing Linux kernel as little as possible. Another transaction system which modified an existing OS was Seltzer's log-structured file system, modified to support transaction processing [37]. Seltzer et al's simulations of transactions embedded in the file system showed that file system transactions can perform as well as a DBMS in disk-bound configurations [35]. They later implemented a transaction processing (TP) system in a log-structured file system (LFS), and compared it to a user-space TP system running over LFS and a read-optimized file system [37].

Microsoft's TxF [19, 43] and QuickSilver's [33] database file systems leverage the early incorporation of transactions support into the OS. TxF exploits the transaction manager which was already present in Windows. TxF uses multiple file versions to isolate transactional readers from transactional writers. TxF works only with NTFS and relies on specific NTFS modifications and how NTFS interacts with the Windows kernel. QuickSilver is a distributed OS developed by IBM Research that makes use of transactional IPC [33]. QuickSilver was designed from the ground up using a microkernel architecture and IPC. To fully integrate transactions into the OS, QuickSilver requires a departure from traditional APIs and requires each OS component to provide specific rollback and commit support. We wanted to allow existing applications and OS components to remain largely unmodified, and yet allow them to be augmented with simple begin, commit, and abort calls for file system operations. We wanted to provide transactions without requiring fundamental changes to the OS, and without restricting support to a particular file system, so that applications can use the file system most suited to their work load on any standard OS. Lastly, we did not want to incur any overheads on non-transactional processes.

Inversion File System [24], OdeFS [5], iFS [26], and DBFS [21] are database file systems implemented as user-level NFS servers [17]. As they are NFS servers (which predate NFSv4's locking and callback capabilities [38]), the NFS client's cache can serve requests without consulting the NFS server's database; this could allow a client application to write to a portion of the file system that has since been locked by another application, violating the client application's isolation. They do not address the problem of supporting efficient transactions on the local disk.

2.3 Database Access APIs

The other common approach to providing a transactional interface to applications is to provide a user-level library to store data in a special page file or B-Tree maintained by the library. Berkeley DB offers a B-Tree, a hash table, and other structures [39]. Stasis offers a page file [34]. These systems require applications to use database-specific APIs to access or store data in these library-controlled page files.

Based on our experiences with KBDBFS, we chose to prototype a transactional file system, again built on BDB, but in user space. Our prototype, Amino, utilized Linux's process debugging interface, `ptrace` [8], to service file-system–related calls on behalf of other processes, storing all data in an efficient Berkeley DB B-tree schema. Through Amino we demonstrated two main ideas. First, we revealed the ability to provide transactional semantics to user-level applications. Second, we showed the benefits that user-level programs gain when they use these transactional semantics: programming model simplification and application-level consistency [44]. Although we extended `ptrace` to reduce context switches and data copies, Amino's performance was still poor compared to an in-kernel file system for some system-call–intensive workloads (such as the configuration phase of a compile). Finally, although Amino's performance was comparable to Ext3 for metadata workloads (such as Postmark [14]), for data-intensive workloads, Amino's database layout resulted in significantly lower throughput. Amino was a successful project in that it validated the concept of a transactional file system with a user-visible transactional API, but the performance we achieved could not displace traditional file systems. Moreover, one of our primary goals is for transactional and non-transactional programs to have access to the same data through the file system interface. Although Amino provided binary compatibility with existing applications, running programs through a `ptrace` monitor is not as seamless as we liked. The `ptrace` monitor had to run in privileged mode to service all processes, it serviced system calls inefficiently due to additional memory copies and context switches, and it imposed additional overhead from using signal passing to simulate a kernel system call interface for applications [44]. Other user level approaches to providing transactional interfaces include Berkeley DB and Stasis.

Berkeley DB. Berkeley DB is a user library that provides applications with an API to transactionally update key-value pairs in an on-disk B-Tree. We discuss Berkeley DB's relative performance in depth in Section 4. We benchmark BDB through Valor's file system extensions. Relying on BDB to perform file system operations can result in large overheads for large serial writes or large transactions (256MiB or more). This is because BDB is being used to provide a file interface, which is used by applications with different work-loads than applications that typically use a database. If the regular BDB interface is used, though, transaction-oblivious processes cannot interact with transactional applications, as the formed use the file system interface directly.

Stasis. Stasis provides applications a transactional interface to a page file. Stasis requires that applications specify their own hooks to be used by the database to determine efficient undo and redo operations. Stasis supports nested transactions [7] alongside write-ahead logging and LSN-Free pages [34] to improve performance. Stasis does not require applications to use a B-Tree on disk and exposes the page file directly. Like BDB, Stasis requires applications to be coded against its API to read and write transactionally. Like BDB, Stasis does not provide a transactional interface on top of an existing file system which already contains data. Also like BDB, Stasis implements its own private, yet redundant page cache which is less efficient than cooperating with the kernel's page cache (see Section 4).

Reflecting on our experience with KBDBFS and Amino, we have come to the conclusion that adapting the file system interface to support ACID transactions does indeed have value and that the two most valuable properties that the database provided to us were the logging and the locking infrastructure. Therefore, in Valor we provide two key kernel facilities: (1) extended mandatory locking and (2) simple write ordering. Extended mandatory locking lets Valor provide the isolation that in our previous prototypes was provided by the database's locking facility. Simple write ordering lets Valor's logging facility use the kernel's page cache to buffer dirty pages and log pages which reduces redundancy, improves performance, and makes it easier to support transactions on top of existing file systems.

3 Design and Implementation

The design of Valor prioritizes (1) a low complexity kernel design, (2) a versatile interface that makes use of transactions optional, and (3) performance. Our seamless approach achieves low complexity by exporting just a minimal set of system calls to user processes. Functionality exposed by these system calls would be difficult to implement efficiently in user-space.

Valor allows applications to perform file-system operations within isolated and atomic transactions. Isolation guarantees that file-system operations performed within one transaction have no impact on other processes. Atomicity guarantees that committing a transaction causes all operations performed in it to be performed at once, as a unit inseparable even by a system crash. If desired, Valor can ensure a transaction is durable: if the transaction completes, the results are guaranteed to be safe on disk. We now turn to Valor's *transactional model*, which specifies the scope of these guarantees and what processes must do to ensure they are provided.

Transactional Model. Valor's transactional guarantees extend to the individual inodes and pages of directories and regular files for reads and writes. A process must lock an entire file if it will read from or write to its inode. Appends and truncations modify the file size, so they also must lock the entire file. To overwrite data in a file, only the affected pages need to be locked. When performing directory operations like file creation and unlinking, only the containing directory needs to be locked. When renaming a directory, processes must also recursively lock all of the directory's descendants. This is the accepted way to handle concurrent lockers during a directory rename [27]. More sophisticated locking schemes (e.g., intent locks [3]) that improve performance and relieve contention among concurrent processes are beyond the scope of this paper.

We now turn to the concepts underlying Valor's architecture. These concepts are implemented as components of Valor's system; they are illustrated in Figure 1.

1. Logging Device. In order to guarantee that a sequence of modifications to the file system completes as a unit, Valor must be able to undo partial changes left behind by a transaction that was interrupted by either a system crash or a process crash. This means that Valor must store some amount of auxiliary data, because an

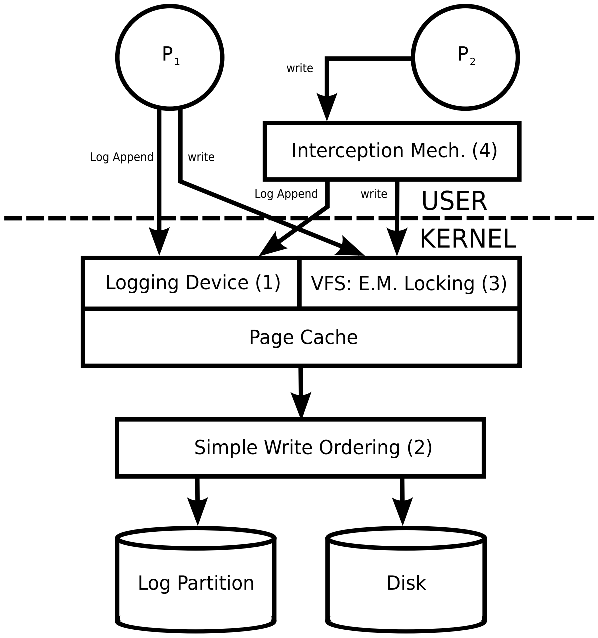

Figure 1: Valor Architecture

unmodified file system can only be relied upon to atomically update a single sector and does not provide a mechanism for determining the state before an incomplete write. Common mechanisms for storing this auxiliary data include a *log* [7] and WAFL [13]. Valor does not modify the existing file system, so it uses a log stored on a separate partition called the *log partition*.

2. Simple Write Ordering. Valor relies on the fact that even if a write to the file system fails to complete, the auxiliary information has already been written to the log. Valor can use that information to undo the partial write. In short, Valor needs to have a way to ensure that writes to the log partition occur before writes to other file systems. This requirement is a special case of *write ordering*, in which the page cache can control the order in which its writes reach the disk. We discuss our implementation in Section 3.1, which we call *simple write ordering* both because it is a special case and because it operates specifically at page granularity.

3. Extended Mandatory Locking. Isolation gives a process the illusion that there are no other concurrently executing processes accessing the same files, directories, or inodes. Transactional processes can implement this by first acquiring a lock before reading or writing to a page in a file, a file's inode, or a directory. However, an OS with a POSIX interface and pre-existing applications must support processes that do not use transactions. These *transaction-oblivious* processes do not acquire locks before reading from or writing to files or directories. *Extended mandatory locking* ensures that all processes acquire locks before accessing these resources. See Section 3.2.

4. Interception Mechanism. New applications can use special APIs to access the transaction functionality that Valor provides; however, pre-existing applications must be made to run correctly if they are executed inside a transaction. This could occur if, for example, a Valor-aware application starts a transaction and launches a standard shell utility. To do this, Valor modifies the standard POSIX system calls used by unmodified applications to perform the locking necessary for proper isolation. Section 3.3 describes our modifications.

The above four Valor components provide the necessary infrastructure for the seven Valor system calls. Processes that desire transactional semantics must use the Valor system calls to log their writes and acquire locks on files. We now discuss the Valor system calls and then provide a short example to illustrate Valor's basic operation.

Valor's Seven System Calls. When an application uses the following seven system calls correctly (e.g., calling the appropriate system call before writing to a page), Valor provides that application fully transactional semantics. This is true even if other user-level applications do not use these system calls or use them incorrectly.

`Log Begin` begins a transaction. This must be called before all other operations within the transaction.

`Log Append` logs an *undo-redo record*, which stores the information allowing a subsequent operation to be reversed. This must be called before every operation within the transaction. See Section 3.1.

`Log Resolve` ends a transaction. In case of an error, a process may voluntarily *abort* a transaction, which undoes partial changes made during that transaction. This operation is called an *abort*. Conversely, if a process wants to end the transaction and ensure that changes made during a transaction are all done as an atomic unit, it can *commit* the transaction. Whether a `log resolve` is a commit or an abort depends on a flag that is passed in.

`Transaction Sync` flushes a transaction to disk. A process may call `Transaction Sync` to ensure that changes made in its committed transactions are on disk and will never be undone. This is the only sanctioned way to achieve durability in Valor. `O_DIRECT`, `O_SYNC`, and `fsync` [9] have no useful effect within a transaction for the same reason that nested transactions cannot be durable: the parent transaction has yet to commit [7].

`Lock, Lock Permit, Lock Policy` Our `Lock` system call locks a page range in a file, an entire directory, or an entire file with a shared or exclusive lock. This is implemented as a modified `fcntl`. These routines provide Valor's support for transactional isolation. `Lock Permit` and `Lock Policy` are required for security and inter-process transactions, respectively. See Section 3.2.

Cooperating with the Kernel Page Cache. As illustrated in Figure 1, the kernel's page cache is central to Valor, and one of Valor's key contributions is its close cooperation with the page cache. In systems that do not support transactions, the `write(2)` system call initiates an asynchronous write which is later flushed to disk by the kernel page cache's dirty-page write-back thread. In Linux, this thread is called `pdflush` [1]. If an application requires durability in this scenario, it must explicitly call `fsync(2)`. Omitting durability by default is an important optimization which allows `pdflush` to economize on disk seeks by grouping writes together. Databases, despite introducing transaction semantics, achieve similar economies through *No-Force* page caches. These caches write auxiliary log records only when a transaction commits, and then only as one large serial write, and use threads similar to `pdflush` to flush data pages asynchronously [7]. Valor is also No-Force, but can further reduce the cost of committing a transaction by writing nothing—neither log pages nor data pages—until `pdflush` activates. Valor's simple write ordering scheme facilitates this optimization by guaranteeing that writes to the log partition always occur before the corresponding data writes. In the absence of simple write ordering, Valor would be forced to implement a redundant page cache, as many other systems do. Valor implements simple write ordering in terms of existing Linux `fsync` semantics which returns when the writes are scheduled, but before they hit the disk platter. This introduces a short race where applications running on top of Valor and the other systems we evaluated (Berkeley DB, Stasis, and ext3) could crash unrecoverably. Unfortunately, this is the standard `fsync` implementation and impacts other systems such as MySQL, Berkeley DB, and Stasis [45] which rely on `fsync` or its like (i.e., `fdatasync`, `O_SYNC`, and direct-IO).

One complexity introduced by this scheme is that a transaction may be completely written to the log, and reported as durable and complete, but its data pages may not yet all be written to disk. If the system crashes in this scenario, Valor must be able to complete the disk writes during recovery to fulfill its durability guarantee. Similar to database systems that also perform this optimization, Valor includes sufficient information in the log entries to *redo* the writes, allowing the transaction to be completed during recovery.

Another complexity is that Valor supports large transactions that may not fit entirely in memory. This means that some memory pages that were dirtied during an incomplete transaction may be flushed to disk to relieve memory pressure. If the system crashes in this scenario, Valor must be able to rollback these flushes during recovery to fulfill its atomicity guarantee. Valor writes *undo* records describing the original state of each affected page to the log when flushing in this way. A page cache that supports flushing dirty pages from uncommitted transactions is known as a *Steal* cache; XFS [41], ZFS [40], and other journaling file systems are No-Steal, which limits their transaction size [42] (see Section 2). Valor's solution is a variant of the ARIES transaction recovery algorithm [20].

An Example. Figure 2 illustrates Valor's writeback mechanism. A process P_2 initially calls the `Lock` system call to acquire access to two data pages in a file, then calls the `Log Append` system call on them, generating the two 'L's in the figure, and then calls `write(2)` to update the data contained in the pages, generating the two 'P's in the figure. Finally, it commits the transaction and quits. The processes did not call `transaction sync`. On the left hand side, the figure shows the state of the system before P_2 commits the transaction; because of Valor's non-durable No-Force logging scheme, data pages and corresponding undo/redo log entries both reside in the page cache. On the right hand side, the process has committed and exited; simple write ordering ensures that the log entries are safely resident on disk, and the data pages will be written out by `pdflush` as needed.

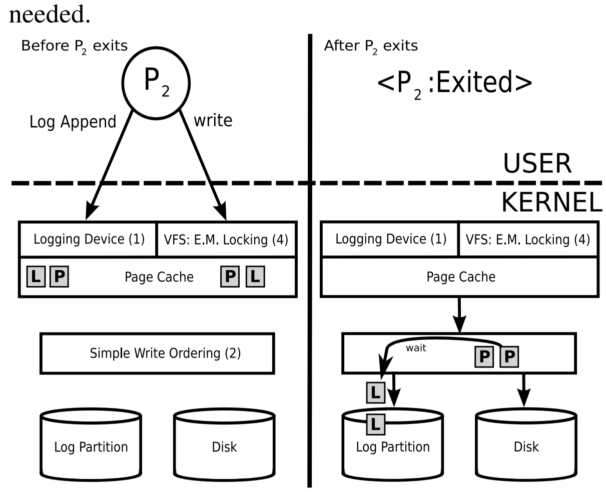

Figure 2: Valor Example

We now discuss each of Valor's four architectural components in detail. Section 3.1 discusses the logging, simple write ordering, and recovery components of Valor. Section 3.2 discusses Valor's extended mandatory locking mechanism, and Section 3.3 explains Valor's interception mechanism.

3.1 The Logging Interface

Valor maintains two logs. A *general-purpose log* records information on directory operations, like adding and removing entries from a directory, and inode operations, like appends or truncations. A *page-value log* records modifications to individual pages in regular files [2]. Before writing to a page in a regular file (*dirtying* the page), and before adding or removing a name from a directory, the process must call `Log Append` to prepare the associated undo-redo record. We refer to this undo-redo record as a *log record*. Since the bulk of file system I/O is from dirtying pages and not directory operations, we have only implemented Valor's page log for evaluation. Valor manages its logs by keeping track of the state of each transaction, and tracking which log records belong to which transactions.

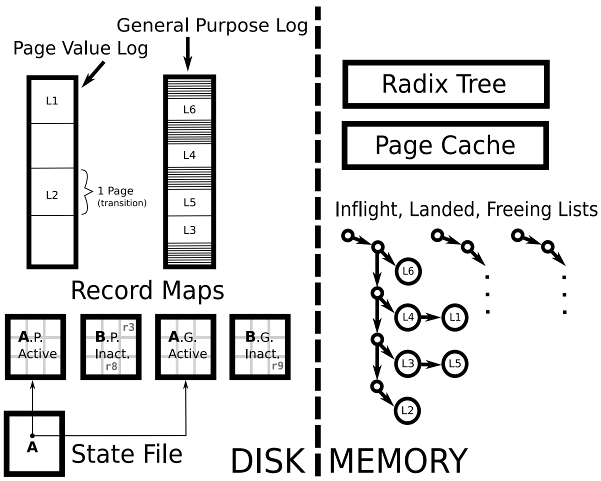

Figure 3: Valor Log Layout

3.1.1 In-Memory Data Structures

There are three states a transaction can be in during the course of its life: (1) *in-flight*, in which the application has called `Log Begin` but has not yet called `Log Resolve`; (2) *landed*, in which the application has called `Log Resolve` but the transaction is not yet safe to deallocate; and (3) *freeing*, in which the transaction is ready to be deallocated. Landed is distinct from freeing because if an application does not require durability, `Log Resolve` causes neither the log nor the data from the transaction to be flushed to disk (see above, *Cooperating with the Kernel Page Cache*).

Valor tracks a transaction by allocating a *commit set* for that transaction. A commit set consists of a unique *transaction ID* and a list of log records. As depicted in Figure 3, Valor maintains separate lists of in-flight, landed, and freeing commit sets. It also uses a radix tree to track free on-disk log records.

Life Cycle of a Transaction. When a process calls `Log Begin`, it gets a transaction ID by allocating a new log record, called a *commit record*. Valor then creates an in-memory commit set and moves it onto the inflight list. During the lifetime of the transaction, whenever the process calls `Log Append`, Valor adds new log records to the commit set. When the process calls `Log Resolve`, Valor moves its commit set to the landed list and marks it as *committed* or *aborted* depending on the flag passed in by the process. If the transaction is committed, Valor writes a *magic value* to the commit record allocated during `Log Begin`. If the system crashes and the log is complete, the value of this log record dictates whether the transaction should be recovered or aborted.

One thing Valor must be careful about is the case in which a log record is flushed to disk by `pdflush`, its corresponding file page is updated with a new value, and the file system containing that file page writes it to disk, thus violating write ordering. To resolve this issue, Valor keeps a flag in each page in the kernel's page cache. This flag can read *available* or *unavailable*; between the time Valor flushes the page's log record to the log and the time the file system writes the dirty page back to disk, it is marked as unavailable, and processes which try to call `Log Append` to add new log records wait until it becomes available, thus preserving our simple write ordering constraint. For hot file-system pages (e.g., those containing global counters), this could result in bursty write behavior. One possible remedy is to borrow Ext3's solution: when writing to an *unavailable* page, Valor can create a copy. The original copy remains read-only and is freed after the flush completes. The new copy is used for new reads and writes and is not flushed until the next `pdflush`, maintaining the simple write ordering.

We modified `pdflush` to maintain Valor's in-memory data structures and to obey simple write ordering by flushing the log's super block before all other super blocks. When `pdflush` runs, it (1) moves commit sets which have been written back to disk to the freeing list, (2) marks all page log records in the inflight and landed lists as unavailable, (3) atomically transitions the disk state to commit landed transactions to disk, and (4) iterates through the freeing list to deallocate transactions which have been safely written back to disk.

Soft vs. Hard Deallocations. Valor deallocates log records in two situations: (1) when a `Log Append` fails to allocate a new log record, and (2) when `pdflush` runs. *Soft deallocation* waits for `pdflush` to naturally write back pages and moves a commit set to the freeing list to be deallocated once all of its log records have had their changes written back. *Hard deallocation* explicitly flushes a landed commit set's dirty pages and directory modifications so it can immediately deallocate it.

3.1.2 On-Disk Data Structures

Figure 3 shows the page-value log and general-purpose log. Valor maintains two *record map* files to act as superblocks for the log files, and to store which log records belong to which transactions. One of these record map files corresponds to the general-purpose log, and the other to the page-value log. For a given log, there are exactly the same number of entries in the record map as there are log records in the log. The five fields of a record map entry are:

Transaction ID The transaction (commit set) this log record belongs to.

Log Sequence Number (LSN) Indicates when this log record was allocated.

inode Inode of the file whose page was modified.

netid Serial number of the device the inode resides on.

offset Offset of the page that was modified.

General-purpose log records contain directory path names for recovery of original directory listings in case of a crash. Page value log records contain a specially-encoded page to store both the undo and the redo record. The state file is part of the mechanism employed by Valor to ensure atomicity. It is described in Section 3.1.3 along with Valor's atomic flushing procedure.

Transition Value Logging. Although the undo-redo record of an update to a page could be stored as the value of the page before the update and the value after, Valor instead makes a reasonable optimization in which it stores only the XOR of the value of the page before and after the update. This is called a *transition page*. Transition pages can be applied to either recover or abort the on-disk image. A pitfall of this technique is that idempotency is lost [7]; Valor avoids this problem by recording the location and value of the first bit of each sector in the log record that differed between the undo and redo image. Although log records are always page-sized, this information must be stored on a per-sector basis as the disk may only write part of the page. (Because meta-data is stored in a separate map, transition pages in the log are all sector-aligned.) If a transaction updates the same page multiple times, Valor forces each `Log Append` call to wait on the Page Available flag which is set by the simple write ordering component operating within `pdflush`. If it does not have to wait, the call may update the log record's page directly, incurring no I/O. However, if the call must wait, then a new log record must be made to ensure recoverability.

3.1.3 LDST: Log Device State Transition

Valor's in-memory data structures are a reflection of Valor's on-disk state; however, as commit sets and log records are added, Valor's on-disk state becomes stale until the next time `pdflush` runs. We ensure that `pdflush` performs an atomic transition of Valor's on-disk state to reflect the current in-memory state, thus making it no longer stale. To represent the previous and next state of Valor's on-disk files, we have a *stable* and *unstable* record map for each log file. The stable record maps serve as an authoritative source for recovery in the event of a crash. The unstable record maps are updated during Valor's normal operation, but are subject to corruption in the event of crashes. The purpose of Valor's LDST is to make the unstable record map consistent, and then safely and atomically relabel the stable record maps as unstable and vice versa. This is similar to the scheme employed by LFS [32, 37].

The core atomic operation of the LDST is a pointer update, in which Valor updates the state file. This file is a pointer to the pair of record maps that is currently stable. Because it is sector-aligned and less than one sector in size, a write to it is atomic. All other steps ensure that the record maps are accurate at the point in time where the pointer is updated. The steps are as follows:

1. Quiesce (block) all readers and writers to any on-disk file in the Valor partition.
2. Flush the inodes of the page-value and general-purpose log files. This flushes all new log records to disk. Log records can only have been added, so a crash at this point has no effect as the stable records map does not point to any of the new entries.
3. Flush the inodes of the unstable page-value and general-purpose record map files.
4. Write the names of the newly stable record maps to the state file.
5. Flush the inode of the state file. The up-to-date record map is now stable, and Valor now recovers from it in case of a system crash.
6. Copy the contents of the stable (previously unstable) record map over the contents of the unstable (previously stable) record map, bringing it up to date.
7. Un-quiesce (unblock) readers and writers.
8. Free all freeing log records.

Atomicity. The atomicity of transactions in Valor follows from two important constraints which Valor ensures that the OS obeys: (1) that writes to the log partition and data partitions obey simple write ordering and (2) that the LDST is atomic. At mount time, Valor runs recovery (Section 3.1.4) to ensure that the log is ready and fully describes the on-disk system state when it is finished mounting. Thereafter, all proper transactional writes are preceded by `Log Append` calls. No writes go to disk until `pdflush` is called or Valor's `Transaction Sync` is called. Simple write ordering ensures that in both cases, the log records are written before the in-place updates, so no update can reach the disk unless its corresponding log record has already been written. Log records themselves are written atomically and safely because writes to the log's backing store are only made during an LDST. Since an LDST is atomic, the state of the entire system advances forward atomically as well.

3.1.4 Performing Recovery

System Crash Recovery. During the `mount` operation, the logging device checks to see if there are any outstanding log records and, if so, runs recovery. During `umount`, the Logging Device flushes all committed transactions to disk and aborts all remaining transactions. Valor can perform recovery easily by reading the state file to determine which record map for each log is stable, and reconstructing the commit sets from these record maps. A log sequence number (LSN) stored in the record map allows Valor to read in reverse order the events captured within the log and play them forward or back based on whether the write needs to be completed to satisfy durability or rolled back to satisfy atomicity. Recovery finds all record map entries and makes a commit set for each of them which is by default marked as aborted. While traversing through record map entries if it finds a record map entry with a magic value (written asynchronously during `Log Resolve`) indicating that this transaction was committed, it marks that set committed. Finally all commit sets are deallocated and an LDST is performed. The system can come on line.

Process Crash Recovery. Recovery handles the case of a system crash, something handled by all journaling file systems. However, Valor also supports user-process transactions and, by extension, user-process recovery. When a process calls the `do_exit` process clean-up routine in the kernel, their `task_struct` is checked to see if a transaction was in-flight. If so, then Valor moves the commit set for the transaction onto the landed list and marks the commit set as aborted.

3.2 Ensuring Isolation

Extended mandatory locking is a derivation of mandatory locking, a system already present in Linux and Solaris [10, 18]. Mandatory locks are shared for reads but exclusive for writes and must be acquired by all processes that read from or write to a file. Valor adds these additional features: (1) a locking permission bit for owner, group, and all (LPerm), (2) a lock policy system call for specifying how locks are distributed upon `exit`, and (3) the ability to lock a directory (and the requirement to acquire this lock for directory operations). System calls performed by non-transactional processes that write to a file, inode, or directory object acquire the appropriate lock before performing the operation and then release the lock upon returning from the call. Non-transactional system calls are consequently two-phase with respect to exclusive locks and well-formed with respect to writes. Thus Valor provides degree 1 isolation. In this environment, then, by the degrees of isolation theorem [7], transactional processes that obey higher degrees of isolation can have transactions with repeatable reads (degree 3) and no lost updates (degree 2).

Valor supports inter-process transactions by implementing inter-process locking. Processes may specify (1) if their locks can be recursively acquired by their children, and (2) if a child's locks are released or instead given to its parent when the child exits. These specifications are propagated to the Extended Mandatory Locking system with the `Lock Policy` system call.

Valor prevents misuse of locks by allowing a process to acquire a lock only under one of two circumstances: (1) if the process has permission to acquire a lock on the file according to the LPerm of the file, or (2) if the process has read access or write access, depending on the type of the lock. Only the owner of a file can change the LPerm, but changes to the LPerm take effect regardless of transactions' isolation semantics. Deadlock is prevented using a deadlock-detection algorithm. If a lock would create a circular dependency, then an error is returned. Transaction-aware processes can then recover gracefully. Transaction-oblivious processes should check the status of the failed system call and return an error so that they can be aborted. This works well in practice. We have successfully booted, used, and shutdown a previous version of the Valor system with extended mandatory locking and the standard legacy programs. A related issue is the locking of frequently accessed file-system objects or pages. The default Valor behavior is to provide degree 1 isolation, which prevents another transaction from accessing the page while another transaction is writing to it. For transaction-oblivious processes, because each individual system call is treated as a transaction, these locks are short lived. For transaction-aware processes, an appropriate level of isolation can be chosen (e.g., degree 2—no lost updates) to maximize concurrency and still provide the required isolation properties.

3.3 Application Interception

Valor supports applications that are aware of transactions but need to invoke subprocesses that are not transaction-aware within a transaction. Such a subprocess is wrapped in a transaction that begins when it first performs a file operation and ends when it exits. This is useful for a transactional process that forks subprocesses (e.g., `grep`) to do work within a transaction. During system calls, Valor checks a flag in the process to determine whether to behave transactionally or not. In particular, when a process is `fork`ed, it can specify if its child is transaction-oblivious. If so, the child has its

Transaction ID set to that of the parent and its in-flight state set to Oblivious. When the process performs any system call that constitutes a read or a write on a file, inode, or directory object, the in-flight state is checked, and an appropriate `Log Append` call is made with the Transaction ID of the process.

4 Evaluation

Valor provides atomicity, isolation, and durability, but these properties come at a cost: writes between the log device and other disks must be ordered, transactional writes incur additional reads and writes, and in-memory data structures must be maintained. Additionally, Valor is designed to provide these features while only requiring minor changes to a standard kernel's design. In this section we evaluate the performance of our Valor design and also compare it to stasis and BDB. Section 4.1 describes our experimental setup. Section 4.2 analyzes a benchmark based on an idealized ARIES transaction logger to derive a lower bound on overhead. Section 4.3 evaluates Valor's performance for a serial file overwrite. Section 4.4 evaluate Valor's transaction throughput. Section 4.5 analyzes Valor's concurrent performance. Finally, Section 4.6 measure Valor's recovery time. All benchmarks test scalability.

4.1 Experimental Setup

We used four identical machines, each with a 2.8GHz Xeon CPU and 1GB of RAM for benchmarking. Each machine was equipped with six Maxtor DiamondMax 10 7,200 RPM 250GB SATA disks and ran CentOS 5.2 with the latest updates as of September 6, 2008. To ensure a cold cache and an equivalent block layout on disk, we ran each iteration of the relevant benchmark on a newly formatted file system with as few services running as possible. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student's-t distribution. In each case, unless otherwise noted, the half widths of the intervals were less than 5% of the mean. Wait time is elapsed time less system and user time and mostly measures time performing I/O, though it can also be affected by process scheduling. We benchmarked Valor on the modified Valor kernel, and all other systems on a stable unmodified 2.6.25 Linux kernel.

Comparison to Berkeley DB and Stasis. The most similar technologies to Valor are Stasis and Berkeley DB (BDB): two user level logging libraries that provide transactional semantics on top of a page store for transactions with atomicity and isolation and with or without durability. Valor, Stasis, and BDB were all configured to store their logs on a separate disk from their data, a standard configuration for systems with more than one disk [7]. The logs used by Valor, Stasis, and BDB were set to 128MiB. Since Valor prioritizes non-durable transactions, we configured Stasis and BDB to also use non-durable transactions. This configuration required modifying the source code of Stasis to open its log without `O_SYNC` mode. Similarly, we configured BDB's environment with `DB_TXN_WRITE_NOSYNC`. The `ext3` file system performs writes asynchronously by default. For file-system workloads it is important to be able to perform efficient asynchronous serial writes, so non-durable transactions performing asynchronous serial writes were the focus during our benchmarking. BDB indexed each page in the file by its page offset and file ID (an identifier similar to an inode number). We used the B-Tree access method as this is the suitable choice for a large file system [44].

4.2 Mock ARIES Lower Bound

Figure 4 compares Valor's performance against a mock ARIES transaction system to see how close Valor comes to the ideal performance for its chosen logging system. We configured a separate logging block device with `ext2`, in order to avoid overhead from unnecessary journaling in the file system. We configured the data block device with `ext3`, since journaled file systems are in common use for file storage. We benchmarked a 2GiB file overwrite under three mock systems. MT-ow-noread performed the overwrite by writing zeros to the `ext2` device to simulate logging, and then writing zeros to the `ext3` device to simulate write back of dirty pages. MT-ow differs from MT-ow-noread in that it copies a pre-existing 2GiB data file to the log to simulate time spent reading in the before image. MT-ow-finite differs from the other mock systems in that it uses a 128MiB log, forcing it to break its operation into a series of 128MiB copies into the log file and writes to the data file. A transaction manager based on the ARIES design must do at least as much I/O as MT-ow-finite. Valor's overhead on top of MT-ow-finite is 35%. Stasis's is 104%. The cost of MT-ow reading the before images as measured by the overhead of MT-ow on MT-ow-noread is only 2%. The cost of MT-ow-finite restricting itself to a finite log is 16% due to required additional seeking. Stasis's overhead is more than Valor's overhead due to maintaining a redundant page cache in user space.

4.3 Serial Overwrite

In this benchmark we measure the time it takes for a process to transactionally overwrite an existing file. File transfers are an important workload for a file system. See Figure 5. Providing transactional protection to large file overwrites demonstrates Valor's ability to scale with larger workloads and handle typical file system operations. Since there is data on the disk already, all systems but `ext3` must log the contents of the page before

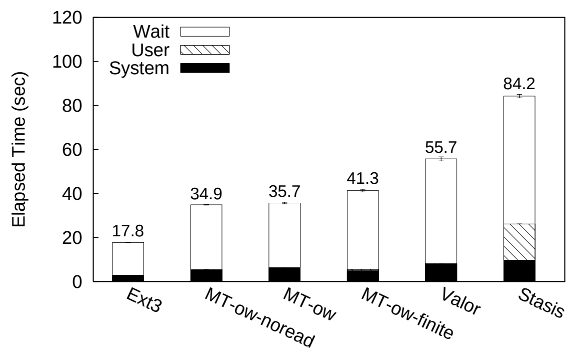

Figure 4: Valor and Stasis's performance relative to Mock ARIES Lower Bound

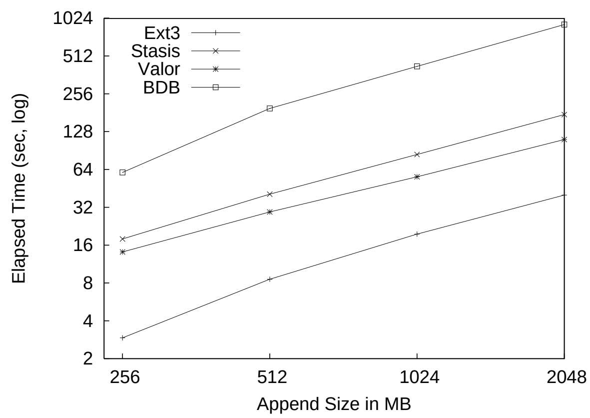

Figure 5: Async serial overwrite of files of varying sizes

overwriting it. The transactional systems use a transaction size of 16 pages. The primary observation from these results is that each system scales linearly with respect to the amount of data it must write. Valor runs 2.75 times longer than `ext3`, spending the majority of that overhead writing Log Records to the Log Device. Stasis runs 1.75 times slower than Valor. It spends additional time allocating pages in user space for its own page cache, and doing additional memory copies for its writes to both its log and its store file. For the 512 MiB over write of Valor and Stasis, and the 256 MiB over write of Stasis the half-widths were 11%, 7%, and 23% respectively. The asynchronous nature of the benchmark caused Valor and Stasis' page cache to introduce fluctuations in an otherwise stable serial write. BDB's on-disk B-Tree format, which is very different from Stasis's and Valor's simple page-based layout, makes it difficult to perform well in this I/O intensive workload that has little need for $log(n)$ B-Tree lookups. Because of this Valor runs 8.22 times the speed of BDB.

4.4 Transaction Granularity

We measured the rate for processing small durable transactions with varying transaction sizes. This benchmark establishes Valor's ability to handle many small transac-

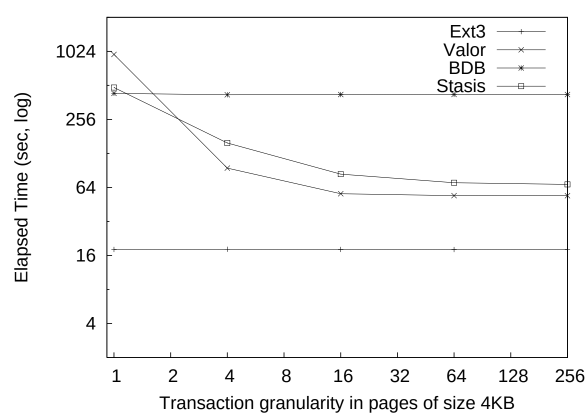

Figure 6: Run times for transactions, increasing granularity

tions, and indicates the overhead of beginning and committing a transaction on a write. We measured BDB, Valor, Stasis, and `ext3`. For `ext3` we simply used the native page size on the disk. See Figure 6. The throughput benchmark is simply the overwrite benchmark from Section 4.3, but we vary the size of the transaction rather than the amount of data to write. We see the typical result that the non-transactional system (`ext3`) is unaffected: transactional systems converge on a constant factor of the non-transactional system's performance as the overhead of beginning and committing a transaction approaches zero. BDB converges on a factor of 23 of `ext3`'s elapsed time, Stasis converges on a factor of 4.2, and Valor converges on a factor of 2.9. It is interesting that Valor has an overhead of 76% with respect to Stasis, and Stasis has an overhead of 25% with respect to BDB for single page transactions. BDB is oriented toward small transactions making updates to a B-Tree, not serial I/O. As the granularity decreases, Stasis and BDB converge to less efficient constant factors of the non-transactional `ext3`'s performance than what Valor converges to. This would imply that Valor's overhead for Log Append is lower than Stasis's since Valor operates from within the kernel and eliminates the need for a redundant page cache. For one page transactions BDB has already converged to a constant factor of `ext3`s performance starting at 1-page transactions: for transactions less than one page in size, BDB began to perform worse.

4.5 Concurrent Writers

Concurrency is an important measure of how a file interface can handle seeking and less memory while writing. One application of Valor would be to grant atomicity to package managers which may unpack large packages in parallel. To measure concurrency we ran varying numbers of processes that would each serially overwrite an independent file concurrently. Each process wrote 1GiB of data to its own file, and we ran the benchmark with 2, 4, 8, and 16 processes running concurrently. Figure 7 il-

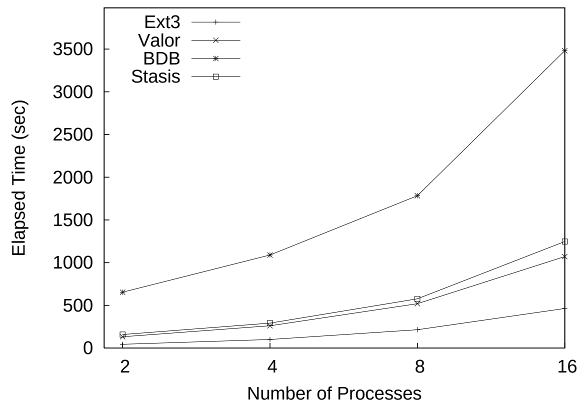

Figure 7: Execution times for multiple concurrent processes accessing different files

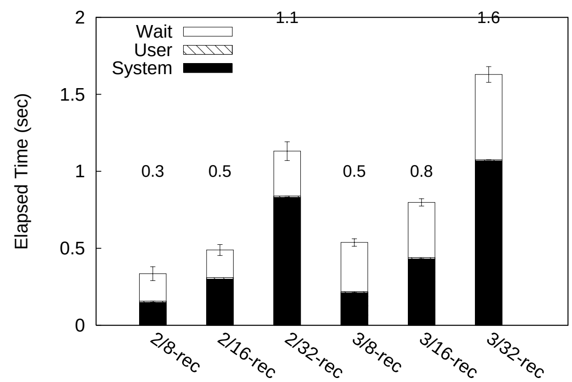

Figure 8: Time spent recovering from a crash for varying amounts of uncommitted data and varying number of processes

lustrates the results of our benchmark. For low numbers of processes (2, 4, and 8) BDB had half-widths of 35%, 6%, and %5 because of the high variance introduced by BDB's user space page cache. Stasis and BDB run at 2.7 and 7.5 times the elapsed time of `ext3`. For the 2, 4, 8, and 16 process cases, Valor's elapsed time is 3.0, 2.6, 2.4, and 2.3 times that of `ext3`. What is notable is that these times converge on lower factors of `ext3` for high numbers of concurrent writers. The transactional systems must perform a serial write to a log followed by a random seek and a write for each process. BDB and Stasis must maintain their page caches, and BDB must maintain B-Tree structures on disk and in memory. For small numbers of processes, the additional I/O of writing to Valor's log widens the gap between transactional systems and `ext3`, but as the number of processes and therefore the number of files being written to at once increases, the rate of seeks overtakes the cost of an extra log serial write for each data write, and maintenance of on-disk or in-memory structures for BDB and Stasis.

4.6 Recovery

One of the main goals of a journaling file system is to avoid a lengthy `fsck` on boot [11]. Therefore it is important to ensure Valor can recover from a crash in a finite amount of time with respect to the disk. Valor's ability to recover a file after a crash is based on its logging an equivalent amount of data during operation. The amount of total data that Valor must recover cannot exceed the length of Valor's log, which was 128 MiB in all our benchmarks. Valor's recovery process consists of: (1) reading a page from the log, (2) reading the original page on disk, (3) determining whether to roll forward or back, and (4) writing to the original page if necessary. To see how long Valor took to recover for a typical amount of uncommitted data, we tested the recovery of 8MiB, 16MiB and 32MiB of uncommitted data. In the first trial, two processes were appending to separate files when they crashed, and their writes had to be rolled back by recovery. In the second trial, three processes were appending to separate files. Process crash was simulated by simply calling `exit(2)` and not committing the transaction. Valor first reads the Record Map to reconstitute the in-memory state at the time of crash, then plays each record forward or back in reverse Log Sequence Number (LSN) order. Figure 8 illustrates our recovery results. Label `2/8-rec` in the figure shows elapsed time taken by recovery to recover 8MiB of data in the case of 2 process crash. We see that although the amount of time spent recovering is proportional to the amount of uncommitted data for both the 2 and 3 process case, that recovering 3 processes takes more time than for 2 because of additional seeking back and forth between pages on disk associated with log records for 3 uncommitted transactions instead of 2. 2/32-rec is 2.31 times slower than 2/16-rec and 2/16-rec is 1.46 times slower than 2/8-rec due to varying size of recoverable data. Similarly, 3/32-rec is 2.04 times slower than 3/16-rec and 3/16-rec is 1.5 times slower than 3/8-rec. Keeping the amount of recoverable data same we see that 3 processes have 44%, 63%, and 60% overhead compared to 2 process with recoverable data of 8MiB, 16MiB, 32MiB, respectively. In the worst case, Valor recovery can become a random read of 128MiB of log data, followed by another random read of 128MiB of on-disk data, and finally 128MiB of random writes to roll back on-disk data.

Valor does no logging for read-only transactions (e.g., `getdents`, `read`) because they do not modify the file system. Valor only acquires a read lock on the pages being read, and, because it calls directly down into the file system to service the read request, there is no overhead.

Systems which use an additional layer of software to translate file system operations into database operations and back again introduce additional overhead. This is why Valor achieves good performance with respect to other database-based user level file system implementations that provide transactional semantics. These alternative APIs can perform well in practice, but only if applications use their interface, and constrain their workloads to reads and writes that perform well in a standard database rather than a file system. Our system does not have these restrictions.

5 Conclusions

Applications can benefit greatly from having a POSIX-compliant transactional API that minimizes the number of modifications needed to applications. Such applications can become smaller, faster, more reliable, and more secure—as we have demonstrated in this and prior work. However, adding transaction support to existing OSs is hard to achieve simply and efficiently, as we had explored ourselves in several prototypes.

This paper has several contributions. First, we describe two older prototypes and designs for file-based transactions: (1) KBDBFS which attempted to port a standalone BDB library and add file system support into the Linux kernel—adding over 150,000 complex lines-of-code to the kernel, duplicating much effort; (2) Amino, which moved all that functionality to user level, making it simpler, but incurring high overheads.

The second and primary contribution of this paper is our design of Valor, which was informed by our previous attempts. Valor runs in the kernel cooperating with the kernel's page cache, and runs more efficiently: Valor's performance comes close to the theoretical lower bound for a log-based transaction manager, and scales much better than Amino, BDB, and Stasis

Unlike KBDBFS, however, Valor integrates seamlessly with the Linux kernel, by utilizing its existing facilities. Valor required less than 100 LoC changes to `pdflush` and another 300 LoC to simply wrap system calls; the rest of Valor is a standalone kernel module which adds less than 4,000 LoC to the stackable file system template Valor was based on.

Future Work. One of our eventual goals is to explore the use of Log Structured Merge Trees [25] to optimize our general purpose log and provide faster name lookups (e.g. decreasing the elapsed time of `find`).

Another interesting research direction is to use NFSv4's compound calls to implement network-based file transactions [38]. This may require semantic change to NFSv4 so as to not allow partial success of some operations within a compound, and to allow the NFS server to perform atomic updates to its back-end storage.

Finally we intend to further investigate the ramifications of weakening `fsync` semantics in light of current trends in hard drive write cache design. We want to explore the possibility of extending asynchronous barrier writes based on native command queueing to the user level layer so that systems which use atomicity mechanisms across multiple devices (e.g., via a logical volume manager or multiple mounts) can retain atomicity. We believe we could avoid hard drive cache flushes [23] using tagged I/O support for SATA drives and export this write ordering primitive to layers higher than the block device and file system implementation. We also are interested in analyzing the probability of failure when using varying semantics for `fsync` as well as analyzing the associated performance trade-offs.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Ohad Rodeh, for their helpful comments. Special thanks go to Russel Sears, Margo Seltzer, Vasily Tarasov, and Chaitanya Yalamanchili for their help evaluating this evolving project. This work was made possible in part thanks to an NSF award CNS-0614784.

References

[1] D. P. Bovet and M. Cesati. *Understanding the LINUX KERNEL*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebatopol, CA 95472, 2005.

[2] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proc. of the Annual USENIX Technical Conf., FREENIX Track*, pp. 19–28, Boston, MA, Jun. 2004.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 3rd edition, 2000.

[4] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Proc. of the Annual USENIX Technical Conf.*, pp. 89–104, Anaheim, CA, Apr. 2005.

[5] N. H. Gehani, H. V. Jagadish, and W. D. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In *Proc. of the 20th International Conf. on Very Large Databases*, pp. 249–260, Santiago, Chile, Sept. 1994. Springer-Verlag Heidelberg.

[6] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[7] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[8] M. Haardt and M. Coleman. *ptrace(2)*. Linux Programmer's Manual, Section 2, Nov. 1999.

[9] M. Haardt and M. Coleman. *fsync(2)*. Linux Programmer's Manual, Section 2, 2001.

[10] M. Haardt and M. Coleman. *fcntl(2)*. Linux Programmer's Manual, Section 2, 2005.

[11] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, pp. 155–162, Austin, TX, Oct. 1987.

[12] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 3–6, Copper Mountain Resort, CO, Dec. 1995.

[13] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proc. of the USENIX Winter Technical Conf.*, pp. 235–245, San Francisco, CA, Jan. 1994.

[14] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[15] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. of the Summer USENIX Technical Conf.*, pp. 238–247, Atlanta, GA, Jun. 1986.

[16] J. MacDonald, H. Reiser, and A. Zarochentcev. Reiser4 transaction design document. `www.namesys.com/txn-doc.html`, Apr. 2002.

[17] D. Maziéres. A toolkit for user-level file systems. In *Proc. of the Annual USENIX Technical Conf.*, pp. 261–274, Boston, MA, Jun. 2001.

[18] R. McDougall and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, Upper Saddle River, New Jersey, 2006.

[19] Microsoft Corporation. Microsoft MSDN WinFS Documentation. `http://msdn.microsoft.com/data/winfs/`, Oct. 2004.

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[21] N. Murphy, M. Tonkelowitz, and M. Vernal. The Design and Implementation of the Database File System. `www.eecs.harvard.edu/~vernal/learn/cs261r/index.shtml`, Jan. 2002.

[22] MySQL AB. MySQL: The World's Most Popular Open Source Database. `www.mysql.org`, Jul. 2005.

[23] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, pp. 1–14, Seattle, WA, Nov. 2006.

[24] M. A. Olson. The Design and Implementation of the Inversion File System. In *Proc. of the Winter 1993 USENIX Technical Conf.*, pp. 205–217, San Diego, CA, Jan. 1993. USENIX.

[25] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[26] Oracle Corporation. Oracle Internet File System Archive Documentation. `http://otn.oracle.com/documentation/ifs_arch.html`, Oct. 2000.

[27] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow. The nfs version 4 protocol. In *In Proc. of the 2nd International System Administration and Networking Conf.*, page 94, 2000.

[28] Calton Pu, Jim Johnson, Rogério de Lemos, Andreas Reuter, David Taylor, and Irfan Zakiuddin. 06121 report: Break out session on guaranteed execution. In *Atomicity: A Unifying Concept in Computer Science*, 2006.

[29] Calton Pu and Jinpeng Wei. A methodical defense against tocttou attacks: The edgi approach. In *Proc. of the International Symposium on Secure Software Engineering (ISSSE'06)*, pp. 399–409, Mar. 2006.

[30] V. K. Reddy and D. Janakiram. Cohesion Analysis in Linux Kernel. *apsec*, 0:461–466, 2006.

[31] H. Reiser. ReiserFS. `www.namesys.com/`, Oct. 2004.

[32] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[33] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pp. 239–253, Pacific Grove, CA, Oct. 1991.

[34] R. Sears and E. Brewer. Stasis: Flexible Transactional Storage. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

[35] M. Seltzer and M. Stonebraker. Transaction Support in Read Optimized and Write Optimized File Systems. In *Proc. of the Sixteenth International Conf. on Very Large Databases*, pp. 174–185, Brisbane, Australia, Aug. 1990. Morgan Kaufmann.

[36] M. I. Seltzer. Transaction support in a log-structured file system. In *Proc. of the Ninth International Conf. on Data Engineering*, pp. 503–510, Vienna, Austria, Apr. 1993.

[37] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proc. of the Ninth International Conf. on Data Engineering*, pp. 503–510, Vienna, Austria, Apr. 1993.

[38] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3530, Network Working Group, Apr. 2003.

[39] Sleepycat Software, Inc. *Berkeley DB Reference Guide*, 4.3.27 edition, Dec. 2004. `www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html`.

[40] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[41] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. of the Annual USENIX Technical Conf.*, pp. 1–14, San Diego, CA, Jan. 1996.

[42] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, Jul. 2000. `http://olstrans.sf.net/release/OLS2000-ext3/OLS2000-ext3.html`.

[43] S. Verma. Transactional NTFS (TxF). `http://msdn2.microsoft.com/en-us/library/aa365456.aspx`, 2006.

[44] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):1–42, Jun. 2007.

[45] Peter Zaitsev. True fsync in linux (on ide). Technical report, MySQL AB, Senior Support Engineer, Mar. 2004. `lkml.org/lkml/2004/3/17/188`.