# Extending ACID Semantics to the File System

CHARLES P. WRIGHT
IBM T. J. Watson Research Center

RICHARD SPILLANE, GOPALAN SIVATHANU, and EREZ ZADOK
Stony Brook University

An organization's data is often its most valuable asset, but today's file systems provide few facilities to ensure its safety. Databases, on the other hand, have long provided transactions. Transactions are useful because they provide atomicity, consistency, isolation, and durability (ACID). Many applications could make use of these semantics, but databases have a wide variety of non-standard interfaces. For example, applications like mail servers currently perform elaborate error handling to ensure atomicity and consistency, because it is easier than using a DBMS. A transaction-oriented programming model eliminates complex error-handling code, because failed operations can simply be aborted without side effects. We have designed a file system that exports ACID transactions to user-level applications, while preserving the ubiquitous and convenient POSIX interface. In our prototype ACID file system, called Amino, updated applications can protect arbitrary sequences of system calls within a transaction. Unmodified applications operate without any changes, but each system call is transaction protected. We also built a recoverable memory library with support for nested transactions to allow applications to keep their in-memory data structures consistent with the file system. Our performance evaluation shows that ACID semantics can be added to applications with acceptable overheads. When Amino adds atomicity, consistency, and isolation functionality to an application, it performs close to Ext3. Amino achieves durability up to 46% faster than Ext3, thanks to improved locality.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*; D.4.3 [**Operating Systems**]: File Systems Management—*Access methods, Directory structures, File organization*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; H.2 [**Database management**]: Database Applications

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: File system transactions, Recoverable memory, Databases, File systems, `ptrace` monitors

## 1. INTRODUCTION

File systems offer a convenient and standard interface for user applications to store data, which is many organizations' most valuable asset. Computer hardware and software can be replaced, but lost or corrupted data can not. Providing reliable file system access is therefore an important goal of any operating system.

Database systems provide strong guarantees for the safety and consistency of data, but each database uses its own interface. Four key requirements define a transaction: atomicity, consistency, isolation, and durability—collectively known as the *ACID* properties. Despite their importance, most file systems have made no provisions to ensure that operations meet all four of these stringent requirements. Our goal is to combine the best part of databases, their reliability (embodied by the ACID properties)—with the best part of file systems, their common and easy-to-use

POSIX API [IEEE/ANSI 1996].

Next, we describe the ACID requirements, and how they relate to file systems.

***Atomicity.*** Atomicity means that operations must complete or fail as a whole unit. Traditionally, file systems provided only limited atomicity (e.g., renaming a file either fails or succeeds). Many applications undertake arduous procedures to try to perform atomic operations. For example, if Sendmail fails when attempting to append new mail messages to a mailbox, it then attempts to truncate the file to erase a partially written message [Sendmail Consortium 2004]. Yet if the truncation fails, then the mailbox is left corrupted. To solve these problems, a file system should allow a sequence of operations to be encapsulated in a single atomic transaction. This has two key benefits: (1) error handling becomes easier, because transactions can simply be aborted, and (2) data corruption cannot occur, because no corrupted data ever reaches the file system. With this new functionality, Sendmail's append operations could be wrapped in a transaction. If they all succeeded, then Sendmail would commit the transaction. Otherwise, Sendmail would abort the transaction and the file-system state would not change.

***Consistency.*** In the context of a database system, consistency means that the database enforces pre-defined integrity constraints. Examples of integrity constraints in a database system are that social security numbers must be unique or that a checking account must have a positive balance. File systems have similar constraints (e.g., inode numbers are unique and no directory entry points to a non-existent inode). By wrapping related operations in database transactions, a file system can maintain a consistent on-disk state.

Applications also have consistency requirements. For example, when committing files to CVS [Berliner and Polk 2001], lock files are created to protect against concurrent accesses. An integrity constraint in this example is that lock files only exist while an instance of CVS is updating the repository. In an unmodified CVS implementation, there are circumstances in which lock files are not properly deleted (e.g., on unexpected termination or occasionally when the user presses Control-C). Using transactions greatly improves error handling—with only four lines of code we were able to prevent CVS from leaving stale lock files. Additionally, we eliminated the possibility of some files being committed, and others not (e.g., if the process is terminated half-way through a commit). If CVS were to have used a transactional model from the start, then hundreds of lines of code through several source files could have been eliminated. Moreover, because the transactional interface does not commit data until all operations succeed, error-handling is much more robust than the several ad-hoc functions that are currently in use.

***Isolation.*** Isolation (or serialization) means that one transaction will not affect the execution of another concurrently running transaction. This is not available in current file systems. For example, a set-UID program cannot use `access` to check whether a user has permission to create a file, because another process could create a symbolic link to a sensitive file between the `access` and the creation. This is known as a time-of-check-time-of-use (TOCTOU) security vulnerability. With a file system that maintains isolation, for example, `access` and file creation can safely be performed in a single transaction so that no other operations could be interleaved between the `access` and the creation; to improve performance, however,

Table I. File system support for ACID. Current file systems cannot provide all ACID properties across multiple operations, but many do provide a subset of the ACID properties for a single operation (i.e., a system call or VFS-level operation). Amino provides all of the ACID properties for an arbitrary sequence of multiple operations.
* FFS-no-SU denotes FFS without SoftUpdates, and FFS+SU denotes FFS with SoftUpdates.

| | **Ext2 and FFS-no-SU*** | **Ext3** | **FFS+SU*** | **Amino** |
|---|---|---|---|---|
| **Atomicity** | No | Single op | No | Multiple ops |
| **Consistency** | No | Multiple ops | Multiple ops, but resources may leak | Multiple ops |
| **Isolation** | Single op | Single op | Single op | Multiple ops |
| **Durability** | Only with O_SYNC | Only with O_SYNC | Only with O_SYNC | Legacy: each op. Enhanced: on commit. |

other operations may be interleaved, but the database management system ensures that the results are as if there was no interleaving.

***Durability***. Once a transaction is committed to disk, the data remains intact even across a software or a hardware crash. This is a desirable property for every application, but often operating systems (OSes) choose to sacrifice durability for better performance, because the synchronous I/O required for durability results in poor performance. Databases employ optimizations such as sequential logs, group commit, and ordered writes to provide durability more efficiently.

As seen in Table I, current file systems do not support full ACID properties. Traditional file systems do not provide atomicity. For example, during `rename`, Ext2 and FFS can both create the file's new name, and then fail before the old name is removed. Journaling file systems like Ext3 provide atomicity for a single operation, so a rename operation cannot fail half-way through, but they do not provide atomicity for a sequence of multiple operations, which is vital for user applications. Many file systems do not provide consistency, which has resulted in the need to run a consistency checker before mounting them (`fsck`). Journaling file systems and SoftUpdates ensure that each operation is consistent, so the composition of many operations is also consistent [McKusick and Ganger 1999]. Current file systems use VFS-level locking to provide isolation for a single operation. For example, a directory is locked before it is modified. However, there is no mechanism to isolate one sequence of operations from another operation (or sequence). To improve performance, current file systems do not provide durable writes unless the `O_SYNC` option is specified.

We believe that the ACID properties are desirable for many applications, especially applications like email that are expected to be highly reliable, or applications that require atomicity and isolation for security (e.g., updating a user's credentials). Therefore, we have designed a file system called *Amino* that extends ACID semantics to standard applications that use the POSIX interface. Legacy support is essential: unmodified applications and file systems continue to work as they have in the past. To exercise fine-grained control over transactions, existing applications need only slight modifications, and benefit from improved reliability.

It can be argued that databases are already taking over for the file system when reliable storage is required. For example, some commercial email systems store messages in databases instead of the file system [Sendmail, Inc. 2004], and it is becoming more common for revision-control systems to store information in a database [CollabNet, Inc. 2004]. However, we believe that writing applications that use the file system interface has inherent advantages over writing applications that use the database interface. When an application is written to a database API, it severely limits its interoperability and adds to the burden of programmers and administrators. For example, with a mail server using a file system, an individual user's mail file can simply be copied to create a backup, or deleted to remove all of the user's messages (from personal experience working at an ISP, this is a not uncommon request). Moreover, any standard text processing package can be used to edit the file. When data is only accessible through a database interface, these types of convenient access are no longer possible. Instead, special applications must be written for each of these functionalities.

We have built Amino on top of the Berkeley Database (BDB) [Seltzer and Yigit 1991]. BDB is an embedded database package that provides efficient transaction-protected access to key-value pairs in hash tables or balanced trees. BDB provides the crucial database infrastructure such as logging, locking, and caching. However, BDB, does not provide or require the use of SQL, stored procedures, a specialized database server, or other heavyweight components often associated with a DBMS. This makes it ideal for use by other operating system components. Using BDB allows us to leverage almost 200,000 lines of time-tested industrial-strength code.

If we were to implement Amino as a traditional file system that interfaces with the VFS, we would have to use the inode, dentry, and page caches. If a transaction aborted, then these caches would become stale with respect to the database. Therefore, we chose to implement Amino as a user-level monitor using the process-tracing facility (`ptrace`) provided by Linux. This interface allows us to intercept all system calls and use only the internal BDB caches. For internal Amino data structures, we developed a recoverable virtual memory (RVM) system on top of BDB that provides support for nested transactions and is transparent to applications.

We evaluated our prototype, and show that it can add atomicity, consistency, and isolation to existing applications with negligible performance overheads. To provide durability, existing file systems require the `O_SYNC` option. Amino can implicitly provide durability, and is 46% faster than a traditional file system in synchronous mode. If a programmer informs Amino when transactions begin and end, durable performance is 173% better than a traditional file system. Given that Amino is an unoptimized user-level prototype, we find these results encouraging and expect that performance can improve with more tuning.

The rest of this article is organized as follows. Section 2 provides an overview of our design. Section 3 describes our current Amino prototype. Section 4 evaluates Amino's performance. Section 5 describes related work. We conclude and discuss future work in Section 6.

## 2. DESIGN

The key decision to make when extending ACID semantics to a file system is whether to graft additional code to provide transactions onto an existing file system, or to build a file system on top of a system that already provides transactional semantics. The advantages of adding code to the file system is that you may end up with less overall code, which is more specialized to the task at hand. However, adding even a subset of the required code to an existing file system can take years. For example, Ext3 shares most of its code with Ext2 and only adds atomicity to single file system operations, but it took more than two years to develop. To get a rough idea of how large a file system is versus a transactional processing system, we can compare the number of lines of code in Ext3 to the number of lines in version 4.1 of the open-source MySQL server [MySQL AB 2005] and version 4.2.52 of the Berkeley Database (BDB) [Seltzer and Yigit 1991; Sleepycat Software, Inc. 2004]. In Linux 2.6.11.12, Ext3 has 21,629 lines of code (including the block journaling layer, *jbd*, which is used only for Ext3). BDB has over 16,870 lines of code in just its transaction-related components, and BDB is a subset of MySQL's overall transaction code (MySQL uses BDB to provide transactional tables). Aside from the transaction-related components, BDB provides efficient data access methods for key-value pairs (e.g., BDB's balanced-tree implementation is 16,843 lines of code). We therefore chose to build our file system on top of BDB, because we can leverage the already existing transactions infrastructure and efficient access methods.

Once we decided to build the file system on top of a transaction-processing system, the next question was what transaction-processing system is an appropriate host for the file system. One option would have been to use an SQL server such as MySQL, PostgreSQL, or Oracle. We rejected using a full-fledged SQL server, because they require significant runtime resources. Moreover, each database update or query requires communication over a socket, thus degrading performance by adding extra context switches and data copies. We therefore chose to use an embedded database, which runs directly in the address space of the client—thereby eliminating context switches and data copies. BDB fits our needs well. It is widely deployed, and has been thoroughly tested. BDB also scales both up and down: it can have a small memory footprint of less than 500KB, yet it also can be configured for databases as large as 256TB. BDB's codebase is still tractable at about 200,000 lines of code. There are two key reasons that BDB's codebase is manageable. First, BDB does not require or support SQL parsing, query planning, or other features often associated with a DBMS. As these features are not needed for a file system, having less code is a distinct advantage. Second, BDB has a modular design and the application designer can choose which components to use (e.g., the transaction subsystem can be used with normal files, or the access methods can be used without logging). Even though BDB is a relatively small DBMS, it still provides the key infrastructure for full ACID semantics: logging, locking, recovery, and a full-featured transactions API. It also provides four data access methods: a sorted balanced search tree, extended linear hashing, a fixed-length record queue, and access by logical record number.

The rest of this section is organized as follows. Section 2.1 provides an overview of BDB. Section 2.2 describes our database schema. Section 2.3 describes our internal use of transactions. Section 2.4 describes our use of transactional memory. Section 2.5 describes the transactions API that we expose to applications.

## 2.1 BDB Overview

BDB provides a uniform API to access both hash tables and balanced search trees in a transactional manner. To open and use BDB databases, a database environment is opened first. The database environment provides caching, logging, and locking functionality for one or more databases (or even simple files). Transactions are associated with the environment, and they have three operations: begin, commit, and abort. Other database operations are protected by the transaction. If a transaction is committed, then all of the protected operations are applied to stable storage as a whole. If the transaction is aborted, then it has no effects. A single transaction can span multiple databases, but the databases must all belong to the same environment. Before a database is opened, a database handle is created and associated with an environment. Next, the handle's parameters are set (e.g., the page size, sorting or hashing function, etc.). Finally, the database is opened inside of a transaction using the fully configured handle. After the database or databases are opened, key-value pairs can be stored using a PUT operation and retrieved using a GET operation. These primitives take the database handle, a transaction, the key, and the value (for PUT) as arguments. Also, BDB provides support for *cursors*, which efficiently iterate through items in the database. The primary cursor operations we are concerned with are `DB_SET`, `DB_SET_RANGE`, and `DB_NEXT`, which find a given key, the first key that is greater than a given key, and the next key, respectively. There are many other BDB operations and parameters, which we omit here for brevity [Sleepycat Software, Inc. 2004].

## 2.2 File System Schema

The database schema defines the format of our file system. The schema dictates the topology of the data, which in turn is directly related to what operations are possible, and how efficient each operation is. Our primary goal in developing our schema was to minimize the number of database accesses required for any given operation, because I/O operations are many orders of magnitude slower than in-memory operations. An organization that is appropriate for a normal disk-based file system is not necessarily appropriate for a database. For example, most FFS-like file systems use simple mappings of integers to disk blocks [McKusick et al. 1984]. For example, to read a block from a file, first the root inode number is mapped to a disk block. After the root inode is read, the root directory's data blocks are scanned to find the inode number of the next pathname component. Reading each data block essentially maps a logical block in the file to a physical disk block using the inode's direct and indirect pointers. This procedure must be repeated for each pathname component, until the file is found.

BDB, on the other hand, provides more complex and efficient data structures. In BDB, the schema is defined by the set of databases and their key-value pairs. A file system can conceptually be divided into two halves: (1) a naming component and (2) a data storage component. For example, FreeBSD has a separate UFS component for naming and an FFS component for storage. Our schema, shown in Table II, has a similar division. We use a *Path* database to map pathnames to unique file identifiers, and a *Data* database to map unique file identifiers to file data. The *Orphan* database contains a list of identifiers that are not accessible

Table II. Our database schema. Directory-reading and lookup operations use the Path database, which maps full path names to path-local meta-data. Read, write, truncate, and other data-oriented operations use the Data database. The Data database has two types of keys: a file identifier points to its meta-data, and a file identifier concatenated with a page index point to the page's data. Files without any names are stored in the Orphan database.

| **Database** | **Key** | **Value** |
|---|---|---|
| Path | Full Path | ID\|\|Path-local meta-data (e.g., `stat` information for a file without hardlinks) |
| Data | ID | Reference Count \|\| Data-local meta-data (e.g., `stat` information for a hard linked file) |
|  | ID \|\| Page index | Page's data |
| Orphan | ID | Path-local meta-data (e.g., `stat` information for a file without hard links) |

through the name space, but is otherwise equivalent to the Path database.

In the rest of this section we describe our schema's design considerations. First we discuss each database in turn: the Path database, the Data database, and then the Orphan database. We then describe path-local and data-local meta-data.

*The Path Database.* The Path database is used for both lookup and directory-reading operations. Each file has a unique identifier, which is analogous to an inode number. In the Path database, the key is a full pathname and the value is a unique identifier. We designed our schema such that a given file can be looked up using a single database access. For any given path name we can quickly find the path's unique identifier, without the need to traverse each component's directory separately as is done in most Unix file systems. The Google file system uses a similar scheme [Ghemawat et al. 2003]. When using a hash function, this yields constant time lookups. Using a balanced tree with a fan-out of 100 keys per page, four disk accesses are always sufficient to find any of $10^8$ files.

The Path database is also suitable for the directory-reading operation. As the access method for the Path database, we selected a balanced tree structure using a customized sort function. In our database, pathnames are first sorted by depth (i.e., by an ascending number of pathname components) and then by standard lexicographic order. Using this sorting function means that for any given directory, every name is contiguous within the database. To read a directory, we use BDB's `DB_SET_RANGE` operator to position a cursor at the first path name within the directory. To read each subsequent entry we use the cursor's `DB_NEXT` operator until we encounter a path name in a different directory.

For the `lookup` operation, the sort function is not critical, as a name can be located correctly with any total ordering. However, our sorting function proves advantageous when reading a directory and performing `stat` operations on the entries. Because each path in the directory is located close to one another, fewer pages must be read in from disk. This type of operation is quite common (e.g, by `ls -l` or recursive tree scans), which is why NFSv3 introduced a single protocol primitive called READDIRPLUS for it [Callaghan et al. 1995].

*The Data Database.* To store the data pages, we use a balanced tree. If a file's unique identifier is stored in the tree, then the given file exists. We assign the identifier randomly, but as the tree is sorted, it is possible to influence data layout policies by modifying the identifier assignment and sort function. For each identifier, the database stores the file's reference counts and meta-data. There are two reference counts: one for the number of path names that reference it (a.k.a. a link count), and another for the number of open instances of the file.

The actual data associated with the file is also stored in the Data database. For a given page of the file, the key is the file's identifier concatenated with the page index. We first sort the tree by the file's identifier and then by the page index. This means that all of a file's data pages are allocated contiguously in the tree, thereby improving locality and allowing the use of database cursors.

Selecting database parameters properly is of the utmost importance for the Data database. In our experiments we found that there can be a factor of ten difference in performance based on page size, cursor use, and other database-tuning parameters. The page size is a particularly important parameter for data-intensive operations. BDB uses a configurable database page size of powers-of-two between 512 bytes and 64KB. It is often useful to make this page size the native page size of the underlying file system, so that BDB reads and writes pages that are compatible with the OS's native page size. The BDB page size also determines when and how *overflow pages* are used. For the Data database, most records are rather large, so they are stored in overflow pages, which means that they are not stored directly with the key. We have found that BDB will store only a single record within an overflow page. Therefore, if the database page size is larger than our file system's transfer unit (for the remainder of this paragraph we refer to our file systems page as a transfer unit to avoid confusion with BDB pages), then the remainder of the database's overflow page is wasted, reducing available disk space and imposing unnecessary I/O overheads. Similarly, if the overflow page size is less than or equal to the file system transfer unit, then BDB stores a small amount of internal meta-data in the beginning of the overflow page, and the first part of the actual data in the remainder of the first overflow page. Another complete overflow page is used for any remaining data, and the rest of it is wasted.

BDB's overflow page allocation behavior means that the file system transfer unit must be carefully selected to avoid performance conflicts with BDB. For example, with a file system transfer unit of 4,096 bytes and the default BDB page size of 16,384 bytes, only 4,122 bytes on each overflow page are used (4,096 for the data, and 26 bytes for BDB's internal meta-data), wasting the remaining $\frac{3}{4}$ of the page. This not only wastes space, but hurts performance because useless data needs to be sent to and from the disk. With a database page size of 4,096 and an equal transfer size, 26 bytes of meta-data are stored on the first overflow page and only 4,070 bytes of actual file-system data can be stored. On the second overflow page only the remaining 26 bytes of file-system data are stored—wasting nearly half of the space. One possible solution is to use a non-standard transfer unit of 4,070 for our file system. Although, well-behaved applications should execute the `fstat` system call to find the optimal transfer unit, poorly-behaved applications do not and memory-mapped accesses inherently require standard page-sized access. The solution we have chosen is to use 64KB database pages, which allowed several full 4,096 byte transfer units to fit within the tree nodes (without the need for overflow pages).

We have also found that using database cursors is essential for good sequential read performance. Simply iterating through the Data database using the GET primitive without cursors can be twice as slow as sequentially reading the file with a cursor. Therefore, whenever possible we use cursor reads with the more efficient `DB_NEXT` flag instead of simple GET operations. We do not use write cursors as they are incompatible with transactions, and require locking an entire database environment.

*The Orphan Database.* Files that have been unlinked, but are still open, are stored in the Orphan database. The Orphan database is identical to the Path database, except that instead of storing the name, only the file's unique identifier is stored. In case of a system crash, we can quickly locate and remove all such orphaned files using a database cursor during the next mount.

*Path-local and Data-local Meta-data.* The `stat` system call returns vital information about a file, such as its size, owner, and access permissions. The performance of `stat` is quite important, as it constitutes a large portion of many workloads. Ellard's traces of NFS-mounted home directories show 24.6–72.4% of all calls were GETATTR and ACCESS, which both require `stat` information [Ellard and Seltzer 2003]. Because each file has a single set of attributes, the file's unique identifier determines the `stat` information even if the file has multiple pathnames. This means that the `stat` attributes are a *functional dependency* of the unique identifier. To avoid *logical redundancy*, or having the same data stored in two different places, and its associated pitfalls in a traditional SQL database, the `stat` information should be stored in a database with the unique identifier as the key [Lewis et al. 2002]. In our schema, logical redundancy would introduce *update anomalies* in which one copy of the data could be updated, but the other might not. However, if `stat` information could be stored in the Path database, then performance would be improved because `stat` would require only one database access.

To solve this problem, we take advantage of the flexibility provided by BDB's key-value pair model to develop a more dynamic schema. Meta-data is divided into two classes: (1) *path-local* meta-data and (2) *data-local* meta-data. Path-local meta-data (PLMD) includes all meta-data that is specific to one path of a file. Data-local meta-data (DLMD) includes all meta-data that may refer to more than one path. For example, a newly created file's `stat` information is stored as PLMD, because there is no other path name that references this `stat` information. However, if a hard link to the file is created, then the PLMD is promoted to DLMD, as both names could be used to reference the same underlying file. If one of the links is removed, then the DLMD could be demoted to PLMD. Dividing meta-data into path-local and data-local components allows our schema to avoid the pitfalls associated with logical redundancy. Yet when the data has no logical redundancy, the `stat` information is stored right with the pathname to improve performance.

## 2.3 Internal File System Transactions

It is essential that each operation in an ACID file system be protected by a transaction. This is true even when the application that is executing that operation is not concerned with ACID semantics, because other applications must access a single consistent view of the database to ensure the isolation property. Also, to ensure that the file system is consistent, certain integrity constraints must be maintained.

We define our file system to be consistent, if and only if it meets the following seven integrity constraints:

UNIQID Each file identifier is unique.

REFCOUNT Each file's link reference count is equal to the number of path names that reference it.

NOORPHANEDFILES Each DLMD data block has a positive link or open instance count. Files with a link count of zero must exist in the Orphan database.

NOORPHANEDBLOCKS Each data page has an associated DLMD block.

HARDLINKUSESDLMD If and only if a file has a link reference count greater than one, then it uses DLMD.

PAGESMATCHSIZE A file has no data pages with an index greater than or equal to $\lceil \frac{FileSize}{TransferUnit} \rceil$.

LASTPAGEMATCHESSIZE If there is page at index $\lfloor \frac{FileSize}{TransferUnit} \rfloor$, then it is no larger than $FileSize \bmod TransferUnit$ bytes.

Each of these integrity constraints is equivalent to a similar invariant in a standard file system and is also equivalent to common integrity constraints enforced by a database system. For example, REFCOUNT is equivalent to a foreign key constraint, and standard file systems verify the same when performing a `fsck`. In traditional file systems, constraints similar to PAGESMATCHSIZE and LASTPAGE-MATCHESSIZE are checked by `fsck` to ensure that no orphaned blocks exist, and that stale data does not reappear, respectively.

Our file system does not require a `fsck`, nor does it explicitly enforce the integrity constraints. Instead, each file system operation is designed to transition from one consistent file system state to another consistent file system state. Because each file system operation is surrounded by a transaction, it is atomically applied or it has no effect. Therefore, our file system is always consistent (because it meets the required integrity constraints). This strategy is different from *enforcement*, in that enforcement would require validating the constraints before committing every transaction. To recover the file system after a crash, it is enough to open the database with BDB's `DB_RECOVERY` flag, which replays the database log, and to remove any orphaned files (we efficiently locate these files using the Orphan database). BDB's internal support for recovery obviates the need for us to take complicated recovery steps in our file system code.

## 2.4 Transactional Memory

One major difficulty with any system that supports transactional semantics is how to deal with an abort operation. Transactional systems should be able to rewind to the state they were at just before the transaction began. This is part of supporting atomic behavior: the effects of a sequence of operations are realized if and only if the transaction containing that sequence is committed. If a transaction is aborted, the operations that were already performed must be reversed so that the state returns to how it was just before the transaction had begun. Of course, this is not restricted to the file system data: caches and other book-keeping memory regions that describe the state of the file system also need to be reversible in this manner (e.g., the process's open file table).

Through the use of BDB's support for application-specific recovery, we built a recoverable virtual memory (RVM) library. Our library supports rolling back allocation, deallocation, and writes to a recoverable region. Because one of our requirements was to allow applications to use nested transactions, our RVM library supports nested transactions. By allocating memory regions related to the file system state with our recoverable memory routines, we can easily rewind our state to the proper one upon abort. Our library internally uses `mmap`, `mprotect`, and signal handlers to protect memory regions transparently.

After catching the page faults, we log the memory's content. This allows us to access recoverable memory transparently using traditional memory references, without the need for error-prone explicit logging functions. This is especially important if the library is to be used to retrofit transactional semantics onto existing applications or infrastructure. It is relatively easy to locate all of the points where data structures are allocated and deallocated, whereas locating each access to a data structure can be very difficult.

## 2.5 Transactions API for Applications

Legacy applications need no changes to enjoy the benefits of a consistent file system, which uses transactions for each individual operation (as applications do today with a journaling file system). However, some applications require more stringent atomicity, consistency, isolation, and durability properties. For example, a mail server must append large messages to the end of a mailbox, and a password update system must consistently update `/etc/passwd` and `/etc/shadow` together. Importantly, both legacy and enhanced applications can coexist and use the same data—without the need to access a data store using a specialized interface.

For these types of applications, our file system exports a transactions API to user applications. Our primary design goal for the API was to avoid any changes to existing system calls, which means that we could not add a transaction argument to each call. To begin a transaction, an application issues a new system call that associates a *current transaction* with the process (or thread in multi-threaded applications). Each file system operation after that point is protected by the current transaction. The application can then commit or abort the transaction, with the expected semantics: an aborted transaction has no effect on the file system, and a committed transaction is safely written to stable storage. Aborting a transaction can greatly simplify error handling code, but developers still must take care not to persistently change state during an aborted transaction (e.g., internal application data structures). One simple way to ensure this property is to exit after an abort (many programs already exit on unexpected failures). A better option is to use our RVM facilities to rewind data structures transparently. We believe that one reason many applications are structured such that error handling consists of shutting down the current process or thread is that ad-hoc error recovery is so difficult, hard to debug, and error-prone that fault-tolerant applications, despite their benefits, are often impractical to develop on current systems. We believe that if transactional semantics for the file system and data structures were provided, then programmers may structure their programs to be more robust in the face of failures rather than coding their programs to exit upon failure.

Using BDB's support for nested transactions, each of the file system's internal transactions is started as a child of the current transaction. This simplifies error handling in the file system, because a transaction for a failed system call can just be aborted. If the child transaction is committed, then it is committed to stable storage only if the parent transaction is committed as well. If a child transaction is aborted, then its effects are undone, but the parent transaction can continue. Our design makes use of this, by wrapping each individual system call in a transaction. In this way, our file system can abort transactions, even if the application is wrapping a set of system calls into a transaction. This functionality is also exposed to user applications. If a process already has a current transaction, and a new transaction is created, then the new transaction is created as a child of the existing transaction, thus creating a stack of nested transactions associated with the process.

We employ a simple shared-memory like API to allow processes to share transactions, and we support multiple concurrent transactions without changing the existing system call API. When a transaction begins, it is assigned a unique identifier that the process can then use to manipulate the transaction. A process with sufficient permissions can set its current transaction by attaching to the unique identifier. In this way, two processes can share the same transaction. Similarly, a process can detach from its current transaction, so that future operations are not transaction protected. If all processes have detached from a transaction, then it is automatically aborted (this policy ensures that no transaction-protected data reaches the file system if it was not explicitly committed). If a process temporarily wants to stop using a transaction, but not abort it, then it may suspend the transaction (e.g., to temporarily switch between transactions). The suspend and detach primitives allow processes to switch between transactions without adding system call arguments. For example, a network server may concurrently service many separate clients. Each client's data should be protected by separate transactions. On exit, all uncommitted transactions are automatically detached.

Transactions can be automatically inserted into an existing application's system call stream using pre-defined *profiles*. For example, a profile can protect an entire application by inserting a begin-transaction call on `exec`, and a commit-transaction call on `exit`. Another profile could use file sessions to insert transactions [Santry et al. 1999]: on the first `open` system call, a transaction is begun; on each subsequent successful open, a counter is incremented; and decremented on close. When the counter reaches zero, then the transaction is committed. Other transaction profiles can be designed and developed, either for a general class of applications or even for the behavior of a specific application.

## 3. IMPLEMENTATION

We developed a prototype ACID file system on Linux, called *Amino*. The key implementation question for our file system is how to intercept calls and direct them to the database transparently. We evaluated six techniques with respect to the following two criteria:

—Legacy applications should not be modified. In the best case, unmodified binaries can run without recompiling or relinking. We also considered techniques in which the application must be recompiled or relinked, but its source code is unmodified.

—The interception technique must not insert caches between the application's system calls and the database. This is because any caches that are not managed by the database suffer from two problems. First, if a transaction that spans multiple operations is aborted, then the cache becomes stale. Second, if the caches are accessed without consulting the database, then the isolation property is violated.

Finally, we considered the implementation effort and attempted to minimize changes to existing infrastructure. We considered five choices.

*In-kernel file system.* The most direct approach would be to write a standard in-kernel file system, which do not require relinking of binaries and into the existing kernel architecture. They also have the advantage of running in kernel mode, so they can minimize data copies and context switches.

In-kernel file systems, however, have two key disadvantages. The first is that standard in-kernel file systems are intimately tied together with caches. This means that substantial code changes would be required to ensure coherency between the internal database caches and the external VFS caches. The second disadvantage is that all of the database code would need to be ported to the kernel, and then execute within the kernel address space. Although this is not an insurmountable problem, it would introduce a code base into the kernel that is ten times larger than most existing file systems.

*FUSE.* FUSE or Filesystem in Userspace is a hybrid user-kernel approach [Szeredi 2005]. Like a standard kernel-level file system, no application modifications are required. A standard kernel file system is used to interface with the VFS, but VFS calls are sent to a user-space demon via a device. The user-space demon executes the call and returns the data and status codes to the kernel-level file system, which in turn passes them on to the user. This means that the database code need not run within the kernel, eliminating one concern about developing an in-kernel file system. Unfortunately, this approach still suffers from the same caching problems as a standard kernel level file system, in that cached accesses do not consult the DBMS. As FUSE file systems run outside of the kernel, and have less control over the VFS than a standard file system, these problems would be more difficult to solve than with a standard kernel-level file system.

*User-level NFS server toolkits.* A user-level NFS server toolkit, like the SFS-toolkit [Maziéres 2001], has many of the same advantages and disadvantages as FUSE: applications need not be modified and the database can run in user level, but the kernel caches information inside of the NFS client, thereby violating the isolation property and creating coherence problems with the database caches. Additionally, user-level NFS servers require additional data copies through the network stack, as well as context switches.

*Library Modifications.* Another option is to run our file system directly in the address space of user processes and intercept system-call wrappers using a modified C library [Korn and Krell 1990] or the `LD_PRELOAD` runtime-linker mechanism. This approach has three main advantages. First, as file-system calls are intercepted at the highest possible level, there are no cache coherency or isolation issues to contend with. Second, the database does not need to run in the kernel. Third, data copies between the process and the kernel are not required. There are, however, four disadvantages. First, system calls that do not use the library wrappers are not intercepted, so not all code would work with this approach. Second, statically linked binaries must be recompiled to use the `LD_PRELOAD` or C library approach (for the C library approach all binaries must be recompiled). Third, with the `LD_PRELOAD` approach the C library itself continues to use the existing calls, so every call of interest must be intercepted (e.g., `fprintf` must be intercepted because applications use it to write to the file system). Fourth, a modified C library introduces circular dependencies with the BDB library. For example, BDB needs the `fwrite` library call, but that call in turn would depend on BDB.

*`ptrace`.* The final option we considered was using the process-tracing facility, `ptrace` [Haardt and Coleman 1999]. The process-tracing facility allows a *monitor* to intercept and modify system calls and signals. From the perspective of the application, the monitor is equivalent to the OS, so no application modifications are required. As shown in Figure 1, the monitor runs in user-level, so BDB does not need to execute within the kernel. Unlike the library approach, a single instance of the monitor can handle multiple processes, so it is simpler to share data, caches, and other resources.

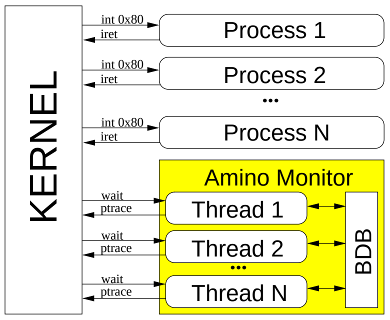


Fig. 1. The Amino monitor can trace an arbitrary number of processes. At system call entry, the kernel signals the monitor via the `wait` system call, and Amino manipulates the monitored processes' state with `ptrace` primitives.

The major disadvantage of the `ptrace` approach is that performance may suffer for system-call–intensive programs, as more context switches are required for each system call. However, we felt that ease of development and cache consistency outweighed performance concerns.

In Section 3.1 we describe the process tracing primitives. In Section 3.2 we describe the structure of the Amino monitor. In Section 3.3 we describe Amino's process control blocks, and in Section 3.4 we describe Amino's path resolution and mount framework. In Section 3.5 we discuss address space issues.

## 3.1 Process Tracing Primitives

The `ptrace` framework provides three primitives to establish tracing: the monitor can issue `PTRACE_ATTACH` to begin tracing a currently running process, the monitor can issue `PTRACE_DETACH` to stop tracing, and one of the monitor's children can issue `PTRACE_TRACEME` to be traced by the monitor. Our monitor begins by forking a new child, issuing `PTRACE_TRACEME`, and then executing the to-be-traced executable. From this point onward, the monitor is notified via the `wait` system call whenever the child needs attention.

The monitor uses three primitives to control the execution of the child process. (1) `PTRACE_SYSCALL` continues execution until the next entry or exit from a system call. If the child is in user-mode, then the child process is stopped before the kernel enters the system call handler, so that the monitor can change the arguments, or even the system call to be executed. If the child process is in the midst of executing a system call, then the kernel completes the routine and the monitor can examine and change any return values. (2) `PTRACE_CONT` continues execution until the child receives a signal. (3) `PTRACE_SINGLESTEP` continues execution until the next instruction.

When the child is in the stopped state, the monitor uses four primitives to observe and manipulate the child process: `PTRACE_GETREGS`, `PTRACE_SETREGS`, `PTRACE_PEEKDATA`, and `PTRACE_POKEDATA`.

`PTRACE_GETREGS` retrieves the values of the registers saved during a context switch from the kernel's process control block. On the Intel 80x86 architecture, the `eip` register contains the program counter, the `eax` register indicates what system call the process wants to execute, and the remaining general purpose registers contain the system call's arguments. Our current implementation is tied to the 80x86 architecture, because it references these registers, but it would not be difficult to add support for other architectures as the ABI is similar on all Linux platforms. In our prototype, only 451 out of 12,187 lines of code reference 80x86 specific registers.

The monitor can also manipulate the registers with the `PTRACE_SETREGS` primitive. Before a system call, the call to execute can be changed by setting `eax`, and the arguments can be changed by updating the corresponding registers. After a system call is executed, the return value can be set by updating the value of `eax`. At any point in time, the execution flow of the program can be changed by modifying `eip`. This is required when a single system call must be implemented in terms of several other system calls. Finally, there are two primitives to examine and update a word in the child process's memory: `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`. These primitives are used when the system call takes pointer arguments (e.g., file names are passed as strings, and `stat` fills in a user-supplied buffer).

Figure 2 shows an example of how the Amino monitor handles a `read` system call destined for the database file system on behalf of a user process. There are ten steps involved in this call:

(1) The user process issues a system call using `int 0x80`. The system call to execute is stored in `eax`.
(2) The wait system call in the monitor returns the process ID of the user process.
(3) The monitor issues a `PTRACE_GETREGS` call to retrieve the value of `eax`. Based on `eax` and the call's arguments, Amino determines whether this call is destined for the database. If the call is not destined for the database, then Amino allows the process to continue with no further intervention.
(4) If this call is destined for the database, then Amino changes the registers to prevent the kernel from handling the call. In the case of `read`, Amino sets `eax` to $-1$, thus the kernel essentially ignores the call because no handler is

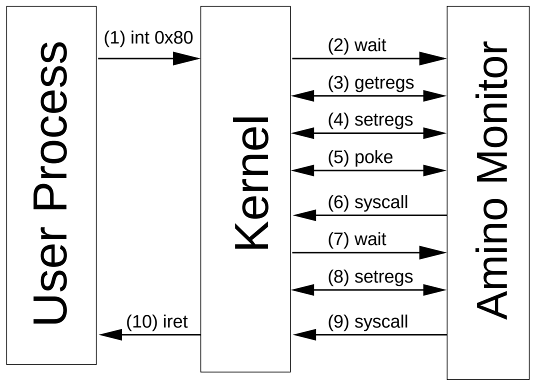


Fig. 2. `ptrace` primitives used to handle a `read` system call. Arrows indicate control transfer. Double arrows indicate that the function was called and returned immediately.

associated with –1.

(5) Amino performs the database `read` operation, and uses the `PTRACE_POKEDATA` primitive to write the returned data into the user process's address space (we also have an optimized mechanism described in Section 3.5).

(6) Amino instructs the kernel to continue execution until the end of the call and calls `wait` (in this case the call returns immediately without performing any service, because `eax` was set to –1 in step 5).

(7) The kernel executes the system call, and returns from `wait`.

(8) Amino uses the `PTRACE_SETREGS` primitive to store the return value of the previously executed read in `eax`.

(9) Amino uses the `PTRACE_SYSCALL` primitive to allow the user process to continue executing.

(10) The kernel issues an `iret` instruction to return control to the user process. The user process reads the return value from `eax`, and it is as if the system call were serviced by the kernel.

## 3.2 Amino Structure

The Amino monitor begins by forking a child process to trace. After the fork, the child executes the program to be monitored. All of the process's descendants are also monitored, and each monitored process is assigned a state. The two most common states are InUser and InCall, which indicate that the process is executing user-level code or that it is executing a system call, respectively. To service requests, Amino calls the `wait` system call. When a process requires attention, usually because it is entering or exiting a system call, the kernel returns its process ID as the result of the `wait` system call (`wait` also returns when a signal is delivered or a process exits).

After returning from `wait`, Amino retrieves the current process's state and performs an appropriate action. There are currently 19 states (including InUser and InCall). Most of the states indicate that the user process is in the midst of a specific call, for example `clone`, `exec`, `chdir`, or `dup`. One of the most important states is InForceRet, which indicates that the return value of the presently executing system call should be overridden by a given value. This state is used by most database calls to pass back status information. In the example in Section 3.1, the return value of the `read` is determined in step 5, but is not yet returned. When the return value is determined in step 5, the monitor sets the state to InForceRet. After step 7, Amino looks up the state and because it is InForceRet, Amino sets the value of `eax` to the proper return value. Two other states of note are RedoCall which indicates that the current system call should be repeated, and RestoreRegs which indicates that the process's registers should be set to their original values. RedoCall allows us to insert a new system call into the stream (e.g., to create shared memory regions), and RestoreRegs is used when we need to change system call arguments (e.g., when rewriting file names).

## 3.3 Process Control Blocks

The monitor maintains each process's state in a private *process control block* (PCB). The monitor's PCB is independent of the OS PCB, and contains the process ID to use as a search key, a copy of the process's registers, the current state of the process (e.g, InForceRet), and all state-specific information (e.g., the return value to be passed back to the application). Encapsulating all of this information in a single structure allows the monitor to handle concurrent processes.

Like an OS PCB, the monitor's PCB contains an open-file table and present working directory (PWD). The open-file table is a simple array with a slot for each possible file descriptor. If a given file descriptor is connected to an Amino file, then its slot contains a pointer to a structure describing the file; otherwise it is empty (`NULL`). If a system call uses a file descriptor as an argument, it is looked up in the open-file table. If the file descriptor's slot is empty, then the system call proceeds with no further intervention. Otherwise, Amino extracts the schema data (i.e., the database and environment handles) and the unique file identifier from the open-file table and directs the call to BDB.

Amino cannot arbitrarily assign file descriptors to the user-level process, because the kernel would not know that a given file descriptor is in use. To handle this situation, Amino uses *shadow descriptors*. When opening a file in the database, Amino changes the path name to "`/`" before letting the system call proceed. The resulting file descriptor (in the child process) is used as a place holder, and no system calls are issued against it. The kernel does not assign the resulting descriptor to any other file, so Amino can correctly identify the calls that it handles.

## 3.4 Mount Subsystem

The Amino monitor maintains a mount table to associate pathnames with database schemas. On startup, an Amino configuration file provides a list of paths to manage, and for each path, the mount type and data (the configuration file is essentially equivalent to `/etc/fstab`). Currently, Amino supports BDB mounts that take the database pathname as an argument. When Amino encounters a system call that references one of these paths, Amino passes it to the appropriate routine.

Pathnames passed to system calls can be rather complex. If they are relative path names, then they depend on the process's context. Any path can use the "`..`" operator to move one level up the directory tree. We store paths as stacks, with the root path represented as an empty stack, and a path such as `/usr/local/bin` is represented by a stack containing `usr`, `local`, and `bin`. If a path is managed by Amino, then it is a child of one of the mount-table entries described in the configuration file. To rapidly determine if one path is a child of another, the path structure also contains a depth, and a length for each path component.

Each PCB contains a path stack for the PWD. When a `chdir` or `fchdir` system call is issued, the new PWD is stored as a candidate. If the system call is successful, then the candidate becomes the PWD. The mount table also uses a path stack to identify the path for each mount.

To resolve a path that is passed to a system call, first the process's PWD is copied to a new stack. If the path begins with a "`/`," then the stack is emptied. Each subsequent component is pushed onto the stack. If the component is "`..`," then an element is popped off the stack (unless of course the stack is already empty). After converting the string pathname into a path stack, the monitor searches the mount table for any mount that contains this path. The path structure is optimized for this purpose: if the path has a lower depth than the mount, then it cannot be a child; and the length is stored with each component so the component names only need to be compared if they have equal length. If one is found, then the path components after the root of the mount are extracted (e.g., if the path is `/usr/local/src/amino` and the mount is rooted at `/usr/local`, then `src/amino` is extracted). The mount private data containing the database handles and the extracted path are then passed to the BDB call. If the path name is not contained in a mount, then Amino allows the system call to go through without any changes.

## 3.5 Address Spaces

There are two distinct address spaces involved in executing the Amino monitor: (1) the address space of the monitor and (2) the address space of the user process. The `ptrace` primitives to access the user process's address space are rather limited—they can only examine or change one word at a time. Thankfully, Linux provides a more powerful interface to it through the `/proc` file system. A process with permission to `ptrace` another process may read from the traced process's memory using the `/proc/pid/mem` file, where `pid` is the PID of the traced process. This allows the transfer of up to a page (1,024 words on the 80x86) in a single system call. Linux also has support to write to `/proc/pid/mem`, but it is disabled by default. For our prototype, we have enabled a writable `/proc/pid/mem` to allow bi-directional bulk transfers. Finally, we also allow regions of the child's address space to be memory-mapped into the monitor, thus providing a zero-copy transfer method. If the `/proc/pid/mem` interface is not available for reading or writing, then Amino falls back to `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`.

All system call arguments must be in the user processes' address space. For example, the first argument to `open` is a pointer to a string. If Amino needs to update these values, then it must manipulate the child's address space. It is not always possible to manipulate the file name in place, because the new file name may be longer than the existing file name, and the memory segment may be read only. To address this issue, previous `ptrace` monitors have modified either the stack, or the first writable segment. In Amino, we establish a System-V shared-memory region between each user process and the monitor. When the first system call is issued with an argument that needs to be updated, the monitor creates a shared memory region. Next, the monitor inserts a shared-memory attach operation in to the child's system call stream. At this point, Amino writes the new file name into its own address space, and updates the child's registers to point to the shared memory in the child's address space. After the call, the child's original registers are restored. Subsequent arguments can be rewritten by simply updating the local region, and the child's registers. This approach has the advantage of requiring no data copies, and the child's existing memory is not modified, therefore the child's memory does not need to be restored after the call.

## 3.6 `ptrace` Enhancements

The standard `ptrace` interface requires at least six context switches for each system call: (1) the traced process traps into the kernel; (2) the kernel transfers control to the monitor; (3) the monitor transfers control to the kernel; (4) after executing the system call, the kernel transfers control back to the monitor so that the return value can be manipulated; (5) the monitor transfers control back to the kernel; and finally, (6) the kernel transfers control back to the traced process. In reality, more context switches are required as the monitor must retrieve the values of traced process's registers, issue system calls to provide OS-like services, etc.

Clearly, reducing the number of times that the monitor is called improves performance. For most calls the monitor needs to be notified only on entry. If the call is not destined for an Amino file system, the monitor does not need to examine the return value so the call could execute without further intervention by the monitor. If the call will be handled by the Amino file system, the return value could be set and the monitor need not be notified. Unfortunately, these two modes of operations are not possible under the current `ptrace` interface.

We created two new `ptrace` operations: `PTRACE_CHECKEMU` and `PTRACE_SYSSKIP`. The `PTRACE_CHECKEMU` operation is similar to the `PTRACE_SYSEMU` operation that was recently introduced to improve the performance of User Mode Linux [Dike 2000]. The primitive `PTRACE_SYSEMU` allows all of a process's system calls to be emulated, but it is not suitable for the Amino monitor, because we emulate only a subset of the system calls. Our `PTRACE_CHECKEMU` interface allows the monitor to determine whether emulation is required after examining the registers. The UML developers agree that our more general `PTRACE_CHECKEMU` interface is an improvement over the existing `PTRACE_SYSEMU` [Giarrusso 2005]. The corollary to `PTRACE_CHECKEMU` is `PTRACE_SYSSKIP`. When the Amino monitor does not implement a call, then it issues `PTRACE_SYSSKIP` instead of `PTRACE_SYSCALL` to bypass notification of this system calls return value and go directly to the start of the next system call. Together, these primitives reduced traps into the monitor by 30.8% during an OpenSSH compile.

Finally, there are also many non-file-system system calls that the monitor need not intercept at all (e.g., `time` or `getpid`). To reduce the number of extraneous calls into the monitor, we added an optional bitmap of system calls to the task structure. By using a new `ptrace` primitive, `PTRACE_SELECT`, the monitor selects precisely the set of calls that need to be traced. This method reduced the number of traps to the monitor by an additional 12.8% during an OpenSSH compilation. Overall, these techniques reduced the number of traps to the monitor by 43.7%.

These three improvements can benefit a wide variety of `ptrace` monitors. For example, the `PTRACE_CHECKEMU` grew out of work for User Mode Linux, but provides a more flexible interface that can be used by a monitor that emulates a subset of system calls. Many security-oriented monitors need to examine only which system calls are being executed and their arguments, but not their return value. For these types of monitors, `PTRACE_SYSSKIP` would greatly improve their performance. The `strace` program provides support for filtering the set of system calls to display (e.g., file system, process, or IPC related calls), but this filtering is done in user-space. By using `PTRACE_SELECT`, `strace` could have the kernel perform this filtering.

## 4. EVALUATION

We evaluated the performance of our system by running several general-purpose workloads and micro-benchmarks. We chose three general-purpose benchmarks: the Postmark benchmark [Katcher 1997] (Section 4.1), an OpenSSH compile (Section 4.2), and a Sendmail benchmark (Section 4.3). We also ran two sets of micro-benchmarks: meta-data–intensive micro-benchmarks (Section 4.4) and data-intensive micro-benchmarks (Section 4.5).

For all our benchmarks we used a dual 2.8Ghz Xeon machine running Fedora Core 4 with all updates as of February 20, 2006. All experiments were located on a dedicated 147GB 10,000RPM Fujitsu U320 SCSI disk (model MAP3147NC). The benchmark scripts, system utilities, and results were stored on an identical disk. We compared Ext3 to Amino using BDB databases stored on Ext2. We used Ext2 as the underlying file system for Amino, because BDB provides ACID semantics even without a journaling file system. We chose to use Ext3 as a basis for comparison, because it provides a limited subset of the ACID properties, whereas Ext2 does not. To ensure a cold cache, we remounted the file systems between each iteration of a benchmark. For all tests, we computed the 95% confidence intervals for the measured quantity the Student-$t$ distribution. Unless otherwise noted, the half-widths of the intervals were less than 5% of the mean.

We used the following nine configurations for our tests:

VANILLA The benchmark is run on Ext3.

VANSYNC The benchmark is run on Ext3, but the file system is mounted with the `sync` mount option to provide durability.

STRACE The benchmark is run on Ext3, but is monitored by `strace -cf`. This configuration shows the overhead of the `ptrace` facilities by counting system calls, but does not modify any calls or produce any output during execution.

AMINOTRACE The benchmark is run on Ext3, but is monitored by the Amino monitor. This configuration shows the overhead of `ptrace` and our path-name resolution infrastructure.

AMINONULL The benchmark is run on Ext3, but all operations are emulated by the Amino monitor. This measures the overhead of `ptrace`, our path-name resolution and the operations required to implement a file system using `ptrace`.

AMINOACI The benchmark is run through the Amino monitor with a BDB database stored on an Ext2 file system. BDB is configured to provide atomicity, consistency, and isolation, but not durability.

AMINOACID This configuration is the same as AMINOACI, but durability is also provided because BDB flushes the log to disk on each commit.

AMINOTXN This configuration is the same as AMINOACI, but the benchmark is modified to insert calls to begin and commit Amino transactions. This measures the overhead of adding transactional code to the applications and the transactional code in the database, without incurring durable writes.

AMINODTXN This configuration is the same as AMINOTXN, but with durability enabled. This configuration improves performance over AMINOACID, because data needs to be flushed to disk only after the transaction is committed, rather than after every system call.

As you can see, we often use two similar configurations one with and another without durability to separate the cost of synchronous writes from the cost of other functionality.

## 4.1 Postmark

Postmark 1.5 is an I/O-intensive benchmark that stresses the file system by performing a series of file system operations such as directory look ups, creations, and deletions on small files [Katcher 1997]. Postmark is typically configured by specifying a number of initial files, and a fixed number of *transactions* (this is Postmark's term for an operation, and is distinct from Amino transactions) to run. Postmark then creates the initial pool of files, performs the fixed number of transactions, and removes any leftover files. Unfortunately, running a fixed number of transactions makes it difficult to compare two configurations that have large differences in the amount of time they take to run (e.g., a durable vs. non-durable configuration), because a configuration large enough to stress the non-durable configuration takes too long on the durable configuration, and vice versa. To solve this problem, we modified Postmark such that it still takes an initial number of files as a parameter, and in addition a time limit. Our modified Postmark creates the initial pool of files, performs transactions for the specified time, and then removes any leftover files. The metric of interest in our modified Postmark is the number of transactions per second. We also measured the CPU utilization of Postmark and our monitor.

The first Postmark configuration we chose is to create 2,500 files ranging from 512 bytes to 10KB, and perform transactions for three minutes. We used the `read` and `write` system calls (as opposed to Unix buffered I/O), and a transfer size of 4,096 bytes for both Ext3 and Amino.

The Postmark results are shown in Figure 3. The VANILLA configuration performed 1,591 transactions/second and had a CPU utilization of 40.3%. The VANSYNC configuration synchronously writes data and meta-data to disk to provide durability. The VANSYNC configuration was slower than VANILLA by a factor of 100.2, due to additional synchronous disk writes. The STRACE and AMINONULL perform 37.7% and 23.7% fewer transactions than VANILLA, respectively. This shows the overhead of the process-tracing facilities. Overall, Amino's CPU utilization is slightly less than `strace`. The Amino monitor uses 53% less system time than `strace` (the monitor in the STRACE configuration), because it accesses all process registers using a single system call instead of one system call for each register. However, Amino uses 60.0% more user time than `strace` because it has to resolve each path name. The AMINONULL configuration performs 30.1% fewer transactions than VANILLA, but is 8.3% worse than AMINOTRACE. This shows the added overhead of performing the operations within the monitor.

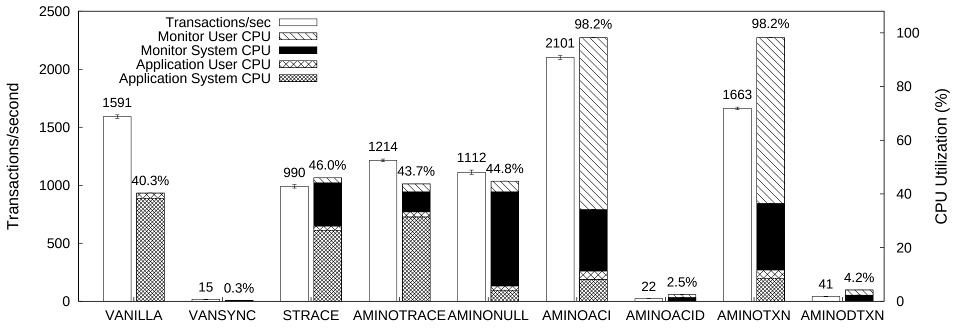


Fig. 3. Postmark. The left axis is transactions/second and the right axis is CPU utilization.

AMINOACI provides atomicity, consistency, and isolation using BDB. It performs 31.9% more transactions than VANILLA, but at a cost of increased CPU utilization—98.2%. Two factors increased CPU utilization. First, Amino requires more data copies than VANILLA or AMINONULL, because it copies data from the kernel into the BDB cache and then from the BDB cache into the user-space process. Second, BDB has more complex data structures and thus uses more CPU than Ext3. However, this is offset by more efficient I/O. AMINOACI wrote 86.8% fewer sectors than VANILLA and these I/O operations took 99.8% less time. This shows that a file system built on a database can provide atomicity, consistency and isolation with good performance, even for I/O-intensive applications, because we can quickly access files and directories with our schema and BDB efficiently writes data to the log.

AMINOACID provides all four ACID properties: atomicity, consistency, isolation, and durability. To provide durability, the database log must be synchronously written to disk after each transaction. This leads to an expected overhead of a factor of 72.3 over VANILLA, but AMINOACID provides semantics closer to VANSYNC. When compared to VANSYNC, AMINOACID improves performance by 46%.

In AMINOACI, each individual system is protected by a transaction. In AMINOTXN, we modified Postmark to begin and end Amino transactions before each high-level operation (i.e., create, remove, read, or write a file) that Postmark refers to as a transaction. In this configuration, Amino provides application-level consistency, so there are never any partially written files. This decreases the transactions per second that Amino can sustain by 20.8%, because more CPU is required to manage the transactions and the CPU was already saturated at 98.2%. The final configuration we used was AMINODTXN, which combines the consistency properties of AMINOTXN with the durability of AMINOACID. The AMINODTXN configuration is 81.6% faster than the AMINOACID configuration and 2.6 times faster than the VANSYNC configuration. Moreover, the AMINODTXN configuration provides application-level consistency, whereas the VANSYNC and AMINOACI configurations do not.

In sum, we show that Amino can provide performance as good as Ext3, but has a higher CPU utilization. Moreover, with only small modifications, applications can improve durable performance and benefit from full ACID semantics.

*Alternate Configurations.* We also ran two alternate Postmark configurations that are slight modifications of the first configuration. The first alternative configuration has ten times larger files: from 5,120–102,400 bytes. The second configuration has ten times more files: we increased the number of initial files to 25,000; for this configuration we introduced 250 subdirectories, so that Ext3 would not have to perform linear scans over 25,000 files for some operations.

*Larger Files.* The results for VANILLA, STRACE, AMINOTRACE, and AMINONULL were similar to the original configuration. VANILLA performed 495 transactions per second, and STRACE, AMINOTRACE, and AMINONULL performed 44.6%, 26.1%, and 35.1% fewer transactions per second, respectively. The AMINOACI and VANILLA configurations performed a statistically indistinguishable number of transactions. However, the AMINOACI configuration used significantly more CPU: 93% vs. 27.6%. This shows that Amino loses some of its performance advantage for this benchmark as the benchmark shifts from a more meta-data–intensive benchmark to a more data-intensive benchmark. The AMINOTXN configuration performed 18.6% fewer transactions than the AMINOACI configuration.

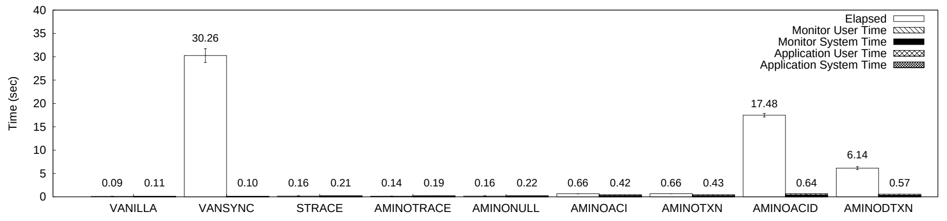
The VANSYNC, AMINOACID, and AMINODTXN configurations performed 147.6, 91.2, and 21.9 times fewer transactions than VANILLA, respectively. These results are similar to the original Postmark configuration: AMINOACID was 61.8% better than VANSYNC and AMINODTXN was 4.0 times better than AMINOACID. The major difference is that AMINODTXN performed 4.0 times better than AMINOACID rather than 81.6% better. The reason that AMINODTXN outperformed AMINOACID more than in the previous configuration is that more individual `write` operations were required for each transaction, so a larger number of writes could be coalesced into a single synchronous log write.

*More Files.* VANILLA performed 346 transactions per seconds, and was outperformed by AMINOACI by a factor of 4.3. The reason is that VANILLA spreads the files through many cylinder groups, but AMINOACI stores them together in a balanced tree, improving locality thereby reducing wait time. However, this comes at a cost of increased CPU utilization, AMINOACI used 94.1% of the CPU and VANILLA only used 9.3%. The STRACE, AMINOTRACE, and AMINONULL configurations performed as expected: 32.3%, 26.9%, and 32.0% slower than VANILLA, respectively.

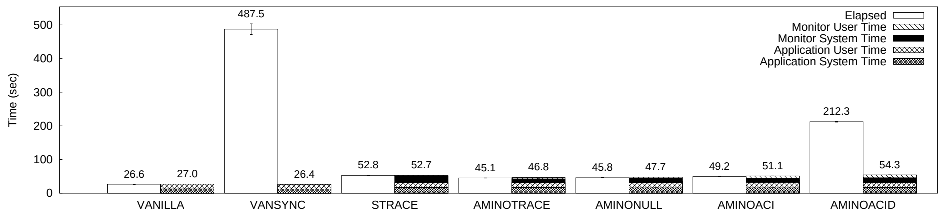
As expected, the synchronous configurations were slower: VANSYNC and AMINOACID had a 24.7 and 17.5 times slowdown, respectively. Again, AMINODTXN was the most efficient synchronous configuration with only a 10.2 times slowdown over VANILLA. This demonstrates that explicitly marking transactions, combined with BDB's highly-tuned logging infrastructure, improves durable performance.

## 4.2 OpenSSH Compile

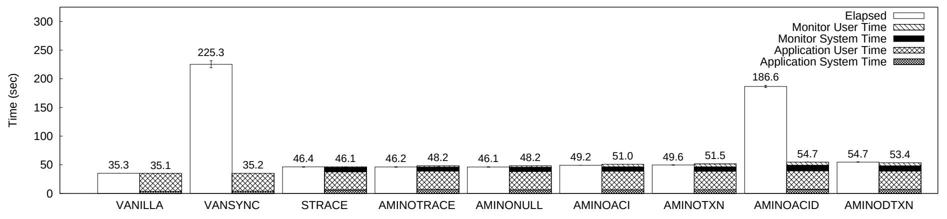
To simulate a more CPU-intensive typical user workload, we adapted the SSH build workload [Seltzer et al. 2000], but used OpenSSH 4.2p1 as it builds cleanly on our systems whereas SSH 1.2.26 did not. This workload stresses the Amino monitor, as it requires significant amounts of additional CPU time in order to intercept system calls. The compile benchmark is divided into three phases: (1) unpack, (2) configuration, and (3) build. We measured the elapsed, system, and user time of each of the phases separately to isolate their different characteristics. In contrast to our previous benchmarks, lower values are better than higher values. In the unpack phase, the package is uncompressed and new files are created by `tar`. In the system-call-intensive configuration phase, the `configure` shell script preforms many small configuration tests, which involve a fair mix of file-system operations. The build phase is more CPU-intensive and builds 157 object files, two libraries, eleven executables, and sixteen man pages. In the unpack and build phase, AMINOTXN and AMINODTXN use transactionally-modified versions of GNU Make and `tar`, which transactionally protect each operation (in Make we define an operation as a build command and in tar each file extraction is an operation). In the other configurations, we use the standard GNU Make and `tar`. In the configuration phase, AMINOTXN is identical to AMINOACI and AMINODTXN is identical to AMINOACID.



(a) Unpack results.



(b) Configuration results.



(c) Build results.

Fig. 4. Each configuration has two bars grouped together. The left bar is for elapsed time; the right bar consists of each of the CPU time components. Configurations with lower CPU and elapsed time perform better than configurations with higher CPU and elapsed time. In some instances, the CPU time components may be larger than the elapsed time, because processes occasionally execute concurrently during compilation.

Figures 4(a), 4(b), and 4(c) show the results of each phase of the OpenSSH compile benchmark. The unpack phase, shown in Figure 4(a), took 0.09 seconds on VANILLA. The STRACE configuration added an overhead of 84.4%, AMINOTRACE had an overhead of 59.9%, and AMINONULL had an overhead of 84.8%. For all three of these Ext3 configurations, the benchmark completed quickly, because no disk writes were performed during program execution due to the buffer cache. The AMINOACI configuration took 0.66 seconds to complete, which is a factor of 7.4 slower than Ext3. The reason that AMINOACI is slower than Ext3 is that the CPU time used increased by 0.31 seconds from 0.11 seconds to 0.42 seconds. The AMINOTXN configuration is similarly 7.5 times slower than VANILLA, because of an increase in CPU time; however, it provides application-level consistency (i.e., no partially written files). The last three configurations we tested were VANSYNC, AMINOACID, and AMINODTXN. The VANSYNC configuration is 340 times slower than VANILLA, because changes are written to the disk synchronously. The AMINOACID configuration provides the same functionality; it is only 194 times slower than VANILLA, because BDB is optimized for durable performance. AMINODTXN is only 69 times slower than VANILLA, but provides application-level consistency and durability.

The second phase of the benchmark, configuration, is shown in Figure 4(b). On VANILLA, this phase took 26.6 seconds. This phase of the benchmark is CPU and system-call intensive, so the STRACE and AMINOTRACE configurations had overheads over VANILLA of 98.2% and 69.6%, respectively. The AMINONULL configuration had an overhead of 72.1%. The AMINOACI configuration has an overhead over VANILLA of 84.7%. When compared with AMINOTRACE, the overhead of AMINOACI is only 9.0%. This demonstrates that our file system is relatively efficient, though the CPU intensive nature of this workload causes the context switches and data-copying induced by the monitoring infrastructure to degrade performance. Of note, our `ptrace` monitoring infrastructure including the file system is faster than STRACE alone. When durability is added, VANSYNC is 18.3 times slower than VANILLA, and AMINOACID is 7.9 times slower than VANILLA. Again, this demonstrates that Amino efficiently provides durable performance.

The build phase, shown in Figure 4(c), took 35.3 seconds on VANILLA. Even though this phase is the most CPU intensive phase of all, it is the least system call intensive. Therefore, the monitoring infrastructure has a lower overhead than in the configuration phase: 31.5% for STRACE and 31.1% for AMINOTRACE. The elapsed times for the STRACE and AMINOTRACE configurations were statistically indistinguishable, but AMINOTRACE used 4.5% more CPU time. The AMINONULL configuration had an overhead of 30.9%, which is statistically indistinguishable from AMINOTRACE. The AMINOACI configuration had an overhead of 39.5%. Most of this was due to a 45.2% increase in CPU time from 35.1 seconds to 51.0 seconds, caused by BDB operations, additional data copying, and context switches. The AMINOTXN configuration had an overhead of 40.8%, which is 0.9% higher than AMINOACI. The additional overhead is caused by a 1% increase in CPU utilization for tracking transactions. The VANSYNC configuration was 6.4 times slower than VANILLA, and AMINOACID was 5.3 times slower than VANILLA. The AMINODTXN configuration performed much better, with an overhead of only 55.1% over VANILLA, just 15.6% more than AMINOACI. This demonstrates that although durability degrades performance, much of the loss can be made up for by inserting explicit transactions.

## 4.3 Sendmail

We ran a Sendmail 8.13.4 server and varied the backing store for the `/var/mail` directory which contains user mailboxes. We used a 2.8Ghz Xeon with 2GB as the client and a 1.7Ghz Pentium 4 with 1GB of RAM as the server. The `/var/mail` directory was stored on a dedicated 7200RPM Maxtor 40GB IDE disk. We did not run Sendmail through our monitor, because it does not access the mail files. Instead, it delegates that task to the local mailer. For the VANILLA configuration, we used the default local mailer. To provide isolation and an approximation of atomicity, the local mailer performs locking and complex checks (e.g., repeatedly calling `stat` to ensure that the file does not change). To ensure that mail is not lost (i.e., provide durability), the local mailer calls `fsync` after writing the message. These checks are unnecessary under Amino, as our file system transparently provides isolation to applications, without the need for explicit locking calls or repeated checks. We wrote a simple mailer replacement that uses an Amino transaction to provide ACID properties for the AMINOTXN configuration. Moreover, thanks to the use of transactions, our delivery agent is six times shorter than the default local mailer (72 lines vs. 441 lines); yet ours provides stronger consistency guarantees. The STRACE and AMINOTRACE configurations monitored the `mail.local` program. The DEVNULL configuration discarded the message.

We developed a Perl script that stress tests the mail server by continuously sending mail using 32 concurrent threads. We created a pool of 100 users to receive the mail, and each message had a randomly selected recipient. The messages sizes were normally distributed with a mean of 5,993 bytes and a standard deviation of 4,166. We chose the size parameters based a 2.5%-trimmed mean of our non-spam email for the past year. The test begins with a 60 second warmup period, in which the test runs without measurement to avoid startup effects. After the warmup, messages are sent for five minutes, and we record the mean achieved rate.

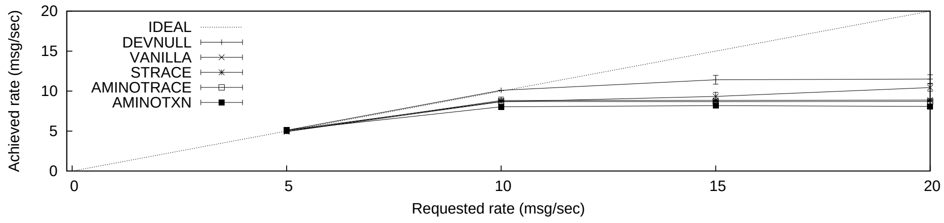


Fig. 5. Local mailer: requested vs. achieved load.

We ran the test for requested rates of 5–20 messages per second (MPS), and plotted the requested rate against the achieved rate in Figure 5. Ideally, the server would process exactly the same number of messages as were requested, but in practice the network, CPU, and disk act as bottlenecks.

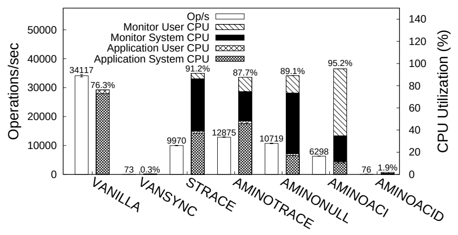
All configurations handled 5 MPS well, achieving the requested rate. The DEVNULL configuration achieved 10.1, 11.4, and 11.5 MPS for request rates of 10, 15, and 20 MPS, respectively. This shows how many messages the machine could handle if the disk was not a bottleneck. The VANILLA configuration achieved 8.7, 9.3, and 10.4 MPS for request rates of 10, 15, and 20 MPS, respectively. This represents a decline of 13.9%, 18.4%, and 9.6% from the DEVNULL configuration, respectively. The STRACE configuration achieved 8.8–8.9 MPS for requested rates of 5–20 MPS, and AMINOTRACE achieved 8.7 MPS for a requested rate of 5–20 MPS. At 15 MPS, this represents an overhead of 5.3% and 7.0% for STRACE and AMINOTRACE, respectively. At 20 MPS, this overhead increases to 16.6% and 15.0%, respectively. AMINOTXN had degraded performance compared to the other configurations. It was only able to handle 8.0–8.2 MPS for a requested rate of 5–20 MPS. Compared to VANILLA, this is a reduction of 7.2% for a requested rate of 10 MPS, 12.4% for a requested rate of 15 MPS, and 22.6% for a requested rate of 15 MPS. Compared to AMINOTRACE, the overheads are between 5.7–7.7%. The AMINOTXN overheads are clearly coming from two sources: (1) `ptrace` monitoring and (2) the Amino file system itself. The Amino performance is poorer than Ext3, because Sendmail uses multiple processes to deliver mail, thus causing increased lock contention to provide the isolation. BDB uses page-level locking for the Path and Data databases, thus falsely limiting concurrency compared to Ext3 (which uses per-file locking). One possibility for improving performance is to investigate alternative schema designs that may yield a higher degree of concurrency (e.g., moving the data-local meta-data to the end of the file would improve append performance, at the possible expense of sequentially reading the file). Even though AMINOTXN is slower, the code is significantly smaller and simpler, which means that fewer bugs and security flaws are possible, and the system is more reliable. Moreover, our local mailer provides improved guarantees. If the Sendmail local mailer exits successfully, then the message has reached stable storage, but if the local mailer does not exit successfully (e.g., due to power failure or an operating system error), then the mailbox can be corrupted. With our local mailer, the mailbox always remains in a consistent state—regardless of whether the mailer exits successfully or not.

## 4.4 Meta-data Micro-benchmarks

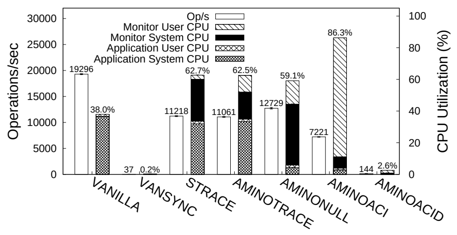
We ran several micro-benchmarks on Amino to evaluate the overheads of primitive file system functions. We broadly classify our micro-benchmarks into metadata (described in this section) and data benchmarks (described in Section 4.5). The meta-data operations we evaluated are `create` (and `mkdir`), `unlink` (and `rmdir`), `stat`, and `readdir`. We chose these meta-data operations because they are a broad cross-section of file system operations, and together with data operations account for the vast majority of operations [Ellard and Seltzer 2003].

To generate metadata operations, we developed a C program that operates on several directories each containing a fixed number of files. We used this method rather than a generic data set (e.g., the source of a package), because when evaluating one specific method we did not want to use directory-reading operations or lookups to determine which files must be operated upon. For all the metadata workloads, we disabled `atime` updates on both in Ext3 and in Amino to isolate the overheads of the metadata operation to be tested.

*Create.* To evaluate the `create` and `mkdir` operations, we used our C program to create 1,000,000 files evenly spread across 5,000 directories (i.e., 200 files per directory). We spread the files among the directories, to avoid unfairly penalizing Ext3 for its linear lookup operation. For the durable configurations, which take significantly longer than the non-durable configurations, we created only 100,000 files evenly spread across 500 directories. configurations. To account for this difference, we normalize the elapsed time to operations per second and CPU utilization.



(a) Creation micro-benchmark.



(b) Deletion micro-benchmark.

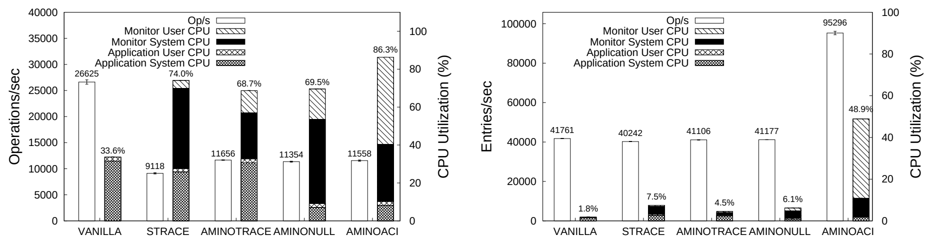
Fig. 6. Creation and deletion micro-benchmark results.

As seen in Figure 6(a), VANILLA created 34,117 files per second. The STRACE configuration created 9,970 files per second, or a reduction of 70.8%. The AMINOTRACE configuration had a slightly lower reduction of 62.3% compared with VANILLA. The AMINONULL configuration had a 68.6% reduction compared with VANILLA. Much of this overhead derives from the context switches required for tracing, as evidenced by STRACE, AMINOTRACE, and AMINONULL using an additional 4.1, 3.0, and 3.7 times more CPU time, respectively.

AMINOACI created 5.4 times fewer files per second than VANILLA. There was a 6.7 times increase in CPU time as well. The application CPU time decreased by 18.6%, because the application did not perform any creates in the kernel. However, the monitor used an additional 6.0 times as much CPU time as the original process. 23.1% of this increase is attributable to the monitoring, the remaining 76.9% is caused by setting registers, context switches, comparisons traversing B-trees, and locking overheads. The AMINOACID configuration ran 4.4% faster than VANSYNC.

*Unlink.* To evaluate the performance of `unlink` and `rmdir`, we removed the files and directories created by the create workload described above. We unmounted and remounted the file system to ensure cold cache between the create and `unlink` workloads. Figure 6(b) shows that the STRACE configuration performed 41.9% fewer deletions per second than VANILLA, mostly because of the context switches of `ptrace`. AMINOTRACE performed 42.7% fewer deletions per second than VANILLA, and AMINONULL performed 34.0% fewer deletions than VANILLA. AMINOACI ran 2.67 times slower than VANILLA. The break up of the overhead is similar to that of the `create` workload. AMINOACID ran 3.9 times faster than VANSYNC, because of a 51.3% decrease in the wait time, as a result of 74.0% fewer sectors being written. AMINOACID writes fewer sectors, because Ext3 must update the inode bitmap, directory block, and deleted inode which are stored in different locations on disk, whereas Amino only needs to update a leaf node of the B-tree.

*Stat.* Directory lookups are one of the most common operations, because they are a precursor for almost every meta-data operation (e.g., opening a file, creating an entry, etc.). To evaluate the lookup operation, we ran `stat` on 5,000 directories with 200 files each. After unmounting and remounting the file system, we performed a `stat` system call on each of the files. Figure 7(a) shows the results for this workload. The STRACE configuration performed 65.8% fewer lookups per second than VANILLA and used 6.4 times as much CPU time. The AMINOTRACE and AMINONULL configurations performed 56.2% and 57.3% fewer lookups than VANILLA and used 4.5 and 4.8 times more CPU time, respectively. The overhead for this workload is caused by two factors. First, the monitor context switches contribute to the increased system time. Second, the increase in user time is caused by resolving each path to determine if it is destined for a BDB mount in the monitor.



(a) `stat` micro-benchmark. (b) `readdir` micro-benchmark.

Fig. 7. Directory operation micro-benchmark results.

The AMINOACI configuration performed 56.8% fewer lookups per second than VANILLA. This is statistically indistinguishable from tracing alone (AMINOTRACE), but AMINOACI uses 26.7% more CPU time than AMINOTRACE. The increased CPU time over AMINOTRACE is attributable to two factors: a 25.1% increase in monitor system time (for reading pages from the database), and a 4.0 times increase in the monitor's user time to perform B-tree traversal. This is slightly offset by a 73.2% decrease in application system time, because the kernel does not perform directory searches on behalf of the process. Wait time is reduced by 54.9%, because 77% fewer read I/O operations are required (even though 48.4% more sectors are read).

*Readdir.* We used the same working set as in the `stat` micro-benchmark to evaluate the performance of the `readdir` operation. We performed a `readdir` on each of the 5,000 directories in sequence. The results are shown in Figure 7(b). The STRACE, AMINOTRACE, and AMINONULL configurations are all within 5% of VANILLA. The reason is that the directory reading is I/O-bound on Ext3, and reading directory entries in sequence takes advantage of read-ahead. This allows the slight increase in CPU time of at most 5.7% to overlap with I/O operations. Additionally, as Linux provides the `getdents` call to read directory entries, the total number of operations performed in the `readdir` workload is smaller than the lookup workload, thus the monitor incurs fewer context switches.

The AMINOACI configuration ran 2.3 faster than vanilla Ext3 for this workload. The improvement is mainly due to a 77.8% decrease in wait time. Wait time is reduced because Ext3 requires seeks to read each directory, as it does not place directories close to each other on the disk. This is evidenced by Amino's read requests taking 46.8% less time even though 3.3 times as many sectors are read. AMINOACI stores the path names in a B-tree and hence has better spatial locality. Therefore it requires fewer and shorter seeks to read directories. The use of B-trees to store metadata and data makes Amino suitable for metadata-intensive workloads which benefit from this locality.

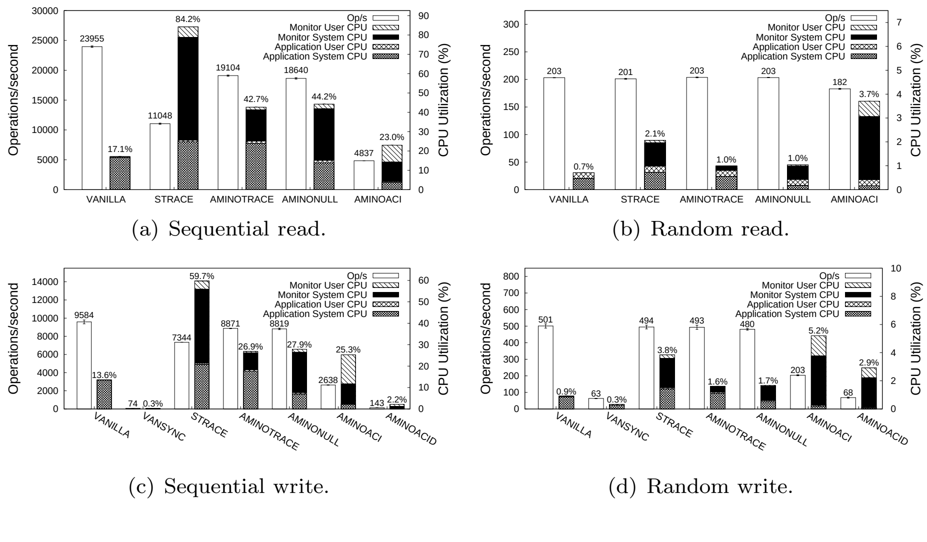
## 4.5 Data Micro-benchmarks

To evaluate the performance of data operations we ran random read, random write, sequential read, and sequential write micro-benchmarks. For each benchmark we created an 8GB file, which is twice the size of the machine's memory. This reduces the effects that caches have on the workload, because the workloads cycle through their list of blocks before rereading a block. For sequential operations, we operated (read or write) on consecutive pages of the file in sequence. For random operations, we generated a pre-populated pattern by randomly shuffling a sequential list of page numbers, and operated on the file using the shuffled list. This method ensures that there are no repeated pages so that caching does not affect the results.

For random and sequential reads we ran the workload for a 30 second warmup period followed by a 150 second measurement period. The warmup period allows the system to reach a steady state before measurement. For the sequential write workloads, we used a warmup period of 120 seconds, ran the benchmark for ten minutes and rebooted the machine between iterations. We used a longer write warmup period, because writes are very fast until the cache is filled, at which point the number of operations drops dramatically. We also used a longer measurement period, because the synchronization phase (to clear dirty cached pages) takes several minutes. The read benchmarks reached a steady state faster, so this additional time was not required. For all benchmarks, we report the number of operations performed per second and the percent of CPU utilization.

*Sequential Read.* The overheads of Amino under the sequential read workload are shown in Figure 8(a). The VANILLA configuration achieved 23,955 operations per second and used 17.1% of the CPU. The STRACE configuration performed 53.9% fewer reads and used 4.9 times as much CPU. The AMINOTRACE and AMINONULL configurations performed better, with 20.3% and 22.2% fewer operations, respectively. The AMINOTRACE and AMINONULL configurations also used less CPU time than STRACE, with an increase of 149.9% and 159.0% over VANILLA, respectively.

The AMINOACI configuration performed 79.8% fewer operations than VANILLA. The small increase in user and system time is because of data copies and B-tree comparison overheads. The increase in wait time is due to two factors: (1) sequential reads require more sector reads and fewer requests are merged in Amino than in Ext3 as the data layout in Amino is a B-tree, and (2) Amino does not provide any explicit read-ahead. The lack of read-ahead for Amino is exacerbated by the Linux read-ahead policy. As soon as a non-sequential access to a file is made, read-ahead is turned off. As BDB periodically accesses the database out-of-order, this results in Linux performing I/O operations that are on average 2.9 times larger for VANILLA than for AMINOACI. As part of our future research, we plan to investigate more efficient data layouts, possibly including a hybrid model that combines a flat file and a database structure.



(a) Sequential read. (b) Random read.

(c) Sequential write. (d) Random write.

Fig. 8. Data micro-benchmark results.

*Random Read.* The results of the random read benchmark are shown in Figure 8(b). The VANILLA configuration performed 203.5 operations per second. The STRACE, AMINOTRACE, and AMINONULL configurations were all within 1% of VANILLA. The CPU utilization for all of these configurations was low, with STRACE being the highest at 2.1%. The AMINOACI configuration had an overhead of 10.0%, and CPU utilization increased to 3.7%. Amino performed 0.99 disk read operations per `pread` system that was issued, whereas Ext3 performed 1.19 disk read operations per `pread`. However, Amino read 120.0 sectors for each `pread` request, whereas Ext3 read only 9.6. These differences can be attributed to the fact that Ext3 was configured to use 4KB blocks, but Amino's Data database had a page size of 64KB. The 21% decrease in disk read operations is caused by Amino finding some of the blocks it needs to read in the cache. The 12.5 times increase in the number of sectors read is caused by Amino reading the 14 adjacent file pages from the database each time it reads a page.

*Sequential Write.* Figure 8(c) shows the sequential write workload results. The VANILLA configuration performed 9,584 write operations per second. The STRACE configuration had an overhead of 23.4% over VANILLA due to a 4.3 times increase in CPU utilization. The AMINOTRACE and AMINONULL configurations were 7.4% and 8.4% slower than VANILLA, respectively. However, they used 93.1% and 100.7% more CPU, respectively. The AMINOACI configuration performed 72.5% fewer writes, and used 112.6% more CPU than VANILLA. Amino in its AMINOACID configuration ran 92.3% faster than Ext3 in its synchronous mode of operation.

The difference is primarily due to an increase in merged disk write requests because Ext3 requires non-contiguous changes to blocks and block bitmaps, whereas Amino just needs to commit changes to the B-tree leaf nodes.

*Random Write.* The random write workload does not produce normally distributed, results, because the Linux dirty buffer flushing daemon is quite sensitive to various cut-offs [Wright et al. 2003]. Figure 9 shows a time line of the number of operations per second performed during the benchmark (sampled at one second intervals). It is clear that there are large spikes (thousands of times larger than the mean), which cause a high variance. The initially high numbers of operations per second occur before the memory has many dirty buffers. Once a given number of the buffers are dirty (e.g., 10%), Linux begins to write them out in the background. This does not greatly affect the measured application, because it can still dirty buffers without penalty. The precipitous drop is caused when the number of dirty buffers exceeds a specified threshold (e.g., 40%). At that point, the application must write out a number of buffers (e.g., 48) for every buffer that it writes. When the combination of this throttling and the background flushing daemon bring the total number of dirty buffers below 40%, this synchronous behavior is removed. If the synchronous threshold were always 40%, then we would not expect this behavior, but Linux dynamically changes the threshold, thus causing oscillations. Aside from the large spikes, if we examine the values in the range from 100–1,000, we can see that there is still significant variation. If we were to use longer measurement periods, we would smooth some of the variation, but the large spikes are so much larger (and unpredictable) that achieving a stable result is exceedingly difficult.

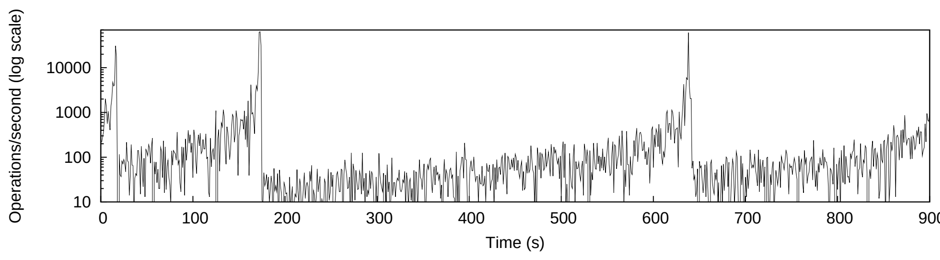


Fig. 9. Random write: operations/second (log scale) vs. time for VANILLA. The Linux dirty buffer flushing is very sensitive to certain thresholds, so the data is both periodic and highly erratic.

Other configurations also suffer from the same periodic and erratic behavior. For example, STRACE has periods of fast writes approximately every three to four minutes (STRACE has more of these periodic increases than VANILLA, because it writes buffers at a slower rate therefore the cache can drain more often). Including a warmup period does not eliminate these kind of startup effects. Instead we ran the random write benchmark for 15 minutes and measured the total number of operations per second. The results of the benchmark are shown in Figure 8(d). Figure 10 shows a box plot of the one second samples showing the first quartile, median, third quartile, and outliers (i.e., points that are more than 1.5 times the interquartile range ($Q_3 - Q_1$) from the median) for each configuration. Note that although AMINOACI and AMINOACID show outlying points near zero, the VANILLA, STRACE, AMINOTRACE, and AMINONULL also have points in this range, but they are not shown because the interquartile range for these configurations includes zero.

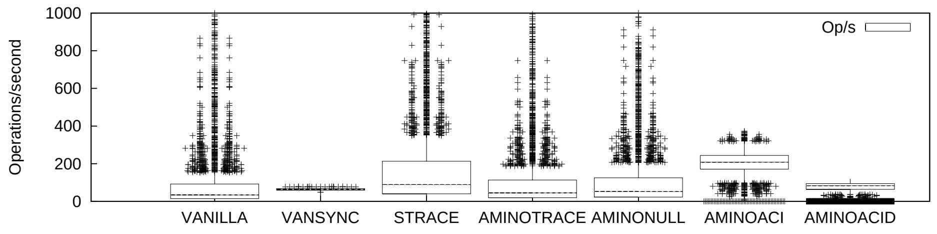


Fig. 10. Random write box plot: Each box represents 50% of the data points, the horizontal bar through the box is the median data point. The vertical lines extend 1.5 times the interquartile range ($Q_3 - Q_1$) from the median. Outlying points above 1,000 are not shown to conserve space.

Figure 8(d) shows the overheads associated with Amino for the random write workload. The VANILLA configuration performed 501 operations per second and utilized 0.9% of the CPU. When we consider each second of the benchmark independently, the interquartile range for this benchmark was 14–92 operations per second (i.e., 25% of the samples were less than 14 operations per second, 50% of the samples were between 14–92 operations per second, and 25% of the samples were greater than 92 operations per second). The highest number of operations achieved in one second was 68,864. The STRACE, AMINOTRACE, and AMINONULL performed 1.2%, 1.5%, and 4.0% fewer operations per second than VANILLA. Most of the one second samples were faster than vanilla for these configurations, with interquartile ranges of 39–213, 19–113, and 23–125 for STRACE, AMINOTRACE, and AMINONULL, respectively. However, the maximum achieved values were lower at 13,619, 36,854, and 34,614 operations per second, respectively. These two effects balanced out, with the mean value for VANILLA, STRACE, and AMINOTRACE being statistically indistinguishable and the AMINONULL having a slight 4.0% overhead due to an increase in CPU usage. The AMINOACI configuration performed 59.3% fewer writes than VANILLA. This can be attributed to Amino writing 7.5 times as many sectors per write operation, because the Data database uses a 64KB page size, whereas VANILLA can write data in 4KB units.

As expected, the durable configurations performed worse than the non-durable configurations: VANSYNC was 7.9 times slower than VANILLA, and AMINOACID was 7.9% faster than VANSYNC.

## 5. RELATED WORK

In this section we discuss four classes of related work. In Section 5.1 we describe systems that integrate databases with the file system. In Section 5.2, we discuss log-structured and journaling file systems. We discuss transactional memory systems in Section 5.3, and in Section 5.4 we describe various system-call interception methods.

## 5.1 Databases and File Systems

Previous simulations of transactions embedded in the file system showed that file system transactions can perform as well as a DBMS in disk-bound configurations [Seltzer and Stonebraker 1990]. The same simulations showed that for CPU-bound configurations, file system transactions usually have an overhead caused by system call costs of less than 20%.

The Inversion File System [Olson 1993] is a user-level wrapper library with file-system–like functions that stores files in a POSTGRES database. Inversion uses POSTGRES to support transactions and fast crash recovery. Unfortunately, Inversion operates in its own namespace, separate from that of other file systems, and uses different functions from the traditional Unix API, making it unsuitable for integrating legacy and transactional applications.

OdeFS [Gehani et al. 1994], Oracle's iFS [Oracle Corporation 2000], and DBFS [Murphy et al. 2002] are user-level NFS servers that use databases as a backing store. This approach allows unmodified applications to use the file system, but ACID properties cannot be extended to the application because the NFS client cache can serve requests without consulting the database system. Also performance suffers due to data copies and context switches related to the network stack. OdeFS is a file-system interface to objects already in the Ode object-oriented database. For each type of Ode object, new methods must be defined for read, write, and other file-system operations. DBFS is a block-structured file system developed using BDB. DBFS exceeds FFS's performance for meta-data operations. However, for page-sized data operations it is 5 to 40 times slower.

WinFS is part of an upcoming version of Microsoft Windows [Microsoft Corporation 2004] (originally WinFS was slated for Longhorn, but has been delayed to "some future date"). WinFS will integrate a full-fledged SQL DBMS into the OS. Using a heavyweight DBMS with SQL enables powerful queries, but could add significant code complexity. Additionally, overheads may be significant depending on schema design and query processing. WinFS uses the database as well as an NTFS file system as a backing store for all files. WinFS changes the basic unit of data storage from a file to an item (an object with attributes). The WinFS API supports explicit transactions for items, but since the API is so radically different, applications must change to take advantage of its new features.

QuickSilver is a distributed operating system developed by IBM research that makes use of transactional IPC [Schmuck and Wylie 1991]. QuickSilver was designed from the ground up using a microkernel architecture and IPC. Every IPC request has a transaction ID, and servers must be able to rollback requests on abort and write them to non-volatile storage on commit. All resource management and notification in QuickSilver is handled by transactions. For example, on process termination (commit or abort) the window manager destroys all windows. Amino integrates transactions into the file system using simpler and more widely-used OS primitives than QuickSilver. Unlike Quicksilver, in which each OS component must provide specific transaction support for rollback and commit, Amino leverages BDB so that each OS component or application can use simple begin, commit, and abort calls, without managing its own rollback or commit.

## 5.2 Log-structured and Journaling File Systems

Log-structured and journaling file systems borrowed the technique of write-ahead logging from databases [Rosenblum and Ousterhout 1991; Hagmann 1987]. The key difference between a log-structured file system and a journaling file system is that in a log-structured file system the log is the permanent home of the data, whereas in a journaling file system the log is a temporary location until the data is checkpointed to a permanent location on disk. In this respect, BDB, and hence Amino, is more similar to a journaling file system than a log-structured file system. When updates are made, they are first written to the database log file and then written to their permanent locations within the database file. In log-structured file systems, journaling file systems, and Amino, synchronous writes have improved performance because they are written sequentially to the log, obviating the need to seek to many locations of the disk for a single update.

Log-structured and journaling file systems write "transactions" to their log, but these transactions are completely controlled by the file system software—user applications cannot surround multiple file system operations in a single atomic transaction. Additionally, the transactions in a log-structured or journaling file system do not provide all of the elements of ACID. Instead, they provide atomicity and consistency for well-defined operations within the file system, and durability can be provided by flushing the log to disk. Notably, log-structured and journaling file systems do not include provisions for isolation apart from other facilities provided by the OS (e.g., directory-level semaphores). Amino provides atomicity, consistency, isolation, and durability for arbitrary sequences of file system operations.

## 5.3 Memory Transactions

Lightweight Recoverable Virtual Memory (LRVM) was developed to simplify Coda servers [Satyanarayanan et al. 1994]. LRVM is designed to handle transactionally protected memory-mapping of a file into a process's address space. To simplify LRVM's design, the file should be a small portion of the total storage: the undo log was stored in memory. Durability was provided by writing a redo log to disk. This type of functionality is closer to Amino's support for memory-mapped files than our in-memory transactions, as our memory-mapping essentially provides recoverable virtual memory. Our in-memory transactions, however, are designed to provide more efficient volatile transactions (i.e., without the need for any redo logging). LRVM was developed as a user-library and requires explicit calls to indicate that a given region of memory will be written to. We believe that our page fault handling mechanism for identifying writes is more convenient and robust. Indeed, the LRVM authors point out that the most common types of bugs were missing calls before manipulating a region, and suggest that language support for LRVM calls would be a good solution to these missing calls.

The Rio, or RAM I/O, project sought to bring persistence to standard memory [Chen et al. 1996]. If memory is persistent, then file systems can avoid writing data indefinitely, thereby improving performance by an order of magnitude. The key observation is that most data in memory is lost because of either power failures (which can be solved with a UPS) or software errors. To cope with software errors, Rio uses page protection and checksums to prevent an errant instruction from writing to memory. To update a page, it must be made writable, then the update is performed, and finally the page is made read-only again. Checksums are stored along with the data so that errors can be detected. This raises the bar for updating memory, so that an errant instruction is unlikely to corrupt Rio memory.

The authors implemented a file cache with Rio, and showed that it can be as reliable as a traditional disk-based file system under a variety of software faults. However, the two major problems with the Rio architecture are that not all architectures support warm reboot (e.g., an x86 cannot be rebooted without destroying RAM contents), and Rio also assumes that hardware and power failures are so rare as to be ignored. Unfortunately, hardware is becoming an increasingly large source of faults, as hardware components increase in number and complexity, and cost pressures force the use of less reliable components [Ghemawat et al. 2003].

Vista is a transactional RVM built on top of Rio [Lowell and Chen 1997]. Vista greatly improves the performance of an RVM system, because memory is assumed to survive a system crash—avoiding synchronous writes. Because Vista is built on top of Rio, it does not require a redo log, and the code complexity is much simpler than that of previous RVM systems, at around 700 lines. In our system, we provide a more rich transactional interface that includes nested transactions. Rather than implementing redo logging and its associated complexities, we leverage existing BDB code. Our completed transactional memory library is only 625 lines of code.

## 5.4 System Call Interception

The Ufo Global File system uses a similar interposition technique as our monitor [Alexandrov et al. 1997]. Ufo provides transparent access to remote files via FTP or HTTP. Ufo's monitor uses the Solaris `/proc` file system. The monitor operates on system calls such as `open`, `close`, and `stat`. When an access to a remote file is detected, the file is transparently fetched, and the system call is changed to open the local copy. Ufo does not implement other calls such as `read`, `write`, `getdents`, or `stat` internally, because the file's local copy can be used without modifying the application. To implement `getdents` and `stat` properly, however, sparse files are used to create *stubs* for files that are not yet locally cached. Creating this hierarchy of stub files hurts performance. The `ptrace` interface was used by the Janus framework to sandbox untrusted applications [Goldberg et al. 1996]. Janus monitors file-system and network-related system call invocations, and applies configurable policies to allow or deny system call execution.

Several other interposition techniques operate at the same logical system-call level as Amino, but use a customized interface. SLIC [Ghormley et al. 1998] is an OS extensibility system that allows kernel-level extensions or user-level servers to register handlers for system calls, signals, and other OS entry points. SLIC has been used to patch security holes, encrypt files, and provide a restricted execution environment. SLIC extensions that run as user-level servers are quite similar to the `ptrace` interface. Interposition agents provide higher-level abstractions for system call interception [Jones 1993]. The key insight for interposition agents is that system calls can be divided into classes that operate on independent sets of objects (e.g., path names or file descriptors).

## 6. CONCLUSIONS

Applications use an easy-to-use and standard POSIX API to access file systems, but file systems do not provide transactional semantics. Databases provide transactions, yet have differing and difficult-to-use APIs. Many applications can benefit from transactions, which can greatly improve error handling and provide atomic operations. For example, atomicity obviates the need for complex error handling, because a transaction can simply be aborted without any ill-effects. Atomicity can be used as a tool to ensure consistency, so that specialized and complex recovery code is not required. Isolation allows applications to provide race-free updates. Finally, durability ensures that data that was written actually reaches the persistent storage. Because transactions are so useful and the file system interface is convenient and ubiquitous, we therefore believe that file systems should provide transactional semantics to applications. Furthermore, we contend that the combination of file system transactions and recoverable memory will enable developers to use more robust and elegant error recovery methods than simply "giving up" and terminating an application upon a failure.

We have designed and developed *Amino*, a prototype file system with ACID semantics. Amino uses the Berkeley Database (BDB) as a backing store, with an efficient file system schema. Using BDB's flexible key-value pair access model, meta-data properly migrates between the Path and Data databases—improving performance for common operations while avoiding the pitfalls of logical redundancy. Amino exports an easy-to-use, yet powerful, nested-transactions API to user space. Applications can begin, commit, and abort transactions. We designed a simple API to enable cooperating processes to share transactions. Using the same API, a single application can support multiple concurrent transactions. To provide powerful transactions to application data structures, we developed an RVM library with support for nested transactions and transparent logging.

We have evaluated our prototype, and have shown that it has acceptable performance. Amino configured for atomicity, consistency, and isolation is only 15.4% slower than Ext3 even though it runs in user space and has additional overheads due to `ptrace`. When durability is required, performance inevitably suffers because of synchronous disk writes. Whereas providing durability for unmodified applications on Ext3 degrades performance by a factor of 25, Amino provides modified applications durable performance with a slowdown of only 10 times. Moreover, Amino provides applications with the additional benefits of atomicity, consistency, and isolation. This validates our decision to build Amino on top of a database rather than an existing file system.

### 6.1 Future work

Applications are currently responsible for handling their own data structures during a transaction. If the application has internal state, and a transaction is aborted, then its state and the file system state are not coherent. We plan to create an API that will let applications use file system transactions to protect in-memory updates. This way, applications can safely update their in-memory structures together with an associated file. If the transaction aborts, then both the application's memory and the file are restored.

In our present design, the performance aspects of the ACID properties and building a file system on a database are conflated into a single metric. We are currently exploring the alternative approach described in Section 2 of adding additional ACID semantics to an existing file system. By adding simple kernel primitives (e.g., more flexible mandatory locking), we can provide isolation. In concert with a standardized and portable kernel-level undo and redo logging facility, we believe that we can provide full ACID semantics to those applications that require it, with little or no modification to existing applications.

We plan to improve the Amino's performance by improving our data schema and the `ptrace` monitoring interface. Currently our Data database uses a balanced tree. We plan to create a custom access method that will give us control over data placement by write pages to a file or disk directly, thus allowing us to align data properly with the native OS page size and improve performance for data operations, which suffer compared to a standard disk based file system. We also plan to investigate changes to the Path database that would improve concurrent access. Because the database schema is so flexible compared to a file system layout, we also plan to explore application specific schemas (e.g., changing B-trees to hash tables, adding fields, or introducing indices).

To further improve the `ptrace` interface, we believe that we should reduce both the number of context switches and data copies between the kernel and the monitor. We believe that two key ways to do this are: (1) the kernel should use a shared-memory segment to manipulate the user process's registers so that data copies and context switches are reduced; and (2) several `ptrace` operations could be bundled into a single system call (e.g., waiting for notification could be combined with retrieving registers) to reduce context switches [Purohit et al. 2003].

To obtain copies of the micro-benchmark programs used in this article go to `www.fsl.cs.sunysb.edu/~cwright/benchmarks/`.

## Acknowledgments

Gaurav Poothia, Nagesh Chetan, and Sandhya Menon developed the initial kernel-level memory-mapping interface for `/proc/pid/mem`. Tzi-cker Chiueh, Alex Mohr, and Margo Seltzer, were generous of their time and provided significant helpful input and comments on the work. Rob Johnson, R. C. Sekar, and the anonymous TOS reviewers also provided useful feedback. This work was partially made possible thanks to NSF CAREER award CNS-0133589, NSF awards CCR-0310493 and CNS-0614784, and HP/Intel gifts numbers 87128 and 88415.1.

## REFERENCES

Alexandrov, A. D., Ibel, M., Schauser, K. E., and Scheiman, C. J. 1997. Extending the Operating System at the User Level: the Ufo Global File System. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, Anaheim, CA, 77–90.

Berliner, B. and Polk, J. 2001. Concurrent Versions System (CVS). `www.cvshome.org`.

Callaghan, B., Pawlowski, B., and Staubach, P. 1995. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, Network Working Group. June.

Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajmani, G., and Lowell, D. 1996. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the*

*Seventh International Conference on Architectural Support for Programming Langauges and Operating Systems (ASPLOS-VII)*. ACM, Cambridge, MA, 74–83.

CollabNet, Inc. 2004. Subversion. `http://subversion.tigris.org`.

Dike, J. 2000. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*. USENIX Association, Atlanta, GA, 63–72.

Ellard, D. and Seltzer, M. 2003. New NFS tracing tools and techniques for system analysis. In *Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*. USENIX Association, San Diego, CA.

Gehani, N. H., Jagadish, H. V., and Roome, W. D. 1994. OdeFS: A File System Interface to an Object-Oriented Database. In *Proceedings of the Twentieth International Conference on Very Large Databases*. Springer-Verlag Heidelberg, Santiago, Chile, 249–260.

Ghemawat, S., Gobioff, H., and Leung, S. T. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM SIGOPS, Bolton Landing, NY, 29–43.

Ghormley, D. P., Petrou, D., Rodrigues, S. H., and Anderson, T. E. 1998. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the Annual USENIX Technical Conference*. ACM, Berkeley, CA, 39–52.

Giarrusso, P. 2005. Fwd: Re: [patch 1/4] UML Support - Ptrace: adds the host SYSEMU support, for UML and general usage. `www.uwsg.iu.edu/hypermail/linux/kernel/0507.3/1992.html`.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. 1996. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *Proceedings of the Sixth USENIX UNIX Security Symposium*. USENIX Association, San Jose, CA, 1–13.

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

Hagmann, R. 1987. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM Press, Austin, TX, 155–162.

IEEE/ANSI. 1996. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. Tech. Rep. STD-1003.1, ISO/IEC.

Jones, M. B. 1993. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '93)*. ACM, Asheville, NC, 80–93.

Katcher, J. 1997. PostMark: A new filesystem benchmark. Tech. Rep. TR3022, Network Appliance. `www.netapp.com/tech_library/3022.html`.

Korn, D. G. and Krell, E. 1990. A New Dimension for the Unix File System. *Software-Practice and Experience 20,* S1 (June), 19–34.

Lewis, P., Bernstein, A., and Kifer, M. 2002. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison Wesley, Boston, MA, Chapter 8: Database Design II: Relational Normalization Theory, 211–260.

Lowell, D. E. and Chen, P. M. 1997. Free transactions with Rio Vista. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP '97)*. ACM, Saint Malo, France, 92–101.

Maziéres, D. 2001. A toolkit for user-level file systems. In *Proceedings of the Annual USENIX Technical Conference*. USENIX Association, Boston, MA, 261–274.

McKusick, M. K. and Ganger, G. R. 1999. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*. USENIX Association, Monterey, CA, 1–18.

McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. 1984. A fast file system for UNIX. *ACM Transactions on Computer Systems 2,* 3 (August), 181–197.

Microsoft Corporation. 2004. Microsoft MSDN WinFS Documentation. `http://msdn.microsoft.com/data/winfs/`.

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

MySQL AB. 2005. MySQL: The World's Most Popular Open Source Database. `www.mysql.org`.

Olson, M. A. 1993. The Design and Implementation of the Inversion File System. In *Proceedings of the Winter 1993 USENIX Technical Conference*. USENIX, San Diego, CA, 205–217.

Oracle Corporation. 2000. Oracle Internet File System Archive Documentation. `http://otn.oracle.com/documentation/ifs_arch.html`.

Purohit, A., Wright, C., Spadavecchia, J., and Zadok, E. 2003. Develop in User-Land, Run in Kernel Mode. In *Proceedings of the 2003 ACM Workshop on Hot Topics in Operating Systems (HotOS IX)*. USENIX Association, Lihue, Hawaii, 109–114.

Rosenblum, M. and Ousterhout, J. K. October 1991. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery SIGOPS, Asilomar Conference Center, Pacific Grove, CA, 1–15.

Santry, D. S., Feeley, M. J., Hutchinson, N. C., Veitch, A. C., Carton, R. W., and Ofir, J. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. ACM, Charleston, SC, 110–123.

Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. 1994. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems 12,* 1, 33–57.

Schmuck, F. and Wylie, J. 1991. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*. ACM Press, Pacific Grove, CA, 239–253.

Seltzer, M. and Stonebraker, M. 1990. Transaction Support in Read Optimized and Write Optimized File Systems. In *Proceedings of the Sixteenth International Conference on Very Large Databases*. Morgan Kaufmann, Brisbane, Australia, 174–185.

Seltzer, M. and Yigit, O. 1991. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*. USENIX Association, Dallas, TX, 173–184.

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

Sendmail Consortium. 2004. Sendmail home page. `www.sendmail.org`.

Sendmail, Inc. 2004. Sendmail Advanced Message Server. `www.sendmail.com/products/mailcenter/sams/`.

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

Szeredi, M. 2005. Filesystem in Userspace. `http://fuse.sourceforge.net`.

Wright, C. P., Dave, J., and Zadok, E. 2003. Cryptographic File Systems Performance: What You Don't Know Can Hurt You. In *Proceedings of the Second IEEE International Security In Storage Workshop (SISW 2003)*. IEEE Computer Society, Washington, DC, 47–61.

Received December 2005; revised December 2006; accepted Month Year