Extending ACID Semantics to the File System via `ptrace`

A Dissertation Presented

by

Charles Philip Wright

to

The Graduate School

in Partial fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-06-04

May 2006

Copyright by
Charles Philip Wright
2006

Abstract of the Dissertation

Extending ACID Semantics to the File System via `ptrace`

by

Charles Philip Wright

Doctor of Philosophy
in
Computer Science

Stony Brook University

2006

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 safety mechanisms in the form of transactions. Transactions are useful because they provide atomicity, consistency, isolation, and durability (ACID). Although many applications could make use of these semantics, databases have a wide variety of non-standard interfaces. As a result, applications currently perform elaborate error handling by themselves to ensure atomicity and consistency, because it is easier than using a DBMS. A transaction-oriented programming model eliminates such 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 file-system interface. In our prototype ACID file system, called Amino, updated applications can group arbitrary sequences of system calls within a transaction. Unmodified applications operate without any changes, but each system call is protected by a transaction.

Amino stores all of its meta-data and data in Berkeley DB, a user-level open-source embedded database. Berkeley DB provides an easy-to-use record-management API, an efficient B-tree access manager, caching, and ACID transactions. We developed Amino entirely in user space by intercepting system calls using the standard `ptrace` API, because developing Amino in the kernel would have required complex kernel changes. For example, the kernel's file system interfaces are intertwined with caches, thus preventing a DBMS from providing proper locking to ensure isolation. Additionally, user-level development proceeds more quickly than kernel development because more debugging and development tools are available. We show that our `ptrace` monitor can be used to develop a variety of user-level file systems with less complexity than other user-level file system approaches. Our performance evaluation shows that for general purpose benchmarks Amino has acceptable overheads (22.6–84.7%), and in some cases even exceeds the performance of Ext3.

To George:

his mother's zaika—a cute little bunny,
his father's little crocodile—a future apex predator,
and the greatest blessing God has bestowed on us.

Contents

List of Figures **vii**

List of Tables **viii**

Acknowledgments **x**

1 Introduction **1**

2 Amino Design **5**
2.1 BDB Overview 6
2.2 File System Schema 7
2.2.1 The Path Database 8
2.2.2 The Data Database 9
2.2.3 The Orphan Database 11
2.2.4 Path-local and Data-local Meta-data 12
2.2.5 Example File System 12
2.3 Internal File System Transactions 14
2.4 Transactions API for Applications 15
2.5 Transactional Applications 18
2.5.1 Postmark 18
2.5.2 GNU Make 19
2.5.3 GNU `tar` 22
2.5.4 `mail.local` 22

3 Monitor Design **24**
3.1 Process Tracing Primitives 27
3.2 Amino Structure 29
3.3 Process Control Blocks 31
3.4 Mount Subsystem 32
3.5 Address Spaces 34
3.5.1 Accessing User Memory 34
3.5.2 Rewriting System Call Arguments 35
3.6 Memory-Mapped Operations 37
3.7 `ptrace` Enhancements 39
3.8 Implementation limitations 41

4 File System Complexity **44**
4.1 Pass-Through Layer . . . 45
4.2 AES Encryption Layer . . . 46
4.2.1 Encryption scheme . . . 47
4.2.2 Extended attributes . . . 48
4.2.3 Implementation . . . 49
4.3 ISO9660 File System . . . 50
4.4 Complexity Evaluation . . . 50
4.4.1 Framework implementation . . . 50
4.4.2 Pass-through layer . . . 52
4.4.3 Encryption file system . . . 52
4.4.4 ISO9660 file system . . . 52
4.4.5 Amino file system . . . 53

5 Evaluation **54**
5.1 Meta-data Micro-benchmarks . . . 55
5.2 Data Micro-benchmarks . . . 60
5.2.1 Single Threaded Results . . . 60
5.2.2 Multi-Threaded Results . . . 65
5.2.3 Cached Results . . . 70
5.3 Micro-benchmark Summary . . . 72
5.4 Postmark . . . 73
5.4.1 Alternate Configurations . . . 75
5.5 OpenSSH Compile . . . 76
5.6 Sendmail . . . 80
5.7 General-Purpose Benchmark Summary . . . 82

6 Related Work **84**
6.1 Transactions and File Systems . . . 84
6.2 File Systems built on Databases . . . 86
6.3 Log-structured and Journaling File Systems . . . 88
6.4 Memory Transactions . . . 88
6.5 System Call Interception . . . 90

7 Conclusions **92**
7.1 Future work . . . 93
7.1.1 Possible `ptrace` enhancements . . . 93
7.1.2 Address Space Manipulation . . . 95
7.1.3 Transactions API . . . 95
7.1.4 Database Deadlocks . . . 96
7.1.5 Schema Improvements . . . 97
7.1.6 ExtAcid . . . 98

A Additional Performance Evaluation **106**
A.1 Additional Tar Benchmarks 106
A.2 `ptrace` Primitives 108

B Monitor VFS Operations **111**
B.1 Internal VFS Operations 111
B.1.1 Mount Operations 111
B.1.2 File Operations 112
B.1.3 Directory Name Operations 113
B.1.4 Notification Operations 114
B.2 System Call VFS Operations 115
B.2.1 Operations with Generic Implementations 116
B.2.2 Operations on File Descriptors 117
B.2.3 Operations on Path Names 118
B.2.4 Unimplemented System Calls 118
B.2.5 New System Call 118

List of Figures

2.1 Sample file system schema. 13
2.2 Transactions API summary. 17
2.3 Simplified `start_job_command` function. 21
2.4 Simplified `reap_children` function. 22

3.1 The Amino monitor's structure. 26
3.2 `ptrace` primitives for a `read` system call. 28
3.3 Simplified monitor state transitions. 29
3.4 Monitor state transitions for `open`, `chdir`, and `exec`. 30
3.5 Path name resolution flow chart. 33
3.6 Monitor state transitions for file name rewriting. 36
3.7 Using `PTRACE_CHECKEMU` for a `read` system call. 40
3.8 Monitor state transitions with `PTRACE_CHECKEMU` and `PTRACE_SYSSKIP`. 41
3.9 Memory-mapping `/proc/pid/mem` interactions with copy-on-write pages 43

4.1 The `unlink` method for the pass-through file system layer. 46

5.1 Creation micro-benchmark results. 56
5.2 Deletion micro-benchmark results. 57
5.3 `stat` micro-benchmark results. 58
5.4 `readdir` micro-benchmark results. 59
5.5 Data micro-benchmark results: Sequential read. 61
5.6 Data micro-benchmark results: Random read. 62
5.7 Data micro-benchmark results: Sequential write. 63
5.8 Random write: operations/second vs. time 63
5.9 Random write: operations/second vs. time (STRACE) 64
5.10 Data micro-benchmark results: Random write. 65
5.11 Data micro-benchmark results: Random write box plot. 66
5.12 Data micro-benchmark results: Multi-threaded sequential read. 67
5.13 Data micro-benchmark results: Multi-threaded random read. 68
5.14 Data micro-benchmark results: Multi-threaded sequential write. 69
5.15 Data micro-benchmark results: Multi-threaded random write. 71
5.16 Data micro-benchmark results: Cached sequential read. 71
5.17 Postmark: 2,500 files. 74
5.18 Postmark: 2,500 Files; 5,120–102,400 bytes. 76
5.19 Postmark: 25,000 Files 77

5.20 OpenSSH Unpack results. 78
5.21 OpenSSH Configuration Results. 79
5.22 OpenSSH Build Results. 79
5.23 Local mailer: requested vs. achieved load. 81

A.1 Unpack results: Linux 2.6.16. 107
A.2 Unpack results: Linux 2.6.16 (non-durable configurations). 107
A.3 Unpack results: Ten copies of Linux 2.6.16. 108

List of Tables

1.1 File system support for ACID. 3

2.1 Our database schema. 7
2.2 Lines of Code for Make 20
2.3 Lines of Code for `tar` 22

4.1 Code complexity metrics. 51

A.1 `getpid` micro-benchmark results. 109
A.2 `stat` micro-benchmark results. 110
A.3 `open` micro-benchmark results. 110

Acknowledgments

This work would not have been possible without the loving support of my wife Mariya. She has suffered through weeks of her fiancee and husband spending every waking moment in the lab. Without her patience, kindness, and perseverance I never would have been able to accomplish this much. My advisor, Erez Zadok, has always been available, helpful, and without him I would not have been able to complete this research. Rick Spillane and Gopalan Sivathanu both contributed in substantial ways this work. Rick worked tirelessly on code during several submissions. Gopalan helped with the analysis, and is an excellent "devil's advocate," challenging me to explain why things are the way I think they are. Gaurav Poothia, Nagesh Chetan, and Sandhya Menon developed the initial kernel-level memory-mapping interface for `/proc/pid/mem`.

My other lab mates have also been great to work with. I would be remiss if I did not acknowledge Kiran Reddy who sat next to me debugging race conditions for months on end during the early years of my study. Eugene Miretskiy introduced me to the wonders of Perl (vs. simply Perl), without which my benchmarking would have been significantly more difficult. More recently, Avishay Traeger and Nikolai Joukov have been my closest partners in crime, making the lab a pleasant place to work. Sean Callanan never passed up a lively debate. During my studies I have also collaborated with Joseph Spadavecchia, Jeffrey Osborn, Ariye Shater, Amit Purohit, Michael Martino, Andrew Himmer, Jay Dave, Akshat Aranya, Puja Gupta, Harikesvan Krishnan, Mohammad Zubair, Abhijith Das, Aditya Kashyap, Devaki Kulkarni, Abhishek Rai, Timothy Wong, Dave Quigley, Arun Krishnakumar and Jeff Sipek.

My committee members, Tzi-cker Chiueh, Alex Mohr, and Margo Seltzer, were generous of their time and provided helpful input and comments on the work. Tzi-cker in particular provided key insights during a Graduate Research Conference presentation in 2004. After my proposal Margo gave me detailed and helpful comments on the writing and also new ideas for technical avenues of exploration. Rob Johnson provided useful pointers with regard to recoverable virtual memory. R. C. Sekar also provided useful feedback after my CSE 659 presentation.

The departments systems and secretarial staff provided valuable administrative support. This work was partially made possible thanks to NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

Chapter 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. Reliability is therefore an important goal for any file system.

Database systems use transactions to provide strong guarantees for data safety and consistency. Transactions provide four key properties: atomicity, consistency, isolation, and durability—collectively known as the *ACID* properties. Despite their importance, most file systems do not ensure that operations meet all four of these stringent requirements. Our goal is to combine the database reliability (embodied by the ACID properties) with file system's ease of use (embodied by the common POSIX API [32]).

Consider the ACID requirements as they relate to file systems.

Atomicity Atomicity means that a set of operations must complete or fail as a unit. Traditionally, file systems provide atomicity only for single operations (e.g., renaming a file either fails or succeeds). Many applications undertake arduous procedures to attempt to perform atomic operations. For example, if Sendmail [77] fails when attempting to append new mail messages to a mailbox, it then attempts to truncate the file to erase a partially written message. Yet if the truncation fails, then the mailbox is corrupted. A file system that allows a sequence of operations to be applied atomically solves this problem providing two key benefits: (1) error handling is easier, because transactions can simply be aborted, and (2) data corruption cannot occur, because corrupted data never reaches the file system. Atomic sequences make Sendmail's append operations transactional. If they all succeed, then Sendmail commits the transaction. Otherwise, Sendmail aborts the transaction and the file-system state remains unchanged.

Consistency In a database system, consistency means the database enforces pre-defined integrity constraints. Uniqueness of social security numbers or requiring a positive account balance are examples of integrity constraints. A more complex integrity constraint is that if a table has an secondary index, then the index must reflect the values stored in the table. File systems have similar constraints (e.g., inode numbers are unique and no directory entry points to a non-existent inode). A file system

can maintain a consistent on-disk state by wrapping related operations in database transactions

Applications also have consistency requirements. For example, CVS [4] creates lock files to protect against concurrent access when files are committed.

The corresponding integrity constraint is that lock files exist only 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 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 that only some files were committed, while others were not (e.g., if the process terminates half-way through a commit). Using transactions from the start from the start from the start would have eliminated hundreds of lines of code through several source files in CVS. Moreover, because the transactional interface does not commit data until all operations succeed, error-handling is more robust than the ad-hoc mechanism that is currently used.

Isolation Isolation (or serialization) means that one transaction does not affect the execution of another concurrently running transaction. This functionality is not available in current file systems. For example, a set-UID program cannot use `access` to check that 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 exploitation is called a time-of-check-time-of-use (TOCTOU) security vulnerability. With a file system that maintains isolation, for example, `access` and file creation can be performed safely in a single transaction so no other operations can intervene between the `access` and the creation. In practice, operations must be executed concurrently for good performance. Therefore, other operations may be interleaved, but the file 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. OSs often make this choice because the synchronous I/O that is often required for durability can result in poor performance. Databases employ optimizations such as sequential logs, group commit, and ordered writes to provide durability efficiently.

Table 1.1 presents current file systems summarizing their support or lack thereof for 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. Traditional file systems like Ext2 and FFS do not provide consistency, resulting in the need to run a consistency checker before mounting them (`fsck`). Journaling file systems ensure that each operation is consistent, so the composition of many operations is also consistent

	Ext2 and FFS-no-SU*	Ext3	FFS+SU*	Amino
Atomicity	No	Single op	No	Multiple ops
Consistency	No	File system	File system, but resources may leak	Application level
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.

Table 1.1: 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.

from the file system's perspective. SoftUpdates is an interesting point along the consistency spectrum. In SoftUpdates disk writes are carefully ordered such that a pointer never references an inconsistent object [46]. This means that the file system is consistent, except for reference leaks. These reference leaks must be corrected to prevent the file system from slowly losing disk space, therefore a consistency checker is still required; but because the file system is consistent in other respects it is possible to run it in the background. File system consistency is a necessary condition for application-level consistency, but applications require higher level guarantees—the semantics of the data must make sense in that application's context. For example, a mail file should not have half-written messages. Because Amino provides multiple operation atomicity, applications can transition from one consistent state to another. 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.

ACID properties are desirable for many applications, especially highly reliable applications like email, or applications that require atomicity and isolation for security (e.g., updating a user's credentials). We 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. Existing applications need only slight modifications to exercise fine-grained control over transactions and benefit from improved reliability.

The alternative approach has databases take 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 [78], and it is becoming more common for revision-control systems to store information in a database [10]. However, writing applications that use the file system interface has inherent advantages over writing applications that use the database interface. Writing to a database API severely limits interoperability and burdens 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 an uncommon request). Moreover, any standard text processing package can be used to edit the file. When data is accessible only through a database interface, these types of convenient access are impossible. Instead, special applications must be written for each of these functionalities.

We built Amino on top of Berkeley DB (BDB) [74]. 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, neither provides nor requires 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 be required 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 implemented 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. We have also developed small and generic modifications to the kernel `ptrace` that improve the performance of `ptrace` monitors. We developed three other file systems using our monitor framework and show that they have a low complexity compared to other methods of developing user-level file systems.

Our prototype evaluation shows that although `ptrace` significantly increases the CPU utilization, resulting in reduced micro-benchmark performance, Amino can provide atomicity, consistency, and isolation to existing user-level applications with a small performance impact (and in some cases even exceeds the performance of Ext3). Additionally, in many cases, our `ptrace` monitor provides file-system functionality with better performance than the standard `strace` tool, which only traces system calls without adding additional functionality. Amino implicitly provides durability, matching or exceeding the performance of Ext3 with durability. Moreover, If a programmer informs Amino when transactions begin and end, durable performance can be several times 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 dissertation is organized as follows. Chapter 2 provides an overview of our transactional file system's design, including several sample applications. Chapter 3 describes our `ptrace`-based Amino prototype. Chapter 4 describes three simple `ptrace`-based file systems and evaluates their complexity. Chapter 5 evaluates Amino's performance. Chapter 6 describes related work. We conclude and discuss future work in Chapter 7.

Chapter 2

Amino Design

The key decision to make when providing ACID semantics to the POSIX file system interface 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 [53] and version 4.3.28 of the Berkeley Database (BDB) [55, 74, 82]. 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 19,776 lines of code in just its transaction-related components,[1] and BDB is a subset of MySQL's overall transaction code (BDB is one of MySQL's engines for 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 17,312 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 a 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, adding extra context switches and data copies. These context switches and especially data copies could hurt file system performance. We therefore chose to use an embedded database, running directly in the address space of the client—thereby eliminating context switches and data copies. There are several embedded databases available including BDB, MySQL in embedded mode, and SQLite [31]. All three options provide transactions, but MySQL and SQLite require SQL, whereas BDB provides lower-level primitives. We

[1] It should be noted that other BDB components make extensive use of the transactional components counted here, so the number 19,776 is a minimum and not a maximum for the number of transaction-related lines of code.

chose BDB for two reasons. First, bypassing the SQL interface improves performance because SQL parsing, query optimization, and other features often associated with a DBMS are not required. As these features are not needed for a file system, having less code is a distinct advantage. More importantly bypassing SQL allows us to control our data layout more precisely, producing more opportunities for performance optimization and tuning. Second, BDB has a highly modular design and the application designer can choose which components are required (e.g., the transaction subsystem can be used with normal files, or the access methods can be used without logging). In addition, BDB is widely deployed, has been thoroughly tested, and 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. 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 B-tree[2], 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 and its supported operations. Section 2.2 describes our database schema. Section 2.3 describes our internal use of transactions. Section 2.4 describes the transaction API that we expose to applications. Section 2.5 describes three example applications to which we have added transactional semantics.

2.1 BDB Overview

BDB provides a uniform API to access hash tables, B-trees, queues, and record-number keyed collections in a transactional manner. For a file system, hash tables and B-trees are the most appropriate underlying data structure, because file systems map names (keys) to files (values) and positions in a file (keys) to the file's data (values). We selected B-trees, because they are sorted, allowing us to control locality and iterate easily through records. The first step to use a BDB database is to open a database environment. The database environment provides caching, logging, and locking functionality for one or more databases (or even simple files). One or more databases can then be opened in the context of that environment. Transactions are associated with an environment, and they have five operations: begin, prepare, commit, abort, and discard. The prepare and discard operations are used only for distributed transactions, so we do not discuss them further. Transactions are used to protect other database operations. 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

[2]Specifically, a B+ linked tree [11].

Database	Key	Value
Path[3]	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)

Table 2.1: 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.

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 use 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 [82].

2.2 File System Schema

The database schema defines the format of our file system. The schema dictates the layout of the data, which in turn is directly related to what operations are possible, and the efficiency of each operation. Our primary goal in developing our schema was to minimize the number of database accesses required for any given operation. This is important for two reasons: (1) uncached database operations result in I/O operations, which are many orders of magnitude slower than in-memory operations; and (2) even for cached accesses each database operation requires additional function calls, locking, logging, and B-tree traversal. 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 [47]. When an FFS-like file system reads 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

[3]To provide shorter prefixes the Path database key is actually Depth||Path, which can be derived from path. This is discussed more thoroughly in the text.

component for storage. Our schema, shown in Table 2.1, has a similar division. All of our meta-data and data is stored in three BDB databases. We use a *Path* database to map pathnames to unique file identifiers, and a *Data* database to map unique file identifiers to the file's data blocks. The *Orphan* database contains a list of identifiers that are not accessible through the name space, but is otherwise equivalent to the Path database.

In the rest of this section we describe the design considerations when developing our schema. 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.

2.2.1 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 lookup requires 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 [18]. 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. Essentially, this makes a trade-off between traditional path-name lookup which depends on the user-determined depth of a path and traversing B-tree nodes which is dependent on the number of files in the file system. Additionally, renaming a directory requires updating all of the directory's children.

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. Theoretically, the key (i.e., the full path name) is sufficient for this sort function, because the depth can be derived from the full path name. However, storing only the full path name greatly increases the length of the prefix that is required to distinguish two keys, because the depth cannot be determined until the final slash is encountered. Ideally, most keys would have a short common prefix to take advantage of BDB's prefix compression within internal nodes of the B-tree. To solve this problem, we explicitly include the depth of the path name as the first four bytes of the key. This allows us to simply compare the depth, followed by a simple string comparison of the remainder of the key. For keys with different depth, the prefix length is four bytes. For keys with the same depth, the prefix length is four bytes plus the length of the path's common prefix. For example the prefix length for `/usr/bin` and `/usr/lib` is $4 + 5 = 9$, because the common prefix is `/usr/`.

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 [7].

2.2.2 The Data Database

To store data pages, we use a balanced tree. If a file's unique identifier is stored in the tree, then a file with that identifier exists[4]. 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. We store the number of open instances for the file in the database, because open files can be unlinked. Unfortunately, this introduces additional updates when opening and closing the file. However, it obviates one design problem and another problem that is an artifact of our implementation. First, storing this counter in memory would result in inconsistencies on transaction abort. Second, in our implementation independent processes can access the same file system (without any shared resources aside from the databases), therefore the database provides a consistent method of accessing a shared resource.

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. We considered two primary alternatives for selecting the page size: (1) using the OS's native page size (4KB on Linux) and (2) using the largest page size available (i.e., 64KB). Using a native OS page size, allows us to match our disk transfer unit to the OS, but requires overflow pages. Using a larger page size allows us to take advantage of locality and eliminates the need for overflow pages. Using a smaller page size than the native OS page size would put Amino at a great disadvantage, because more I/O transfers would be needed to read the same amount of data. Intermediate page sizes would not realize the full benefits of bulk I/O transfers associated increasing the page size.[5]

Native Page Size The first alternative we considered was using a page size that is equal to the native page size of the underlying file system (4KB on Linux). This ensures that BDB

[4]The existence of a file does not imply that a pathname points to it. This situation occurs when an open file is unlinked.

[5]In fact, our initial tests showed that 8KB pages exacerbate the problems of 4KB pages, because overflow pages are still used. Additionally, neither 16KB or 32KB pages yield the performance of 64KB pages.

reads and writes in units that the file system can handle efficiently. 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. Because of these considerations, when a 4,096-byte page is used the transfer unit for our file system is 4,070. Although this is a non-standard size, well-behaved applications should execute the `fstat` system call to find the optimal transfer unit stored in the `st_blksize` field. Poorly behaved applications work as expected, but with degraded performance. Our benchmarks show that when applications using a 4,096 block size, there is a 4% slow down for sequential reads, and an 18% slow down for sequential writes. Random operations have a greater performance penalty, because they do not benefit from locality as the sequential workloads do: reads are slowed by 48.4% and writes by 57.1%. The Inversion file system made a similar trade-off [56]: it uses a transfer unit that is slightly less than POSTGRES's 8KB page size.

Large Database Pages The second alternative we considered was using the largest page size available. The advantage of this alternative is that we can present applications with a standard 4,096-byte transfer unit without the use of overflow pages in the database. Additionally, using a larger page automatically provides read-ahead, because for each transfer unit that is read into the database cache 15 adjacent transfer units are also read into the cache. The major concern with this page size is that random I/O performance could be reduced, because the database performs I/O operations in terms of whole database pages. This means an I/O operation for a single transfer unit requires reading or writing 15 adjacent transfer units of data.

Our benchmarks show that using 64KB pages with 4,096-byte transfer units provides a 26.3% improvement for sequential read; a 17.7% improvement for sequential write; and a 3.9% improvement for random read. However, random write throughput was reduced by

4.9%. Overall, we concluded that using a 64KB page size is a significant improvement over the 4KB page size in terms of performance and also application compatibility. Therefore, we use a 64KB page size by default (and in our evaluation in Chapter 5).

Cursor Usage We found that using database cursors is essential for good sequential read performance. Simply retrieving each item in 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.

Ordering Database Operations Files in Amino support traditional Unix semantics including the ability to have sparse regions (a.k.a holes) inside of the file or at the end of the file. Our original implementations of the `read` and `write` operations were very much like a standard Unix file system: we read the file's meta-data, and then proceeded to read the data. On a traditional Unix file system, this is required, because the meta-data contains block pointers which must be read to read the actual file data. This is also convenient, because reads past the end of the file can be eliminated or truncated to an appropriate size—eliminating the need for special cases.

However, on an Amino file system, reading the meta-data before accessing a block is not required, because storing pointers to individual blocks is delegated to the DBMS. In fact, reading meta-data first proved quite harmful for concurrent performance. For a sequential read workload, adding additional processes results in a steady decrease in the number of operations performed. Worse yet, the additional lock contention causes two processes to be several times worse than a single process.[6]

Our current implementation reverses this intuitive approach. To read a file, the specific data page is read first and only then is the meta-data read and updated. This slightly complicates the implementation of sparse files. A possible hole at the end of the file is detected by encountering the data-local meta-data of the next file in the database (or equivalently the last record in the database). To determine whether the hole actually exists, the data-local meta-data of the file is accessed and if the size is greater than the position of the last byte that was read, then the hole is filled. As we show in Section 5.2.2, this approach prevents random and sequential reads from degrading as concurrency increases.

2.2.3 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.

[6]The processes must wait for each other after each I/O and thus serially perform what becomes random I/O. After a fourth process is added, the I/O rate improves because the head seeks for each I/O become shorter.

2.2.4 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 [13]. 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 [40]. In our schema, logical redundancy introduces *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 includes all meta-data that is specific to one path of a file. Data-local meta-data includes all meta-data that may refer to more than one path. For example, a newly created file's `stat` information is stored as path-local meta-data, because there is no other path name that references this `stat` information. However, if a hard link to the file is created, then the path-local meta-data is promoted to data-local meta-data, as both names can used to reference the same underlying file. If one of the links is removed, then the data-local `stat` information is demoted to path-local meta-data. 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.

2.2.5 Example File System

To illustrate the relations in our schema, we present a concrete example of an Amino file system with three directories and four files. Figure 2.1 shows the file system's databases. The keys are rectangles with sharp corners and the values are rectangles with rounded corners. The solid arrows show key-value relationships within a database, and the dashed arrows show logical relationships between the databases. The database manages the key-value relationships, but our file system maintains the logical relations. The data shown in the Data database is only an example, in reality entire data pages would be stored.

This file system contains three directories: `/`, `/doc`, and `share`. The attributes for all directories are stored in the Path database, because directories cannot have hard links. The directories have a single entry in the Data database, indicating that their unique identifier is in use. The sorting algorithm is also demonstrated by this example. The root directory, "`/`" is first, because it has the smallest depth (i.e., zero). The directories with a depth of one are next (i.e., direct descendants of the root directory). Within a given depth, the directories are sorted in lexicographic order. This has the effect of sorting a directories

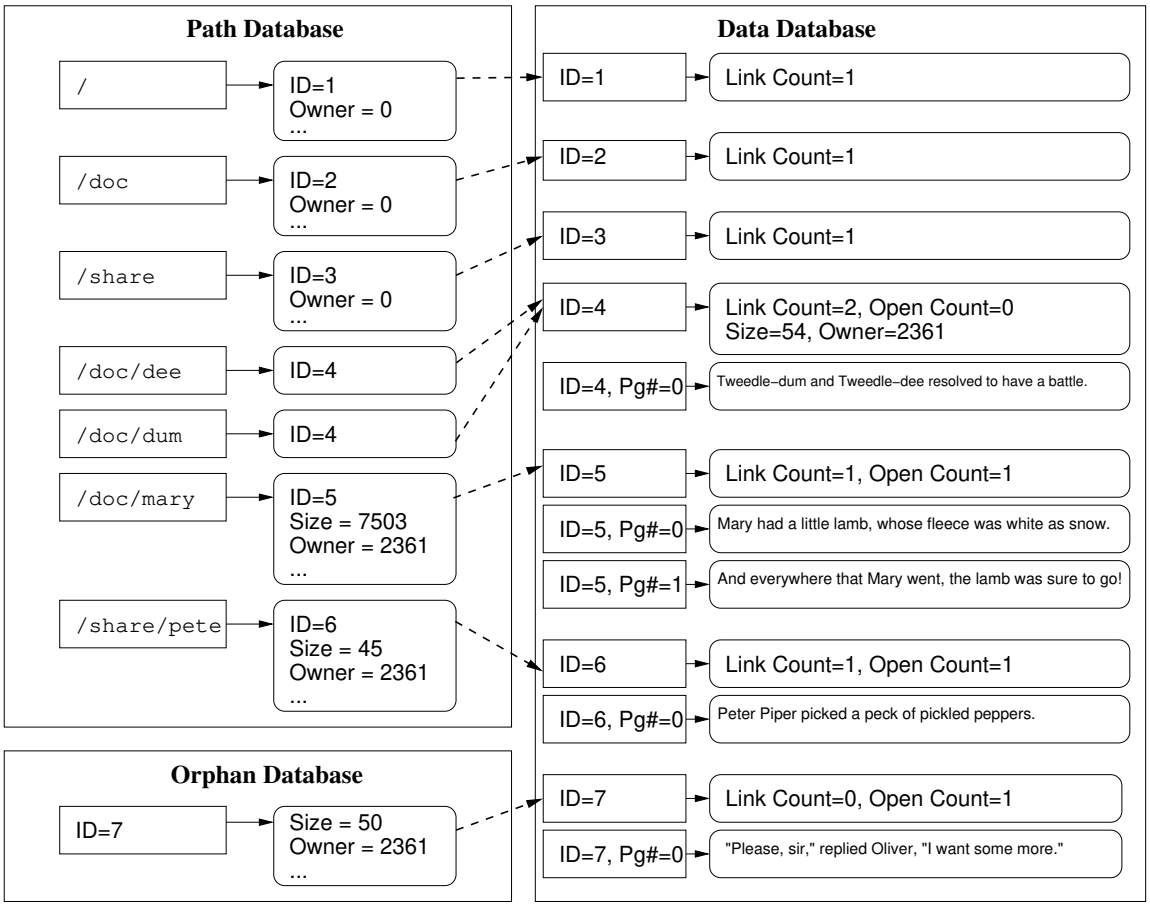


Figure 2.1: *A sample Amino file system. Within a database, keys (rectangles) and data (rounded rectangles) are connected by solid arrows. Logical relationships between databases are denoted by dashed arrows.*

children consecutively in the database.

There are two files without hard links in this example: `/doc/mary` and `/share/pete`. Like the directories, the meta-data is stored in the Path database, because there are no hard links for these files. In the Data database, there are entries for the unique identifier. For example, `/doc/mary` has a unique identifier of five, and the corresponding entry indicates that no other file may use that identifier. These files also have data pages, that are contained entirely within the Data database. For `/doc/mary`, there are two pages; and `/share/pete` has a single page. A file need not have consecutive pages, for example a sparse file does not store any page for holes. Pages also need not be as long as the maximum page size. For example, `/share/pete` is only 45 bytes long, so its first page is 45 bytes. When BDB returns data from a database, it also returns the data's size. Amino uses the record's size as returned by BDB to handle sparse files. The file `/doc/mary` also has pages that are less than 4,096 bytes. The first page contains only 55 bytes of data ("Mary had a little lamb, whose fleece was white as snow."). When a user process reads bytes 56–4,096, zeros are automatically returned. Similarly, the second page contains only 55 bytes of data ("And everywhere that Mary went, the lamb was sure to go!"). Because the file is 7,503 bytes long, Amino automatically returns zero for bytes 4,151–7,503.

The file with the unique identifier four has two names, `/doc/dee` and `/doc/dum`. Because storing the file's meta-data in the Path database introduces logical redundancy, the meta-data is instead stored in the Data database together with the file's reference count. The file's page is stored identically to those files without hard links.

Finally, this file system has an orphan file. Orphan files are files that do have no name. They come into existence on Unix systems when an open file is removed. After the file is closed, the file's data is removed. If the system crashes while orphan files are open, Amino locates these files by iterating through the Orphan database and deleting them. In other respects, orphan files are identical to the other files.

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 executing that operation is not concerned with ACID semantics, because other applications must see 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. Defining and verifying integrity constraints for most disk-based file systems is difficult, because the combination of caches and asynchronous writes yield complex states that must be managed. Recent work to address this adapted Linear Temporal Logic (LTL) to disk-based file systems [61, 81], however this type of logic is not suitable (or necessary) for file systems based on a transactional DBMS. This is because the DBMS handles the complexities of ensuring that caches and the disk-based image are consistent, and it allows the file system to apply multiple updates atomically.

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 data-local meta-data block has a positive link count or open instance reference count. If the link count is zero, then an entry for this file must exist in the Orphan database.

NOORPHANEDBLOCKS Each data page in the Data database has an associated data-local meta-data block.

HARDLINKUSESDLMD If and only if a file has a link reference count greater than one, then it uses data-local meta-data.

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 LASTPAGEMATCHESSIZE 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). 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. An alternative strategy is *enforcement*. If we selected an RDBMS that enforced integrity constraints, the database system would validate the constraint before committing each transaction, thus simplifying our file system's design at the cost of decreased performance (because each integrity constraint must be verified).

2.4 Transactions API for Applications

Legacy applications need no changes to enjoy the benefits of a consistent file system that 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 on Linux 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 applications that need fine-grained control of transactions, we export a transactional API (shown in Figure 2.2 on Page 17) 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. We created a new system call called `amino` that performs all of the operations for our API based on its first argument. To begin a transaction, an application issues a begin 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. Our API supports the BDB flags `DB_TXN_NOSYNC`, `DB_TXN_SYNC`, and `DB_TXN_NOWAIT`. These flags are directly passed to BDB and instruct the database to skip flushing the log on commit, flush the log on commit, and return `DB_LOCK_DEADLOCK` immediately instead of blocking on locks.

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 a new current transaction is created as a child of the existing current transaction. This creates a stack of nested current transactions associated with the process.

Aborting a transaction simplifies 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 recoverable virtual memory facilities to rewind data structures transparently [69]. 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.

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 *current transaction* is established, the transaction stack (i.e., the transaction and all of its children) is assigned a unique identifier. A process set its current transaction by attaching to the unique identifier. In this way, two processes can share the same transac-

amino(BEGIN_TXN, *pathname*, flags) Creates a new current transaction for this process that is associated with the mount containing *pathname*. If *pathname* is not located on an Amino file system, then ENOSYS is returned.

If this process already has a current transaction, then a child transaction is created. This forms a stack of transactions. If the parent transaction is for a different mount point than *pathname*, then EXDEV is returned. The process's new current transaction is the child.

The DB_TXN_NOSYNC, DB_TXN_SYNC, and DB_TXN_NOWAIT flags are passed to BDB.

On success, a unique identifier for the transaction stack (i.e., the transaction and all of its parents) is returned.

amino(COMMIT_TXN, *flags*) If the process has a current transaction, then it is committed. If there is no current transaction, then EINVAL is returned.

The *flags* parameter relates to shared transactions and can be 0, TXNOP_WAIT, or TXNOP_FORCE. If the transaction is not shared, it is always committed immediately. If the transaction is shared (i.e., the reference count is greater than one), then the action depends on the flag. If the flag is 0, then EBUSY is returned. If the flag is TXNOP_WAIT, the transaction is committed after all other users have detached from it. If the flag is TXNOP_FORCE, then the transaction is committed immediately and marked invalid so that other processes cannot use it.

amino(ABORT_TXN, *flags*) The ABORT_TXN operation is analogous to the COMMIT_TXN call, but aborts the transaction instead of committing it.

amino(ATTACH_TXN, *id*) If the process has no current transaction, then transaction stack with the identifier *id* is associated with the process and its reference count is increased. If the process has a current transaction, then EINVAL is returned. If no transaction stack has the identifier *id*, then ENOENT is returned.

amino(DETACH_TXN) If the reference count of the current transaction is greater than one, then decrement the transaction's reference count and disassociate this process from its the current transaction.

amino(SUSPEND_TXN) Disassociate the process's current transaction with the process. The transaction's reference count is not updated, and the transaction is added to this process's *suspended transaction list*. On process exit, the transaction's reference count is decremented. If it reaches zero, then the transaction is aborted.

amino(QUERY_TXN) Returns the current transaction identifier.

Figure 2.2: *Our transactions API has seven operations. Three operations begin and end transactions BEGIN_TXN, COMMIT_TXN, and ABORT_TXN. Three more operations allow sharing of transactions: DETACH_TXN, ATTACH_TXN, and SUSPEND_TXN. The QUERY_TXN operation returns the current transaction's identifier.*

tion. 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.[7] 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, we developed a profile to 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 [68]: 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.

2.5 Transactional Applications

We added transactional semantics to four applications: Postmark, GNU Make, GNU `tar`, and `mail.local`. Postmark is a benchmark that stresses meta-data operations [37], GNU Make is used to build software packages [16], GNU `tar` extracts files from an archive, and `mail.local` is the Sendmail component that delivers mail to a user's mailbox [77]. Our small modifications to Postmark, GNU Make, and GNU `tar` are described in Sections 2.5.1, 2.5.2, and 2.5.3, respectively. Our reimplementation of `mail.local` is described in Section 2.5.4.

2.5.1 Postmark

Postmark is a relatively simple benchmark (1,500 lines of code) that creates an initial pool of files, performs four types of operations on that pool, and then removes the pool. The four Postmark operations are:

Create Adds a new file to the pool and write data to it.

Unlink Removes a file from the pool.

Read Sequentially reads an entire file from the pool.

Append Appends data to an existing file in the pool.

[7] Although the API supports switching transactions, it is likely to be error-prone in our current prototype, because the dead lock detector does not know that the transactions are part of the same thread of control. In practice, we have used the suspend functionality to "hand off" transactions to child processes. We describe a method that could provide switching transactions in Section 7.1.4.

All four operations stress meta-data operations, and to a lesser degree the create, read, and append operations stress data operations. Because of Postmark's simplicity it is often used for file system benchmarking.

We modified Postmark 1.5 to use the Amino transactions API described in Section 2.4. Modifying Postmark to use transactions was a rather simple task, requiring only 33 lines of code. Postmark was already structured around the notion of a transaction, so all that was required was marking the beginning and end of the four transaction functions with Amino calls. Each of these functions takes the form:

```
if (amino(BEGIN_TXN, file_table[free_file].name, 0) == -1) {
   fprintf("begin: %s\n", strerror(errno));
   exit(1);
}

/* Function Code Goes Here. */

if (amino(COMMIT_TXN, 0) == -1) {
   fprintf("commit: %s\n", strerror(errno));
   exit(1);
}
```

As can be seen, beginning a transaction requires four lines and committing it takes another four. Therefore, each of the four functions requires eight lines each, for a total of 32 lines of code. Combined with a single line to include the `amino.h` header file, this means that Postmark required only 33 lines of code. Part of the reason that Postmark requires so few lines of code, is that it is never necessary to abort a transaction, because if a failure occurs it already exits.[8] Because Amino automatically aborts transactions on process exit, this results in simple code.

2.5.2 GNU Make

GNU Make is a popular software package that manages the dependencies of source files, intermediate files, and output files—typically for source code, object code, and executables. It is the core of building most Unix packages (e.g., OpenSSH, which we use as a compile benchmark in Section 5.5).

Unlike Postmark, Make does not read or write any of the files we are interested in transactionally protecting. Instead, it uses a series of rules that are defined in a `Makefile` to execute external programs that operate on these files (e.g., `gcc -c` to compile a file and then `ld` to link the object files).

Rather than modifying each of these external programs individually, we chose to modify Make such that it wraps each individual program in a transaction. If the external program succeeds (or if the Makefile directs make to ignore the error code), then we commit the transaction. If the program fails, then we abort the transaction. This is similar to the

[8]The original Postmark 1.5 does not exit, but rather silently ignores errors. We used a slightly modified version described in Section 5.4 that uses improved timing and exits on failure.

Lines of Code	Purpose
1	Include `amino.h`.
1	Include transaction ID in Make's child process data structure.
1	Verify that parent has no current transaction before `fork`.
19	Begin transaction in parent before `fork`.
4	Suspend transaction in parent after `fork`.
4	Attach to transaction after child completes `fork`.
4	Abort transaction on failure.
4	Commit transaction on success.
38	**Total Lines of Code for Make.**

Table 2.2: Our modified Make required 38 new lines of code to transparently provide transactions for child processes.

standard Make behavior, which deletes the targets after failure. The difference in behaviors is that an unmodified Make may leave temporary files of which it is unaware, but our modified Make undoes all changes.

To implement this Make extension required 38 lines of code as detailed in Table 2.2.[9] One line of code was required to include the Amino header, and another was used to add the transaction ID to Make's internal data structure that represents a child process.

About half of our code is in the `start_job_command` function, which is responsible for starting external programs and recursive instances of Make (called sub-make processes). We present a simplified version of this code in Figure 2.3 on page 21. We do not transaction protect sub-make processes, because they protect their own sub-processes. Before Make starts a new child process, we verify that there is no child transaction (1 line). If we have a transaction, this would indicate that our parent began a transaction, which is most likely due to a programming error in which we did not recognize that Make was recursively calling itself. Next, we determine Make's present working directory and begin a transaction on that Amino file system if we are not executing a sub-make (19 lines). At this point Make begins the child process using the `vfork` system call. The child process immediately executes the external program and is unmodified. The parent process suspends the current transaction and the child's job is started (4 lines).

The remainder of our code is in the `reap_children` function. We present a simplified version of the code in Figure 2.4 on page 22. After a child returns Make looks up its child data structure. If there is a transaction associated with the child (which is true, unless it was a submake), then we attach to that transaction (4 lines). If the child process failed and the error should not be ignored, then we abort the transaction (4 lines). Otherwise, we commit the transaction (4 lines). This has two effects: (1) if a build process fails, then we no leave stale temporary files; and (2) if the build process is durable, no synchronous writes are required until the external program completes.

[9]The number of lines inserted is actually 68. The number 38 omits comments and blank lines.

```
/* Determine what job is to be run, assemble argv, and
 * initialize stdin. */

/* Make itself should never have a transaction, because we
 * do not create one for recursive makes. */
assert(amino(QUERY_TXN) == 0);

/* If this is not a recursive make, then start a
 * transaction, using the pwd as the transactional file
 * system. */
if (!(child->flags & COMMANDS_RECURSE)) {
    pwd = getcwd(NULL, 0);
    child->txnid = amino(BEGIN_TXN, pwd, 0);
} else {
    child->txnid = 0;
}

child->pid = vfork();
if (child->pid == 0) {
    /* Execute child job, this never returns. */
}

/* We are the parent, so we suspend this transaction. */
if (child->txnid)
    amino(SUSPEND_TXN);
```

Figure 2.3: *A simplified version of the* `start_job_command` *function, which begins a transaction for the external program. We omit error handling and code that is not related to our modifications.*

```
/* Reap a child and determine its exit code. */

/* We attach to the child's transaction. */
if (c->txnid)
    amino(ATTACH_TXN);

if (child_failed && !c->noerror) {
    /* The commands failed, so delete targets. */
    if (c->txnid)
        amino(ABORT_TXN, TXNOP_WAIT);
} else {
    if (c->txnid)
        amino(COMMIT_TXN, TXNOP_WAIT);
}
```

Figure 2.4: *A simplified version of the `reap_children` function, which commits or aborts a transaction for the external program. We omit error handling and code that is not related to our modifications.*

Lines of Code	Purpose
1	Include `amino.h`.
5	Begin a transaction.
5	Function to abort a transaction.
5	Function to commit a transaction.
1	Extern declaration for abort function.
18	Commit/abort function calls (and required extra braces).
35	**Total Lines of Code for `tar`.**

Table 2.3: Our modified `tar` required 35 new lines of code to protect each extracted file.

2.5.3 GNU `tar`

We modified GNU `tar` to transactionally protect each file extracted. This prevents applications from leaving partially extracted files and from unsuccessfully overwriting existing files. In fact, GNU `tar` already includes 468 lines of code for managing backup files to restore the last overwritten file when extraction fails. Our solution is only 35 lines long and provides more robust error handling in the face of hardware and software errors. Table 2.3 describes the changes, which are to begin a transaction (5 lines), create functions for committing and ending that transaction (10 lines), and to call that function in the appropriate places (18 lines of which 8 are the function calls and the remainder are newly required braces).

2.5.4 `mail.local`

The `mail.local` program is the Sendmail component that is responsible for delivering mail to users on a local machine. This operation boils down to appending the data from

the standard input stream to a file specified on the command line (e.g., `/var/mail/cwright`). The `deliver` function of `mail.local` performs this append and is 451 lines long and has a McCabe cyclomatic complexity of 83 [44], which counts the number of control points.

Our transactional local mailer is 113 lines long. The delivery function is 72 lines long, and has a cyclomatic complexity of 14. This means that our mailer is six times less complex than the non-transactional equivalent. The delivery function uses two transactions. The first transaction is used to execute the `stat` and `open` system calls in isolation. The `stat` call verifies the owner of the file and its mode. The `open` system call opens the verified file, creating it if it does not exist. This transaction does not require durability, because if the file is not created the only update is the file's access time. If the file is created, then the only change is its existence as a zero-sized file. If the machine crashes before this update reaches the disk, then the file is recreated when the next message is delivered.

The second transaction protects the actual writing of the message to the mail file. This transaction provides atomicity, consistency, isolation, and durability. Atomicity is required so that either the entire message is written to the mailbox or none of it is. Isolation is required so that two concurrently delivered messages are not intermixed. Together these properties yield application-level consistency: that the mailbox is properly formatted series of complete messages. Finally, durability is required so that by the time the local mailer exits successfully, Sendmail can safely remove the message from its queue.

The mailer application would work correctly with a single transaction, but we chose to use two different transactions to provide improved concurrency. If we used one longer-lived transaction, then the long-held lock on the path database would reduce concurrency.

We handle deadlocks differently for the two transactions. For the first transaction, if a deadlock occurs we repeat the `stat` and `open`. The second transaction is more complicated because as the message is appended to the mail file, we consume it from the `stdin` stream. If the transaction fails mid-way, we have already consumed part of the message, and cannot roll back `stdin`. To solve this, we exit with the status `EX_TEMPFAIL`, which instructs Sendmail that the mail was not delivered, but that it should attempt delivery again. An alternative approach would have been to store the message in a deadlock free temporary file,[10] and then read from that file before sending the message. However, we chose to use the simple and convenient retry method built into Sendmail.

[10]For example, the temporary file could be stored on a non-transactional file system to ensure that it is deadlock free.

Chapter 3

Monitor Design

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 without source code modification.
- The interception technique should 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 spanning 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 six choices.

In-kernel file system The most direct approach would be to write a standard in-kernel file system. In-kernel file systems do not require relinking of binaries, and such file systems fit into the existing kernel architecture. They also have the advantage of running in kernel mode, so they 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 intertwined with caches. This means that substantial code changes are required to ensure coherency between the internal database caches and the external VFS caches. The second disadvantage is that all of the database code needs to be ported to the kernel and then executed within the kernel address space. Although this is not an insurmountable problem,[1] it introduces a code base into the kernel that is ten times larger than most existing file systems.

[1] In fact, we have previously ported a subset of BDB to the kernel [35, 36], and the lessons learned from that project have motivated this thesis.

FUSE FUSE or Filesystem in Userspace is a hybrid user-kernel approach [86]. 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. Finally, because FUSE file systems are limited to a strictly-defined VFS-like interface they have no notion of processes. Our design requires process information for current transactions and transaction sharing, which is another reason that FUSE is not appropriate for our system.

User-level NFS server toolkits A user-level NFS server toolkit, like the SFS-toolkit [43], 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. The NFS protocol also divorces the file system from processes and is not easily extensible for new transactional primitives.

`LD_PRELOAD` library Another option is to run our file system directly in the address space of user processes and intercept system-call wrappers using 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, statically linked binaries cannot use the file system, so they must be recompiled. Second, 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). Third, system calls that do not use the library wrappers are not intercepted, so not all code would work with this approach. Fourth, as the file system runs in the separate address space of each process, sharing data among them becomes more difficult.

Modified C library Directly modifying the C library is another option to extend new file-system functionality to applications [39]. The advantages and disadvantages are similar to the `LD_PRELOAD` mechanism, but high-level calls like `fprintf` do not need to be modified if the corresponding low-level library calls like `write` are handled correctly. Three additional disadvantages of using a modified C library instead of an `LD_PRELOAD` are that all applications must be relinked with the new C library, modifying the C library requires

significant implementation effort, and that circular dependencies would exist between the BDB library and the C 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` [25]. 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 3.1, 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. To enable the concurrent execution of system calls, the monitor uses a separate thread for each user-level process that is being traced. On Linux processes and threads almost identical, so the monitor can trace a multi-threaded application just as if it were tracing two separate processes.[2]

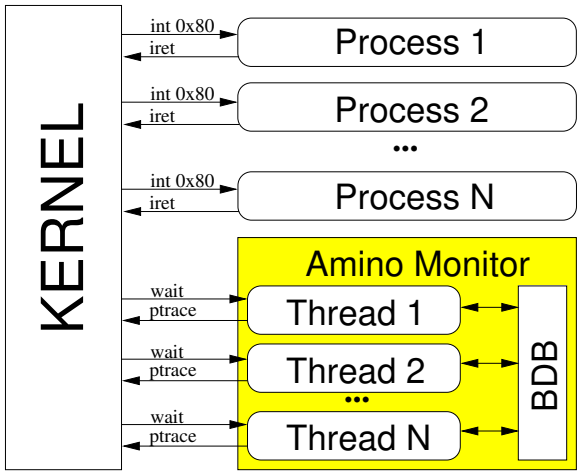


Figure 3.1: *The Amino monitor can trace an arbitrary number of processes. Each application process is traced by a separate thread within the monitor. At system call entry, the kernel signals the monitor via the* `wait` *system call. Amino manipulates the monitored processes' state with ptrace primitives. BDB executes within the monitor's address space and uses standard system calls.*

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

[2]This decision also has the convenient side-effect of causing each thread in the monitor that issues Berkeley DB operations to map exactly to a thread of control in the application. Without this property, it would be possible for undetectable deadlocks to occur in BDB, because BDB does not permit two outstanding transactions in the same thread of control.

Section 3.5 we discuss address space issues. Section 3.6 describes our `mmap` implementation and discusses conflicts between ACID semantics and the POSIX memory-mapping interface. Section 3.7 describes our `ptrace` enhancements, and Section 3.8 discusses limitations of our current prototype.

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 440 out of 14,227 lines of code reference 80x86 specific registers.

The monitor can also manipulate the registers with the `PTRACE_SETREGS` primitive. Before a system call executes, the monitor can change the call to be executed 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).

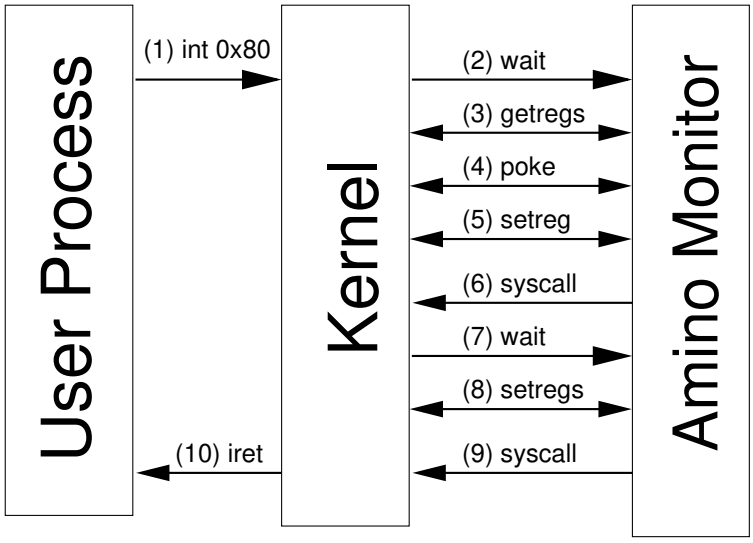


Figure 3.2: *`ptrace` primitives used to handle a `read` system call. Arrows indicate control transfer. Double arrows indicate that the function was called and returned immediately.*

Figure 3.2 shows an example of how the monitor handles a `read` system call destined for an Amino 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. 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 3.5.1).

5. 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 associated with –1.

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 three most common states are INUSER, INCALL, and INFORCERET, which indicate that the process is executing user-level code, the kernel is executing a system call, or the monitor is emulating 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).

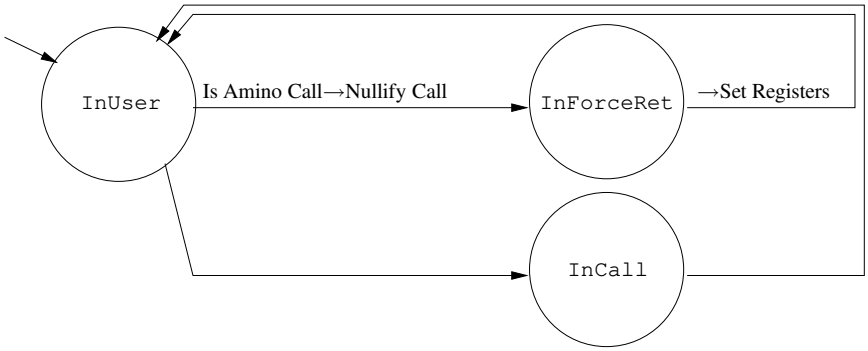


Figure 3.3: *The monitor has 17 states that determine its actions after* `wait` *returns. This diagram shows three of those states:* INUSER*,* INCALL*, and* INFORCERET*. The edges are labeled with implications. If the antecedent is true, then the edge is followed after the actions in the consequent are performed. Unlabeled edges are followed unconditionally.*

After returning from `wait`, Amino retrieves the current process's state and performs an appropriate action. A simplified State diagram is shown in Figure 3.3. Each edge is labeled with one or more implications, which are evaluated from top to bottom. If the antecedent is true, then the edge is followed after performing the actions in the consequent. If the antecedent is empty (or the edge is unlabeled), then the edge is followed unconditionally. If the consequent is empty (or the edge is unlabeled) no action is performed. After each transition the user-level program's execution is resumed. We can see that the monitor begins in the INUSER state, because the user-level program begins by executing user-level code. After a trap into the monitor, the process's registers are examined, and if the call is not destined for Amino, the monitor transitions into the INCALL state. After the call completes, the monitor transitions back to INUSER and is ready to service the next system call. If the

call is destined for Amino, then the monitor nullifies the system call (so that the kernel does not service it) and transitions into INFORCERET. After the nullified call completes, the monitor sets the processes registers to the appropriate return value and then returns to INUSER. 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 8, Amino looks up the state and because it is INFORCERET, Amino sets the value of `eax` to the proper return value. To complete the call, Amino sets the state to INUSER and issues `PTRACE_SYSCALL` (step 9).

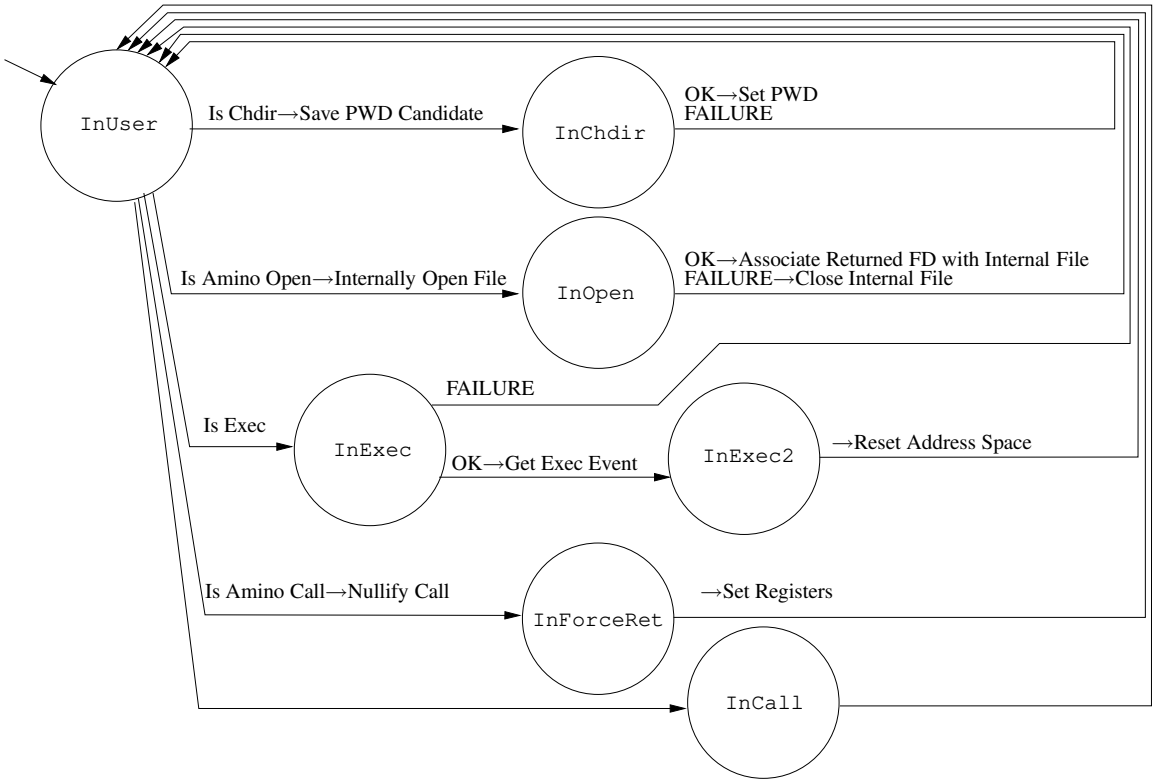


Figure 3.4: *Most of the 17 monitor states are for specific calls. Using the same notation as Figure 3.3, this state diagram shows the transitions for `open`, `chdir`, and `exec`.*

There are currently 17 states, most of the states indicate that the user process is in the midst of a specific call, for example `chdir`, `open`, or `exec`. Figure 3.4 shows the state transitions for these three calls. Before the `chdir` call is executed, the monitor saves the present working directory candidate (i.e., the directory into which the application wants to change). If the `chdir` call is successful, then the monitor sets the process's PWD to the saved value. On either success or failure, the monitor then returns the process to the INUSER state. An Amino open call proceeds similarly to `chdir`, first the monitor internally opens the file and transitions to the INOPEN state. If the kernel cannot reserve a file descriptor for this file, then the file is closed. If the kernel can reserve a file descriptor for this file, then the file is associated with that file descriptor. The `exec` call is slightly more complicated than `chdir`; `exec` requires two states: INEXEC and INEXEC2. First, if the system call being executed is `exec`, the process transitions to the INEXEC state. If

the `exec` call fails, then the process returns to the INUSER state. Otherwise, the monitor transitions verifies that it received an `exec ptrace` event, and then transitions to the INEXEC2. After the INEXEC2 state, the `exec` system call was successful and the monitor resets address-space-specific information about the process (e.g., shared memory regions are destroyed by `exec`).

Three other states of note are REDOCALL, RESTOREREGS, and INFRCEXEC. REDOCALL indicates that the current system call should be repeated, and RESTOREREGS 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). The INFRCEXEC overrides the return value as is done in INFORCERET, but the original system call is executed.

The DOCONT and FORKCONT states indicates that the process should be continued on the next entry. The INSHMAT state is used for shared memory attachment, which is described in Section 3.5.2. The remaining five states are used to handle the `dup`, `mmap`, `mremap`, `clone` calls (clone requires two states).

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; in case of a software error, most calls on this file descriptor fail with `EISDIR` (because "`/`" is a directory). For efficiency, Amino reuses this file descriptor with `dup` on subsequent `open` calls.

3.4 Mount Subsystem

The Amino monitor must maintain 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, our monitor supports Amino mounts that take the BDB database pathname as an argument, pass-through and AES mounts that take a directory to stack over as an argument, and ISO mounts which take an ISO image as an argument. When the monitor encounters a system call that references one of these paths, it passes the call to the appropriate file system's routine.

This architecture is somewhat similar to the prefix table in Sprite [58]. In Sprite, a dynamic table of path name prefixes was used to resolve servers. This is somewhat different than the classical vnode interface, which does not rely on an explicit table of mounts for path name resolution, but rather has a vnode field that points to the root of a new file system if the vnode is a mount point. The static mount table in Amino, however, is less complex than the Sprite prefix table, which dynamically changes based on the state of file servers.

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.

Figure 3.5 (page 33) is a flowchart of how the monitor resolves path names that are passed to a system call. First, the monitor creates an empty path stack. Next it checks whether the path begins with a "`/`." If not this means the path is a relative path, so the process's PWD is copied to a new stack. 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 depth less 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 a mount 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.

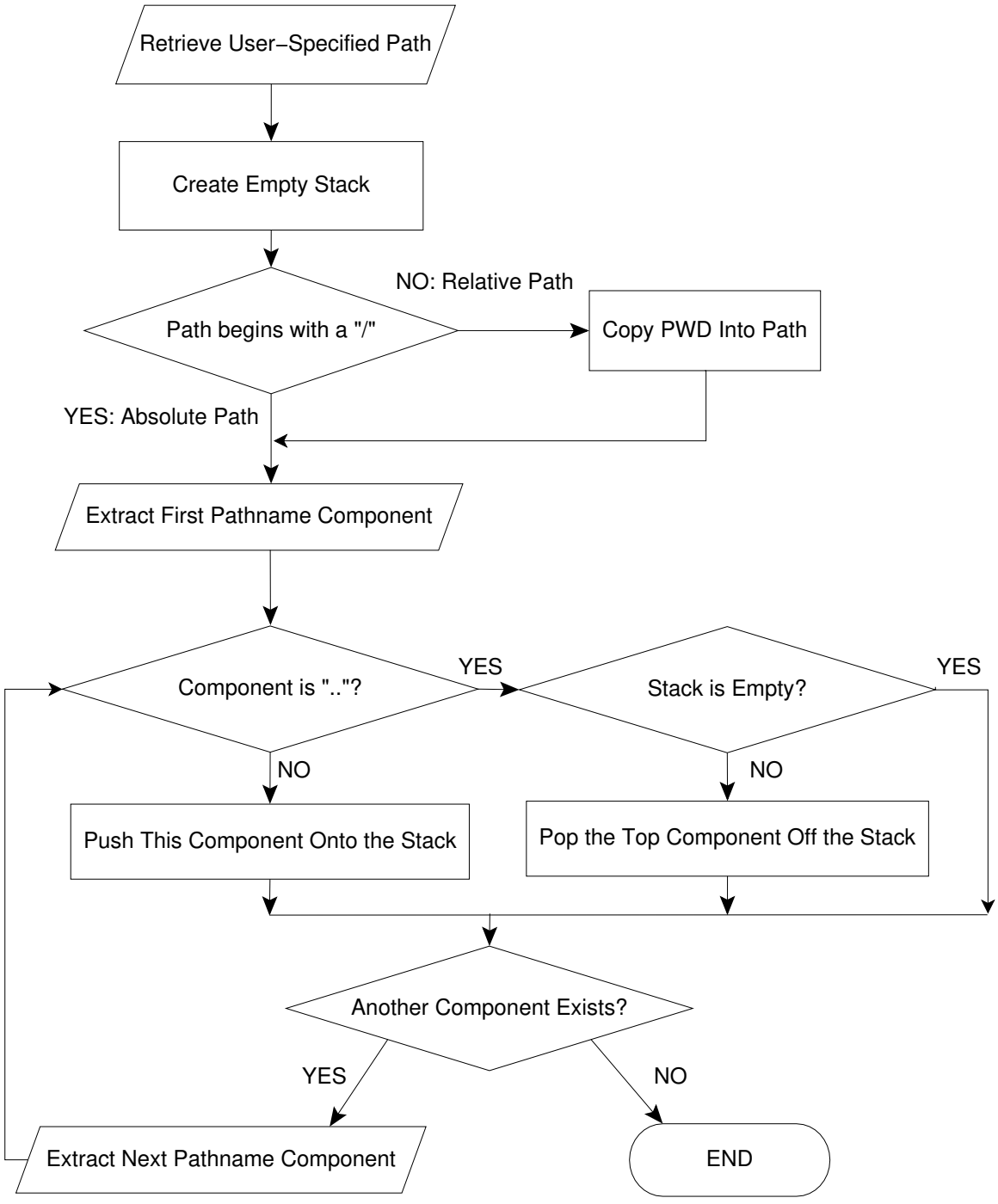


Figure 3.5: *The monitor uses stacks to represent path name components. When a new component is encountered, it is pushed onto the stack. When a "..." is encountered a component is popped off the stack.*

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. In Section 3.5.1 we discuss accessing the user process's memory. In Section 3.5.2 we discuss rewriting system call arguments.

3.5.1 Accessing User Memory

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 enabled a writable `/proc/pid/mem` to allow bi-directional bulk transfers. If the `/proc/pid/mem` interface is not available for reading or writing, then Amino falls back to `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`.

To further improve on this interface, we created a kernel patch that allows the monitor to map arbitrary regions of the user process's address space into its own. After a mapping is established, the monitor no longer needs to rely on system calls to transfer data to and from the user process instead using standard memory access. These mappings have two key performance advantages over the existing interfaces: (1) the data does not need to be copied from the user-process to the kernel before being copied to the monitor (or vice versa), (2) fewer system calls are required because there is no artificial limit on transfer sizes.

Using the rather simple kernel component that establishes mappings, the monitor tracks which mappings are established between itself and the user-level process using a multi-level page table. When those mappings become invalid (e.g., through successful `brk`, `mmap`, `munmap`, `exec`, or `fork` calls), then the monitor removes the invalidated mappings.

Using the more flexible memory-mapped interface automatically eliminates one data copy, but we realized further reductions by changing the monitor. For example, without the memory-mapped `/proc/pid/mem` interface, before invoking Amino's `read` operation the monitor creates a temporary buffer that is large enough for the result. Amino then reads the data into that buffer (one data copy is required to transfer data from the BDB cache into the buffer), and it is copied to the user application (requiring a two data copies, one to transfer data to the kernel and a second to transfer data to the application). When a memory-mapped `/proc/pid/mem` is available, then Amino reads directly into the user process's address space—requiring only one data copy (the one from BDB's cache into the user application). Though it is in theory possible to remove this copy without altering the `ptrace` interface it is difficult in practice for two reasons. First, BDB provides support for reading data into an arbitrary buffer provided by the user or a buffer allocated using the OS's `malloc`. Neither of these methods is suitable for zero-copy reads using the existing `ptrace` interface. To provide zero-copy reads, a new BDB method must be added that returns the address of data in the cache. The second reason the existing ptrace interface

makes zero-copy reads problematic is that the core of Amino's file system operations would need knowledge of `ptrace` and its limitations (e.g., only one 4,096-byte aligned page can be written to). It is certainly possible to add this knowledge to Amino file system operations, but it breaks the abstraction between the core of the monitor and the file system, increasing the complexity of both.

3.5.2 Rewriting System Call Arguments

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 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. The monitor writes the new file name into its own address space and updates the child's registers to point to the shared memory region in the child's address space. After the call, the child's original registers are restored. 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.

Establishing the shared memory region requires inserting a new system call into the process's system call stream. The state transitions used for this procedure are shown in Figure 3.6 (page 36) using the notation of Figure 3.3 (page 29). Amino calls do not require rewriting the file name, because the kernel does not execute them, therefore they are unchanged by the introduction of the shared memory region.[3] When the first system call is issued with an argument that needs to be rewritten, the shared memory region is not yet established. To establish the shared memory region the following steps are taken:

1. The monitor creates a System V shared memory region and attaches to it in its own address space.

2. From user applications, the `shmat` call attaches to a shared region memory region, but on Linux `shmat` is implemented in terms of a larger `ipc` system call. The `ipc` system call returns the address that a shared-memory region is attached by updating a pointer in the process's address space. This means that to establish the region, we must store this return address somewhere within the child's address space.

   The monitor locates the first private writable address within the process for this return value.

3. The contents of the process's address space in which `ipc` will write its return value are saved.

4. The process's registers are changed to perform the `ipc` call by setting `eax` to `__NR_ipc` and the other registers to the appropriate arguments.

[3]The `open` call actually rewrites the filename to `/`, but we omit this detail for clarity. Rewriting a path for an Amino call works exactly like rewriting a path for a kernel call.

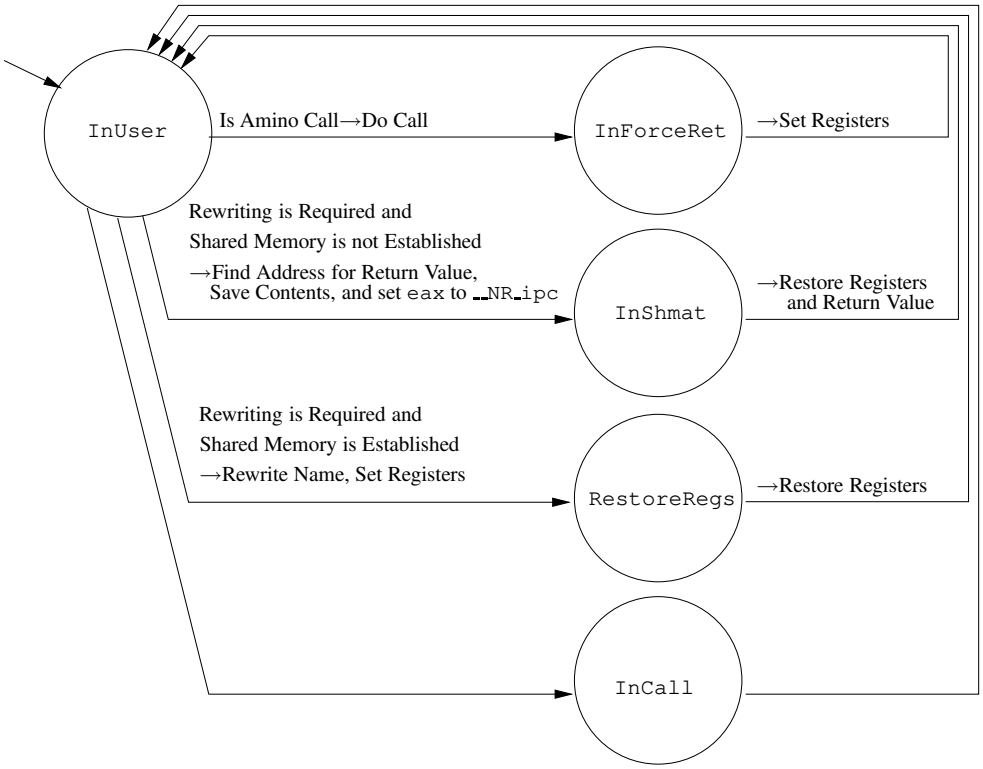


Figure 3.6: *Using the same notation as Figure 3.3, this state diagram shows the transitions for rewriting file name arguments to system calls. Amino calls do not require rewriting, so they are unaffected. If a shared memory region is already established, then it is used to rewrite the file name. If rewriting is required, but the shared memory region is not yet established the monitor establishes the region and restores the process's registers including the instruction pointer (EIP). After transitioning back to the* INUSER *state, the process attempts to execute the original system call and can use the newly established region.*

5. The state is set to INSHMAT, and the process continues.

After the `ipc` system call completes, control is returned to the monitor. Because the process is in the INSHMAT state, the monitor stores the value returned by `ipc` in the PCB. This address is later used to update the process's registers when rewriting file names. The process's original registers and the contents of the overwritten memory are restored, and the process is allowed to continue. Because the instruction pointer (`eip`) register was restored along with the others, the user process re-executes the original system call.

When re-executing the original system call, the shared memory region has already been established. The file name is rewritten into the monitor's shared memory segment, and the user process's registers are updated to point to its copy of the shared memory segment. The process's state is set to RESTOREREGS, and it is allowed to execute the call. Upon return, the monitor restores the registers to their original value. Arguments to subsequent system calls can be rewritten by simply updating the local region and the child's registers.

3.6 Memory-Mapped Operations

Many applications (e.g., linkers) take advantage of memory-mapped operations, which allow access to the file system through an efficient memory-like interface. Therefore, supporting `mmap` is essential to providing good compatibility for POSIX applications. The monitor provides support for memory-mapped operations by intercepting the `mmap` system call and any `SIGSEGV` signals that are delivered to a monitored process.

Upon intercepting an `mmap` system call destined for one if its file systems, the monitor behaves much like an OS kernel: it establishes an empty region and services page faults for that region. To create the empty region, the monitor converts the process's `mmap` system call to an anonymous region that the process is not allowed to read or write. The monitor also records the address of the memory-mapped region and its backing file in the process's PCB using a multi-level page table structure. The multi-level page table structure, which is identical to the structure used by most OSs, allows the monitor to find the region corresponding to a given address using two table lookups. The structure is also efficient in terms of space, requiring only 4MB to address 4GB of memory.[4]

Normally, the OS handles the page faults through a hardware trap triggered by the MMU. The monitor handles page faults through a software trap. Whenever an application accesses an invalid page (either because it does not have permission or the page does not exist), the OS sends it a `SIGSEGV` signal. Before a monitored application receives the signal, the monitor examines the signal information including the address that faulted. If the monitor finds a region corresponding to the address, then the monitor reads the page into the process's address space. Next, the monitor issues an `mprotect` system call in the context of the application to allow the process to read the page. If the address is not found, then a `SIGSEGV` signal is delivered to the process, usually resulting in a core dump.

Memory-mapped files can be either private or shared. For shared mappings, writes to the memory region are reflected in the file, whereas for private mappings writes are not

[4]In practice, the amount of memory required is smaller. As only Amino mappings are tracked, the vast majority of the structure is sparse. Moreover, memory mappings are clustered together so if a second-level table is required, then it is likely used more than once.

reflected in the backing file. For private mappings, only one page fault is required—the one that initially brought the page into memory both reads it and marks it as writable (as any writes to the region are discarded by the file system, so tracking them is not required). However, for shared mappings the system must keep track of which pages in a region have been written to, so that they can be written to disk. When a shared page that was already read into the process's address space is written to, a second page fault is generated. The monitor marks the page's state as dirty. It then allows the process to change the page (using `mprotect`). Unfortunately, the signal information structure informs us only that an access violation occurred; it does not inform us whether the requested access was a read or write. If we had this extra piece of information, we could reduce the number of traps into the monitor that are required for memory-mapped writes, because if a region is written to without being read, then we would not need the first fault.

On `munmap` or `msync`, the monitor writes dirty pages to the backing file. Although the monitor does not currently write dirty pages in other circumstances, it would be possible to create a separate thread for flushing dirty pages (analogous to Linux's `pdflush` or FreeBSD's `vm_pageout`).

On closer inspection, the POSIX memory-mapping interface, particularly for shared mappings, is fundamentally at odds with ACID semantics:

- User-level applications map regions, which can be thought of as a file-level operation. After the mapping is established, traditional load/store instructions (or instructions that reference memory) operate on bytes or words. The operating system on the other hand uses a fundamentally different abstraction for memory-mappings—pages. Ideally, the operating system would provide memory-mapped semantics that are defined to the level of bytes, but hardware limitations essentially dictate the page is the smallest unit which can be used for memory-mapping efficiently.

- The question of when a write should be acknowledged and propagated, makes transparent ACID semantics difficult. For system calls, it is clear when the OS should write data to disk, because they are executed at a single point in time. However, for memory-mappings the application can write data over a long period of time and there is no commonly used signal to indicate that a write operation is completed. In our implementation we use the `munmap` and `msync` system calls, but this method has three drawbacks. First, `munmap` and `msync` are not required. Second, `msync` is a rarely used system call (e.g., we did not observe any instances of it during any of our benchmarks).[5] Third, by not propagating writes until the `munmap` call the system has the potential run out of memory and swap when it otherwise would not. Normally, the OS would periodically flush these writes according to a timer, or when memory is running low. However, if writes are not propagated until `munmap` or `msync`, then the outstanding writes must be kept in virtual memory until they are explicitly flushed.

- The expected semantics of shared mappings, fundamentally propagate unisolated

[5]The `msync` call is also comparatively rare in source code. In the Fedora Core 5 source packages, the `mmap` and `munmap` calls occur on 4,031 and 1,487 lines, respectively. However, `msync` occurs only 192 times (i.e., 7.7 times less than `munmap`).

writes. There are two components to shared-mapping semantics. First, the changes made to a shared mapping are reflected in the backing file. Second, the changes made to a shared mapping are immediately visible by other processes that are mapping the file using a shared mapping (or reading from the file). This second condition, is fundamentally at odds with isolation, because writes are visible by other transactions before the transaction commits. Moreover, causing these applications to block for locking changes the expected semantics in such a way that applications are likely to break (e.g., applications can use shared memory for explicit synchronization).

- Implementation artifacts introduce several undesirable properties. For private mappings, it is unspecified whether changes to the file after the `mmap` system call are visible in the mapped region. This is more than just a gap in the specification, whether updates are visible actually depends on a specific execution of processes. If the page is being accessed for the first time, then it reflects updates; otherwise it does not. Clearly, undefined or ill-defined behavior is at odds with transactional semantics.

Given this conflict, our current design falls closer to a standard POSIX OS: memory-mapped reads take place on the first read from a page; memory-mapped writes take place on `msync` or `munmap`; and we do not attempt to provide isolation for memory-mapped accesses after the initial access.

3.7 `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.

Therefore, 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 (UML) [12]. 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 [20]. Most of the primitives are the same as those described in Section 3.1, but steps 6–9 are replaced with a single operation:

6. The monitor instructs the kernel to skip executing this system call and immediately return to user space.

Aside from reducing the number of ptrace operations and context switches, reducing the number of traps decreases CPU time and other unnecessary operations (e.g., the monitor checks for certain conditions and gets the process's registers after a wait call).

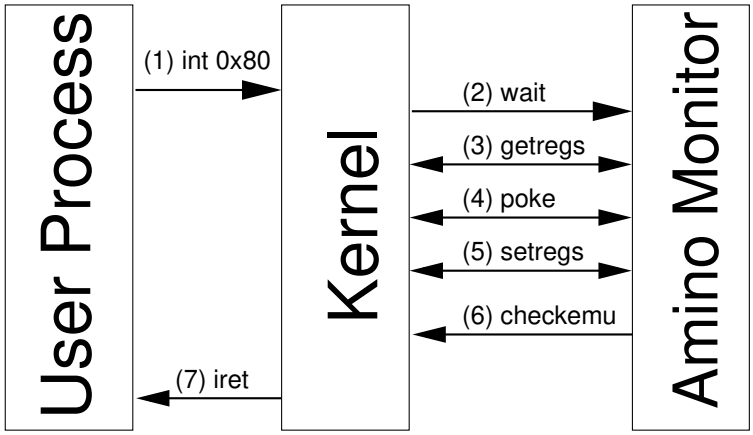


Figure 3.7: *The PTRACE_CHECKEMU call reduces the number of ptrace primitives required for servicing a system call compared to the standard ptrace primitives shown in Figure 3.2.*

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.

In Section 3.2, Figure 3.3 (page 29) showed the states that were required for traditional ptrace primitives. PTRACE_CHECKEMU removes the need for the INFORCERET state and PTRACE_SYSSKIP removes the need for the INCALL state. Figure 3.8 shows the simpler state transition diagram, when these primitives are used. Because these two states are eliminated, fewer transitions are needed, thus reducing 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). Other operating systems, such as Solaris, already provide this functionality, but Linux did not. 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%.

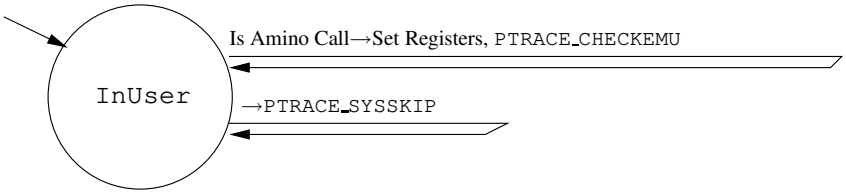


Figure 3.8: *Figure 3.3 assumes the standard* `ptrace` *interfaces. Our new primitives reduce the number of states and transitions needed for many calls.* `PTRACE_CHECKEMU` *removes the need for the* INFORCERET *state and* `PTRACE_SYSSKIP` *removes the need for the* INCALL *state.*

These three improvements can benefit a 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.

3.8 Implementation limitations

Our monitor implementation provides a sound framework for exploring ACID file system development, yet there are four limitations that prevent its use as a production system: (1) no permissions checking, (2) non-standard shared mapping semantics, (3) lack of invalidation for memory-mapped access to `/proc/pid/mem`, (4) and no symbolic link support. None of these limitations are fundamental to the design of our system, but are rather unimplemented features.

First, although Amino stores ownership and permissions information it does not enforce it for file system operations. The existing OS cannot provide enforcement, because all of the files in an Amino file system are stored together on the lower-level file system without regard to the owner. Providing enforcement in the monitor is conceptually rather straightforward. The lower-level Amino files could be owned by a designated user, and the monitor would be a set-UID executable that runs as the user owns the files. After opening the database files, the monitor could drop its escalated privileges and apply the appropriate permissions based on the monitored process's UID and GID. Of course, as with any set-UID program, additional steps must be taken to minimize possible security vulnerabilities.

Second, our current implementation of shared memory-mappings does provide the expected write-back semantics of shared mappings, but does not provide the sharing aspect. Each independent shared mapping is treated as a separate unit, and changes made to that mapping are not reflected by `read` system calls or in other shared mappings until `msync` or `munmap` is called. Even then, previously mapped regions are not updated (unless that page has never been read). Traditional POSIX semantics could be delivered in the follow-

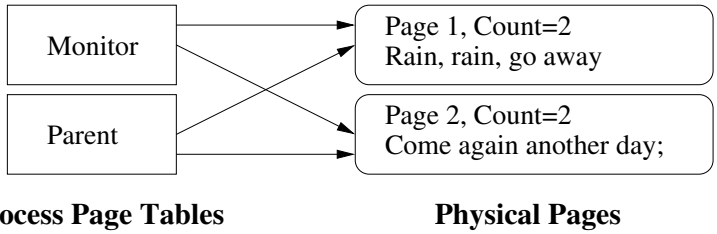
ing manner (as to whether that is desirable see Section 3.6). Each page of a file would be allowed to exist in only a single shared mapping at any instant in time.[6] If a page is to be loaded into a mapping, it is first invalidated in all other mappings and, if necessary, written to the database. Before processes are read the file using `read` system calls, the mapping would be similarly invalidated. This architecture would be quite similar to cache coherency in UCLA's architecture for stackable file systems [28, 29]. As an optimization, instead of using multiple anonymous regions and invalidating pages, we could use a temporary file as a backing store for our shared memory-mapped regions. The underlying OS page cache, would provide the expected sharing semantics in memory, and modified pages could be written out before `read` system calls.

Third, the kernel patch that provides memory-mapped access to `/proc/pid/mem` is suitable for our prototype, but is not suitable for a production system. Specifically, the kernel does not invalidate the mapping's pages after `exec`, `munmap`, `mremap`, `brk`, and `fork`.[7] This can lead to unexpected results. For example, a successful `exec` system call replaces the address space of the process, but if an address was already read by the monitor, then it may access stale data. Similarly, `munmap` and `mremap` can remove data from a process's address space. The `fork` system call is slightly more subtle, but can also cause inconsistencies as shown in Figure 3.9. Though `fork` does not directly change the process's address space, it does mark it as copy-on-write. If the parent process writes to a page before the child, then a new page is created in the parent process's address space. However, the monitor's mapping still references the old page, which is in the address space of the child (thus yielding the data from the child's address space and not the parents). If an application does not correctly manage its mappings, it is possible to read from or write to process's that it should not (e.g., after executing a set-UID program). Therefore, for production use the kernel must properly invalidate pages from these mappings.

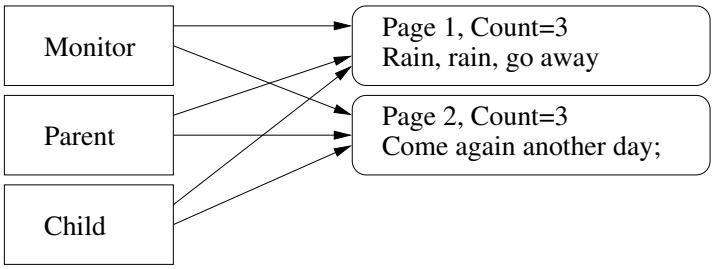
Fourth, Amino does not currently support symbolic links. To add symbolic links, the pathname resolution algorithm described in Section 3.4 must consult the file system while considering each path name component. If that component is a symbolic link, then the link should be read and resolved.

[6] This requirement can be relaxed such that a read-only copy of a page may exist in multiple mappings, but a read-write copy of a page must be the only copy of that page.

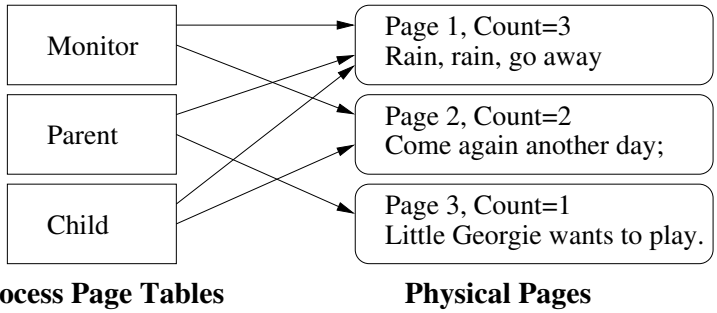
[7] The monitor does perform the proper invalidations from user-level with `munmap`. Properly written applications, can thus access the process's memory in a consistent manner.



(a) Page tables before the `fork` system call. The monitor's page tables point to the pages of the user-level process (labeled "Parent").



(b) After `fork` the Parent and Child process share physical pages.



(c) When the parent writes to page 2, a copy of the page is made and the parent can change it. The monitor points still points to page 2 (which is mapped by the child), thus causing an inconsistency.

Figure 3.9: *Our kernel patch does not currently handled copy-on-write pages after a* `fork`*. In a production system, the kernel should invalidate pages that the monitor maps after a* `fork` *to prevent an inconsistent view of the parent process.*

Chapter 4

File System Complexity

We identified three main advantages of using a `ptrace`-based infrastructure for user-level file system development as opposed to porting code to the kernel:

Avoid kernel changes Our primary reason for using `ptrace` instead of a direct kernel file system is that `ptrace` allows us to bypass VFS caches without changing large portions of the core kernel. This is essential for Amino, as interjecting VFS caches between the application and the database would break both the atomicity and isolation properties. The existing stable of VFSs (FreeBSD, Linux, and Solaris) are all intimately intertwined with caches—a cached object must exist before methods can even be invoked. To separate any of these VFSs from its caches requires significant code changes not only to the VFS, but also to any existing file systems.

Leverage user-level libraries One of our initial observations was that developing a transaction processing system is a huge task in its own right, as evidenced by the size of various transaction processing systems [53, 82]. Utilizing an existing transaction processing library (BDB) allowed us to develop Amino much more quickly than we could have otherwise.

If we were to have developed Amino in the kernel, we first would have to port BDB to the kernel, test it, and still it may not be entirely suitable for use in the kernel. For example, BDB expects to use several large contiguous segments of memory for its cache, but in the Linux kernel there is only a small amount of address space set aside for these types of large allocations (less than 100MB). Worse yet, this address space must be shared among all of BDB's components (e.g., the logging subsystem buffers data before writing it out) and even other kernel modules. Though this example is Linux specific, porting BDB to any OS kernel is likely to run into similar problems.

Simplified debugging Kernel debugging is rather difficult. Some kernel debuggers are available [34], but more often developers choose to insert print statements. These problems, combined with lengthy reboot cycles and the raw complexity of any large OS kernel introduce debugging difficulties that almost never occur in user space. Conversely, developers have a wide variety of good debugging techniques at their disposal in user space [15, 60, 79].

All of these factors contributed to allowing us to develop Amino rather quickly, without a large team of programmers. Moreover, implementing new file systems at the system-call-level has advantages over implementing them using the VFS interface. For example, an encryption file system has more control over a process's memory when implemented at the system call level (e.g., it could insert `mlock` calls to prevent data from reaching an unencrypted swap partition [62]).

Our monitor includes a simple Virtual File System (also known as a File System Switch or VFS). Our VFS is different from others in four ways [90]:

- To varying degrees, existing VFSs perform significant functionality before passing operations down to the file system, thus preventing the file system from changing its behavior. Our hierarchy of methods begins at the system call layer (the highest one available to an OS, or the monitor). This allows file systems to replace the monitor's default behavior from system call entry to exit.

- Existing VFSs struggle with whether to include functionality in the VFS, or to push it down to individual file systems. Our solution to this problem is to provide operations at multiple levels of abstraction. For example, the generic system call methods for `read`, `readv`, and `pread` all call an internal read method. This allows simpler file systems to implement a single read operation, but more complex file systems can individually implement `read`, `readv` and `pread` if the need arises.

- Our architecture is designed such that file systems can be layered with additional function calls only for methods that the layered file system implements.

- Our VFS includes event notifications for various process related events: `exec`, `fork`, and `exit`.

Appendix B describes each method that our VFS provides.

To demonstrate the simplicity of developing file systems using our monitoring framework, we wrote three simple user-space file systems. The major difference between these file systems and Amino is that they are smaller in scope and can be designed with other techniques such as FUSE or as an in-kernel file system. In Section 4.1 we describe a simple pass-through layer that handles file system operations by passing them down to another directory. This pass-through layer serves as the basis for our AES encryption file system described in Section 4.2. In Section 4.3 we describe a user-level ISO file system, which allows users to browse CD-ROM images. In Section 4.4 we evaluate the complexity of these file systems.

4.1 Pass-Through Layer

We developed a simple pass-through file system layer for two reasons. First, it serves as an example for other file system extensions. We developed it in such a way that its operations could be reused for other file systems (e.g., the encryption file system described in

Section 4.2). Second, it provides a suitable basis for evaluating Amino's overhead (Section 5). The pass-through file system takes a single mount-time argument: the name of the directory to which operations should be redirected.

The pass-through file system implements 21 operations, 17 of which are simple wrappers around another system call. It also defines two new operations: `encodename` and `decodename`. File systems built on top of this pass-through layer can override these operations to manipulate file names. These methods translate upper-level file names to the corresponding lower-level names (and vice versa). A representative method of the pass-through file system is `unlink` (shown in Figure 4.1), which has only three function calls: (1) the argument is converted to a lower-level name using `encodename`; (2) the lower-level name is unlinked; and (3) the lower-level name is freed.

```
static int null_unlink(struct pcb * pcb) {
  char *fullpath = NULL;
  int ret;

  ret = pcb->cur_mount->ops->encodename(pcb,
                                        pcb->cur_mount,
                                        pcb->cur_filename,
                                        &fullpath)));
  if (ret)
    return ret;

  if ((ret = unlink(fullpath)) != 0)
    ret = -errno;

  free(fullpath);
  return ret;
}
```

Figure 4.1: *The `unlink` method for the pass-through file system layer.*

The `read`, `write`, and `lseek` methods are similar to the system-call-based wrappers, but take internal monitor objects (i.e., mount and open file structures) as arguments instead of operating at the ABI level. This allows the methods to be re-used for many types of system calls (e.g., the `read` operation is used for both the `read` and `readv` system calls in addition to memory-mapped reads). The last two methods are `open` and `close`, both of which wrap underlying system calls and manage monitor state (e.g., the open file structure).

4.2 AES Encryption Layer

We have developed an AES [54] encryption file system on top of the pass-through layer described in Section 4.1. This encryption layer allows users to encrypt the contents of a directory, thereby preventing a breach of confidentiality if the hard disk is stolen.

4.2.1 Encryption scheme

Our file system layer encrypts both file names and file data. However, to simplify development and administration, we chose to preserve the existing structure of files by encrypting each file separately for four reasons: (1) users are used to dealing with a traditionally organized file hierarchy [6, 89]. (2) backing up a subset of encrypted files is simple; (3) when an encrypted file is deleted, the space is immediately reclaimed; and (4) there is no need to preallocate space for encrypted volumes. This convenience, however, comes at the expense of revealing some information about the structure of the files (e.g., how many files exist in a given directory and their size). Several systems have made theses choice, including CFS [6], NCryptfs [89], and eCryptfs [27]. The data and names in our system are encrypted using a key that is read with `getpass` on startup.

We use a separate scheme for file name encryption and data encryption. For file names, we must encrypt the parent directory name and a name within that directory. Each parent directory has an associated initialization vector (IV), which means that a file with the same name in two different directories does not encrypt to the same text. We chose to use the AES-CBC mode to encrypt file names. This has the disadvantage of causing the file name's length to be rounded up to the nearest cipher block size (16-bytes), but it is more secure: more malleable cipher modes (e.g., CFB and CTR) are inappropriate because they do not permit the reuse of an IV for different cipher texts. After the file name is encrypted, it is base-64 encoded so that illegal (i.e., "`/`" and "`\0`") or control characters, which can disrupt the user's terminal and confuse utilities, are not written to the file system. Before encryption, we include a CRC that is checked on decryption. This CRC allows us to detect files that were encrypted with a different key (or unencrypted files) and omit them from the directory listing.

For data we need a scheme that has four properties:

- Two different files with the same plaintext have different cipher text. This means that different files should have different IVs.
- Two different regions of the same file that contain the same plaintext have different cipher text. This means that the ciphertext should be dependent on the position in the file.
- We can rewrite regions of the file with the same IV. This means that we cannot use more malleable modes of encryption such as CFB or CTR, because an attacker who reads the ciphertext before and after an update could recover the plaintext.
- Random access has a constant penalty, so we cannot use a chaining mode over the whole file.

The scheme we developed, inspired by Blaze's OFB/ECB hybrid [6], is a hybrid of AES-CTR and ECB mode that satisfies each of these properties. This scheme has the advantage over Blaze's that there is no need to store precomputed data; it supports arbitrarily large files; and we use a distinct random stream for each file.

Blaze generated a large random file using OFB mode. To encrypt a block of data, it is first XORed with the data in the same position in the random file. If a file to be encrypted

is larger than the random data, then the position is taken modulo the random file size. The result is then encrypted with ECB mode. In sum, Blaze's scheme is:

$$C_i = ECB_K(P_i \oplus R_{i \bmod s}) \tag{4.1}$$

Where C_i is the i-th block of cipher text. P_i is the i-th block of plaintext data, R_i is the i-th block of random data, K is the key, and s is the size of the random data. This scheme allows random access and only one cipher block ever needs to be reencrypted.

We adapted this scheme to use AES-CTR. AES-CTR essentially generates a stream of random bits based on an IV as follows:

$$R_i = ECB_K(f(i)) \tag{4.2}$$

To encrypt data at position i, it is XORed with R_i and a per-file IV. We used this method to generate an arbitrary-length random stream in place of the precomputed random data. For the IV, we use a 128-bit nonce that is generated when the file is created. The scheme is then similar to Blaze's method:

$$C_i = ECB_K(P_i \oplus n \oplus ECB_K(f(i))) \tag{4.3}$$

using the previous notation where n is the per-file nonce. This scheme has the advantage over Blaze's that there is no need to store precomputed data, and it supports arbitrarily large files. Though this scheme does not require precomputed data, the values of R_i can be precomputed and stored in memory, thus eliminating half of the ECB encryption operations for those precomputed values of R_i. This allows the user to trade off increased memory utilization for decreased CPU utilization.

4.2.2 Extended attributes

For each encrypted file we must store two pieces of information: its initialization vector and its actual size, because the file's size is rounded up to the nearest cipher block size. In both cases we use the extended attribute API supported by Ext2, Ext3, Reiserfs, and many other file systems [24].[1]

Extended attributes are not the only solution, but they are well suited for storing small amounts of per-file data. Other possible solutions we considered are:

- Storing the data in the file itself. We did not store this extra data inside of the file, because it is only 24 bytes long. If we stored it in a block aligned manner, we would need to waste a full block. If we did not store it in a block aligned manner, we would change the file's expected performance characteristics (because applications expect data to be aligned according to disk blocks).

- Storing the data in a per-mount auxiliary file. Storing the data in an auxiliary file would require additional code to organize the data efficiently. Moreover, modern file systems store extended attributes within the file's inode, which must be updated whenever these attributes are updated.

[1] In our previous work, we have developed a stackable file system that adds extended attribute support to any existing file system using a BDB database.

- We could also have used a self-describing padding scheme for the last block, but chose not to because finding file size by opening the file would hurt performance of `stat`.

We believe that for this type of data, which is updated in concert with the inode, extended attributes are superior to any of the previously described methods.

4.2.3 Implementation

When the monitor initializes the AES encryption file system its operations vector is dynamically computed using the pass-through methods described in Section 4.1 as defaults. The encryption file system then overrides 16 of these methods. Most of these methods in turn call a pass-through method, but insert new functionality before or after the method call.

The `mount` operation initializes the AES encryption and decryption keys and locks them in memory so that the OS does not write them to swap. The `unmount` operation zeros out the keys before freeing them. The encryption layer defines four generic VFS-like operations: `open`, `read`, `write`, and `lseek`. It retrieves the file's IV via the extended attribute interface. If the file does not yet have an IV, then a new one is generated. The `read` and `write` functions are more complex than the others because they must correctly handle I/O operations that are not aligned on the AES block size, forcing us to use padding. Because of the padding we use an extended attribute to determine the real size of the file.

The encryption layer implements two internal methods for the pass-through file system: `encodename` and `decodename`. The `encodename` method converts a decrypted file name (e.g., `/home/cwright/amino/thesis.pdf`) to an encrypted file name. For each pathname component, the IV of the parent is first retrieved and then it is encrypted using CBC mode. The result of the encryption is base-64 encoded to avoid writing invalid or control characters to the file system. The `encodename` operation is used for `open`, `mkdir`, and other operations that take a pathname as an argument. Conversely, the `decodename` operation is used for directory-reading operations. It retrieves the IV of the parent, and then decrypts the name.

Finally, the encryption layer implements five system-call-level functions: `stat`, `fstat`, `truncate`, `ftruncate`, and `read`. The `stat` and `fstat` functions retrieve the file size using extended attributes. The `truncate` and `ftruncate` functions fill holes that could be created by sparse files, and align all truncate operations on AES block-size boundaries.

It is important to note that for operations that the encryption file system does not define, the pass-through file system is called directly by the monitor—without the encryption file being called at all. Existing VFS architectures require additional function calls for each stackable layer [90]. Moreover, by providing the extensible `encodename` and `decodename` functions, the pass-through file system obviates the need for derived file systems to implement most methods.

4.3 ISO9660 File System

CD-ROM images, also known as ISOs, are formatted according to the ISO9660 standard. ISOs are a convenient way of transferring large collections of files, such as Linux distributions, software backups, or even family photos. However, to access the files in an ISO, users must first mount it using a loop device. Unfortunately, mounting a file system requires root privileges. It would be possible to create set-UID programs to allow a user to mount ISO images, but even if developed securely, there is always a potential for bugs or misconfigurations that could compromise the security of the system.

To address this issue, we developed a user-level file system using our monitor, built around `libiso9660` from GNU `libcdio` [66]. The monitor's user-space nature allowed us to link against this library and leverage its 3,449 lines of already tested code.

Because ISO9660 file systems are read-only by their nature, we needed to implement only nine methods for this file system: `mount`, `unmount`, `open`, `close`, `read`, `lseek`, `fstat`, `getdents`, and `fcntl`. The most complex method was `read`. For `read`, much of the code complexity was caused by a limitation of the `libiso9660` library, which only allows 2KB-blocks to be read. To implement read efficiently, we wrote more code to avoid extra data copies for unaligned access.

4.4 Complexity Evaluation

We used four metrics to compare the amount of development effort different frameworks require to write pass-through, encryption, and ISO9660 file systems. For each type of system that we evaluated, Table 4.1 (page 51) shows the number of lines, tokens, identifiers, and the McCabe [44] cyclomatic complexity. McCabe's metric is the most precise: it measures the number of control points in a program. It has been shown that programs with a lower cyclomatic complexity are less error prone [30, 80, 87].

To calculate cyclomatic complexity we used the C and C++ Code Counter [41], which is the most robust tool we found for calculating cyclomatic complexity. Unfortunately, it cannot handle all of the complex programming constructs used in operating system code (e.g., old-style function declarations, bit fields, preprocessor directives interspersed with control statements, and some advanced C++). Therefore, we needed to modify the source of most packages for CCCC to parse them correctly. We endeavored to preserve complexity during any modification. Indeed, most of our modifications simply removed structure definitions, macros, or the offending code. These modifications can only decrease the cyclomatic complexity, and our monitor requires no such transformations; so these transformations do not give any advantage to our system.

4.4.1 Framework implementation

Our monitor and the FUSE frameworks require a similar amount of development effort to implement. We chose the SFS toolkit [43] as an example of a user-level NFS server. It is more than twice as complex than either FUSE or our monitor, but it is tightly integrated with a simple pass-through file system, which we did not remove from its complexity metric.

Method	LoC	Tokens	Identifiers	MC
Framework Implementation				
`ptrace`	**7,606**	46,854	**15,940**	**1,272**
Kernel	48,072	255,969	109,081	8,152
FUSE	8,481	**46,051**	17,772	1,338
NFS	18,480	103,960	42,734	2,726
Pass-Through File System				
`ptrace`	785	**4,297**	**1,514**	**136**
Kernel	6,079	34,612	14,146	599
FUSE	**706**	5,010	1,659	149
Cryptographic File System				
`ptrace`	**1,321**	**8,331**	**2,612**	**236**
Kernel	9,780	57,489	23,130	943
FUSE	2,396	19,297	7,468	423
NFS	1,556	8,981	3,663	251
ISO9660 File System				
`ptrace`	**582**	**2,906**	**1,046**	**93**
Kernel	3,769	22,158	8,666	616
FUSE	1,704	11,890	4,315	363
Amino File System				
`ptrace`	6,621	38,454	14,396	960

Table 4.1: *We evaluated different types of file systems implemented using different frameworks according to four metrics.* ***Bold*** *entries are the smallest in their class. (LoC means Lines of Code; MC means the McCabe cyclomatic complexity).*

The kernel's VFS system is the largest framework by any metric. This is not surprising because it cannot rely on external libraries and includes caching, quota management, support for several binary formats, asynchronous I/O, and many other tightly integrated facilities. This tight integration means that a kernel developer has to be familiar with a large body of complex code to develop file systems.

4.4.2 Pass-through layer

The monitor's and FUSE's pass-through file systems have similar complexity: FUSE is 11% shorter, but has 16% more tokens, 10% more identifiers, and a 10% higher cyclomatic complexity. Although our pass-through file system is 26 lines longer, it has more functionality than its FUSE counterpart. Our file system transforms names before passing the operation down to the lower-level file system, which enables us to mount on any lower-level directory (FUSE is limited to "/") and build our encryption file system on our pass-through file system. When this additional functionality is removed from our file system, its cyclomatic complexity is reduced to 97 (or 53% less than FUSE's). The SFS toolkit provides a built-in loopback NFS server, but the toolkit itself is more complex than our monitor or FUSE and the corresponding pass-through file system put together. Wrapfs, a pass-through file system for the Linux kernel, has the highest complexity, because it must perform elaborate operations on reference counts and cached objects, and emulate much of the VFS's functionality.

4.4.3 Encryption file system

We compared our monitor to the in-kernel eCryptfs [27], FUSE's EncFS [22], and the encryption file system from the SFS toolkit. Our file system and the one from the SFS toolkit have similar complexity. This is as expected because both allow a file system layer to extend an existing pass-through layer. Because EncFS was developed in FUSE, where the interface is similar to the VFS's, developers had to implement a lower-level interface, and thus they had to implement more routines. EncFS originally had over 14,000 lines of code and a McCabe complexity of 1,323. Even when we removed all code related to configuration, abstract classes, header files used by C++, and specialized caching, we still found EncFS to be twice as complex as our file system. eCryptfs suffers from the same problems as Wrapfs (because it is essentially a modified copy). Thus it is twice as complex as the other file systems.

4.4.4 ISO9660 file system

We compared our ISO9660 file system to the kernel's and to the FUSE-based `fuseiso` [51]. Our monitor's ISO9660 file system is 582 lines of code with a McCabe complexity of 93. The size and complexity of `fuseiso` was greater: 1,704 lines and a cyclomatic complexity of 423. This increase is for two reasons: (1) our monitor uses `libiso9660`, whereas `fuseiso` does not use any external libraries, so it has code for reading ISO9660 directories, and (2) FUSE requires its file systems to handle more VFS objects than our framework. The kernel implementation is larger than `fuseiso`. This

is because it cannot use external libraries such as `libiso9660` or even system calls, so interfacing with the device it is mounted on is more complicated.

4.4.5 Amino file system

We cannot directly compare Amino implemented in different frameworks, because we only implemented via `ptrace` (indeed, it would not be possible to do it over NFS or FUSE; and would have been more difficult to do it in the kernel). Amino is more complex than the other classes of file systems we investigated. In terms of lines of code, tokens, and identifiers Amino is larger than any user-level file system, and only the in-kernel encryption file system is larger. Amino is more complex in terms of the McCabe complexity than any of the other user-space file systems. The Amino file system is slightly less complex than our monitor framework with 13% less code, 17% fewer tokens, 19% fewer identifiers, and 25% less complexity than the framework.

In sum, FUSE and NFS are comparable to our monitor in terms of complexity. However, neither of these methods would have been able to export the transactional semantics to applications that was required for Amino.

Chapter 5

Evaluation

We evaluated the performance of our system by running several micro-benchmarks and general-purpose workloads. In Section 5.1 we present results for meta-data–intensive micro-benchmarks, and in Section 5.2 we present results for data-intensive micro-benchmarks. Section 5.3 summarizes interesting micro-benchmark results. We chose three general-purpose benchmarks to evaluate our system. In Section 5.4, we present results for the Postmark benchmark [37]. In Section 5.5 we present results for an OpenSSH compile, and we present results for a Sendmail benchmark in Section 5.6. Section 5.7 summarizes the results of the general-purpose workloads.

For all our benchmarks we used a dual 2.8Ghz Xeon machine running Fedora Core 4 with all updates as of July 19, 2005. 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, but does not modify any system 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 the pass-through file system described in Section 4. This measures the overhead of `ptrace`, our path-name resolution infrastructure and the operations required to implement a file system using ptrace. It is important to note that AMINONULL does not require extra data copies for file data, because it reads data directly from the lower-level file system into the user process address space using a memory-mapped `/proc/pid/mem` interface.

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.

These configurations can broadly be broken up into two groups: those that do not provide durability and those that do. The six configurations that do not provide durability are VANILLA, STRACE, AMINOTRACE, AMINONULL, AMINOACI and AMINOTRACE. Within this group of configurations, the are in order of increasing amounts of code provided by the monitor. For example, STRACE includes the overhead associated with `ptrace`, and AMINOTRACE adds the overhead of our path-name resolution on top of that. The three durable configurations are VANSYNC, AMINOACID, and AMINODTXN. The added cost of providing durability in these configurations can be determined by comparing them compared to the non-durable configurations VANILLA, AMINOACI, and AMINOTXN, respectively.

Not all benchmarks use all configurations. For example, the micro-benchmarks do not use AMINOTXN or AMINODTXN, because the operations are not logically grouped together, and read-only benchmarks do not use VANSYNC or AMINOACID.

5.1 Meta-data Micro-benchmarks

We ran several micro-benchmarks on Amino to evaluate the overheads of certain primitive file system functions like file creation and deletion as well as reading and writing data. We broadly classify our micro-benchmarks into metadata and data benchmarks. We describe the meta-data micro-benchmarks in this section, and the data benchmarks in Section 5.2. 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 [13].

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 so to isolate the overheads of the metadata operation to be tested.

Create To evaluate the overhead of 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 we created only 100,000 files evenly spread across 500 directories, because they take significantly longer than the non-durable configurations. To account for this difference, we normalize the elapsed time to operations per second and CPU utilization.

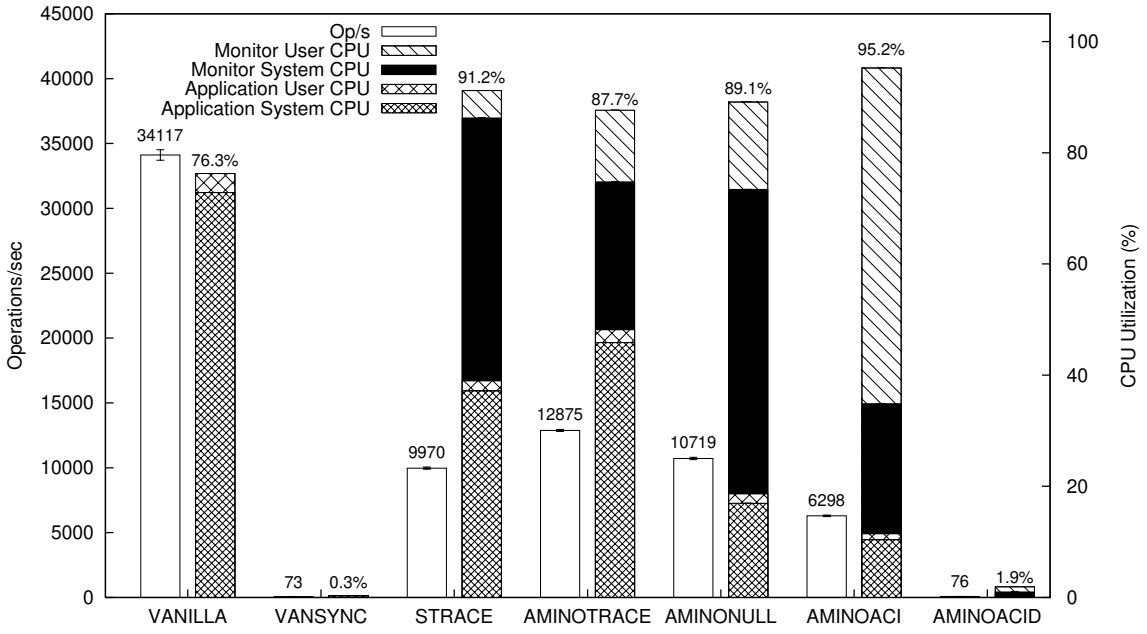


Figure 5.1: *Creation micro-benchmark results. The left axis shows the number of creates per second, the right axis shows CPU utilization.*

As seen in Figure 5.1, 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.

The AMINOACI created 5.4 times fewer files per second than VANILLA. There was 6.7 times increase CPU time as well. The applications 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 the synchronous mode of Ext3.

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 5.2 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. The AMINOACID configuration ran 3.9 times faster than Ext3 in its synchronous mode. This is because of a 51.3% decrease in the wait time, as a result of 74.0% fewer sectors being written. The AMINOACID configuration requires so many 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 the leaf node of the B-tree.

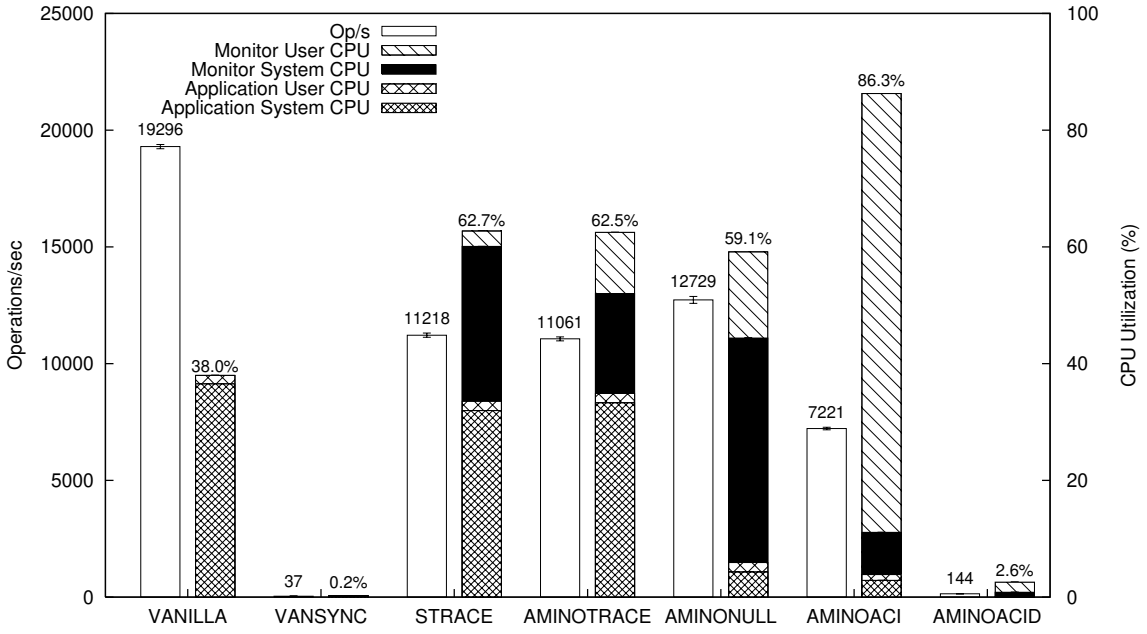


Figure 5.2: *Deletion micro-benchmark results. The left axis shows the number of unlinks per second, the right axis shows CPU utilization.*

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, deleting 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 5.3 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.

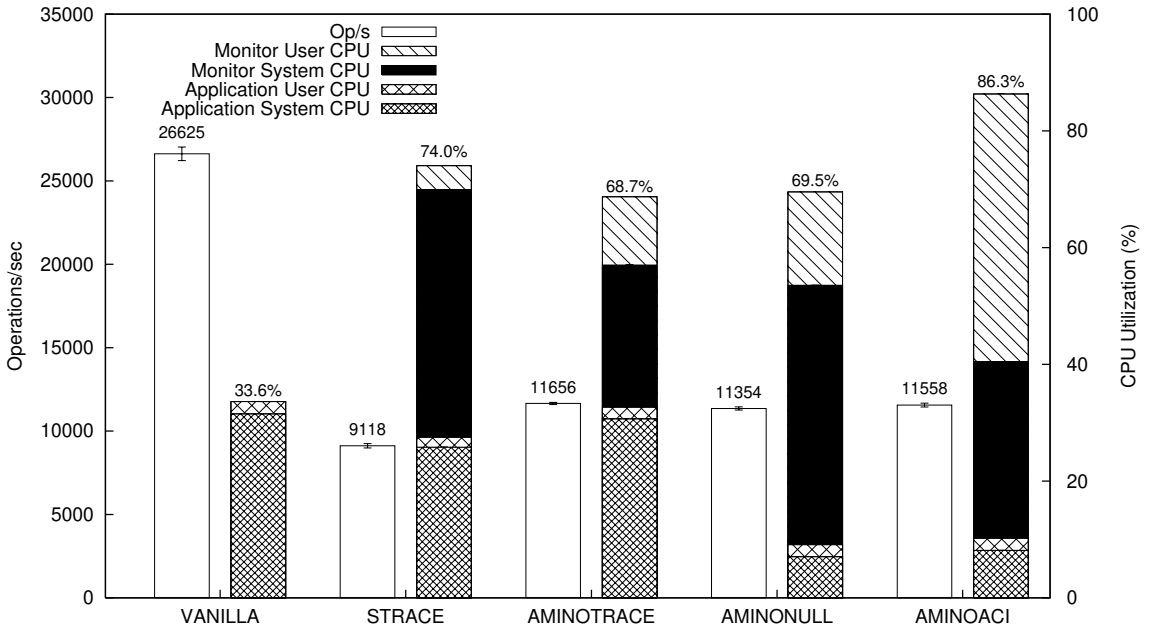


Figure 5.3: *Directory operation micro-benchmark results:* `stat`*. The left axis shows the number of* `stat` *operations per second. The right axis shows CPU utilization.*

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 monitor 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). Because there were no writes in this workload, the durable configurations are identical to their non durable counterparts.

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 5.4. 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.

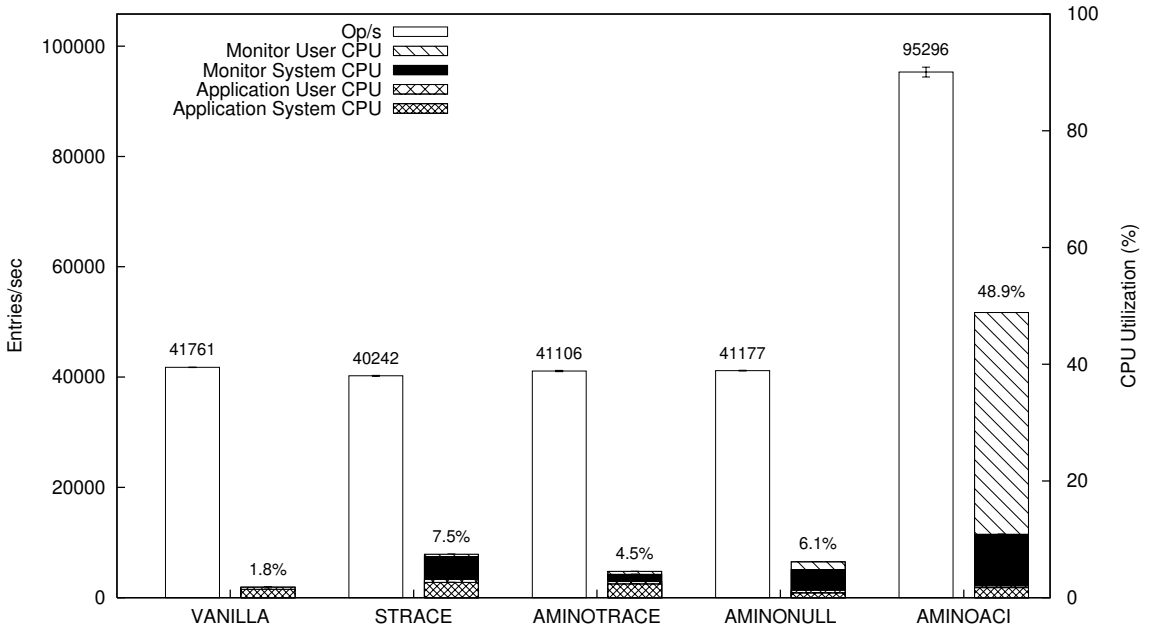


Figure 5.4: *Directory operation micro-benchmark results: `readdir`. The left axis shows the number of directory entries read per second. The right axis shows CPU utilization.*

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. The difference in overheads between VANSYNC and AMINOACID are within 2% of VANILLA and AMINOACI as this is a read-only workloads.

5.2 Data Micro-benchmarks

To evaluate the performance of data operations we ran a random read, random write, sequential read, and sequential write micro-benchmark. 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 read 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 and 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. We describe the random write setup in Section 5.2.1.

In Section 5.2.1 we present uncached single threaded results. In Section 5.2.2 we present uncached multi-threaded results. In Section 5.2.3 we present cached read results.

5.2.1 Single Threaded Results

In this section we describe the single threaded results for sequential read, random read, sequential write, and random write. All benchmarks are run with a cold cache.

Sequential Read The overheads of Amino under the sequential read workload are shown in Figure 5.5. 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.

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 and must begin again. As the database periodically accesses the database out-of-order, this results in Linux performing

I/O operations that are on average 2.9 times bigger 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. The overheads of AMINOACID are comparable to AMINOACI, as this is a read-only workload.

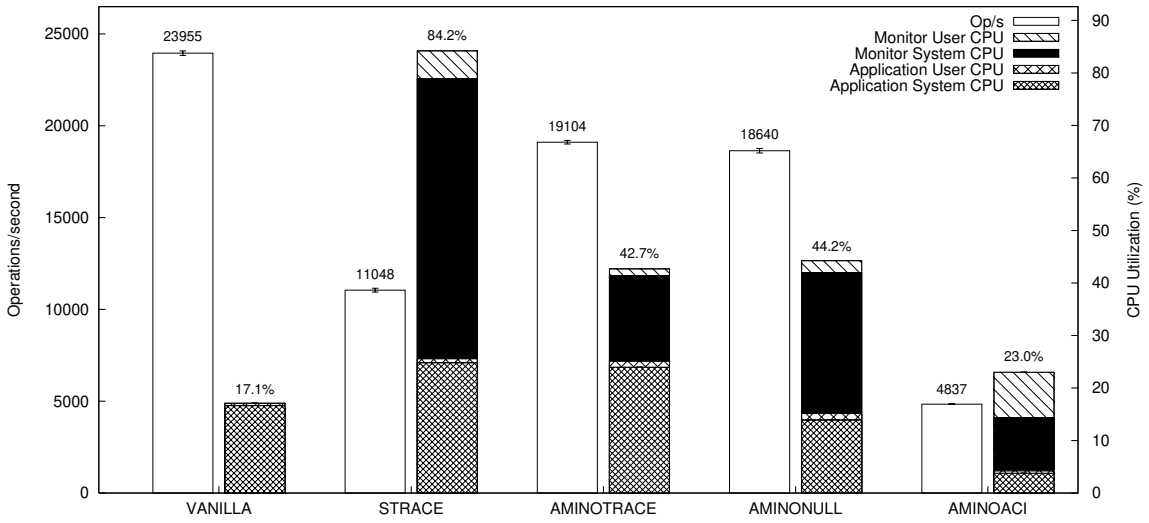


Figure 5.5: *Data micro-benchmark results: Sequential read. The left axis shows the number of operations per second, the right axis shows the CPU utilization.*

Random Read The results of the random read benchmark are shown in Figure 5.6. 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. The overheads of AMINOACID are comparable to AMINOACI, as this is a read-only workload.

Sequential Write Figure 5.7 shows the time taken for Amino for the sequential write workload. 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

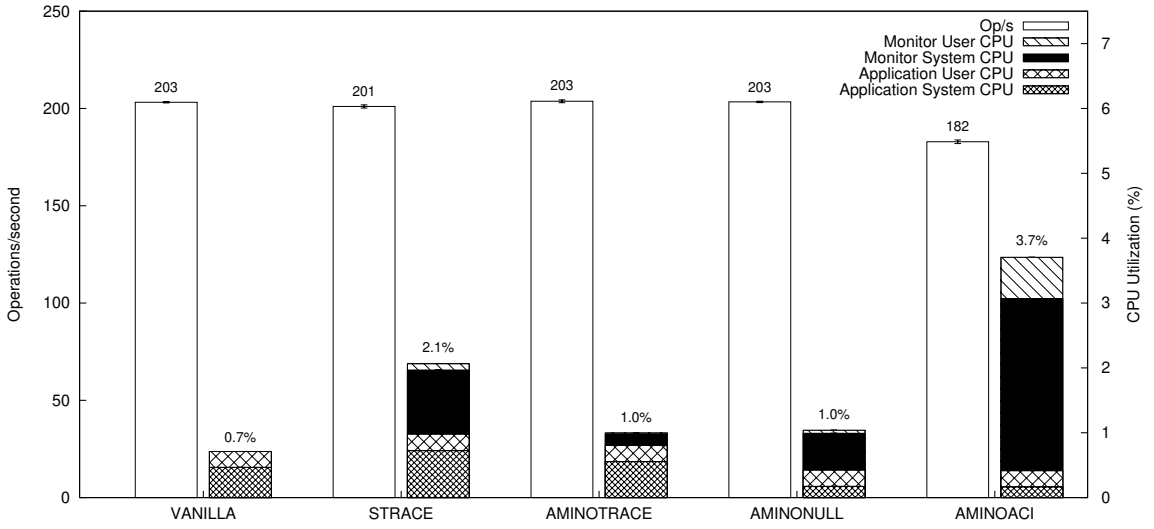


Figure 5.6: *Data micro-benchmark results: Random read. The left axis shows the number of operations per second, the right axis shows the CPU utilization.*

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 [88]. Figure 5.8 shows the number of operations per second performed compared to time during the benchmark (sampled at 10 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 this 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 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.

As can be seen in Figure 5.9, other configurations also suffer from the same periodic and erratic behavior. In this example, STRACE has periods of fast writes approximately every

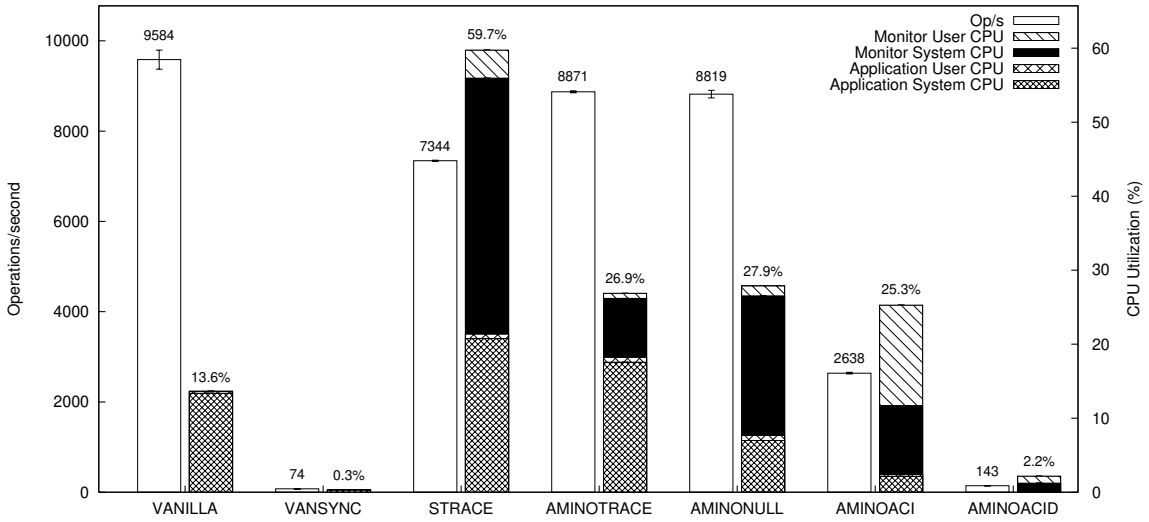


Figure 5.7: *Data micro-benchmark results: Sequential write. The left axis shows the number of operations per second, the right axis shows the CPU utilization.*

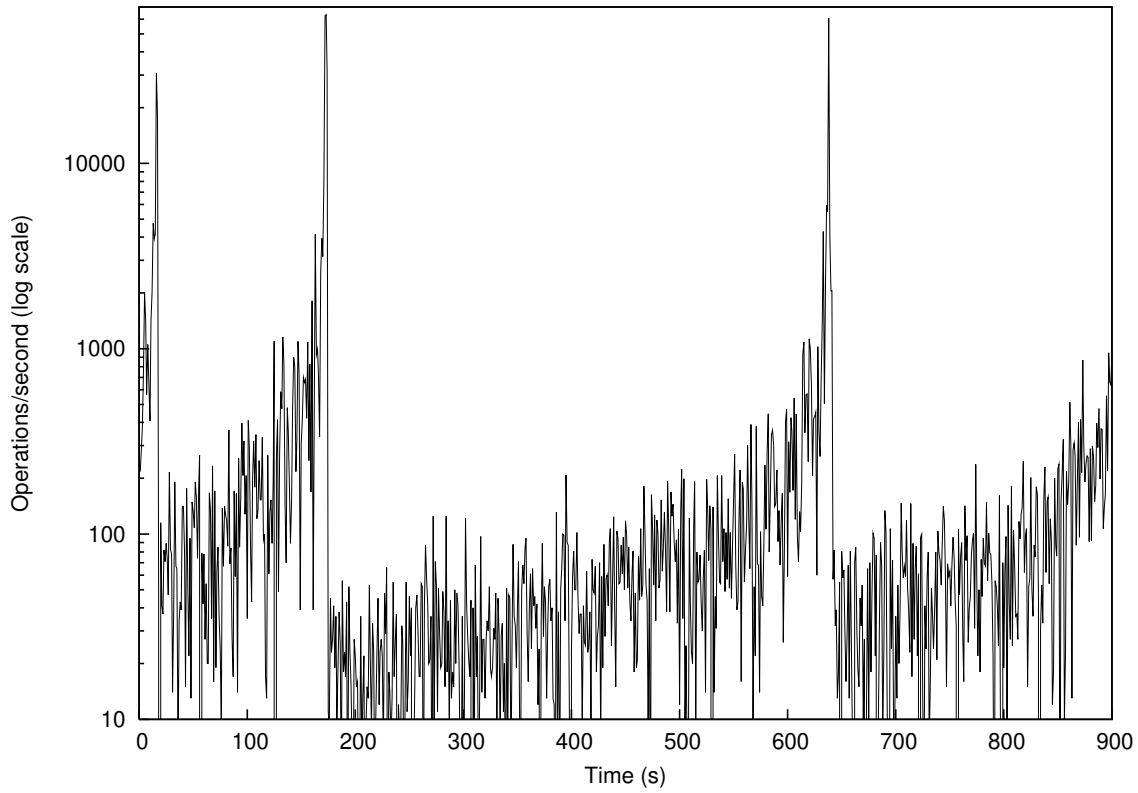


Figure 5.8: *Random write: operations/second (log scale) vs. time using one second samples for* VANILLA. *The Linux dirty buffer flushing is very sensitive to certain thresholds, so the data is both periodic and highly erratic.*

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 run the random write benchmark for 15 minutes and measure the total number of operations per second. The results of the benchmark are shown in Figure 5.10. We also include a box plot of the 1 second samples showing the first quartile, median, third quartile, and outliers[1] for each configuration in Figure 5.11. 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.

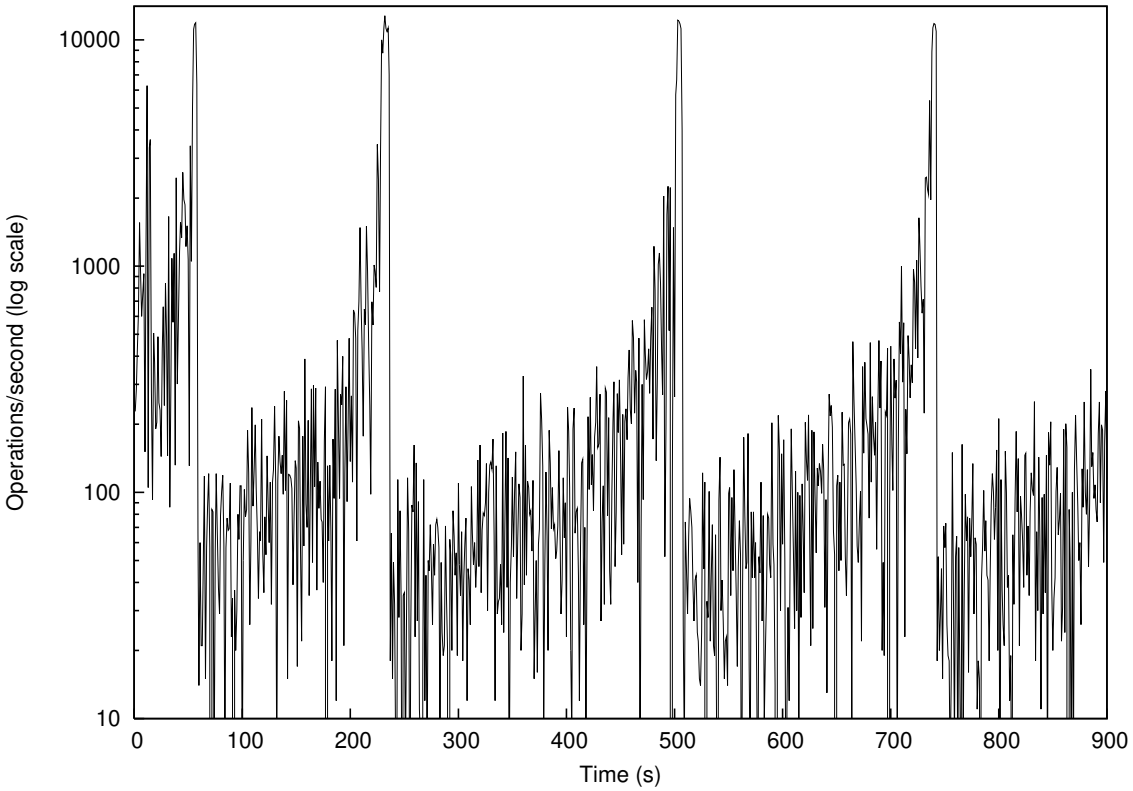


Figure 5.9: *Random write: operations/second (log scale) vs. time using one second samples for* STRACE*. The Linux dirty buffer flushing is very sensitive to certain thresholds, so the data is both periodic and highly erratic.*

Figure 5.10 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 were performed (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).

[1]Outliers are defined as points that are more than 1.5 times the interquartile range ($Q_3 - Q_1$) from the median.

The highest number of operations achieved in a 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. 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.

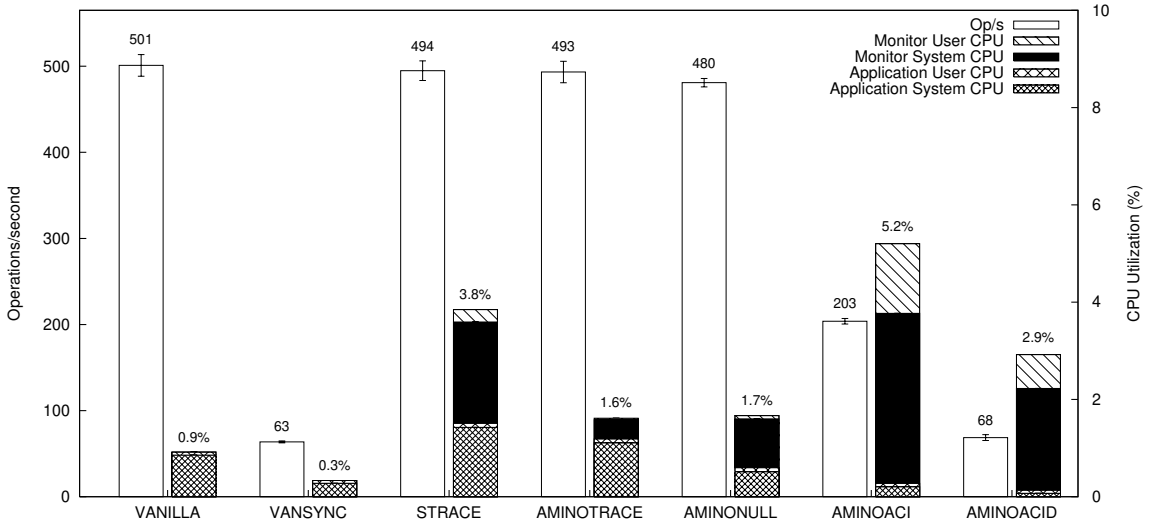


Figure 5.10: *Data micro-benchmark results: Random write. The left axis shows the number of operations per second, the right axis shows the CPU utilization.*

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.

5.2.2 Multi-Threaded Results

In this section we describe results for sequential read, random read, sequential write, and random write from 1–32 threads. Each thread operates on a disjoint subset of the file. An array that contains all of the page indexes in the file is divided into n sections (for example, in the two threaded case the first half of the array and thus file, is one section and the second half of the file is another section). Each thread repeatedly operates on its section from start to finish until the workload terminates. The random workloads operate in the same way, but the array is randomized before dividing the file into sections. All benchmarks are run with a cold cache.

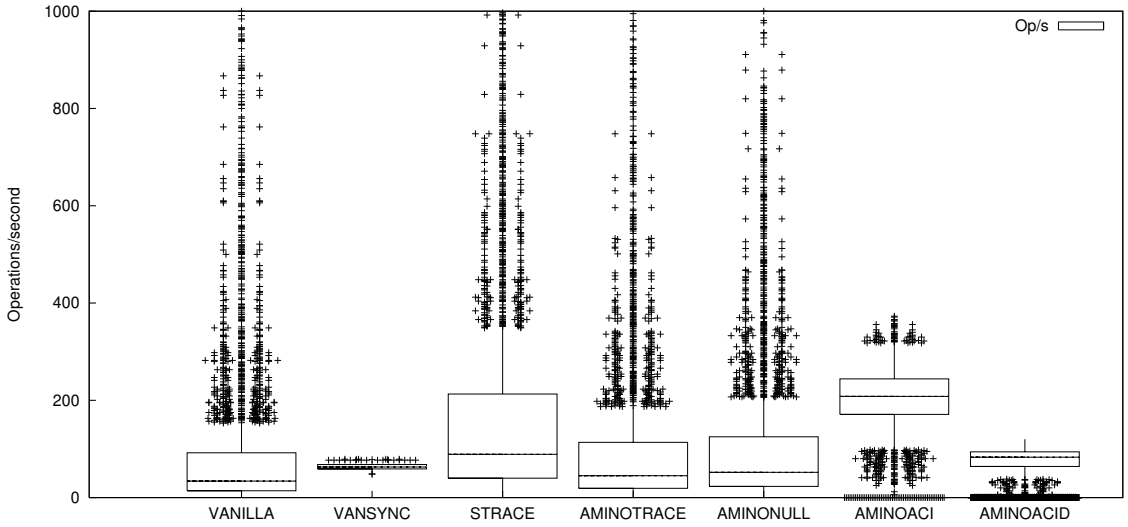


Figure 5.11: *Data micro-benchmark results: Random write box plot. Each box represents 50% of the data points, the horizontal bar through the box is the median data point. The vertical lines extend 1.5 times the interquartile range ($Q_3 - Q_1$) from the median. Outlying points below 1,000 are shown.*

Sequential Read As seen in Figure 5.12, the VANILLA configuration improves as threads are added. For two threads, 44.7% more read operations are performed than for a single thread. For four, eight, sixteen, and 32 threads, 2.7[2], 7.7, 8.9, and 9.5 times as many operations as a single thread are performed, respectively. The tracing configurations were not able to scale along with VANILLA configuration. For two threads, STRACE improved by 46.3%. Because the STRACE monitor is single threaded, as more threads are added there is more competition for the monitor among the threads. For four threads the improvement over a single thread was only 14.6% (i.e., less than the improvement for two threads). For 8–32 threads, STRACE performed 19.6–22.3% more operations than a single thread. For two and four threads AMINOTRACE configuration improved by 23.4% and 14.1% over a single thread, respectively. The CPU time used also increased by 58.5% and 99.5% to 67.7% and 85.1%, respectively. Eight threads produced the best results, with an increase of 39.6% over a single thread. For 16 and 32 threads, the CPU utilization increased even more to 225.7% and 274.3%, respectively.[3] This marked increase in CPU time caused performance to degrade to 13.0% over a single thread for 16 threads, and for 32 threads the number of operations performed per second dropped to 29.3% less than a single thread. The AMINONULL configuration performed 46.3% more operations with two threads than a single thread, meaning that it scaled similarly to VANILLA. However, for 4, 8, 16, and 32 threads it only achieved 14.6–22.3% more operations than for a single thread. The AMINONULL configuration did not degrade below a single threads performance as AMINOTRACE did, because the AMINONULL configuration actually uses less CPU time than the

[2]The half-width for VANILLA sequential read with four threads is 12.8% of the mean.

[3]The maximum CPU utilization for this machine is theoretically 400%, because it has two hyper-threaded processors.

AMINOTRACE configuration. The reason is that the application's system time is reduced by 64.6% from AMINOTRACE to AMINONULL, because the application performs none of its own system calls as the monitor does all of them on its behalf. The AMINOACI configuration scaled similarly to the tracing configurations. Its performance increased by 61.5%, 78.3%, and 63.9% for two, four, and eight threads, respectively. For sixteen threads, the number of operations performed was 1.0% higher than for a single thread. For 32 threads, the number of reads was 17.3% lower than for a single thread.

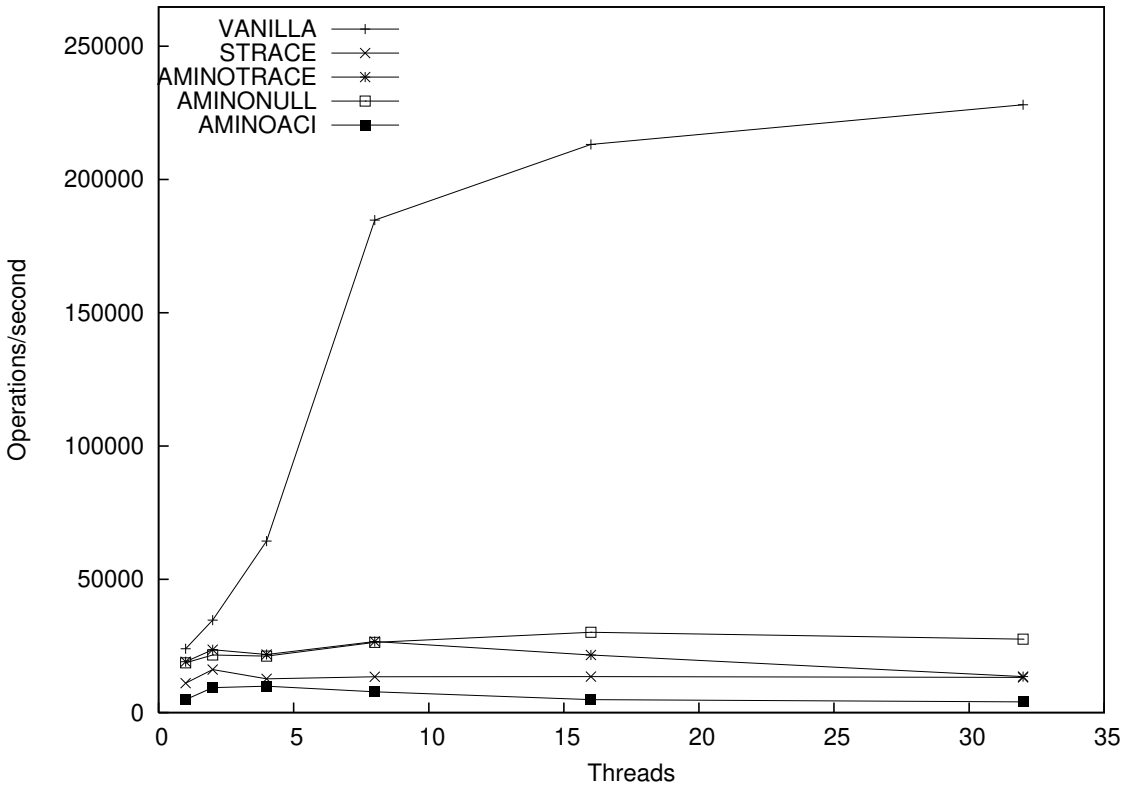


Figure 5.12: *Data micro-benchmark results: Multi-threaded sequential read.*

Random Read For the random read workload, the VANILLA configuration performed 14.1% more operations for a two threads than a single thread. As threads were added, VANILLA made steady gains: 33.7% for four threads, 64.2% for eight threads, 105.5% for sixteen threads, and 143.9% for 32 threads. The STRACE, AMINOTRACE, and AMINONULL configurations were all within 2.1% of the VANILLA configuration for 1–32 threads. The AMINOACI configuration scaled similarly to VANILLA, but was between 7.0% and 11% slower than VANILLA for the same number of threads. The overhead is caused for the same reasons as in a single threaded workload: Amino had to read significantly more data (due to its larger page size), but fewer I/O operations were required (as some pages are found in the cache).

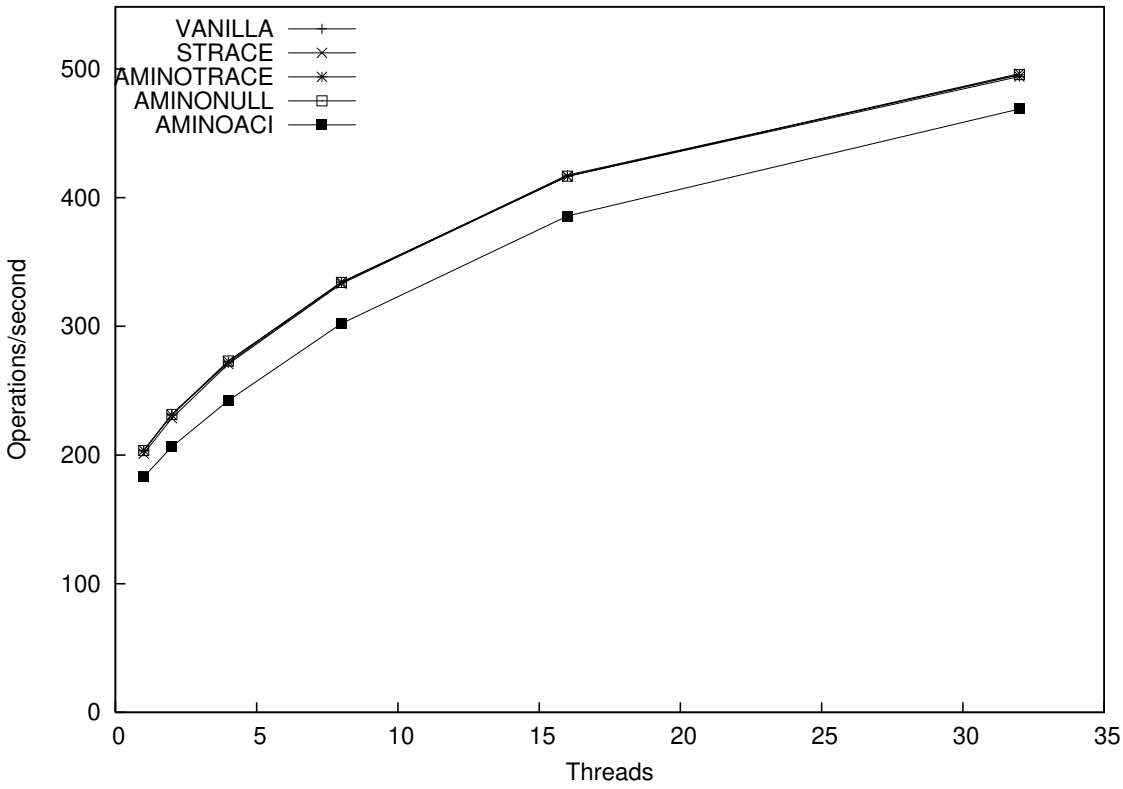


Figure 5.13: *Data micro-benchmark results: Multi-threaded random read.*

Sequential Write The single threaded sequential write workload performed 9,585 operations per second under VANILLA. For two threads, the throughput increased by 46.3% to 14,022 operations per second, but four 4–32 threads there was a slow decline from 2.8% higher than a single thread to 12.4% worse than a single thread. The tracing configurations followed suit, with STRACE declining from 15.2% faster for two threads than a single thread to 10.4% slower than a single thread for 32 threads. The STRACE configuration performs worse than the VANILLA configuration for a single thread, because the monitor is single threaded so cannot take advantage of as much concurrency. The AMINOTRACE configuration performed 55.5% more operations for two threads than a single thread. For 4–32 threads the rate declined from 26.8% faster than a single thread to 22.2% slower than a single thread. The AMINONULL configuration was similar to AMINOTRACE. For two threads, there was a 31.7% improvement over a single thread, which declined to a 10.5% degradation over a single thread for 32 threads. The AMINOACI configuration ran increased by 85.9% for two threads over a single thread. For 4–32 threads it declined from 84.6% to 28.0% faster than a single thread.

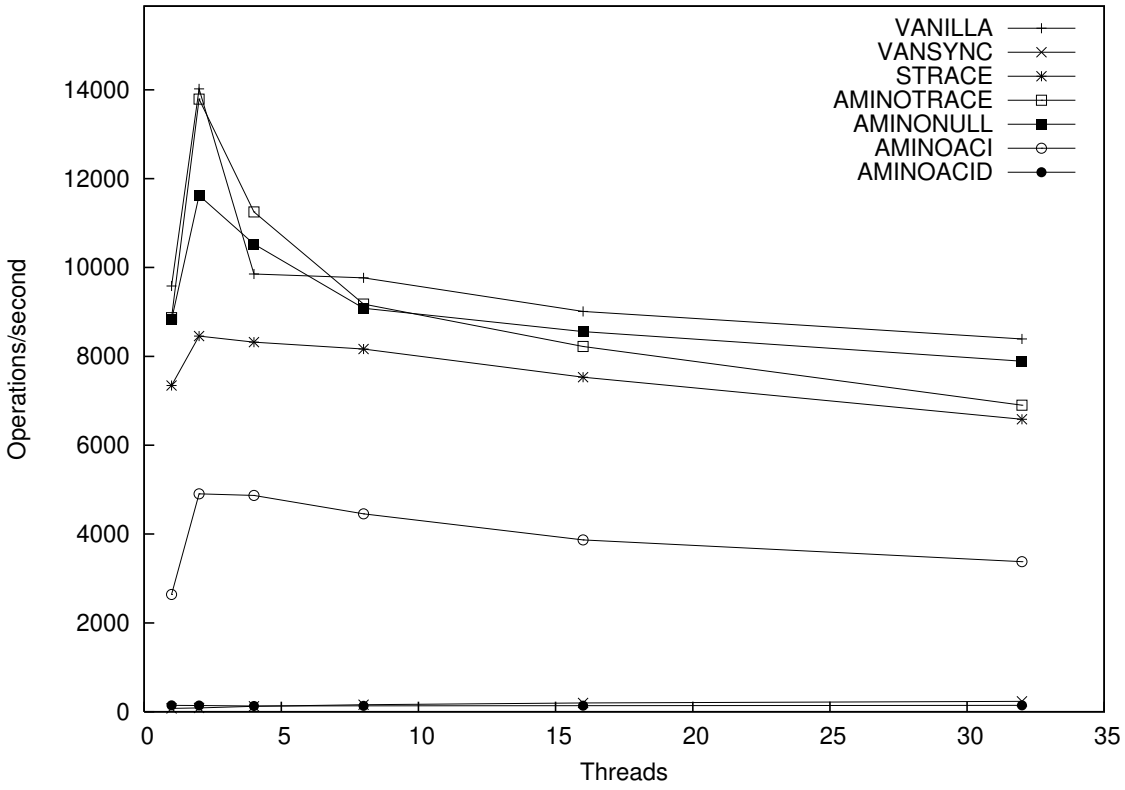


Figure 5.14: *Data micro-benchmark results: Multi-threaded sequential write.*

As expected, the durable configurations are slower than the non-durable configurations. The VANSYNC configuration experienced steady gains as threads were added. Two threads were 16.2% faster than one thread, four threads were 61.9% faster, eight threads were 2.1 times faster, 16 threads were 2.6 times faster, and 32 threads were 3.2 times faster than a single thread. The AMINOACID configuration had relatively constant results. For

2–32 threads it was between 0.5–9.5% slower than for a single thread. The reason that AMINOACID does not scale well is that the file's modification time update requires a write lock for the duration of the log flush. This prevents group commit from occurring, so only one record can be flushed to the log at a time. Although AMINOACID was faster than VANSYNC for 1 and 2 threads, it did not scale as well as VANSYNC, so the VANSYNC configuration was faster for four or more threads.

Random Write The multi-threaded random-write benchmarks are shown in Figure 5.15. The number of operations performed per second remained relatively flat for the VANILLA, STRACE, AMINOTRACE, and AMINONULL configurations. For each configuration, 2–32 threads were within 5.0% of a single thread using the same configuration except for the thirty-two threaded AMINONULL configuration, which was 6.9% faster than a single thread.[4] The AMINOACI configuration performed an increased number of operations as threads were added. Two, four, eight, sixteen, and thirty two threads performed 12.0%, 17.1%, 26.9%, 36.1%, and 30.4% more operations than a single thread, respectively,

The VANSYNC configuration had more pronounced gains than AMINOACI. Two and four threads were 36.3% and 94.6% better than a single thread, respectively. The 8, 16, and 32 thread workloads were 2.6, 3.3 and 4.1 times better than a single thread. The VANSYNC results were also the most consistent, with very few outliers compared to other configurations. The AMINOACID configuration performed 5.2% fewer operations for two threads than a single thread. However, for four threads 6.7% more operations were performed than for a single thread, and slight gains were made as threads were added: 8.7% for eight threads, 12.4% for sixteen threads, and 14.7% for 32 threads.

5.2.3 Cached Results

In this section we describe results for sequential and random read for a single thread. We used a 500MB file for this workload, as it fits entirely in the cache. Before the warmup phase, we sequentially read the file to load it into the caches.

Cached Sequential Read As expected, the cached read workload performed significantly more operations per second than the uncached workload. The VANILLA configuration performed 5.9 times as many read operations, and the CPU utilization increased from 17.1% to 94.5%. All of the other configurations had similarly high CPU utilization, but in addition to the user application consuming CPU the monitors did as well. The STRACE configuration performed 91.2% fewer operations, because the monitor consumed 66% of the CPU. The AMINOTRACE and AMINONULL configurations performed better than STRACE, because they used less CPU for the monitor (though slightly more overall CPU). The AMINOTRACE and AMINONULL configurations had a reduction in I/O operations of 66.7% and 69.1%, respectively. The AMINOACI configuration had a reduction of 79.7%. For AMINOACI, the monitor used 79.2% of the CPU, and the user component of the monitor's time was the highest at 47.6%.

[4]The four threaded VANILLA results have a half-width that is 6.4% of the mean.

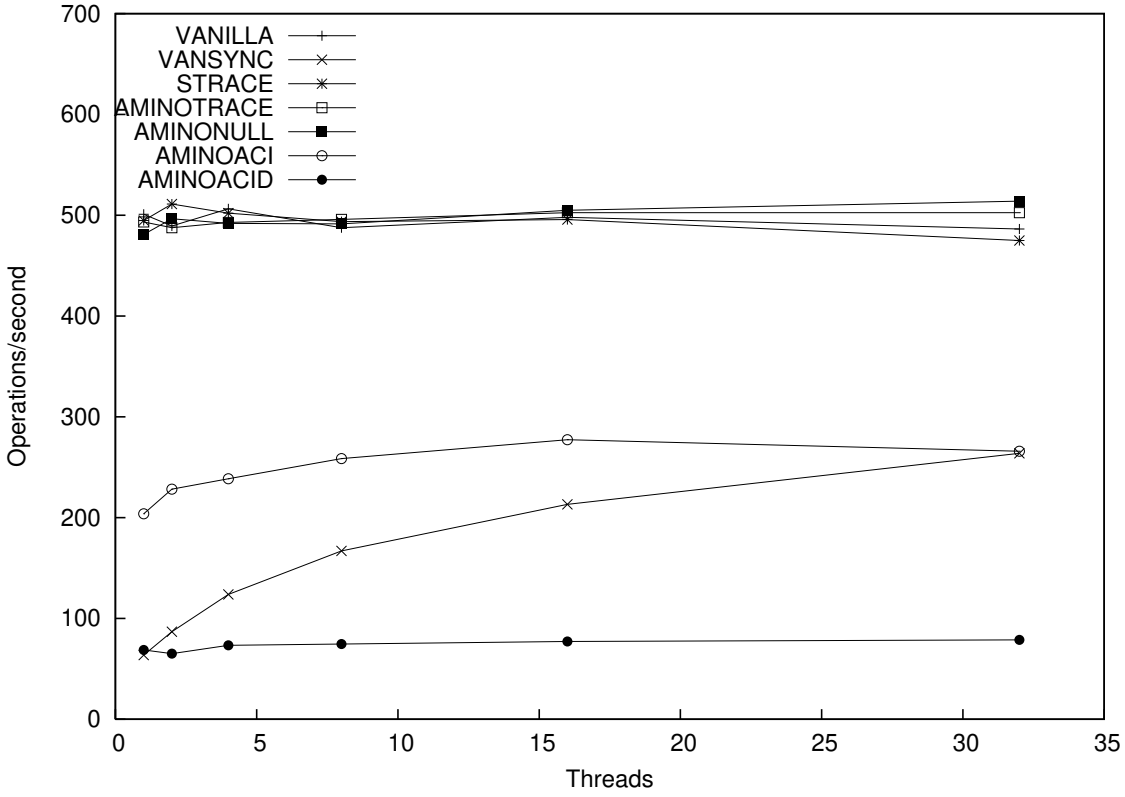


Figure 5.15: *Data micro-benchmark results: Multi-threaded random write.*

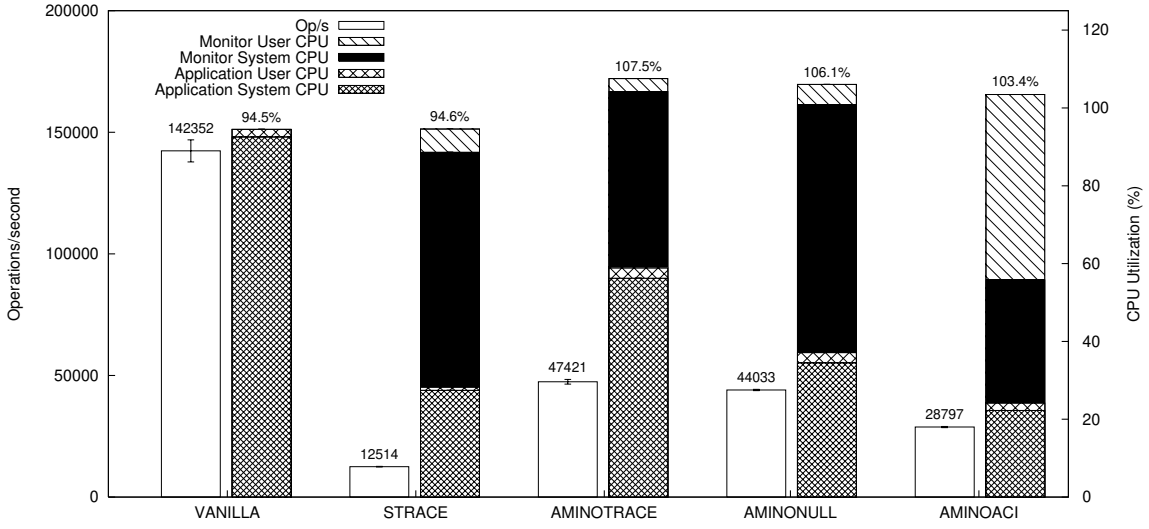


Figure 5.16: *Data micro-benchmark results: Cached sequential read. The left axis shows the number of operations per second, the right axis shows the CPU utilization.*

Cached Random Read The cached random read results are statistically indistinguishable from the cached sequential read results for `vanilla`, STRACE, and AMINOTRACE. For AMINONULL, the sequential read performed 1.7% more operations per second. The AMINOACI configuration performed 7.0% fewer random reads than sequential reads, because the monitor CPU time per read increased by 9.5%.

5.3 Micro-benchmark Summary

To summarize, our micro-benchmarks show that `ptrace` adds additional CPU time, which has the largest impact on CPU-intensive workloads. Our monitor in the AMINOTRACE configuration is faster than STRACE, and the AMINONULL configuration is often faster than STRACE. The AMINOACI configuration utilizes yet more CPU time than AMINONULL, resulting in decreased throughput.

We draw three conclusions from our meta-data micro-benchmarks:

- The `create` and `unlink` workloads are CPU-intensive; therefore, tracing caused a performance decline of from 62.3–70.8% for `create` and 34–42.7% for `unlink`. The AMINOACI configuration used 6.0 and 6.1 times more CPU time for `create` and `unlink`, respectively. This resulted in an 81.6% and a 62.6% decline in operations per second when compared to VANILLA.

- The `stat` workload showed that tracing had a large impact on this workload (56.2–65.8%). However, AMINOACI was statistically indistinguishable from AMINOTRACE. The AMINOACI configuration uses more CPU time than AMINOTRACE because it traverses the B-tree. However, AMINOACI performs fewer I/O operations, canceling out the additional CPU utilization.

- For `readdir`, the tracing configurations are all within 5% of VANILLA, and AMINOACI is 2.3 times faster than VANILLA because it has better locality.

These results tell us that the additional CPU overhead of `ptrace` results in a decrease in throughput for CPU-intensive write workloads (between 2.7–5.4 times). For the read-only `stat` workload, AMINOACI shifted CPU time from the kernel to user-space, and decreased the number of I/O operations. For the read-only `readdir` workload, the improved locality of a B-tree increased throughput.

Our single-threaded data micro-benchmarks show that:

- The AMINOTRACE and AMINONULL configurations performed better than STRACE, because they used less CPU time.

- For the random read and write workloads, AMINOACI's larger database page size resulted in fewer operations per second.

- For the sequential read workload, the smaller I/O operations issued by the kernel combined with the overhead of `ptrace` AMINOACI caused a 79.8% decrease in operations per second.

- The random write workload is highly erratic on Linux. An improved buffer flushing policy could yield more predictable performance [88].

Our multi-threaded data micro-benchmarks show that:

- For the sequential read workload, although Ext3 improves with multiple threads, the tracing configurations do not improve as much. The reason is that the tracing configurations saturate the CPU more quickly, making it a bottleneck.

- The random read workload is I/O-bound, and tracing has negligible overhead. The AMINOACI configuration has relatively constant overhead of 7–11% because of its larger database page size.

- The random write workload is also I/O bound. Whereas the VANSYNC configuration improves as threads are added, the AMINOACID configuration does not because the file's modification time updates cause the log flushes to be serialized.

In many cases, our micro-benchmarks show significant declines in performance. We believe that many of the performance declines would be offset by porting our file system to the kernel and using a representation for the data that does not rely on the BDB access methods.

5.4 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 [37]. 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 left over files.

The primary metrics of interest are elapsed time or transactions per second (a function of elapsed time). Unfortunately, this 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 parameter, and then a time limit. Our modified Postmark creates the initial pool of files, performs transactions for the specified time, and then removes any left over 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 5.17. The VANILLA configuration performed 1,591 transactions/second and had CPU utilization of 40.3%. The VANSYNC configuration

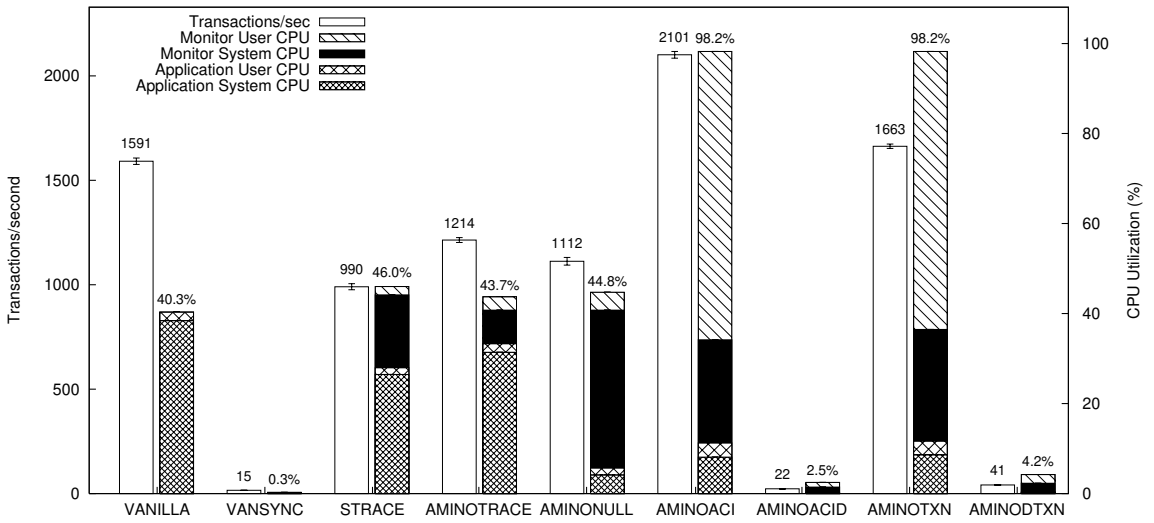


Figure 5.17: *Postmark: 2,500 files. The left axis is transactions per second, and the right axis is CPU utilization.*

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 resolves 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.

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%. This CPU increased CPU utilization results from two factors. 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 utilization. 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 as described in Section 2.5.1. 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 AMINOTXN 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.

5.4.1 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 be required to perform linear scans over 25,000 files for some operations.

Larger Files The results of the first alternative configuration (larger files) are shown in Figure 5.18. 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.

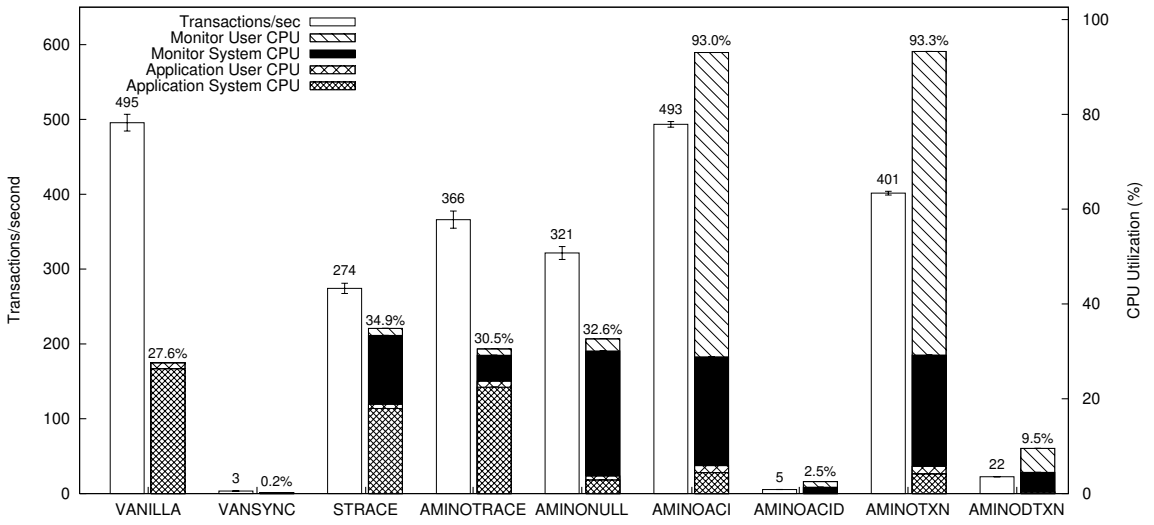


Figure 5.18: *Postmark: 2,500 Files; 5,120–102,400 bytes. The left axis is transactions per second, and the right axis is CPU utilization.*

More Files The results of the second alternative configuration (more files) are shown in Figure 5.19. 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 had a 24.7 times slow down, and AMINOACID had a 17.5 times slowdown. Again, AMINODTXN was the most efficient synchronous configuration with only a 10.2 times slowdown over VANILLA. This configuration demonstrates that explicitly marking transactions combined with BDB's highly-tuned logging infrastructure can improve durable file system performance.

5.5 OpenSSH Compile

To simulate a more CPU-intensive typical user workload, we adapted the SSH build workload [76], but used OpenSSH 4.2p1 as it builds cleanly on our systems whereas SSH 1.2.26 does 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

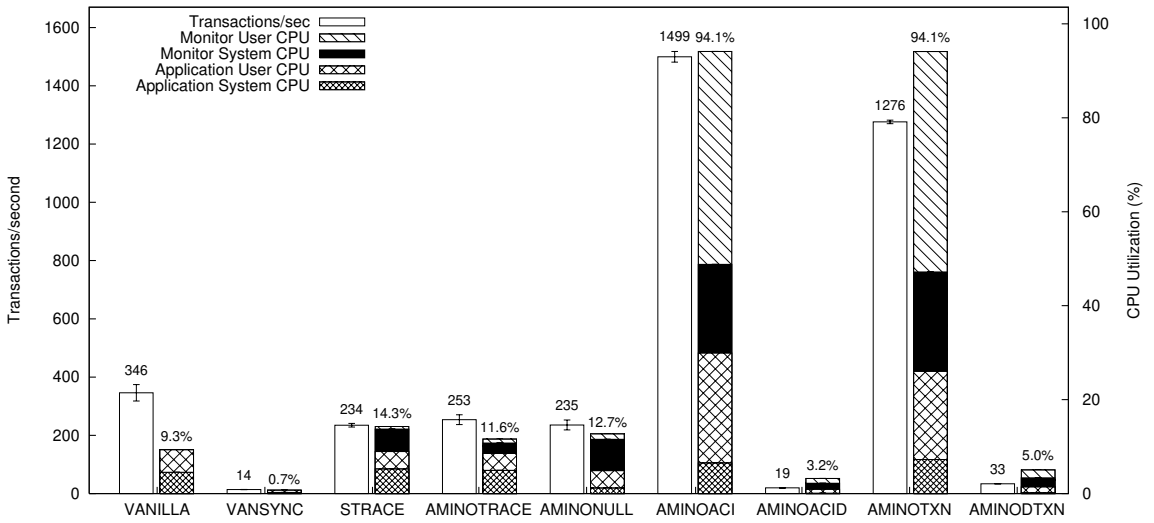


Figure 5.19: *Postmark: 25,000 Files. The left axis is transactions per second, the right axis is CPU utilization.*

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 our modified versions of GNU Make and `tar` described in Section 2.5.2 and Section 2.5.3, respectively. 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.

Figures 5.20, 5.21, and 5.22 show the results of each phase of the OpenSSH compile benchmark. The unpack phase (shown in Figure 5.20) took 0.09 seconds on VANILLA. The STRACE configuration added an overhead of 84.4%, and AMINOTRACE had an overhead of 59.9%. The AMINONULL configuration 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 the VANILLA configuration, 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. The AMINODTXN is only 69 times slower than VANILLA, but provides application-level consistency and durability. In general, this untar phase of the benchmark is quite short, so it does not provide the most accurate (because of timing accuracy) or interesting results.

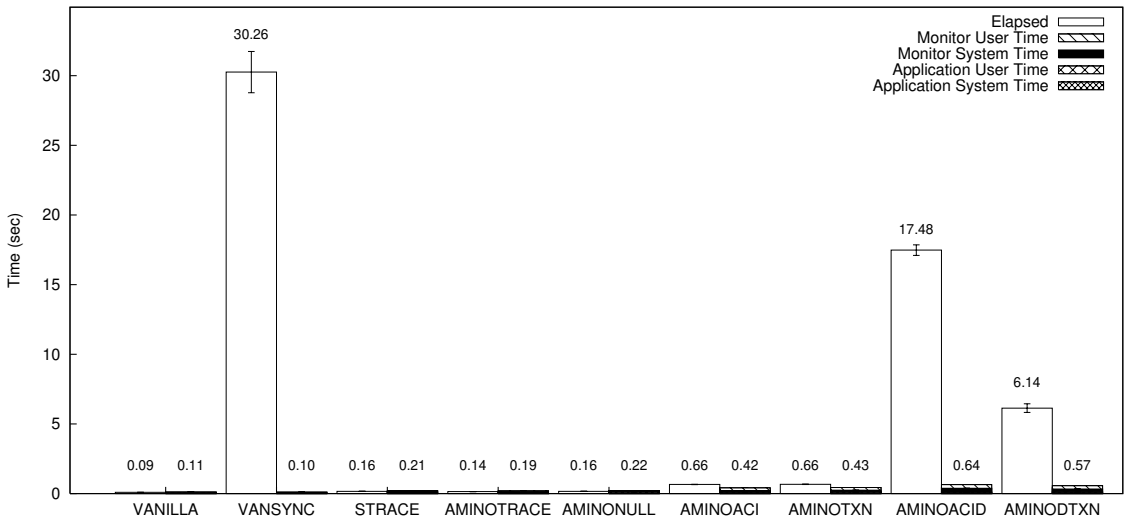


Figure 5.20: *OpenSSH Unpack results. Each configuration has two bars grouped together. The first bar is for Elapsed time, the remaining bar consists of each of the CPU time components. Configurations with lower CPU and elapsed time perform better than configurations with higher CPU and elapsed time.*

Therefore we performed two more untar benchmarks described in Appendix A.1, which shows that AMINOACI is between 3.9 and 7.0 times slower than VANILLA (the Appendix includes results for all of the configurations).

The second phase of benchmark, configuration, is shown in Figure 5.21. 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 5.22) took 35.3 seconds on VANILLA. Even though this phase is the most CPU intensive phase of all, this 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

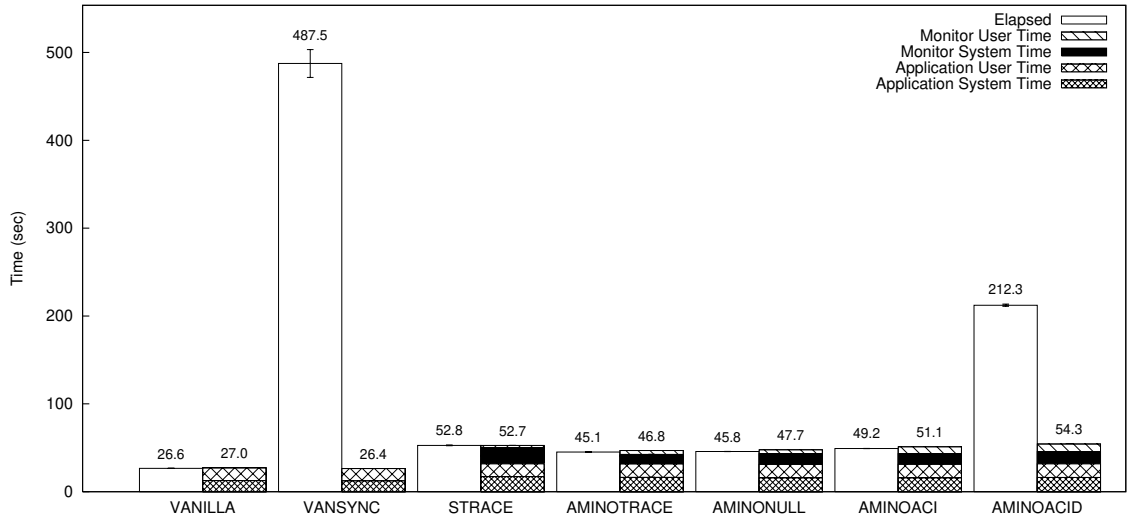


Figure 5.21: *OpenSSH Configuration Results. Each configuration has two bars grouped together. The first bar is for Elapsed time, the remaining bar consists of each of the CPU time components. In some instances, the CPU time components may be larger than the Elapsed time, because processes occasionally execute concurrently during compilation. Configurations with lower CPU and elapsed time perform better than configurations with higher CPU and elapsed time.*

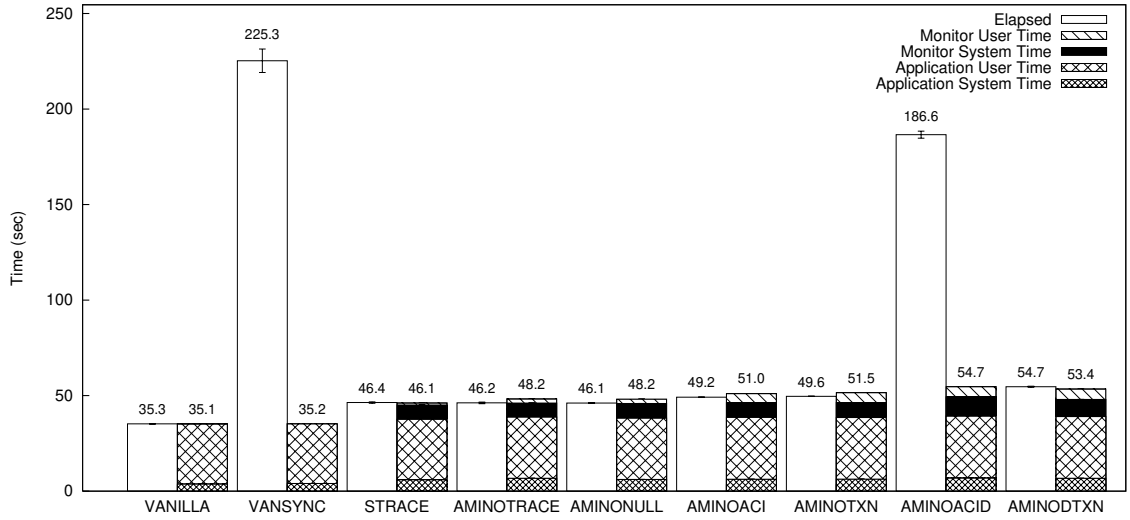


Figure 5.22: *OpenSSH Build Results. Each configuration has two bars grouped together. The first bar is for Elapsed time, the remaining bar consists of each of the CPU time components. In some instances, the CPU time components may be larger than the Elapsed time, because processes occasionally execute concurrently during compilation. Configurations with lower CPU and elapsed time perform better than configurations with higher CPU and elapsed time.*

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 55.1% over VANILLA, just 15.6% more than the AMINOACI configuration. This demonstrates that although durability degrades performance, much of the loss can be made up for by inserting explicit transactions.

5.6 Sendmail

We ran a Sendmail 8.13.4 server and varied the backing store for the `/var/mail` directory where user mailboxes are stored. 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 the Amino monitor, because it does not access the mail files. Instead, it delegates that task to the local mailer. We used the default local mailer for the VANILLA configuration. 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. Instead of using the default local mailer, we wrote a simple replacement that uses an Amino transaction to provide ACID properties for the AMINOTXN configuration (see Section 2.5.4). The STRACE and AMINOTRACE configurations monitored the `mail.local` program. The DEVNULL configuration discarded the message.

For our benchmark, we developed a Perl script that stress tests the mail server by continuously sending mail. 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. The client could use up to 32 concurrent threads to send messages (fewer threads are used if the rate limit is met).

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 5.23. 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 a request rate 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 a request rate of 10, 15, and 20 MPS, respectively. This represents a decline of 13.9%, 18.4%, and

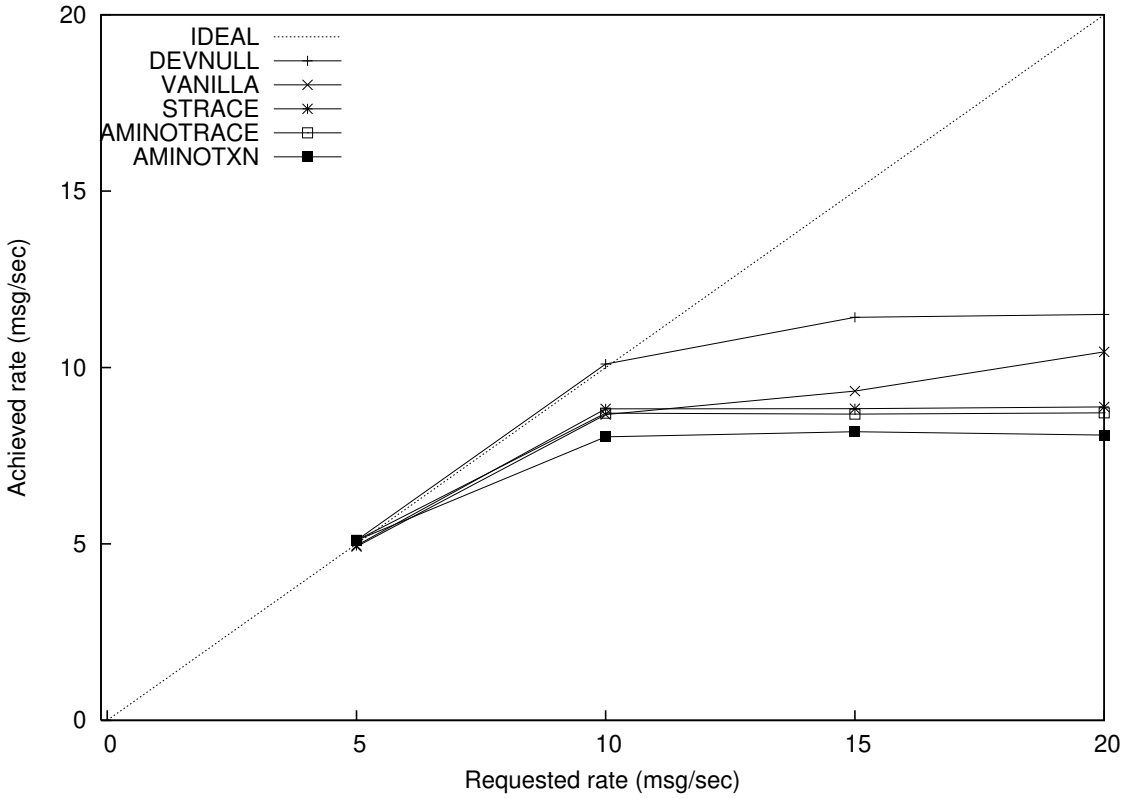


Figure 5.23: *Local mailer: requested vs. achieved load.*

9.6% from the DEVNULL configuration. 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. This is a reduction compared to VANILLA 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.

5.7 General-Purpose Benchmark Summary

Our general-purpose benchmarks showed better performance than the micro-benchmarks.

Although Postmark is I/O-bound on Ext3, it is CPU bound on Amino. Our improved performance for small files resulted in a 31.9% increase in Postmark transactions per second. However, when larger files were used, the performance was identical. The performance drop for larger ten times larger files is expected based on the results of our data micro-benchmarks, which operated on a large file. Increasing the number of files decreased performance for Amino and Ext3, but the decrease for Ext3 (78.3%) was greater than for Amino (28.7%).

The OpenSSH compile is a CPU-bound workload, so tracing overheads cannot be overlapped with I/O. For the system-call–intensive configuration phase, the tracing configurations were between 69.6–98.2%. The AMINOACI configuration was only 9.0% slower than AMINOTRACE. During the build phase, fewer system calls are performed. Therefore, the tracing configurations had smaller overheads ranging from 30.9–31.5% over VANILLA. The AMINOACI configuration had an overhead of 45.2%, caused by a CPU time increase attributable to BDB operations, additional data copies, and context switches.

For Sendmail, the AMINOTXN configuration performed between 7.2–22.6% fewer operations than VANILLA. However, this decreased throughput comes with the benefit of

transactional consistency and significantly lower code complexity.

All in all, our prototype is suitable for user-like workloads, and an optimized implementation is likely to achieve performance that is competitive with Ext3.

Chapter 6

Related Work

In this section we discuss five classes of related work. In Section 6.1 we describe systems that integrate transactions with the file system. In Section 6.2 we describe file systems built on top of databases. In Section 6.3, we discuss log-structured and journaling file systems, which both use similar techniques to database systems to provide atomic updates. We discuss transactional memory systems in Section 6.4, which are complementary to transactional file systems. Section 6.5 we describe various system-call interception methods, which are related to our monitoring infrastructure.

6.1 Transactions and File Systems

In this Section we describe Seltzer's support for transactions in a log-structured file system, WinFS, and Quicksilver.

Transaction Support in a Log-Structured File System Seltzer's simulations of transactions embedded in the file system showed that file system transactions can perform as well as a DBMS in disk-bound configurations [73]. 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%. In later work, Seltzer implemented a transaction processing system embedded in a log structured file system (LFS) and compared it to a user-space transaction processing system running over LFS and the same user-space transaction processing system running over a read-optimized file system [75]. Seltzer found that LFS performed better than a read-optimized file system for the transaction processing workload both in user-space and the kernel. Moreover, a transaction processing system integrated with an LFS performed better than either user-space solution, because before-images do not need to be written to the log in a no-overwrite file system (like LFS) and synchronization was more efficient. The simulations actually predicted that synchronization would be faster in a user-space transaction processing system, but the DECstation on which the benchmarks ran did not have a test-and-set primitive. This required the user-space transaction processing system to use two semaphore system calls for locking. On an architecture with a test-and-set primitives, the user and kernel implementations should

have similar performance. The transactional log-structured file system had only a negligible (1–2%) performance impact for non-transactional workloads.

Viewed in the context of Amino, this work has two major consequences:

1. Running the database transaction manager in user-space instead of the kernel should not have inherent limitations. As evidenced by our evaluation, much of our overhead was derived from `ptrace`. If we can improve the performance of `ptrace`, our file system should be able to perform similarly to a traditional file system.

2. An LFS may be slightly more appropriate for a file system built on a database due to its no overwrite behavior. The simulations indicated the LFS should be faster, because there was no need to write before-images to the log (as they exist elsewhere on disk). However, the simulations did not take into account the LFS cleaner, which would block transactions during cleaning. This narrowed the gap between expected performance based on simulations and actual performance based on benchmarks.[1]

WinFS WinFS is part of an upcoming version of Microsoft Windows [48] (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 (as evidenced by the delay of Longhorn and subsequent removal of WinFS). 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 QuickSilver is a distributed operating system developed by IBM research that makes use of transactional IPC [70]. 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 (assuming the server has non-volatile state). All resource management and notification in QuickSilver are handled by transactions. For example, on process termination (commit or abort) the window manager destroys all windows; the virtual terminal server closes the standard input and output file descriptors; and the task manager kills all of its children. The use of transactions removes the need to handle local vs. remote processes differently. 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.

[1] In hindsight, this result seems somewhat intuitive. The write-ahead log on the read-optimized file system can be viewed similarly to the in-place writes on the LFS. Flushing dirty buffers in the read-optimized file system is analogous to cleaning old segments in the LFS. The LFS probably still has some advantages, however, because the schedule of the LFS cleaning can be more malleable than general data writeback on a traditional file system (which non-transaction processing applications rely on for consistency).

6.2 File Systems built on Databases

In this Section we describe the Inversion File System, the Ode File System, Oracle's Content Management SDK, and DBFS.

Inversion The Inversion File System [56] 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. Our `ptrace`-based design does not require any application modification, and makes it possible to provide more OS services (e.g., memory-mappings). Moreover, the transactions API provided by Inversion is also more limited than the one provided by Amino. Inversion does not support nested transactions or the shared transactions API that Amino does. Our richer transaction API allows us to modify applications such as make to provide transactions for applications that do not support them (e.g., `gcc`). POSTGRES has support for SQL query processing, query-execution planning, network access, and stored procedures. Inversion uses each of these features, making it more resource intensive, and hence less suitable for use by performance-sensitive OS components such as the file system.

Inversion does, however, provide some notable features that Amino does not: *typed* files and *time travel*. Inversion was designed for scientific research and supports typed files, that have a specific structure or meaning to their data. For example, Inversion supports several formats for satellite images with functions to access individual pixels. Inversion also leverages POSTGRES's no-overwrite storage policy to provide time travel for user applications. As data is never overwritten, historical copies of data are always available for users. This allows Inversion to provide functionality that is similar to comprehensive versioning file systems like CVFS [84].

OdeFS OdeFS [17] was designed to provide a file-system interface to objects already in the Ode object-oriented database. This allows standard tools to manipulate these objects, alleviating the need to build a set of tools analogous to those already extant on Unix systems (e.g., `grep`, `vi`, and `lpr`). However, for each type of Ode object, new methods must be defined for read, write, and other file-system operations. This essentially makes OdeFS a framework for developing other types of file systems, because each type of object has differing semantics.

OdeFS is implemented as a user-level NFS servers that combines the Ode database backend with a traditional Unix file systems. Thus, OdeFS includes some aspects of unification [59] as well as access to objects in a database. This leads to some non-intuitive situations in which programs like `cp` do not work, because OdeFS cannot distinguish between writes to Unix files or Ode objects. The use of the NFS protocol comes with some disadvantages. Notably, NFS does not include `open` or `close` operations, so OdeFS cannot determine when a user has finished updating an object. This also makes error reporting problematic, because the data that a user writes to an object may not be correct and this

complicates when that error should be reported. Additionally, the NFS clients may cache data that they write and thus OdeFS "plays games" with file modification times to mitigate this problem. These limitations are related to the major reason that Amino could not be implemented as a user-level NFS server. ACID properties cannot be extended to applications using an NFS server, because there are no mechanisms for adding new primitives (e.g., begin or commit) to NFS and the NFS client cache can serve requests without consulting the database system (thus defeating isolation).

Oracle Content Management SDK Oracle Content Management SDK (formerly known as Oracle Internet File System or iFS) is a file system that is built on top of Oracles relational database [57]. iFS is designed to be robust and scalable shared file system for content management. It provides access to files via several Internet protocols: Samba, NFS, HTTP, FTP, SMTP, POP3, and IMAP4. As a shared content management system, iFS has several features to improve collaboration: file versioning, check in and checkout, searching, and email notifications when documents are modified. iFS provides convenient access to files using a variety of network protocols, but the clients may cache data and transactions are not available using these protocols.

DBFS The Database File System (DBFS) is a block-structured file system developed using BDB [52]. DBFS uses BDB's transaction, caching, and logging components to build a file system that provides consistency similar to FreeBSD SoftUpdates [46] (that is atomicity, consistency, and isolation) at the file system level, but did not extend transactions to applications.

DBFS is implemented as a user-level library and a user-level NFS server. The user-level library does not use a POSIX API, so applications must be modified and relinked to use DBFS. The NFS server alleviates this to some degree, because unmodified applications can use the standard NFS client in the kernel. The use of a library does not eliminate the possibility of extending transactions to applications, but DBFS did not explore this possibility. As we previously described, NFS servers cannot provide transactional semantics to applications.

The database schema of DBFS is somewhat similar to Amino's, but there are several key differences. The DBFS schema is made up of three databases (DBFS refers to them as tables): a `dirtree` database, a `metadata` database, and a `blocks` database. The most similar database is the `blocks` database, which corresponds to our Data database. The `blocks` database maps unique identifiers and a block index to the actual data. The main difference is that our Data database also includes the information in DBFS's `metadata` database. Our design improves locality, because the metadata is stored together with the data. The `dirtree` database is analogous to our Path database. The primary difference is that DBFS's `dirtree` database is structured like a Unix directory tree (i.e., a parent inode number points to a child inode number and name). Additionally, DBFS uses a single key with multiple values for each directory. The DBFS authors state that they cannot skip to a specific directory entry, therefore to read an n-sized directory $O(n^2)$ cursor walks are required. A consequence (though not mentioned by the DBFS authors) of this fact is that the lookup operation is also $O(n)$. However, we do not believe that this

is a fundamental limitation of using a Unix like directory, but more probably an artifact of DBFS's schema implementation. It should be possible using a properly structured data item and sort function to use the `DB_GET_BOTH` cursor flag to provide $O(lgn)$ lookups and the `DB_GET_BOTH_RANGE` cursor flag to resume directory reading.

The primary goal of DBFS was to determine the performance characteristics of a file system that was built on a database compared to FFS. In the abstract the DBFS authors say that DBFS's performance was 50–80% slower than FFS. However, this is true only for asymptotically large data transfers (e.g., 8MB). For page-sized data transfers, the performance was on the order of 5 times slower for reads and 30–40 times slower for writes. However, meta-data operations were an order of magnitude fastr than FFS without Soft Updates, but slower than FFS with Soft Updates. We have found that Amino shows similar performance to DBFS for reads, but is faster for writes (with a slowdown of roughly 3.2 times). For meta-data operations, we found that Amino was slower than Ext3, but much of the slowdown we observed was related to `ptrace` (the DBFS evaluation was performed using direct DBFS library calls).

6.3 Log-structured and Journaling File Systems

Log-structured and journaling file systems borrowed the technique of write-ahead logging from databases [26, 38, 67]. 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. Thus, Amino is similar to a journaling file system in that when updates are made, they are first written to the database log file and then written to their permanent locations within the database file.

Standard 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 (the notable exception being Seltzer's work described in 6.1). 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.

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.

6.4 Memory Transactions

Providing transactions on the file system is a useful first step towards providing fully transactional semantics to applications. However, applications also have in-memory data struc-

tures that need to be kept consistent. Therefore, we believe that providing transactional semantics for in-memory data is complementary to providing transactions for on-disk data. 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.

Lightweight Recoverable Virtual Memory Lightweight Recoverable Virtual Memory (LRVM) was developed to simplify Coda servers [69]. 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. 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 a 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.

Rio The Rio, or RAM I/O, project sought to bring persistence to standard memory [8]. 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 or software errors. Power failures can be solved through the use of UPSs. To cope with software errors, two approaches are taken. First, Rio memory uses page protection and checksums to prevent an errant instruction from writing to it. To update a page, it must be made writable, then the update is performed, and finally the page is made read-only again. Along with the update, checksums are stored along with the data so that errors can be detected. These two mechanisms raise the bar for updating memory, so that an errant instruction is unlikely to corrupt Rio memory, and even if Rio memory is corrupted, the change can be detected with a checksum. The second approach that Rio uses is saving memory across warm reboots. After a system crash, the machine is rebooted, but the memory contents are preserved. Before the OS is fully booted, the memory is written to a swap partition. After the OS is booted, the contents of Rio memory are restored from the swap partition.

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 [18, 49].

Vista is a transactional RVM built on top of Rio [42]. 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.

6.5 System Call Interception

Using the `ptrace` interface allowed us to develop Amino much more quickly than if we were to have modified the kernel directly. In this section, we describe several projects that leverage system call interception to provide new OS-like functionality: the Ufo global file system, Janus, Systrace, SLIC, and Interposition Agents.

Ufo The Ufo Global File system uses a similar interposition technique as our monitor [1]. 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.

This is in marked contrast to the Amino monitor, which does not rely on the lower-level file system for any functionality. Like any design decision this comes with some advantages and disadvantages. The major advantage of Ufo's design is that it is simpler and just "patches" a handful of system calls. This makes compatibility for the remainder of system calls trivial in Ufo, whereas Amino must carefully implement each system call to provide compatibility. This also has some positive performance implications for Ufo, because it does not need to handle calls like `read` and `write`, the number of data copies and context switches are reduced. Not handling all of the system calls also has some negative performance implications: for example, to read a few bytes of a large file in Ufo the whole file must be written to temporary storage during the `open` call. In Amino's architecture the file does not need to be written to temporary storage, instead the monitor simply reads as many bytes as required.

The major disadvantage of Ufo's design is that, because it handles just a handful of system calls it cannot override kernel functionality for the unimplemented system calls. For example, Ufo could not provide encryption functionality, because it would have to write the unencrypted data to disk during `open`; thus defeating the purpose of an encryption file system. Similarly, Ufo is not able to provide fine-grained transactional semantics for file data, because the Ufo file system delegates all file data operations to a standard disk based file system. For example, a transactional file system under Ufo cannot handle concurrency on a single file. The Ufo monitor must manually provide whole-file locking (i.e., it cannot rely on a DBMS as in Amino), because it does not know where reads and writes take place within a file (or even if they take place at all until `close`). Moreover, all directory operations must use similar locking, because of the reliance on stubs.

Interestingly, the Solaris's `/proc` interface provides two pieces of functionality not available under Linux: (1) writing to the process address space is enabled by default, and (2) the monitor can select which system calls to intercept. This provides functionality

that is equivalent to a read-write `/proc/pid/mem` file (described in Section 3.5) and `PTRACE_SELECT` (described in Section 3.7).

The performance of Ufo was poor compared to the existing operating system. For the intercepted system calls the performance was 14.3–21.8 times slower than the standard OS, even for files that Ufo did not handle (this configuration is similar to AMINOTRACE. For files handled by Ufo (this configuration is similar to AMINONULL), the performance was 52.8–128 times worse. Amino performed similarly, causing a slowdown of roughly 10–20 times for tracing and emulation (see Appendix A.2 for full results). For larger-scale benchmarks, Ufo fared better. For the Andrew benchmark, the Ufo was 33% slower for while tracing, and was 67% slower for files that it handled. This benchmark is most similar to our compile benchmark, in which Amino had an overhead of 47% for tracing and 48% for the AMINONULL configuration. For the `iostone` benchmark, Ufo was 8.33 times slower for tracing and 34 times slower for files that it handled.

Janus The `ptrace` interface was used by the Janus framework to sandbox untrusted applications [21]. Janus monitors file-system and network-related system call invocations, and applies configurable policies to allow or deny system call execution.

Systrace Systrace [63] is a system call interception framework for improving host security. Systrace intercepts system calls in the kernel and passes a configurable set of calls to a user-level monitor. In this way it is similar to our `PTRACE_SETCALLS` patch. Systrace enforces user-defined security policies (e.g., the application may bind to port 53 or open `/var/named`) and provides an interactive policy generation tool. Systrace also allows a policy to elevate privileges for individual system calls, thus removing the need to run entire setuid programs as root. Because Systrace is focused on security it takes precautions that `ptrace` monitors can not, including resolving all symbolic links before executing calls and copying all of the process's arguments to the kernel's address space. This prevents malicious processes from exploiting TOCTOU vulnerabilities (e.g., passing innocuous arguments that are verified by the monitor, and then having a sibling thread modify them before the kernel executes the call [85]). Systrace originally required kernel modifications to intercept system calls on Linux, Mac OS X, NetBSD, and OpenBSD, however, recent Linux versions can use `ptrace` to intercept system calls without kernel modification.

SLIC Several other interposition techniques operate at the same logical system-call level as Amino, but use a customized interface. SLIC [19] 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 Interposition agents provide higher-level abstractions for system call interception [33]. 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).

Chapter 7

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 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 believe that file systems should provide transactional semantics to applications. Furthermore, we contend that a 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 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, transactions can be added to unmodified processes. We designed four sample applications using our transactional API: a transactional Postmark, a version of GNU Make that adds transactions to the build process, a transactional version of GNU `tar`, and a transactional local mailer.

We evaluated our `ptrace` file system infrastructure, and showed that file systems can be developed using our monitor with lower complexity than in the kernel and that the complexity is similar to other techniques, such as FUSE, but our monitor provides functionality that FUSE does not. Our performance evaluation shows that our file systems have acceptable performance. For micro-benchmarks, our monitor is often several times slower than Ext3, because of the additional CPU time required for `ptrace` monitoring and additional database operations. Our `ptrace` monitoring infrastructure can add new file-system functionality to applications, and in many cases exceeds the performance of the standard

`strace` tool—which simply traces processes without adding any functionality.

For general purpose workloads, the performance impact of Amino is small. The I/O and meta-data-intensive Postmark workload Amino matches and exceeds the performance of Ext3. For our Sendmail benchmark, Amino had overheads from 7.2–22.6% over Ext3, but provided consistency guarantees that Ext3 can not. For more CPU intensive workloads, such as the compile the overheads were larger, 39.5–84.7%. When durability is required, performance inevitably suffers because of synchronous disk writes. For unmodified applications, Amino meets or exceeds the performance of Ext3. For modified applications, Amino performs several times better than Ext3 in durable mode (even approaching the non-durable configurations for the compile). 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.

7.1 Future work

In this section we describe interesting avenues for further research and possible improvements of our system. In Section 7.1.1 we describe possible `ptrace` improvements. In Section 7.1.2 we discuss address space manipulation. In Section 7.1.3 we describe possible extensions to our transactions API. In Section 7.1.4 we discuss database deadlocks. In Section 7.1.5 we describe schema enhancements. In Section 7.1.6 we describe ExtAcid, a proposed ACID file system that is compatible with Ext2.

7.1.1 Possible `ptrace` enhancements

In this section we describe the problems with $m - n$ threading in Linux's current `ptrace` implementation and suggest an extensible kernel-level `ptrace` filtering mechanism.

$m - n$ threaded `ptrace` monitors Our current monitor is multi-threaded, which is essential for providing good file system performance under concurrent workloads. We used a $1 - 1$ threading model, such that one thread of the monitor monitors a specific child. The main advantage of this model is that it is simpler to implement than a $m - n$ threading model, where m is the number of user-level applications and n is the number of monitors. However, a $m - n$ model may be more suitable for monitoring many processes, particularly those that are user-time intensive such as a compile benchmark.

A $m - n$ threading model would allow a monitor design that is similar to a pre-forked network server, such as Apache [2]. The monitor could dynamically create new monitoring processes as needed, and avoid setting up and tearing down threads for each child process. For example, in the Configuration phase of a compile benchmark, hundreds of very short-lived processes are executed. Eliminating these startup costs could improve performance. Moreover, if all of the processes on an OS are to be monitored, it is unlikely that all of them execute simultaneously and need dedicated threads (e.g., on the author's laptop there are over 100 processes running while the machine is idle).

Unfortunately, `ptrace` does not permit a true $m - n$ threading model, where a given thread can be serviced by any of n monitor threads. Instead, the `ptrace` API ties a thread

to a specific monitor thread, so it is possible to statically assign a subset of threads to a given monitor thread. We believe that a pool of monitor threads should be able to dynamically service a given number of processes. This is a non-trivial kernel change, because there is a direct link from a process to the thread that is monitoring it, but it is possible to modify the kernel such that the link from the monitored process is to the monitoring process (including all of its threads), rather than to just the individual threads within the monitoring process. The `ptrace` API itself could remain essentially unchanged (and if minor changes are required, then the `PTRACE_SETOPTIONS` primitive could enable them), thereby preventing incompatibilities with existing tracing software.

Kernel-level filtering We are also considering porting performance-sensitive subsets of the database and the monitor into the kernel. Selecting which calls to pass to the existing OS vs. handling within the monitor are among the most performance sensitive parts of the monitor (e.g., path name resolution and file-table lookups). Porting these facilities to the kernel is essentially an extension of the `PTRACE_SELECT` primitive that we introduced, but rather than simply selecting calls to receive on the coarse-granularity of system call number, the monitor can select calls based on file descriptor or paths. The most trivial way to implement this would be a direct port of the Amino monitor's pathname resolution and file table maintenance code to the kernel with the appropriate system calls for the monitor to manipulate the mount and file tables. This approach has the disadvantage that is it intimately tied to file-system code and even some of our assumptions about file system code (e.g., the mount a file system is on is determined by a prefix of its path).

A more general solution would be to allow `ptrace` monitors to upload small general-purpose code to the kernel to filter requests. This would be useful because tracing monitors, file-system monitors, security monitors, and others all take advantage of this type of functionality. In addition, a small shared kernel-user segment can be created so that the monitor can dynamically change the filtering criteria (e.g., by updating a file table). Care must be taken any time untrusted code is executed within the kernel. Various methods can be used to ensure kernel safety for such `ptrace` filtering extensions. The Packet Filter allows user-level processes to upload simple byte-code programs for demultiplexing packets into the kernel [50]. The packet filter provides safety by using an interpreted stack-based language that does not allow arbitrary memory-references. Dispatching system calls to the kernel or monitor is essentially a form of demultiplexing, so such a solution is likely to be especially suited for this domain. The BSD Packet Filter extends this model to include registers [45], and BPF+ dynamically verifies a filter's safety and compiles it into native code [3]. VINO [72], SPIN [5], and Exokernel [14] are extensible operating systems. Each of these OSes takes a different route to ensure safety: VINO uses software fault isolation to prevent code from accessing invalid memory and transactions to prevent resource hoarding; SPIN requires extensions to be written in the type-safe Modula-3 language; and Exokernel allows multiple independent library OSes to run on a single machine. Palladium [9] runs kernel modules at a privilege level between that of the kernel and user-space using x86. Cosy allows user functions to run at the kernel level, but prevents them from accessing anything other than a shared buffer using x86 segmentation [64, 65]. One or more of these safety mechanisms could be employed to provide the monitor with sufficient capability to

modify a shared data segment and the kernel-code segment to use them in conjunction with the process's registers and address space contents to make filtering decisions.

7.1.2 Address Space Manipulation

The monitor is able to read and write from the process's address space, but cannot change it. Our monitor needs to manipulate the address space of the user applications as well as its own address space. Unfortunately, the mechanisms Unix provides for address space manipulation are lacking. Specifically, `ptrace` should provide an API that allows to perform the following actions:

- Insert new regions into the a process's address space as if the process were calling `mmap`.
- Remove regions from a process as if the process were calling `munmap`.
- Change the protection bits of pages in a process's address space as if the process were calling `mprotect`.
- Duplicate part of its own address space (or that of a monitored process), into a different part of its address space (or that of a monitored process).

The first three items (`mmap`, `munmap`, and `mprotect`) are a logical extension of the monitor reading to and writing from the process's address space. To get around this limitation our monitor inserts these calls into the system call stream. This is relatively clean for `mmap` and `munmap`, because our monitor only issues these calls when the process is already executing a system call. However, the monitor must force user processes to issue `mprotect` system calls at arbitrary points during the page fault handler. This involves inserting machine code into the process's address space, updating the instruction pointer and other registers, and then restoring them. This works, but is less efficient and more error-prone than the kernel providing a mechanism to accomplish the task.

There are several times throughout the design of our system that a flexible method of grafting parts of one address space onto another would be useful. The most basic example is that the monitor would no longer need to inject system calls for establishing the System V shared memory region. Also, implementing shared mappings could be simplified considerably if the monitor could map the pages that represent a file in the monitor into other processes. Another example that does not directly relate to the monitor is that when implementing a recoverable virtual memory system, it would be convenient if the application could map the same page into its address space read-only in one location and read-write in another. This would allow efficient explicit logging functions in concert with a robust page-fault based logging mechanism. Finally, System V shared memory could be implemented in terms of such an API.

7.1.3 Transactions API

We describe two types of possible modifications to our transactions API: exposing more BDB functionality to user applications and a language for defining transactions profiles.

Exposing BDB Functionality We plan to expand our transactions API to allow applications to select a degree of isolation. By default, BDB provides the highest degree of isolation, *repeatable reads*, which means that a given transaction can repeatedly read a data item without it changing [23]. There are three weaker levels, *cursor stability*, *browse*, and *anarchy* (present file systems support only anarchy). Lower-levels allow one process to interfere with another, but provides better performance. This trade-off is appropriate for some applications (e.g., `locate` does not need repeatable reads, as it only accesses each file once). Fortunately, using the proper locking protocols, each of these isolation levels can peacefully coexist, without negatively impacting transactions at a higher isolation level.

Transaction Profiles In Section 2.4 we proposed using transaction profiles to automatically provide transactions for existing applications. Currently, implementing a transaction profile requires inserting C code into the monitor to begin and end transactions where required. For example, the profile that protects an entire application's execution is inserted into the `on_new_pcb` and `on_free_pcb` events. An improved solution would be to provide a simple language for defining extensions rather than requiring monitor source code changes. For example, if the monitor exposes system entry, exit, and notifications, then declarative rules could be used to create a finite state automaton augmented with instructions to begin, commit, or abort transactions. In this way, the transaction profiles would be quite similar to regular expressions over the language of the system calls.

7.1.4 Database Deadlocks

Any sufficiently complex database application is bound to exhibit deadlocks, and Amino is no exception. Fortunately, BDB provides automatic deadlock detection, and returns the error code to `DB_DEADLOCK` when a deadlock occurs. To detect deadlocks, BDB reads the lock table and constructs a "waits-for" graph [83]. A locker waits for another locker, if it is requesting a lock that the other transaction holds. For example, if transaction A holds a lock and transaction B is requesting the lock, then B waits for A. After constructing this graph, BDB searches for cycles in the graph, which indicate a deadlock has occurred.

The primary limitation with this method is that BDB restricts each thread of control to issuing database calls using a single transaction. When multiple transactions are used in a single thread, *self deadlock* can occur. Consider the situation where a single thread begins transactions A and B. If A requests an object, and then B requests the same object, then neither transaction can make forward progress, because B is waiting for A and A will not be scheduled. This is not detected as a deadlock, because the waits-for graph contains only the edge $B \rightarrow A$. More complicated scenarios also exist. Self deadlocks are possible because Berkeley DB does not know about thread identifiers. If the waits-for graph was augmented such that for each thread waiting on an object, edges are added from other lockers in the same thread of control to that waiting locker, then cycles would be formed and deadlocks could be detected. In the previous example, B is waiting on an object held by A so the graph contains the edge $B \rightarrow A$. Because B is waiting on an object, all other lockers in the same thread wait on B so the edge $A \rightarrow B$ is added, thus forming a cycle. This improvement to deadlock detection would allow concurrent child transactions, an $m - n$

threading model, and prevent deadlocks on transaction suspend. Finally, the OS could also augment the waits-for graph with other relationships (e.g., a blocking pipe reader waits for the writer).

When two or more process's share a transaction stack, we currently serialize their operations using a mutual exclusion lock. Performance would be improved if each of the processes could begin their own child transaction and perform operations concurrently. The reason that concurrent child transactions are not currently supported in BDB is that all of the child transactions are mapped to their parent for locking purposes, because if the children were assigned different locker IDs, then they could self deadlock [71]. However, if they were assigned different locker IDs and were in different threads of control they would not self-deadlock; but true deadlocks are possible and must be detected. One possible solution to this would be to use a new database flag (e.g., `DB_TXN_THREAD`). However, this solution would not permit the use of child transactions in the same thread of control (as BDB now supports) and in different threads of control at the same time. Using the self-deadlock aware waits-for graph would allow such child transactions. After the BDB infrastructure is put in place, our file system's transactions API would need to be modified to support a tree of transactions rather than a stack.

We use a $1-1$ threading model for our monitor for two reasons: (1) it is dictated by `ptrace` limitations, and (2) it ensures that each user thread maps to a single monitor thread. This second constraint prevents self deadlock. If two user threads began transactions and were handled by the same monitor thread, then the monitor thread would interleave calls for two transactions, thus introducing the possibility of self deadlock. If self-deadlock was handled by BDB, then the monitor could use an $m-n$ threading model (assuming `ptrace` was also updated).

The transaction suspend primitive allows applications to issue calls against two transactions at the same time, introducing the potential for self deadlock. If BDB were self-deadlock aware, then this problem would be eliminated.

7.1.5 Schema Improvements

We also plan to further improve the performance of Amino's database schema. Currently our Data database uses a balanced tree. We plan to use a custom access method that will write pages to a file (or possibly raw disk) directly. This will give us more control over data placement, and allow us to align data properly with the native OS page size. BDB's modular design means that we can use the existing locking, logging, transaction, and caching components. This should help 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 and how it is accessed that would improve concurrent access. For example, by reordering operations to the Data database, we were able to provide performance equivalent to a single thread for multi-threaded workloads. Performing similar optimizations for the Path database is likely to yield similarly positive results. We are also considering using a more Unix-like schema for the Path database. In Unix each directory is also a normal file that can be moved by simply unlinking it from one parent and linking it from another. To rename directories using our schema the number of database operations required is linearly dependent on the number of

descendent's of the directory. Although renaming directories is not very common, this does change the expected performance characteristics of a file-system operation. If a more Unix-like schema is used, however, it is likely that certain paths will become a bottleneck (e.g., `/`) while traversing the directory structure. One possible answer to this is using a summary table that has a schema much like the current path database. This summary table can be lazily populated with full path names, and if directory renames occur it can then be entirely removed (a constant time operation). The key performance question here is whether maintaining the summary table is more expensive than the additional queries it is designed to save. Moreover, maintaining the expected performance characteristics of directory rename may not be worth the extra effort. For improved efficiency, the summary table could be stored in a memory-only BDB environment. If this summary table is maintained as an in-memory database, it essentially devolves into a directory-name-lookup cache (DNLC) within the monitor. This scheme ends up being rather similar to a sprite prefix cache, which maps path name prefixes to the appropriate server [58]. In this case, the prefix cache would instead map prefixes to database entries.

7.1.6 ExtAcid

Our investigation centered around developing a transactional file system using database components for data organization. A logical alternative is to use an existing file system format such as Ext2 for data organization. There are a large number of installations with existing Ext2 and Ext3 file systems. The data stored on these file systems is valued, but the code that accesses these file systems is interchangeable. For example, FreeBSD and Linux can both access Ext2 file systems. We call our proposed Ext2 implementation that exposes ACID semantics to the user level *ExtAcid*. ExtAcid should have the exact same on-disk format as Ext2, making migration between an ACID and a non-ACID file system as trivial as passing different options to `mount`. If ExtAcid and Amino provide the same user-level transaction API, applications could use either ExtAcid or Amino seamlessly.

BDB is a highly modular software toolkit. It is possible to use some facilities of BDB without using others. To provide ACID properties using the Ext2 format, one could use the logging, locking, and caching components of BDB. Instead of using a BDB access method (the components that define the on-disk structure of BDB databases), an Ext2 partition can be directly accessed, leveraging existing Ext2 code where appropriate. By mediating these accesses using the BDB lock, logging, and cache management facilities, it is possible to provide ACID guarantees.

As part of this investigation, alternative ExtAcid-specific logging and locking components should be developed. The performance and development time implications of using general-purpose database components vs. more finely tuned components can then be explored. For example, BDB performs page-level locking, which can falsely limit concurrency when many distinct objects are stored on a page (e.g., in the inode bitmap). A custom locking component would be aware of the differences between bitmap, inode, and data pages and thus have finer lock granularity when appropriate.

To obtain copies of the micro-benchmark programs used in this dissertation go to `www.`

fsl.cs.sunysb.edu/~cwright/benchmarks/.

Bibliography

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

[2] The Apache Group. *Apache HTTP Server Reference Manual*, 2.2 edition, December 2005. `http://httpd.apache.org/docs/2.2/mod/prefork.html`.

[3] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 123–134, Cambridge, MA, August/September 1999. ACM.

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

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[6] M. Blaze. A Cryptographic File System for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, 1993. ACM.

[7] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.

[8] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajmani, and D. Lowell. 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)*, pages 74–83, Cambridge, MA, October 1996. ACM.

[9] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 140–153, Kiawah Island Resort, near Charleston, SC, December 1999. ACM SIGOPS.

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

[11] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

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

[13] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*, San Diego, CA, October 2003. USENIX Association.

[14] D. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium*

on Operating System Principles (SOSP '95), pages 251–266, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[15] The Free Software Foundation, Inc. GDB: The GNU Project Debugger. `www.gnu.org/software/gdb/gdb.html`, January 2006.

[16] The Free Software Foundation, Inc. *GNU Make Manual*, April 2006. `www.gnu.org/software/make/manual/`.

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

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

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

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

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

[22] V. Gough. Encfs, November 2005. `http://arg0.net/wiki/encfs`.

[23] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*, chapter 7: Isolation Concepts, pages 375–445. Morgan Kaufmann, San Mateo, CA, 1993.

[24] A. Grünbacher. POSIX Access Control Lists on Linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 259–272, San Antonio, TX, June 2003. USENIX Association.

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

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

[27] M. A. Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, pages 201–218, Ottawa, Canada, July 2005. Linux Symposium.

[28] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[29] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 3–6, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[30] S. Henry, D. Kafura, and K. Harris. On the relationships among the three software metrics. In *Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality*. ACM Press, March 1981.

[31] D. R. Hipp. SQLite. `www.sqlite.org`, February 2006.

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

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

[34] A. S. Kale. The KGDB home page. `http://kgdb.sourceforge.net`, 2001.

[35] A. Kashyap. File System Extensibility and Reliability Using an in-Kernel Database. Master's thesis, Stony Brook University, December 2004. Technical Report FSL-04-06, `www.fsl.cs.sunysb.edu/docs/kbdbfs-msthesis/kbdbfs.pdf`.

[36] A. Kashyap, J. Dave, M. Zubair, C. P. Wright, and E. Zadok. Using the Berkeley Database in the Linux Kernel. `www.fsl.cs.sunysb.edu/project-kbdb.html`, 2004.

[37] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[38] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, and E. R. Zayas. DEcorum File System Architectural Overview. In *Proceedings of the Summer USENIX Technical Conference*, pages 151–164, Anaheim, OH, June 1990. USENIX Association.

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

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

[41] T. Littlefair. *An Investigation into the use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Faculty of Communications, Health, and Science, Edith Cowan University, Mount Lawley Campus, June 2001.

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

[43] D. Maziéres. A Toolkit for User-Level File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 261–274, Boston, MA, June 2001. USENIX Association.

[44] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[45] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Technical Conference*, pages 259–269, San Diego, CA, January 1993. USENIX Association.

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

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

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

[49] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors: a case for recoverable programming models. In *Proceedings of the 9th ACM SIGOPS European workshop*, pages 97–102, Kolding, Denmark, 2000. ACM Press.

[50] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 39–51, Austin, TX, October 1987. ACM Press.

[51] D. Morozhnikov. FUSE ISO File System, January 2006. `http://fuse.sourceforge.net/wiki/index.php/FuseIso`.

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

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

[54] National Institute of Standards and Technology. *FIPS PUB 197: Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, November 2001.

[55] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, Monterey, CA, June 1999. USENIX Association.

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

[57] Oracle Corporation. Oracle Content Management SDK. `http://www.oracle.com/technology/products/ifs/`, March 2006.

[58] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[59] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 25–33, New Orleans, LA, December 1995. USENIX Association.

[60] B. Perens. *efence(3)*, April 1993.

[61] A. Pnueli. Temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):44–60, 1981.

[62] N. Provos. Encrypting virtual memory. In *Proceedings of the Ninth USENIX Security Symposium*, Denver, CO, August 2000.

[63] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington, DC, August 2003.

[64] A. Purohit. A System for Improving Application Performance Through System Call Composition. Master's thesis, Stony Brook University, June 2003. Technical Report FSL-03-01, `www.fsl.cs.sunysb.edu/docs/amit-ms-thesis/cosy.pdf`.

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

[66] H. V. Riedel and R. Bernstein. GNU Compact Disc Input and Control Library. `www.gnu.org/software/libcdio/`, October 2005.

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

[68] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, Charleston, SC, December 1999. ACM.

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

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

[71] M. Seltzer. Deadlock detection and multi-threaded transactions. Personal communication regarding internal Sleepycat Support Request 2997, November 2000.

[72] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the architecture of the VINO kernel. Technical Report TR-34-94, EECS Department, Harvard University, 1994.

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

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

[75] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, Vienna, Austria, April 1993.

[76] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 71–84, San Diego, CA, June 2000. USENIX Association.

[77] Sendmail Consortium. Sendmail home page. `www.sendmail.org`, August 2004.

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

[79] J. Seward, N. Nethercote, and J. Fitzhardinge. Valgrind. `http://valgrind.kde.org`, August 2004.

[80] S. B. Sheppard and E. Kruesi. The effects of the symbology and spatial arrangement of software specifications in a coding task. Technical Report TR-81-388200-3, General Electric Company, Arlington, VA, 1981.

[81] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S Jha. A Logic of File Systems. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST '05)*, pages 1–16, San Francisco, CA, December 2005. USENIX Association.

[82] Sleepycat Software, Inc. *Berkeley DB Reference Guide*, 4.3.27 edition, December 2004. `www.sleepycat.com`.

[83] Sleepycat Software, Inc. *Berkeley DB Reference Guide: Deadlock Detection*, 4.4.20 edition, November 2005. `www.sleepycat.com/docs/ref/lock/dead.html`.

[84] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, CA, March 2003. USENIX Association.

[85] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the Eigth USENIX Security Symposium*, pages 184–189, Washington, DC, August 1999.

[86] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net`, February 2005.

[87] T. J. Walsh. Software reliability study using a complexity measure. In *Proceedings of the National Computer Conference.*, New York, NY, 1979. AFIPS.

[88] C. P. Wright, J. Dave, and E. Zadok. 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)*, pages 47–61, Washington, DC, October 2003. IEEE Computer Society.

[89] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[90] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright. On incremental file system development. *ACM Transactions on Storage (TOS)*, 2(3), August 2006. To appear.

Appendix A

Additional Performance Evaluation

This appendix contains additional performance evaluation and ancillary graphs that are not included in the body of the thesis. Section A.1 describes two additional untar benchmarks. Section A.2 presents a micro-benchmark of our `ptrace` primitives.

A.1 Additional Tar Benchmarks

Although the Unpack phase of the OpenSSH compile produces statistically distinguishable results, it completes too quickly to get measurements that one can be confident in, because of startup effects and the resolution of the CPU time measurements. Therefore, we ran the unpack phase of the experiment using the Linux 2.6.16 source tree as well. The Linux 2.6.16 source tree unpacks to 260MB in 20,433 files and directories. The results for this experiment are shown in Figure A.1 and the non-durable configurations alone are shown in Figure A.2.

The VANILLA configuration completed in 3.5 seconds. The STRACE, AMINOTRACE, AMINONULL configurations had overheads of 146.0%, 88.3%, and 117.5%, respectively. These overheads are slightly higher than for the OpenSSH compile which were 84.4%, 60.0%, and 84.8%, respectively. The AMINOACI configuration was 7.0 times slower than VANILLA (vs. 7.3 times for OpenSSH). Again, this was mainly due to an increase in CPU time from 5.5 seconds to 17.8 seconds. Moreover, the VANILLA configuration utilized the second CPU more efficiently, with a total utilization of 159.3%, whereas AMINOACI only utilized 73.4% of the CPU while it was running. The AMINOTXN configuration added slightly more overhead, and was 7.3 times slower than VANILLA. The VANSYNC configuration was 441 times slower than VANILLA, AMINOACID was 282 times slower than VANILLA, and AMINODTXN was 104 times slower than VANILLA.

Figure A.3 shows a ten-times-larger benchmark for the non-durable configurations. We created an archive with ten copies of Linux 2.6.16, and then measured how long it took to unpack it. The VANILLA configuration takes 90.9 seconds, which is 26 times longer than for a single copy of Linux 2.6.16, because I/O was required. The STRACE, AMINOTRACE, and AMINONULL configurations have overheads of 31.6%, 18.9%, and 26.6%, respectively. The AMINOACI configuration was 3.9 times slower than VANILLA, because CPU utilization increased by 3.6 times and 3.4 times more sectors were written to disk. The AMINOTXN

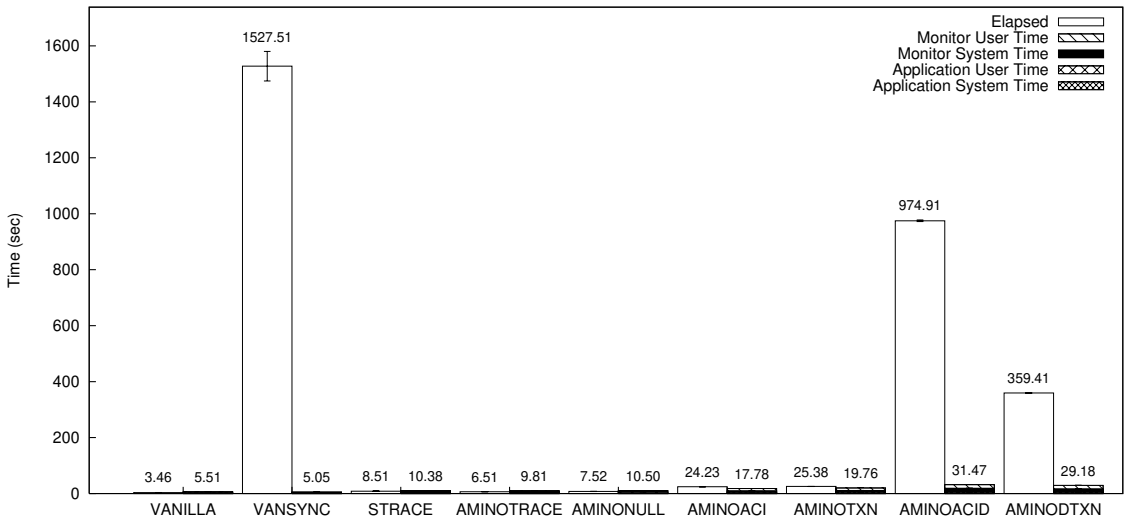


Figure A.1: *Unpack results: Linux 2.6.16. Figure A.2 shows just the non-durable configurations.*

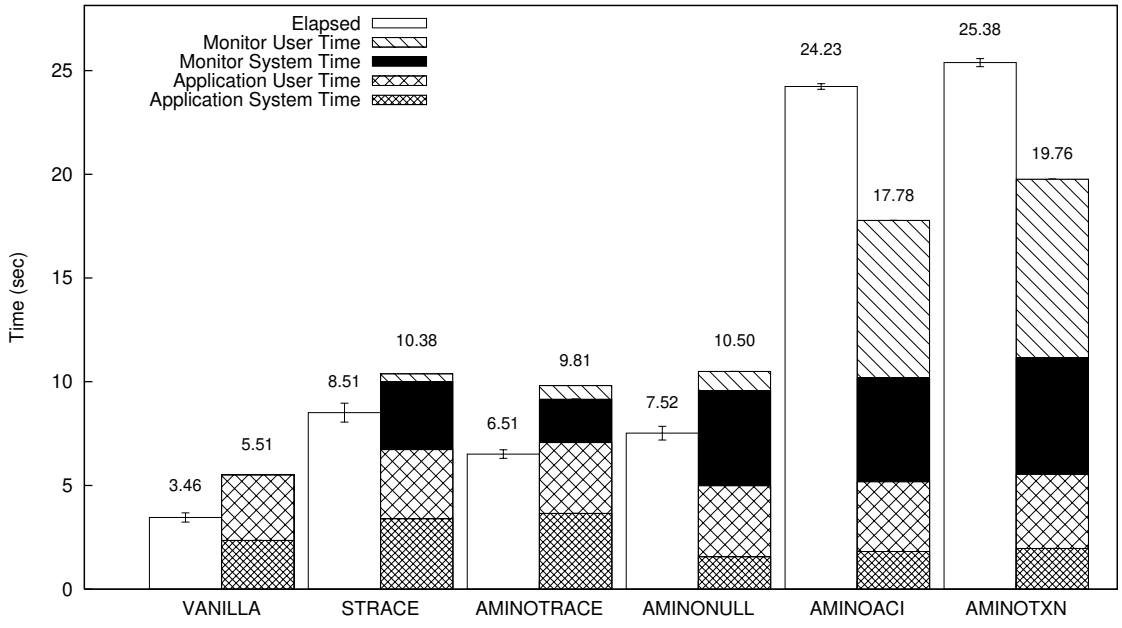


Figure A.2: *Unpack results: Linux 2.6.16. Only non-durable configurations are shown. All configurations are shown in Figure A.1.*

configuration increased CPU utilization by 5.0% over AMINOACI, and was 3.9 times slower than VANILLA.

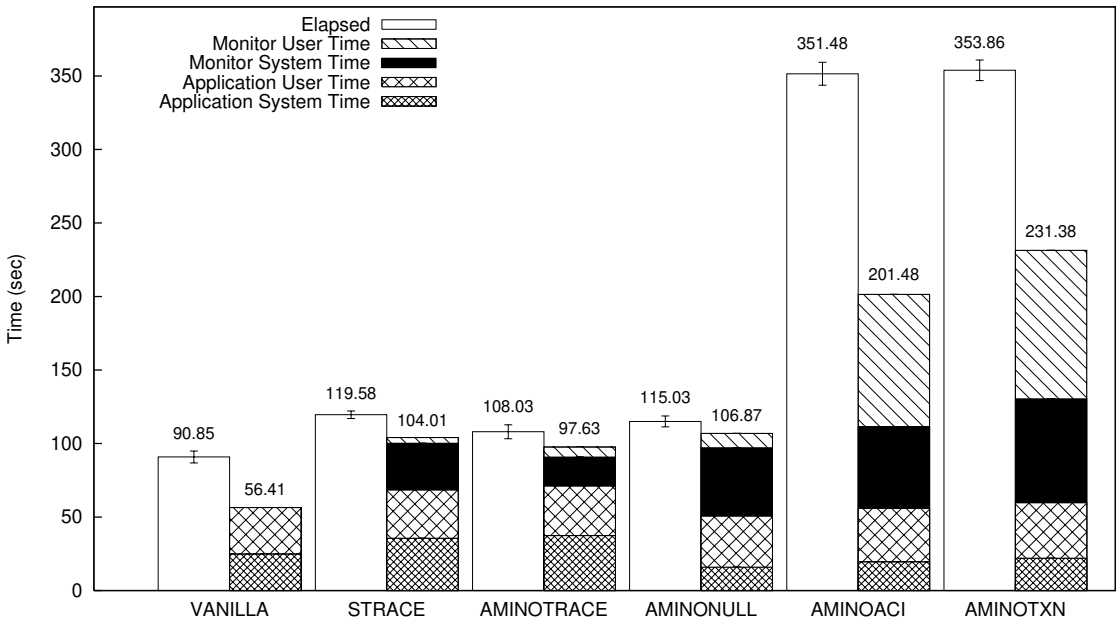


Figure A.3: *Unpack results: Ten copies of Linux 2.6.16.*

A.2 `ptrace` Primitives

We performed micro-benchmarks for `getpid`, `stat`, and `open` to determine the overhead of our ptrace primitives. To time individual system calls, we read the TSC register (which contains the number of cycles since system start) before and after the system call and recorded the difference. We accumulated the total number of calls and cycles for one second, and computed an average. We then repeated this process 50 times and present an overall average.

`getpid` The `getpid` system call shows a worst case overhead for `ptrace`, because it just returns an integer without any additional processing in the kernel. We measured the overhead of six configurations:

- An untraced process.
- An process traced by `strace`.
- An process traced by our monitor, but we used our `PTRACE_SETCALLS` primitive to ignore the call.

- An process traced by our monitor. The `getpid` call entry is traced, and we use `PTRACE_SYSSKIP` to avoid tracing the call's exit.
- An process traced by our monitor. The `getpid` call entry is traced, and we use `PTRACE_CHECKEMU` to force a return value.
- An process traced by our monitor. The `getpid` call is traced, and we use `PTRACE_SYSCALL` to trace the entry and exit.

We present our results in Table A.1. The `getpid` system call takes 556 clock cycles on an untraced process, which we use as our basis of comparison. When the same process is traced by `strace`, the call takes 77,454 cycles or 139 times as long. This is not surprising as `getpid` normally just traps into the kernel, returns an integer, and returns to user mode. When `strace` is monitoring it, the kernel must a context switch from the application to `strace`, and then handle dozens of more complex system calls within `strace` before control is returned to the application.

Configuration	Cycles/call	ns/call	Normalized
Vanilla	556	199	1
`strace`	77,454	27,661	139
Amino—untraced	1,087	388	2
Amino—entry traced	25,726	9,188	46
Amino—emulated	27,980	9,992	50
Amino—entry and exit traced	49,751	17,767	89

Table A.1: *`getpid` micro-benchmark results. Cycles are measured on a 2.8Ghz Xeon. Each cycle is 0.357 nanoseconds. The final column is the time normalized to the vanilla configuration.*

We examined four modes of our monitor, the fastest being when the process is traced, but no traps are made into the monitor (this is the case for non-file-system–related system calls). In this case an overhead of 531 cycles or 96% is incurred, because the kernel must make an additional function call along the system call path to check whether or not the call should be traced. The next two modes of operation trap only on the entry to the `getpid` call. When the entry is traced, there is an overhead of 46 times. This mode is used for file-system-related system calls that are not handled by Amino. When the entry is traced, and a return value is emulated, the overhead is 50 times. This mode is used for most Amino file-system calls. The highest overhead from our monitor comes when both the entry and exit are traced. This mode is only used for a small subset of calls for which the monitor needs to trace the return value (e.g., `open`, `dup`, `exec`, and certain `mmap` operations). In this case, our monitor has an overhead of 89 times, which is 64% of the overhead for `strace`. The reduced overhead is due to Amino more efficiently retrieving the process's registers.

`stat` Table A.2 shows the results for the `stat` benchmark. We used the VANILLA, STRACE, AMINOTRACE, AMINONULL, and AMINOACI configurations described in Section 5. The `stat` call takes 5,980 cycles or 2.1 microseconds on VANILLA, which is 10.6

times as long as `getpid`. The tracing overheads, however, do not increase nearly as much so the normalized overheads are better. STRACE and AMINOTRACE are 14.6 and 14.0 times slower, respectively. Interestingly, the AMINONULL configuration is only 8% worse than the AMINOTRACE configuration. Finally, AMINOACI is 19.8 times worse than VANILLA.

Configuration	Cycles/call	μs/call	Normalized
VANILLA	5,980	2.1	1.0
STRACE	87,239	31.2	14.6
AMINOTRACE	83,512	29.8	14.0
AMINONULL	90,197	32.2	15.1
AMINOACI	118,427	42.3	19.8

Table A.2: *`stat` micro-benchmark results. Cycles are measured on a 2.8Ghz Xeon. Each cycle is 0.357 nanoseconds. The final column is the time normalized to the* VANILLA *configuration.*

`open` The results for `open` (shown in Table A.3) are similar to `stat` in that the VANILLA configuration took more time than `getpid` or `stat` so the overhead is further reduced. The STRACE, AMINOTRACE, and AMINONULL configurations have overheads of 11.4, 10.5, and 11.7 times VANILLA, respectively. The AMINOACI configuration has comparatively higher overhead of 18.9 times VANILLA. The reason that AMINOACI is higher is that more processing is required for an Amino open (e.g., updating the reference count in the database) than for an open on Ext3.

Configuration	Cycles/call	μs/call	Normalized
VANILLA	8,260	3.0	1.0
STRACE	93,954	33.5	11.4
AMINOTRACE	87,024	31.1	10.5
AMINONULL	96,988	34.6	11.7
AMINOACI	155,822	55.6	18.9

Table A.3: *`open` micro-benchmark results. Cycles are measured on a 2.8Ghz Xeon. Each cycle is 0.357 nanoseconds. The final column is the time normalized to the* VANILLA *configuration.*

Appendix B

Monitor VFS Operations

Most of our VFS operations take a `struct pcb` as an argument, which contains information about the currently running process. This context can be used to retrieve register values, examine the open file table, the current transaction, or any other per-process context.

Unless otherwise noted, all operations with an integer return value return 0 or greater on success, and a negative error number (errno) on failure.

The monitor's VFS operations can be divided into two broad classes. Section B.1 describes the first class, which are operations used by the monitor internally. Section B.2 lists VFS operations which map directly to a system call.

B.1 Internal VFS Operations

Amino has two main VFS objects. The first is a `struct mount`, which contains the mount's opaque data related to the mount and an operations vector. The second is a `struct fd_struct`, which represents an open file. The `fd_struct` points to a `mount`, which in turn points to the operations vector.

The monitor has 19 internal VFS operations: three for manipulating mounts (Section B.1.1), seven for files operations (Section B.1.2), two directory-name operations (Section B.1.3), and seven event notifications (Section B.1.4).

B.1.1 Mount Operations

The monitor has three operations to manipulate mounts:

`void` **`cleanup`**`( void *data);`

The `cleanup` operation releases any private `data` associated with the mount. For example, Amino closes the Berkeley DB databases and the encryption file system zeros and frees the key in memory.

`int` **`sync`**`( struct mount *mount);`

The `sync` operation writes all of the `mount`'s outstanding data to disk. When a `sync` system call is issued, this call is executed on each mount.

```
int statfs_internal(struct pcb *pcb,
                    struct mount *mount,
                    struct statfs *buf);
```

Fills the `struct statfs` buffer pointed to by `buf` with the number of used blocks, used files, free blocks, free files, etc. on this file system. This is an internal operation, because it does not access the process's registers, and thus the monitor can call it internally. There are a simple generic `statfs`, `statfs64`, `fstatfs`, and `fstatfs64` wrappers to the `statfs_internal` operation to handle their respective system calls.

B.1.2 File Operations

The monitor provides seven operations for operating on files internally. These operations do not need to be executed in the context of a system call, but do need to be executed in the context of a process. Each of these operations has a corresponding system call wrapper described in Section B.2.

```
struct fd_struct * open_internal(struct pcb *pcb,
                                 struct mount *mount,
                                 char * f_path,
                                 int flags,
                                 int mode,
                                 int *rval);
```

The `open_internal` operation is used to create an open `fd_struct` that can be used for the other internal file operations, or as an entry in the monitor's open file table. The `f_path` argument is the file's path relative to the root of this `mount`. The `flags` argument specifies how the file should be opened (e.g., `O_RDWR`), whether it should be created (`O_CREAT`) or truncated (`O_TRUNC`), etc. If the file is created, then its permissions are `mode`. On success an open file structure is returned. On error, `NULL` is returned a negative error number is stored in the location pointed to by `rval`.

```
int close_internal(struct pcb *pcb,
                   struct fd_struct
                   *fd_struct);
```

Closes an `fd_struct` returned by `open_internal`.

```
int stat_internal(struct pcb *pcb,
                  struct mount *mount,
                  char *f_path,
                  struct stat64 *buf);
```

Fills the `stat64` structure `buf` with information about the file `f_path` (specified relative to the root of `mount`). This is used by the `stat` family of system calls, and also by internal monitor components that need to differentiate files and directories.

```
int read_internal(struct pcb *pcb,
                  struct fd_struct *fd_struct,
                  char *buf,
                  loff_t pos);
```

The `read_internal` operation is equivalent to `pread`, but takes internal monitor objects as arguments: it reads data from `fd_struct` into the location pointed to by `buf` at the position `pos`.

```
int write_internal(struct pcb *pcb,
                   struct fd_struct *fd_struct,
                   const char *buf,
                   loff_t pos);
```

The `write_internal` operation is equivalent to `pwrite`, but takes internal monitor objects as arguments: it writes data from the location pointed to by `buf` to `fd_struct` at the position `pos`.

```
loff_t lseek_internal(struct pcb *pcb,
                      struct fd_struct *fd_struct,
                      loff_t pos,
                      int whence);
```

The `lseek_internal` operation updates `fd_struct`'s current file pointer to `pos`. If `whence` is `SEEK_SET`, then `pos` is treated as an absolute position. If `whence` is `SEEK_CUR` or `SEEK_END`, then `pos` is relative to the current position or end of the file, respectively.

```
int readfile(struct pcb *pcb,
             struct mount *mount,
             char *f_path,
             int fd);
```

The `readfile` sequentially reads the entire contents of the file named `f_path` (relative to the root of `mount`) and copies it to the given Unix file descriptor. This is used for creating temporary copies of files during the `exec` system call. Usually, it is implemented using the `generic_readfile` method, which calls the `open_internal`, `read_internal`, and `close_internal` methods.

B.1.3 Directory Name Operations

The pass-through file system layer defines two operations for managing path names:

```
int encodename(struct pcb *pcb,
               struct mount *mount,
               const char *path,
               char **outpath);
```

The encodename name operation takes a mount and a full path relative to the root of the mount as arguments, and transforms the name into a full path name. The full path name is stored in a buffer allocated with malloc. The buffer's address is stored at the location pointed to by outpath.

```
int decodename(struct pcb *pcb,
               struct mount *mount,
               struct fd_struct *fd_struct,
               char *name,
               char **outname);
```

The decodename operation translates the path name component pointed to by name inside of the open directory fd_struct into a lower-level path name component. The lower-level path name component is stored in a buffer allocated with malloc. The buffer's address is stored at the location pointed to by outname.

B.1.4 Notification Operations

The monitor is notified of various events of interest to file systems by ptrace. Many events are passed to all file systems using a loop of the form:

```
static int do_pre_fork(struct mount *mount, void *pcb) {
  if (!mount->ops->on_pre_fork)
     return 0;
  return mount->ops->on_pre_fork(mount, (struct pcb *)pcb);
}

static int setup_clone(struct pcb *pcb) {
  int ret;

  if ((ret = for_each_mount(0, do_pre_fork, pcb))) {
     setup_return(pcb, ret);
     return 0;
  }
  /* The rest of the clone handling code follows. */
}
```

The monitor passes these events down to the file systems using seven notification operations:

```
int on_exec(struct mount *mount,
            struct pcb *pcb);
```

The on_exec event is triggered when the current process's image has been replaced using the exec system call. The new process image is not necessarily located on the file system pointed to by mount. This can be used to invalidate state that is specific to a process's address space.

```
int on_free_pcb(struct mount *mount,
                struct pcb *pcb);
```

The on_free_pcb event is triggered during the cleanup of pcb's resources. At the time on_free_pcb is called, the pcb structure no longer has any open files or mapped regions. This callback is used by the individual file systems to free their PCB-specific data (e.g., Amino aborts any active transactions).

```
int on_new_pcb(struct mount *mount,
               struct pcb *pcb);
```

The on_new_pcb event is triggered after a PCB is created. File systems can insert private data into the PCB at this point.

```
int on_pre_fork(struct mount *mount,
                struct pcb *pcb);
```

The on_pre_fork event is triggered before the clone system call (or equivalent) is executed. If the notification function returns a non-zero value, then the fork is canceled and the returned value is passed to the user-level application.

```
int on_post_fork(struct mount *mount,
                 struct pcb *child,
                 struct pcb *parent);
```

The on_post_fork event is triggered in the parent process after the clone system call (or equivalent) is executed.

```
int on_post_fork_child(struct mount *mount,
                       struct pcb *child);
```

The on_post_fork_child event is triggered in the child process after the clone system call (or equivalent) is executed.

```
int on_startup(struct mount *mount);
```

The on_startup event is triggered after the monitor has read its configuration file, but the traced program is executed. This can be used for initialization such as starting a BDB checkpointing thread.

B.2 System Call VFS Operations

The monitor provides support for 51 system calls. Each system call VFS operation is named after the corresponding system call and takes a PCB as an argument, and the operation extracts its arguments from the process's registers. The return value of each operation is passed to the user-level process. If a file system does not implement an operation it can use the generic_nosys, generic_opnotsupp, generic_eperm, or generic_ignore functions, which return −ENOSYS, −EOPNOTSUPP, −EPERM, and 0, respectively We do not discuss the arguments for individual calls, as they are identical to the standard Linux kernel application binary interface (ABI).

B.2.1 Operations with Generic Implementations

Assuming that the file system defines the seven internal operations described in Section B.1.2, the monitor provides generic routines for 24 operations:

`int` **`close`**`( struct pcb *pcb);`

The `close` operation implements the close system call. If `close_internal` is defined, then the `generic_close` function may be used.

`int` **`exec`**`( struct pcb *pcb);`

The `generic_exec` operation can be used to execute binaries if the `readfile` operation is defined. The `generic_exec` function creates a temporary file, rewrites the `exec` system call's registers, and unlinks the temporary file.

`int` **`lseek`**`( struct pcb *pcb);`

If the `llseek` method is defined, then the `lseek` method can be implemented using `generic_lseek`.

`int` **`llseek`**`( struct pcb *pcb);`

If the `lseek_internal` method is defined, then the `generic_llseek` can be used.

`int` **`mmap2`**`( struct pcb *pcb);`
`int` **`old_mmap`**`( struct pcb *pcb);`

The `generic_old_mmap` and `generic_mmap2` operations are used to establish mappings. The file system must implement `stat_internal`, `read_internal`, and `write_internal`.

`int` **`open`**`( struct pcb *pcb);`
`int` **`complete_open`**`( struct pcb *pcb);`

The `open` operation is called at the entry to the open system call. If `generic_open` is used, the file system must implement the `open_internal` method.

The `complete_open` operation is called at the exit of the open system call, after the kernel assigns a file descriptor. If `generic_open` is used for the `open` operation, then `generic_complete_open` should be specified.

`int` **`read`**`( struct pcb *pcb);`
`int` **`readv`**`( struct pcb *pcb);`
`int` **`pread64`**`( struct pcb *pcb);`

If the `read_internal` method is defined, then the `read`, `readv`, and `pread64` operations can be implemented in terms of `generic_read`, `generic_readv`, and `generic_pread64`. Note that it is not necessary to use all three generic methods. For example, Amino does not use `generic_readv`, because it must surround each all of the individual read operations in a single transaction.

`int` **`stat`**`( struct pcb *pcb);`
`int` **`fstat`**`( struct pcb *pcb);`
`int` **`stat64`**`( struct pcb *pcb);`

```
int fstat64(struct pcb *pcb);
```

If the `stat_internal` method is called, then the `stat`, `fstat`, `stat64`, and `fstat64` methods can be implemented using the `generic_stat`, `generic_fstat`, `generic_stat64`, and `generic_fstat64` methods, respectively.

```
int truncate64(struct pcb *pcb);
int ftruncate64(struct pcb *pcb);
```

The `truncate64` and `ftruncate64` operations can be implemented in terms of `generic_truncate64` and `generic_ftruncate64`, if the `truncate` and `ftruncate` operations are defined. Neither of these methods supports large files.

```
int statfs(struct pcb *pcb);
int fstatfs(struct pcb *pcb);
int statfs64(struct pcb *pcb);
int fstatfs64(struct pcb *pcb);
```

If the `statfs_internal` method is defined, then the `statfs`, `fstatfs` `statfs64`, and `statfs64` operations can be implemented in terms of `generic_statfs`, `generic_fstatfs`, `generic_statfs64`, and `generic_fstatfs64`, respectively.

```
int write(struct pcb *pcb);
int writev(struct pcb *pcb);
int pwrite64(struct pcb *pcb);
```

If the `write_internal` method is defined, then the `write`, `writev`, and `pwrite64` operations can be implemented in terms of `generic_write`, `generic_writev`, and `generic_pwrite64`. Note that it is not necessary to use all three generic methods. For example, Amino does not use `generic_writev`, because it must surround each all of the individual write operations in a single transaction.

B.2.2 Operations on File Descriptors

The monitor supports 12 operations on file descriptors that the file system may implement (or alternatively use one of the generic error functions). We do not describe these operations, as they use the standard Linux ABI.

```
int fchmod(struct pcb *pcb);
int fchown(struct pcb *pcb);
int fcntl64(struct pcb *pcb);
int flock(struct pcb *pcb);
int fsync(struct pcb *pcb);
int ftruncate(struct pcb *pcb);
int getdents(struct pcb *pcb);
int getdents64(struct pcb *pcb);
int ioctl(struct pcb *pcb);
```

B.2.3 Operations on Path Names

The monitor supports 11 operations on path names that the file system may implement (or alternatively use one of the generic error functions). We do not describe these operations, as they use the standard Linux ABI.

```
int access(struct pcb *pcb);
int chmod(struct pcb *pcb);
int chown(struct pcb *pcb);
int link(struct pcb *pcb);
int mkdir(struct pcb *pcb);
int rename(struct pcb *pcb);
int rmdir(struct pcb *pcb);
int truncate(struct pcb *pcb);
int unlink(struct pcb *pcb);
int utime(struct pcb *pcb);
int utimes(struct pcb *pcb);
```

B.2.4 Unimplemented System Calls

Our monitor has support for six system calls, which none of our example file systems currently implement. However, new file systems can implement these operations without any changes to the monitor.

```
int getxattr(struct pcb *pcb);
int listxattr(struct pcb *pcb);
int readlink(struct pcb *pcb);
int removexattr(struct pcb *pcb);
int setxattr(struct pcb *pcb);
int symlink(struct pcb *pcb);
```

B.2.5 New System Call

Our monitor implements the system call `amino` to manipulate transactions. Only the Amino file system implements this call.

```
int amino(struct pcb *pcb);
```

The amino operation is not part of the existing Linux ABI. The ABI of this call is similar to the description of the amino library wrapper provided in Figure 2.2. The only difference is that the pathname argument for BEGIN_TXN is stored in the ebx register. This makes the amino call more similar to other system calls that take a pathname as an argument. If the call does not take a pathname, then ebx must be NULL. If ebx is NULL, then the operation is passed to all Amino mounts, but only the one that owns the process's current transaction performs the action. The transactional operation is stored in the ecx register, and depending on the call the flags or id parameter is stored in edx. The system call number for Amino is 290, which is stored in the eax register.