Provenance-Aware Storage Systems

Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, Margo Seltzer
Harvard University

Abstract

A Provenance-Aware Storage System (PASS) is a storage system that automatically collects and maintains *provenance* or *lineage*, the complete history or ancestry of an item. We discuss the advantages of treating provenance as meta-data collected and maintained by the storage system, rather than as manual annotations stored in a separately administered database. We describe a PASS implementation, discussing the challenges it presents, performance cost it incurs, and the new functionality it enables. We show that with reasonable overhead, we can provide useful functionality not available in today's file systems or provenance management systems.

1 Introduction

Provenance is traditionally the ownership history of an object. In digital systems, ownership history includes a description of how the object was derived [4]. In software development, build rules express provenance and source code control systems track it. Archivists maintain provenance meta-data to support document viability, renderability, understandability, authenticity, and identity in preservation contexts [21]. Scientific reproducibility requires provenance to identify precisely input data sets, experimental procedures, and parameterization. The business community uses "lineage" to refer to the history of a document. In all of these domains provenance increases an object's value.

Digital provenance is typically stored in standalone database systems, maintained in parallel with the data to which it refers [2, 6, 25]. Separating provenance from its data introduces problems such as: ensuring consistency between the provenance and the data, enforcing provenance maintenance, and preserving provenance during backup, restoration, copies, etc.

Provenance should be maintained by the storage system, since provenance is merely meta-data and storage systems manage meta-data. Managing provenance in the storage system provides many advantages:

- The storage system can generate system-level provenance automatically, freeing users from manually tracking provenance and tool designers from constructing provenance-aware applications.
- Provenance collection and management are transparent. Users take no special actions, so provenance collection is the norm rather than the exception.
- The tight coupling between data and provenance provided by the storage system avoids provenance loss both during normal activity and during management procedures such as backup, restoration, or data migration.
- System-level provenance produces a level of meta-data completeness difficult or impossible to achieve with application-level solutions.

A *provenance-aware storage system* (PASS) is a storage system that automatically collects, stores, manages, and provides search for provenance. Provenance-aware storage offers functionality unavailable in conventional systems. For example, comparing the provenance of two pieces of derived data, such as simulation results, can reveal changes between two program invocations. We can use provenance to identify the particular workflow that produced a document; this provides a tight coupling between workflow management and information life-cycle management (ILM). Searchable provenance enables queries like "Who is using my dataset?" or "On whose data am I depending when I run this experiment?" System-level provenance enables identification of configuration changes that affect applications.

We present the design and evaluation of a prototype PASS. A PASS is both a provenance solution and substrate upon which we can support other provenance systems. Domains characterized by command-line invoca-

tion of data transformations are well served by PASS and may not require any domain-specific solution. Domains that require GUI-driven applications or application environments still derive benefit from integration with provenance-aware storage. This paper contributes: An approach to automatic provenance collection and maintenance; a prototype that demonstrates the efficacy and practicality of the PASS concept; an evaluation of PASS functionality and overhead; and a research agenda for future work on PASS.

The rest of this paper is organized as follows. In Section 2, we present several novel use cases that PASS enables. In Section 3, we compare PASS to existing provenance systems. In Section 4, we outline the requirements for PASS and explain how PASS interacts with existing provenance systems and file system utilities. In Section 5, we describe our PASS implementation. In Section 6, we quantify the overheads PASS introduces. We conclude in Section 7 with a discussion of open research issues and the long-term PASS agenda.

2 Novel Functionality

PASS provides novel features not available in other systems. In this section, we present use cases and examples that demonstrate the power of provenance-aware storage.

2.1 Script Generation

Early PASS users were most excited by PASS's potential to create scripts that reproduce a file originally created through trial and error. We find this one of the most compelling features of PASS.

Recently one of the authors was preparing a paper and discovered that a PostScript figure was being generated with the wrong bounding box, producing a tiny figure surrounded by blank space. After several hours of experimental tinkering as the submission deadline approached, he discovered that creating a huge bitmap, cropping it with an image-manipulation tool, and then converting it to a different format for incorporation into the paper solved the problem. Discovering the precise solution was an iterative multistage process. When the figure finally displayed properly, its provenance allowed automating the processing. Currently, we generate scripts to reproduce a particular object; generalizing these scripts so that they transform objects of one type into objects of another type is a future project.

2.2 Detecting System Changes

Because we collect provenance at the system level, we know the operating system, library versions, and the environment present when an object was created. For example, the output of the `sort` utility depends on the value of the environment variable `LC_COLLATE`. Examining the provenance of files produced by `sort` explains why two seemingly identical invocations of the same command on the same input produce different results. Because we record the process' environment in the provenance, it becomes easy to identify what differs between the two invocations. Tracking changes in libraries and tools is similarly straightforward. The provenance that PASS collects reveals changes in the environment, libraries, operating system, or tools.

2.3 Intrusion Detection

Since a PASS collects provenance inside the operating system and file system, it provides a detailed record of how objects change. This feature can be used in some cases both for intrusion detection and subsequent forensic analysis.

One of the authors uses a UNIX version of the Windows trash bin, aliasing `rm` (remove) to

```
mv !* ~/etc/garbage
```

which moves the target files to a garbage directory. Quite unexpectedly, several recently deleted files appeared in `~/public_html`. This happened on a non-PASS system, so it took nearly a half hour to discover that `~/etc/garbage` had been symbolically linked to `~/public_html`. Had this occurred on a PASS, the provenance of the files in `~/public_html` would have revealed that they had been explicitly moved from their original locations. Had the shell provided application-specific provenance, the `rm` alias would have been readily apparent.

2.4 Retrieving Compile-Time Flags

A common approach to understanding software performance is to recompile with different functionality selected via macros defined on the compiler command line. However, in order to interpret the results, one must keep track of which macros were defined for each program run. Often, users neglect to record this information; then, once the details have been forgotten, it becomes necessary to redo the work. PASS automatically records command-line arguments, as well as the relationship between the various versions of the program and the performance results. It is thus possible to recover the macro arguments and functionality choices.

2.5 Build Debugging

Large build environments are often missing elements of their dependency lists. These become apparent when a

build repeatedly produces a flawed executable. The traditional solution (suggested explicitly for the Linux kernel) is to `make clean` after any change.

PASS easily identifies an inconsistent build: in nearly all build environments, it is incorrect for two different versions of the same source or intermediate file to simultaneously be ancestors of the same output. Examining the complete ancestry of an obsolete file reveals immediately the intermediate file that should have been rebuilt, identifying the missing dependency.

2.6 Understanding System Dependencies

Early experimentation with PASS presented some surprising results. We found that some objects we created unexpectedly included `/bin/mount` in their provenance. This was baffling, since we were executing relatively simple commands such as `sort a > b`.

We discovered that the Linux C library frequently reads the mount table, `/etc/mtab`, to determine where `procfs` is mounted before using it. `mount` and `umount` update `/etc/mtab`, so these programs quite correctly appeared in our sorted file's ancestry. These files also appear in the provenance of *any* process that reads the mount table to find `procfs`.

This behavior is either good or bad, depending on perspective. For a system designer, such information is useful; to a user, it is distracting. To preserve the information for some without burdening others, we provide a provenance truncation utility, `ptrunc`, that removes `/etc/mtab`'s (or any file's) ancestry.

3 Provenance Solutions

Provenance is pervasive in scientific computing, business, and archival. Simmhan et al. categorize provenance solutions in terms of their architecture: database-oriented, service-oriented, and "other" [29]. We borrow and enrich this taxonomy. We extend database-oriented approaches to include file and file-system-oriented approaches such as PASS. The service-oriented architecture encompasses myriad grid-based solutions. Simmhan's "other" category refers to scripting architectures; we treat software development tools, i.e., source code control and build systems, as a specific instance of a scripting architecture. To these three categories, we add "environment architectures" where users perform all tasks in a unified environment that tracks provenance.

3.1 File, File System and Database Approaches

One obvious approach to provenance maintenance is to include provenance inside the corresponding data file. Astronomy's Flexible Image Transport (FITS) format [18] and the Spatial Data Transfer Standard (SDTS) [23] are examples of this approach. A FITS file header consists of a collection of tagged attribute/value pairs, some of which are provenance. Whenever a file is transformed, additional provenance is added to this header. This approach addresses the challenge of making the provenance and data inseparable, but it introduces other disadvantages. It is expensive to search the attribute space to find objects meeting some criteria. Tools that operate on such files must read and write the headers and be provenance-aware. The validity and completeness of the provenance is entirely dependent upon the tools that process the data. Worse yet, there is no way to determine if provenance is complete or accurate.

The Lineage File System (LinFS) [15] is most similar to PASS. LinFS is a file system that automatically tracks provenance at the file system level, focusing on executables, command lines and input files as the only source of provenance, ignoring the hardware and software environment in which such processes run. As shown in Section 2, complete system-level provenance provides functionality unavailable in other systems. A second, and perhaps more important, difference is that LinFS delays provenance collection, performing it at user-level by writing it to an external database. In contrast, PASS manages its provenance database directly in the kernel, providing greater synchronicity between data and provenance; PASTA, our provenance-aware file system, manages both the data and provenance producing a tighter coupling than provided by a separate user-level database.

Trio [27] is to databases what a PASS is to file systems. Trio is a database system that incorporates uncertainty, managing both data and its provenance. It extends SQL to support lineage and accuracy information when requested by a user or application. Trio and PASS are complimentary. Trio focuses on the formalism to describe uncertainty via lineage and operates on tuples within a database framework; PASS focuses on a less structured environment and operates on files.

3.2 Service-oriented Architectures

Many of the computational sciences use provenance systems designed for grid environments since provenance facilitates scientific verification, reproducibility, and collaboration. Most of these systems use a directed-acyclic-graph (DAG) representation to describe workflows. The tools that understand these workflows collect provenance and transmit it to a grid provenance service. For example, Globus [7] is used widely by high-energy physicists and includes the Metadata Catalog Service (MCS) [6] that stores metadata for logical data objects. MCS includes a set of common attributes and permits inclusion

of domain- or application-specific attributes. San Diego SuperComputer's Storage Request Broker [2, 26] has a Metadata Catalog similar to the MCS.

Chimera [8] is a virtual data system providing a virtual data language (VDL) and a virtual data catalog (VDC). The VDC implements a virtual data schema defining the objects and relations used to capture descriptions of program invocations and to record these invocations. The VDL is used for defining and manipulating data derivation procedures stored in the VDC. Chimera can be used to generate Grid workflows from the derivations.

These systems place the responsibility for provenance maintenance with the grid tools, storing provenance in a system parallel to the data storage. They create and maintain provenance only for data that is processed by provenance-aware tools, so there is no mechanism to capture provenance for local experimentation or operations issued outside the bounds of these tools. PASS provides capabilities not found in these systems and is complimentary to their approaches. A grid-accessible PASS provides the advantages of both worlds.

3.3 Scripting Architectures

Software developers manage provenance manually using source code control and build systems. Though powerful, these systems rely more on manual intervention than PASS does. Build systems maintain dependencies *after* those dependencies have been specified; PASS derives dependencies based upon program execution. Source code control systems track differences between manually-declared versions, but a manually-entered commit message is typically the only conceptual expression of the transformation between those two versions. Thus, the quality of provenance in these systems depends on the quality of commit messages and build configuration files. For example, makefiles that track include dependencies properly are considerably more useful than those that do not.

The source code control systems most similar to PASS are ClearCase (and its predecessor DSEE) and Vesta. ClearCase [5] is a source code control system, and like PASS, it is based on a custom file system. The file system serves as the source code repository, and the build system relies on the standard `make` utility. The custom file system tracks and maintains system dependencies to avoid work in future builds and to trigger rebuilds. PASS also captures these dependencies. As is the case with all build systems of which we are aware, ClearCase requires that critical dependencies be specified *a priori*; PASS derives dependencies by observation.

Vesta [12] is a second generation build system developed at DEC Systems Research Center (SRC). The key design goals were making builds repeatable, consistent, and incremental. As with DSEE, Vesta relies on a custom build environment that monitors the build process to extract dependencies and record complete environment information to facilitate repeatable builds. Like DSEE and other source code control systems, it relies on an *a priori* description of the derivation process. As a result, while extraordinarily useful for software development, it ignores the central PASS challenge: automatically generating the derivation rules as a system runs.

3.4 Environment Architectures

Other domains have environments that track work and record provenance as Vesta does for software development. GenePattern [10] is an environment for computational biologists, the Collaboratory for Multi-scale Chemical Sciences (CMCS) [20] is an environment for chemists, and the Earth System Science Workbench (ESSW) [9] is an environment for earth scientists. As long as a user modifies data exclusively in one of these environments, the environment can effectively track provenance. However, operating on data outside the environment or moving data between two different environments breaks the provenance chain. Traditional file system utilities, such as backup and restore, or regular utilities, such as remove and rename, can also break the provenance chain. The semantic information these environments provide is powerful; we propose PASS as a substrate under such environments. This hybrid architecture avoids disruptions in provenance and adds the ability to augment an environment's provenance with provenance about the operating system, libraries, and other system-level information.

3.5 Summary

Provenance-aware storage provides functionality not found in today's provenance solutions while compatible with most of them. Ultimately, end-to-end provenance requires multiple approaches working in concert. As shown by our evaluation, using PASS as a substrate for the end-to-end solution provides significant benefits.

4 The PASS Vision

PASS collects and maintains provenance for all the objects stored in it. We define provenance to be a description of the execution history that produced a persistent object (file). We represent provenance as a collection of attribute/value pairs, referenced by a unique identifier called a *pnode number*. Provenance can contain references to objects that are themselves *provenanced*, i.e., have provenance. Therefore, complete provenance is the transitive closure over all such references.

In both service-oriented architectures and PASS, provenance is captured in a DAG. PASS automatically generates the DAG describing the relationship between processes running on it and the files stored in it. Our prototype tracks provenance at a file granularity, but this is not an inherent property of PASS; a system could track provenance at finer or coarser granularities [16] (e.g., bytes, lines, directories or whole volumes).

Data on a PASS is considered to be either new data or the output of some process. The provenance of a process' output must include:

- A unique reference to the particular instance of the executable that created it.
- References to all input files.
- A complete description of the hardware platform on which the output was produced.
- A complete description of the operating system and system libraries that produced the output.
- The command line.
- The process environment.
- Parameters to the process (frequently encapsulated in the command line or input files)
- Other data (e.g., the random number generator seed) necessary to make pseudo-random computation repeatable.

Collecting all the information listed above poses challenges, but is possible. However, it is not possible to automatically collect provenance that the system never sees; we call such provenance *opaque provenance*. Opaque provenance arises when data originates from a non-PASS source, such as a user, another computer, or another file system that is not provenance-aware.

There are three approaches to opaque data. First, a PASS records any provenance it can deduce about an input, such as its creator, create time, its source (expressed as the global name for a networked object or the full path for a local, non-PASS object) and a unique fingerprint of the source (e.g., a hash value). Second, a PASS permits users to add annotations to files. Annotations provide either provenance for opaque data or semantic information, invisible to the system. Third, a PASS allows storage and retrieval of application-specific provenance. Application-specific provenance records are like annotations, but are provided programmatically from, for example, a driver for a remote data-producing device (e.g., a driver taking data from a telescope) or an application environment such as GenePattern [10]. A PASS distinguishes between internally-collected provenance, annotations, and application-generated provenance so queries can specify which attribute types to consider.

Field	Value
`FILE`	/b
`ARGV`	sort a
`NAME`	/bin/sort,/bin/cat
`INPUT`	*(pnode number of a)*
`OPENNAME`	/lib/i686/libc.so.6, /usr/share/locale/locale.alias, /etc/mtab,/proc/meminfo .#prov.mtab
`ENV`	PWD=/pass USER=root ...
`KERNEL`	Linux 2.4.29+autoprov #17 ...
`MODULE`	pasta,kbdb,autofs4,3c59x,...

Table 1: Provenance of `sort a > b`. The `FILE` record is relative to the root of the PASS volume.

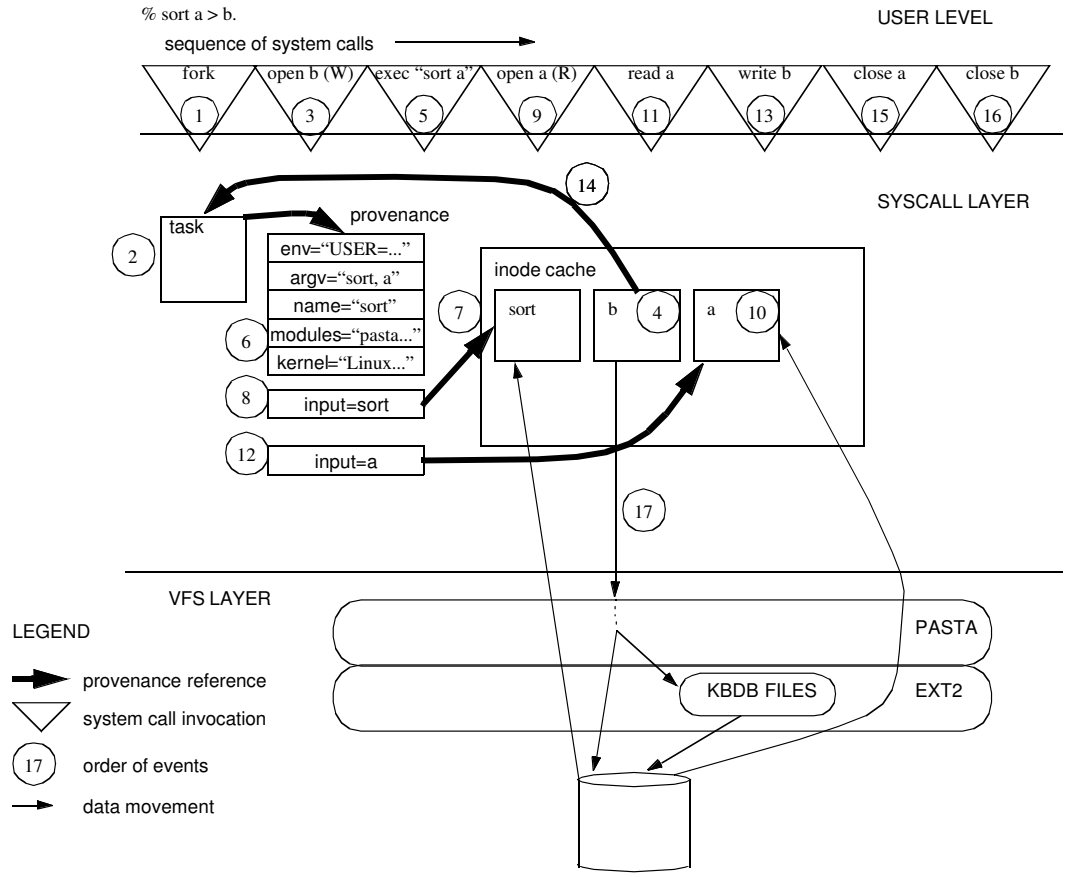
4.1 Requirements

PASS must automatically collect provenance and also provide the functionality outlined in this section.

PASS must support **application-generated provenance**, so that PASS can be used as a substrate for domain-specific provenance solutions. Applications that currently write to domain-specific provenance systems can instead write into a PASS's provenance database, producing an integrated view of application and system provenance. Our implementation uses a simple, low-level data representation that is easily mapped to XML, a relational schema, or any other data format used by an existing provenance solution.

PASS must provide **security for provenance**. Provenance requires access controls different from the data it describes. Consider an employee performance review that includes input from others: while the review itself must be readable by the employee, the provenance must not [15]. We conducted user studies to gather requirements for a provenance security model and determined that the security model for provenance can be divided into two pieces: one that protects access to ancestors and descendants and one that protects access to attributes. We are designing a security model for provenance, but its details are beyond the scope of this paper [3].

Finally, PASS must provide support for **queries on provenance**. Collecting provenance on data is not useful unless that provenance is easily accessed. Provenance queries fall into two categories: conventional attribute lookup and computation of transitive closures of ancestry (or descendancy) information. The latter are particularly challenging for most data management systems.



1. Create a new process via `fork`.
2. Materialize a task_struct structure for the new process.
3. Open output file `b` as stdout.
4. Place `b`'s inode into the inode cache.
5. Exec `sort`
6. Create provenance records for the process.
7. Load the executable; place its inode in the inode cache.
8. Associate the executable's inode to the running process.
9. Open input file `a`.
10. Read `a`'s inode into the inode cache.
11. Read from `a`.
12. Add file `a's` provenance to the current process.
13. Write output to `b`.
14. Transfer process' provenance to `b`.
15. Close `a`.
16. Close `b`.
17. Write provenance to database.

Figure 1: Provenance Collection Example: We show the relevant system calls and provenance collection and maintenance steps for a simple command.

5 Implementation

We implemented PASS in Linux 2.4.29. To provide a framework for the implementation discussion, we begin with an example describing how PASS collects and maintains provenance. We then present an overview of provenance collection followed by the details of its implementation. We conclude this section by evaluating our prototype relative to the requirements presented in the previous section.

Throughout the rest of this section, we use the example command line "`sort a > b`" to describe how PASS collects and stores provenance. Table 1 shows the records that are added to the database, and Figure 1 shows the sequence of important system calls and the provenance collection steps performed while the command executes.

With this structure in mind, we present the details of our implementation in four parts. First, we present an overview of provenance collection. Second, we discuss the collector, which implements the activity shown in the the syscall layer in the figure. Third, we discuss our provenance-aware file system, which resides below the VFS layer. Last, we present our query system, which is not shown in the figure.

5.1 Overview

The system maintains provenance information both in memory and on disk, but the two representations do not map one-to-one. On disk, file ancestry can be expressed as cross-references to other files. In memory, however, PASS must account for processes and for other objects such as pipes or sockets that play important roles in provenance collection and derivation, but are not materialized in the file system.

Processes are provenanced objects, because processes produce files that must incorporate their creating process' provenance. For example, in Figure 1, the process' provenance is the collection of records that are attached to the task structure. Since our implementation does not track explicit data flow within a process, *all* data a process accesses can potentially affect the process' outputs and must be included in its provenance.

In memory, the system maintains an ancestry graph in which files reference processes, and processes reference PASS files, pipes, non-PASS files, and (transitively) even other processes. Many of these objects are never materialized on disk, and some that are written to disk cannot be readily cross-referenced. When provenance is written to disk, in-memory references to persistent objects are recorded in the database as references while the provenance of a non-persistent referenced object (e.g., a pipe) is copied directly into the referring object's provenance. Therefore, each object on disk may correspond to several in-memory objects, and in-memory non-persistent objects may have their provenance recorded multiple times on disk if they are ancestors of multiple files.

The system collects provenance for every process, because it cannot know in advance which processes might ultimately write to a PASS volume. It also collects provenance for non-PASS files, which is retained as long as their inodes are kept in memory.

5.2 The Collector

One of the key challenges in PASS is translating a sequence of low-level events into a sequence of meaningful provenance entries; the collector performs this translation. It intercepts system calls, translating them into in-memory provenance records, which are then attached to key kernel data structures. Figure 1 shows how the collector translates some system calls into provenance records. For example, the `exec` system call (step 5) triggers creation of the ENV, ARGV, NAME, KERNEL, and MODULES provenance records in step 6. It also maintains the ancestry graph for in-memory objects. The final job of the collector is to map the in-memory graph to the on-disk provenance that is passed to the storage layer, shown as step 17 in Figure 1. We refer to on-disk provenance of an object via a unique *pnode number*.

In the `sort` example, the collector's work is relatively simple. The collector generates a provenance record for each provenance-related system call and binds the record to the appropriate structure. Table 2 summarizes the system calls that the collector intercepts, the provenance records generated, and the structures to which records are attached.

Duplicate Elimination In the example, `sort` might issue many `read` system calls if its input file is large. The simple-minded collector creates a new INPUT provenance record for each read, even though the subsequent provenance records are exact duplicates of the first. Our collector tests for and eliminates duplicates at two points. It drops a new record when that record exactly duplicates an existing record already attached to the same kernel object. This does not eliminate all duplicates, however. Consider the shell command `(sort a; sort a) > b`, which writes two sorted copies of `a` into `b`. Both sort processes read `a` and have `a` as an ancestor. When the collector traverses the in-memory graph to write provenance for `b`, two references to `a` appear, so the collector also suppresses duplicates during this traversal. This duplicate elimination removes identical ancestors (e.g., the two occurrences of a), but since the two `sort` processes have different process ids, the

System Call	Record Type	Description	Where attached
execve	ENV	environment	current process
	ARGV	command line arguments	current process
	NAME	process name	current process
	PID	process id	current process
	KERNEL	kernel version	current process
	MODULES	kernel modules loaded	current process
	INPUT	reference to program	current process
open	OPENNAME	pathname of file (non-PASS files only)	target file's inode
read	INPUT	reference to file	current process
write	INPUT	reference to current process	target file's inode

Table 2: Basic provenance collection by system call.

provenance still reveals that there were two invocations of `sort`.

Versions Suppose that we executed `sort a > b` but `b` already existed. The system would truncate `b` and write new data to it, logically creating a new version of `b`. The system needs to ensure that there is a new version for the provenance of `b`, as the new `b` may have no relationship whatsoever to the old. Creating a new provenance version allocates a new pnode number. The collector intercepts the `truncate` operation to make sure this happens at the right time.

Because we are not running on a versioning file system, the data in the old `b` is lost. However, our file system does retain old versions of provenance, because the old `b` could be an ancestor of an important file, and we cannot remove that file's provenance.

In this example, it was easy to determine when to create new versions. However, declaring a new version is more complicated in the general case. Consider a sequence of write system calls. Does each write create a new version? What if a process closes and re-opens the file after each write?

The simple approach of declaring a new version for every write suffers from *provenance explosion*, the creation of too much provenance and it does not suggest when to create new versions for processes. The collector must be able to group multiple writes together into a single version. In the simple case, it is sufficient to declare a new version on a file's last `close` and on any `sync`. The next section discusses the solution for more complicated cases.

Cycles Ideally, only meaningful versions should be given new pnode numbers. Versions generated by the system as a result of the way the application is written or as a result of properties of the collector implementation should be minimized. However, as the workload increases in complexity, reducing the number of versions introduces an even more daunting challenge: cycles. Provenance cycles are problematic, because they represent violations of causality (i.e., an object depending on the existence of its children). Consider the program that does:

```
read a
write a
```

This program reads `a`, becoming a descendant of `a`. Then it writes `a`, making itself also an ancestor of `a`. This produces a cycle unless the write to `a` creates a new version that is not an ancestor of the process.

This particular case is easily suppressed; however, the problem is considerably broader than this. Consider two processes `P` and `Q`:

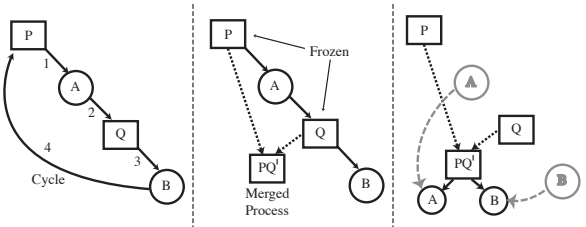
```
  P              Q
read a
                read b
write b
                write a
close a         close a
close b         close b
```

If no new versions are created until close, `Q`'s write of `a` creates a cycle; `Q` is an ancestor of `a`, which is an ancestor of `P`, which is an ancestor of `b`, which is an ancestor of `Q`. More complex examples include dozens of processes and intervening files.

Since we want to avoid creating a new version on every write, there are at least three possible approaches:

1. Ignore it. Let cycles appear in the database and make the query tools deal with them. We consider this approach unacceptable.

2. Declare a new version only on writes that add new provenance information. This approach generates a



1. Cycle Detected | 2. Freeze and Merge Processes | 3. Re-parent Files

Figure 2: Cycle breaking by node merging.

large number of versions, and in our current system, versions are too expensive to allow this, but we are pursuing more efficient version management in our next design that might allow this approach.

3. Detect and break cycles. This is our current solution.

Before adding an in-memory provenance record, the collector checks the in-memory ancestry graph for cycles. If the new record would create a cycle, the collector invokes a cycle-breaking algorithm.

The simplest way to break a cycle is to start a new version for the object that would create the cycle. (If a process were about to read from a file that was its own descendant, we would create a new version for the process; if a process were about to write a file that was its own ancestor, we would create a new version for the file.) This solution has two problems. First, the target might be a file, and creating extraneous versions of files is undesirable. Second, when data flows in a loop, it is often because of a circular operation (e.g., recompiling your compiler) and it is likely to repeat more than once. Creating a new version on every iteration causes a form of provenance explosion that we refer to as *version pumping*.

We developed *node-merging* to avoid these problems. Node-merging treats a set of processes that cycle data among themselves as a single entity for provenance purposes. The provenance of this shared entity is the union of the provenance of the processes that form it. Figure 2 illustrates this approach.

Specifically, if the cycle contains processes P_i and files F_j, the collector starts a new version for each P_i, and merges all those new versions into a single "process" P_{new}. The files F_j are then made descendants of P_{new}.

Implementation Details The implementation of the collector is approximately 4850 lines of code, plus approximately 250 lines of code to intercept system calls.

Each provenanced object within the kernel points to a "virtual pnode" structure (`vpnode`). This structure holds the in-memory provenance records and the graph edges used for cycle detection. Both persistent and non-persistent objects (such as pipes) use the same data structures.

Capturing provenance for data flowing through pipes requires no special effort: a pipe has an inode structure in the kernel just like a file, and this has a vpnode attached to it. Writing to a pipe creates an ancestry reference from the pipe to the source process; reading the data out again creates an ancestry reference from the sink process back to the pipe. Since the pipe is non-persistent, these references are traversed when provenance is written. Entire pipelines can be captured this way.

The collector communicates with the storage layer via five new VFS calls:

1. `getpnid` - returns a file's on-disk pnode number
2. `thaw` - begin a new version of a file
3. `freeze` - end a file version
4. `write_prov_string` - transforms a non-reference in-memory provenance record into a persistent provenance record
5. `write_prov_xref` - transform an in-memory file reference into a persistent provenance record

5.3 The Storage Layer

The storage layer is composed of a stackable file system, called PASTA, that uses an in-kernel database engine to store its meta-data. Pasta uses the FiST [30] toolkit for Linux to layer itself on top of any conventional file system. We use `ext2fs` as our underlying file system.

We use an in-kernel port of the Berkeley DB embedded database library [19], called KBDB [14], to store and index provenance. Berkeley DB provides fast, indexed lookup and storage for key/value pairs. Berkeley DB does not provide a schema in the traditional sense. Instead, applications define their schema by creating indexed databases (tables in relational parlance), secondary indexes, and relationships between the tables. An entry in a Berkeley DB database consists of an indexed key and an opaque data item. The application determines how data items are interpreted.

We store provenance in five Berkeley DB databases, summarized in Table 3. The PROVENANCE database is the primary repository of provenance records. The record types stored in the PROVENANCE database are shown in Table 2. The MAP database maps inode numbers to pnode numbers. A file's pnode number changes

Database	Key	Values
MAP	inode number	pnode number
PROVENANCE	pnode number, record type	provenance data
ARGDATA	sequence number	command line text
ARGREVERSE	command line text	sequence number
ARGINDEX	words from `argv`	pnode number

Table 3: Provenance databases.

Query	Functionality	Use cases
MAKEFILE GENERATION	show script to recreate a file	2.1, 2.4, 2.5
DUMP ALL	show all provenance records for a file	2.2, 2.6, 2.3
COMMAND-LINE LOOKUP	find files modified by a process given matching command line arguments	2.4
LIST ANNOTATIONS	show all user-defined annotations to a file	-

Table 4: Query tool summary. The last column maps the query to the example use cases mentioned in Section 2.

each time it is *thawed*. Command lines and environments are large relative to other provenance and are often repeated. The ARGDATA database implements an optimization, assigning (via Berkeley DB) a sequence number to each unique command line and environment. The ARGREVERSE database maps instances of an environment or command line to the sequence number assigned by ARGDATA. The ARGINDEX database is another secondary index, mapping components of command lines and environments (e.g., individual environment variables and their values) to pnode numbers.

User annotations to a file are stored in the PROVENANCE database with record type ANNOTATION. We provide an annotation `ioctl` that is used to record annotations to a file. This `ioctl` takes the annotation name and a string value as input and records them in the database. Annotations do not pass through the collector and are stored directly into the database.

5.4 The Query Tools

We make the provenance database accessible to users as a standard set of Berkeley DB databases. Making the provenance accessible as standard Berkeley DB files provides many advantages. We can use a variety of programming languages for building query tools (e.g., Python, Perl, Tcl, Java, C), and the standard Berkeley DB utilities (e.g., dump, load) also work.

As we discuss in greater detail in Section 7.1, our prototype does not meet the security requirements identified earlier, but it does allow us to gain experience with PASS, deploy it in constrained situations, and extract as much information as possible to drive development of our later prototypes.

We built an easy-to-use query tool atop the Berkeley DB databases. The query tool has a built-in file system browser, the Provenance Explorer. Users navigate to the file of interest and run queries using the Provenance Explorer. The MAKEFILE GENERATION query generates a set of commands corresponding to the sequence of events that led to the file's current state. Users can set various filters to, for example, remove commands that occurred before or after a given point in time. Another query, DUMP ALL, retrieves the complete provenance for a selected file. The Provenance Explorer also allows users to add or retrieve file annotations. We also support the command-line argument lookup query that allows users to search for files based on arguments to the processes that modified them. For example, a computational physicist can look up files that were modified by a process with an argument, `particle`, set to a particular value. The query capabilities are summarized in Table 4.

The Provenance Explorer is written in Java and uses JNI to access the Berkeley DB databases. To facilitate scripting, we also built command-line utilities that provide functionality equivalent to the Provenance Explorer.

5.5 Prototype versus Requirements

Our prototype has been a source of insight and future challenges despite its limitations. We now analyze these limitations in light of the vision we outlined in Section 4.

The prototype accurately collects and maintains provenance. The provenance it collects for computations that are encapsulated in a process is complete. To the best of our knowledge, this is the first system that captures such information automatically and makes it generally accessible. Unlike many existing systems, it does so without requiring *a priori* specification of the workflow or DAG.

Although we create reasonable provenance for files copied from non-PASS file systems on the local machine, we do not yet create provenance for files that are transmitted across the net. Users and applications can add annotations and application-specific provenance, but for the long-term, we will implement network adapters that observe network activity and create provenanced in-memory objects corresponding to the networked object.

A complete provenance solution requires provenance-aware applications, which are not yet commonly available. However, as we demonstrated in Section 2, the functionality we currently provide does not exist in cur-

rent systems and is complementary to domain-specific solutions. There are existing APIs for provenance-aware applications [11], and the interfaces we have defined for kernel provenance collection offer an alternative. Making an application provenance-aware requires that a programmer identify and then express the dependencies between the objects that an application manipulates. Given the existence of provenance-aware environments[10] and the level of activity in this area, we do not anticipate significant obstacles. Building several provenance-aware applications is part of our next generation design process and will undoubtedly inform the future API.

We have not yet implemented security. This functionality was not important to our first generation users. We wholeheartedly believe that we cannot add security to an already existing system, so we are intentionally designing our next generation PASS from scratch, incorporating what we have learned about provenance collection as well as what we have ascertained about appropriate security models for PASS [3].

The prototype provides simple query capabilities. These capabilities can be extended in a number of ways, but we provide the critical capabilities that we and our users identified. Our current script generation is primitive (it cannot reproduce pipelines without incurring huge overheads), but does meet the needs of our current users.

6 Evaluation

PASS requires in-kernel processing to track provenance, and it produces additional data that is ultimately written to disk. We describe our evaluation platform and then present benchmark results illustrating both the disk-space and time overheads of our prototype. We then present query times for some sample queries run on the output of a Linux build. Finally, we present a sample application and its overhead.

6.1 Evaluation Platform

We evaluated our PASS prototype on a 500Mhz Pentium III machine with 768MB of RAM, running Redhat 7.3. We ran all experiments on a 40GB 7200 RPM Maxtor DiamondMax Plus 8 ATA disk drive. To quantify the overhead of our system, we took measurements on both a PASS and a non-PASS system. We obtain results marked "PASS" by running our modified Linux 2.4.29 kernel (supporting provenance collection) and our provenance-aware file system, PASTA, running on top of ext2fs. We obtain non-PASS results, marked "Ext2", running on an unmodified Linux 2.4.29 kernel and ext2fs, without PASTA.

To ensure a cold cache, we reformatted the file system on which the experiments took place between test runs. Creating the five provenance databases on a newly reformatted file system introduces 160 KB space overhead, which we consider negligible, given today's enormous file systems. We used Auto-pilot [28] to run all our benchmarks. For all experiments, we measured system, user, and elapsed times, as well as the amount of disk space used for provenance. We compute wait time, which usually represents I/O time, as the difference between elapsed time and the sum of system and user times. We ran each experiment 10 times. In nearly all cases, the standard deviations were less than 5%. We do not discuss user time as our code is in the kernel and does not affect the user time.

6.2 Benchmark Performance

We begin with the large and small file microbenchmarks frequently used to evaluate file system performance [24]. These benchmarks exercise the `creat`, `read`, `write`, `fsync`, and `unlink` system calls for a large number of files spread over a directory hierarchy.

Phase	Ext2	PASS	% Overhead
create	10420	12180	16.89
read	10420	12180	16.89
write	10420	13476	29.33
write-sync	10420	13892	33.32
delete	24	3486	14,466

Table 5: Space overhead (in KB) for the small file microbenchmark.

Phase	Ext2	PASS	% Overhead
create	0.91	3.02	232.41
read	0.71	1.51	111.90
write	1.62	4.42	172.09
write-sync	1.67	5.31	217.44
delete	1.13	1.22	7.56

Table 6: Time overhead (in seconds) for the small file microbenchmark.

The small file test uses 2500 4 KB files with 25 files per directory. Table 5 and 6 show that the overheads can be large, but are acceptable for two reasons: First, the absolute numbers are also quite small, so expressing the overhead as a percentage is misleading. Second, this is a particularly challenging benchmark for PASS, because the files are small and the write phases of the benchmark

overwrite existing files. (Only the create phase writes new files.) The overhead during the read phase is due to duplicate elimination done by a PASS. Since the benchmarking process has as many as 2,500 files open, checking for duplicates introduces significant overhead. The overheads during the create, write and write-sync phases are higher than during the read phase, because they generate provenance records in addition to detecting duplicates. The overhead during the delete phase is due to the deletion of entries in the MAP database. The total data size remains unchanged while the provenance grows, suggesting that *provenance pruning*, discussed in Section 7.1, is an important area for further research.

Phase	Ext2	PASS	% Overhead
seq-write	6.16	7.11	15.47
seq-read	0.90	0.92	2.01
rand-write	6.20	7.04	13.58
rand-read	0.90	0.92	2.31
re-read	0.89	0.92	2.44

Table 7: Time overhead (in seconds) for the large file microbenchmark.

The large file benchmark consists of five phases. First, it creates a 100MB file by sequentially writing in units of 256KB; second, it reads the file sequentially; third, it writes 100MB in 256 KB units to random locations in the existing file; fourth, it reads 100MB in 256KB units from random locations in the file; last, it re-reads the file again sequentially. The benchmark has virtually no space overhead, since it uses one file and adds only 10 new records to the provenance databases.

Table 7 shows the time overhead. As a proportion of the running time, these are much better than the small file results, because the "application" is doing more work and the provenance collector proportionally less. The write overheads are due to the stackable file system component of our storage layer. As shown in the next sections the large file results are more indicative of the overheads on realistic workloads.

	Makefile	Dump All
elapsed time	1055.09	91.90
system time	190.98	26.37
user time	817.12	17.57
average elapsed time per file	0.065	0.006
average db lookups per file	485	36

Table 8: Query times, in seconds, for all files present after a Linux build.

For our last benchmark, we built the vanilla Linux 2.4.29 kernel. This build generates 399 MB of data producing 11% space overhead and 10.5% execution time overhead (increasing from 2227 seconds to 2461 seconds). The time overheads are due to the collector and database computational overheads and the I/O time to write the provenance to disk.

We then used the directory tree and files resulting from the Linux compile to measure provenance query times. For each file existing on PASS after the Linux build, we issued two queries: a makefile generation query and a dump_all query. The makefile generation query retrieves records from the PROVENANCE database. The dump_all query retrieves records from the PROVENANCE and ARGDATA databases. The PROVENANCE database had 472,356 records and the ARGDATA database had 5,189 records. The makefile generation query benchmark retrieves 7,911,886 records in all and the maximum ancestor depth is 15. The dump_all query benchmark retrieves 587,494 records in all. Table 8 shows that the queries are fast: even though a makefile query performs nearly 500 lookups per file, it does so in only 65 ms due to the index capabilities of Berkeley DB. However, the writes to these indices contribute to the high write overheads in the small-file benchmark.

Next, we used the same tree to demonstrate how provenance grows as files are modified. We chose 'N' files randomly, appending a comment to each one, and then rebuilt the kernel. Table 9 shows how the provenance grows for different values of 'N'.

N	Size (MB)	% Files changed	% Space growth	% Record growth
0	44	-	-	-
10	44	0.06	0	0.63
110	44	0.67	0	2.31
310	45	1.90	2.27	3.89
1310	49	8.03	11.36	17.18
3310	52	20.28	18.18	34.73
8310	61	50.93	38.64	72.16

Table 9: Provenance Growth benchmark results. where N is the number of files modified and size is the total provenance size after the compile.

The results indicate that provenance growth rate is reasonable considering that changes to the files are small but PASS still has to enter the same number of records into its database irrespective of whether the changes are small or large.

6.3 Application Performance

One of our early users was a computational biologist who regularly uses `blast` [1] to find the protein sequences in one species that are closely related to the protein sequences in another species. Typically, she starts with two files containing the protein sequences of two different species of bacteria. A program, `formatdb`, formats the files and prepares them for `blast`. Then she runs `blast` followed by a series of Perl scripts to produce a list of the proteins in the two species that might be related to each other evolutionarily. When the output is satisfactory, she uses the PASS query tools to generate a script containing the precise sequence of commands that produced the output file.

The time overhead for this real-world workload is minimal, 1.65% (from 309 seconds to 315 seconds), so the new features incur no perceptible cost.

6.4 Evaluation Summary

PASS provides functionality unavailable in other systems with moderate overhead. With the exception of the small file microbenchmark, our time and space overheads are small – typically less than 10% and always under 20%. We and our users are satisfied with these overheads. In addition, we found the system was useful in unanticipated ways such as its ability to generate scripts and its ability to uncover system mysteries.

7 Conclusion

Provenance-aware storage is a new research area that exposes myriad research challenges that we outline in the following section.

7.1 PASS Research Challenges

As we developed our prototype, we encountered problems with our initial design as well as problems that the design did not anticipate. Despite flaws in the prototype, we continued development so we could learn as much as possible before beginning the design of a second generation system. The current prototype has been deployed to a few early adopters, but is not yet in heavy use. We are designing a second prototype based on the experience we have gained. In the following sections, we outline what we have learned and where significant future research challenges remain.

We began the PASS implementation with the simplest and lowest-level schema that could meet our query needs. In parallel with development of the prototype, we undertook an evaluation comparing different provenance storage solutions: our existing one, a relational data model, an XML-based representation, and an LDAP-based representation [13]. Results so far suggest that the Berkeley DB-based implementation provides superior performance, but these results are not yet final [17].

We recognized the security challenges early in the project and that they are a source of myriad research opportunities. We conducted a low fidelity pilot user study to gain insight into this area and identified the two-pronged security model that separately addresses provenance ancestry and attributes.

The cycle-breaking algorithm we described in Section 5.2 is complicated and has been the single greatest source of errors in the system. Cycle-free provenance collection is an explicit goal for our next prototype, but this remains a research challenge.

As currently implemented, the provenance databases are append-only. This is dangerous, wasteful, and not viable in the long-term. Some provenance pruning is easy: deleting a file with no descendants should delete its provenance. However, deleting items with descendants may offer subtle opportunities for pruning, for example, by compressing long chains of pnodes into equivalent single pnodes. Tackling the general provenance pruning problem requires synthesizing research in information flow and garbage collection and applying this work to this new domain.

Provenance-aware storage will be more broadly accepted once we demonstrate integration with existing provenance solutions. Building provenance-aware applications is the first step in this direction. Discussions with users suggests that building a provenance-aware R environment [22] will make the platform attractive to biologists and social scientists. This is next on our agenda, and we hope that others will select their favorite tools to make provenance-aware.

Until all storage systems are provenance-capable, we face interoperability challenges. Moving files through a non-PASS system should not lose provenance, although it may prove difficult. Extended attributes have been in use for many years, yet there is still no safe way to move files and their extended attributes among systems with and without extended attribute support. Ultimately, we must develop provenance-aware network protocols so provenance can be atomically transmitted with data.

Finally, there remain difficult engineering issues. Our current handling of `mmap` is primitive and needs improvement. The `ptrunc` utility mentioned earlier is equally primitive; we prefer, instead, to allow a user or administrator to designate files (e.g., `/etc/mtab`) that should be ignored in provenance.

We built our PASS prototype to facilitate rapid implementation and deployment, but requiring a particular operating system is not a long-term solution. Instead, we need to develop network-attached PASS imple-

mentations complete with client-side plug-ins for NFS and CIFS clients. We will also build a versioning provenance-aware file system; exploring the considerations involved is another open research problem.

7.2 Acknowledgements

We thank our corporate sponsors, Network Appliance and IBM, for supporting this work. We also thank those who reviewed our paper and gave us excellent feedback, particularly our shepherd, Jason Flinn, for repeated careful and thoughtful reviews.

7.3 In Closing ...

We presented provenance management as a task for storage systems and described and evaluated our prototype system that addresses this problem. It provides a substrate offering significant provenance functionality and lends itself to unifying system and application provenance. We described several use cases where system provenance provides new capabilities and demonstrated that we can accomplish it with acceptable overhead.

References

[1] ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., AND LIPMAN, D. J. Basic local alignment search tool. *Molecular Biology 215* (1990), 403–410.

[2] BARU, C., MOORE, R., RAJASEKAR, A., AND WAN, M. SDSC Storage Resource Broker. In *CASCON* (Toronto, Canada, Nov–Dec 1998).

[3] BRAUN, U., AND SHINNAR, A. A Security Model for Provenance. Technical Report TR-04-06, Harvard University, Jan. 2006.

[4] BUNEMAN, P., KHANNA, S., AND TAN, W. Why and Where: A Characterization of Data Provenance. In *International Conference on Database Theory* (London, UK, Jan. 2001).

[5] ClearCase. `http://www.ibm.org/software/awdtools/clearcase`.

[6] DEELMAN, E., SINGH, G., ATKINSON, M. P., CHERVENAK, A., HONG, N. P. C., KESSELMAN, C., PATIL, S., PEARLMAN, L., AND SU, M.-H. Grid-Based Metadata Services. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM04)* (June 2004).

[7] FOSTER, I., AND KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing* (Summer 1997).

[8] FOSTER, I., VOECKLER, J., WILDE, M., AND ZHAO, Y. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR* (Asilomar, CA, Jan. 2003).

[9] FREW, J., AND BOSE, R. Earth system science workbench: A data management infrastructure for earth science products. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management* (2001), IEEE Computer Society, pp. 180–189.

[10] GenePattern. `http://www.broad.mit.edu/cancer/software/genepattern`.

[11] GROTH, P., MOREAU, L., AND LUCK, M. Formalising a protocol for recording provenance in grids. In *Proceedings of the UK OST e-Science Third All Hands Meeting* (Nottingham, UK, Sept. 2004).

[12] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, March 2001.

[13] HODGES, J., AND MORGAN, R. Lightweight Directory Access Protocol (v3). `http://www.rfc-editor.org/rfc/rfc3377.txt`, Sept. 2002.

[14] KASHYAP, A. File System Extensibility and Reliability Using an in-Kernel Database. Master's thesis, Stony Brook University, December 2004. Technical Report FSL-04-06.

[15] Lineage File System. `http://crypto.stanford.edu/~cao/lineage.html`.

[16] MOORE, J. S. The txdt package — interlisp text editing primitives. Manual CSL-81-2, XEROX PARC, Jan. 1981.

[17] MUNISWAMY-REDDY, K.-K. Deciding How to Store Provenance. Technical Report TR-03-06, Harvard University, Jan. 2006.

[18] NOST. Definition of the flexible image transport system (FITS), 1999.

[19] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track* (Monterey, CA, June 1999).

[20] PANCERELLA, C., ET AL. Metadata in the Collaboratory for Multi-scale Chemical Science. In *Dublin Core Conference* (Seattle, WA, 2003).

[21] Data Dictionary for Preservation Metadata. `http://www.oclc.org/research/projects/pmwg/premis-final.pdf`, May 2005.

[22] The R Project for Statistical Computing. `http://www.r-project.org`.

[23] Spatial data transfer standards (SDTS).

[24] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File System Logging versus Clustering: A Performance Evaluation. In *Winter USENIX Technical Conference* (New Orleans, LA, January 1995).

[25] SINGH, G., BHARATHI, S., CHERVENAK, A., DEELMAN, E., KESSELMAN, C., MANOHAR, M., PATIL, S., AND PEARLMAN, L. A Metadata Catalog Service for Data Intensive Applications. In *Proceedings of SC2003 Conference* (November 2003).

[26] FAQ: Frequently Asked Questions on SRB. `http://www.npaci.edu/dice/srb/faq.html`, June 2004.

[27] WIDOM, J. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Conference on Innovative Data Systems Research* (Asilomar, CA, January 2005).

[28] WRIGHT, C. P., JOUKOV, N., KULKARNI, D., MIRETSKIY, Y., AND ZADOK, E. Auto-pilot: A Platform for System Software Benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track* (Anaheim, CA, April 2005), pp. 175–187.

[29] YOGESH SIMMHAN, BETH PLALE, D. G. A Survey of Data Provenance Techniques. Technical Report IUB-CS-TR618, Indiana University, Bloomington, 2005.

[30] ZADOK, E., BĂDULESCU, I., AND SHENDER, A. Extending file systems using stackable templates. In *USENIX Technical Conference* (Monterey, CA, June 1999).