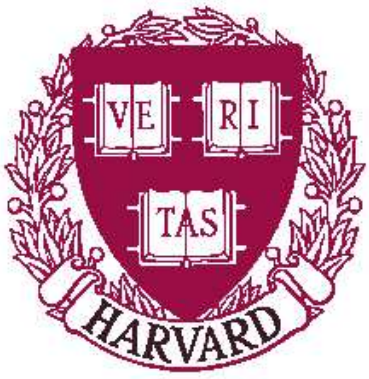
Provenance-Aware Storage Systems

Margo Seltzer
Kiran-Kumar Muniswamy-Reddy
David A. Holland, Uri Braun
and Jonathan Ledlie

TR-18-05



Computer Science Group
Harvard University
Cambridge, Massachusetts

Provenance-Aware Storage Systems

Margo Seltzer, Kiran-Kumar Muniswamy-Reddy
David A. Holland, Uri Braun, Jonathan Ledlie
Harvard University
pass@eecs.harvard.edu

Abstract

Provenance is a type of meta-data that describes the history or ancestry of an object. Although provenance is typically manually generated and stored in a stand-alone database, we make the case that it must be managed by the storage system.

In this paper, we describe *provenance-aware storage systems* (PASS), a new class of storage system that automatically tracks provenance. A PASS takes responsibility for recording provenance meta-data for the objects stored on it and maintaining that provenance over time. We predict that within the next decade, all storage systems will be expected to be provenance-aware.

We describe a PASS prototype, demonstrate that tracking provenance does not incur significant overhead, and present comments from a prospective user indicating that provenance-aware storage helps scientists get their jobs done better than is currently possible.

1 Introduction

Provenance, from the French for "source" or "origin," refers to the complete history or lineage of a document. In the scientific community, provenance refers to the information that describes data in sufficient detail to facilitate reproduction and enable validation of results [2]. In the archival community, provenance refers to the chain of ownership and the transformations a document has undergone [5]. However, in most computer systems today, provenance is an after-thought, implemented as an auxiliary indexing structure parallel to the actual data [6].

An object's provenance is, in many cases, nearly as important as the object's data. In some well-known instances, provenance is sufficient to facilitate the recreation of an object. For example, the software development tool `make` [14] could be considered a provenance management system: in the absence of the object itself, a `Makefile` provides all necessary information for `make` to recreate an object from its components.

Provenance is so important that in many domains, practitioners have developed domain-specific provenance systems. For example, in astronomy, the Flexible Image Transport System, or FITS (a common data format) consists of a primary header followed by the actual data. This primary header is a set of key-value pairs recording meta-data necessary to interpret the data that follows [16]. Provenance frequently appears in this header under `HISTORY` keywords. In chemistry, the Collaboratory for the Multi-scale Chemical Sciences project [4] supports a flexible approach to provenance. The Collaboratory uses the Dublin Core elements [18] to store provenance (and other) meta-data. In biology, the GenePattern system [8] provides a web-based environment in which one can conduct microarray data analysis. The system maintains a database that contains the provenance of any output data files created within GenePattern. Finally, the physics community is heavily invested in Grid technology and uses the Metadata Catalog Service (MCS), which is a centralized Grid meta-data management service [6, 20]. The MCS maps attribute name-value pairs to logical file names. The set of potential attribute names is quite large, but includes provenance attributes; *e.g.,* the experiment that produced the data, or the identity of the input data set.

The fundamental problem with domain-specific approaches is that, by definition, the provenance and the object itself are being managed by two entirely separate data management systems. Usually the data is managed by a file system and the provenance is managed by a database system (for example, an RDBMS). This separation means that it is easy for the data and provenance to get out of sync. It also means that there is no way to guarantee or even "encourage" that provenance is recorded for any particular object. Total solutions such as GenePattern can enforce provenance handling, but only so long as analyses are conducted within the GenePattern environment. As soon as experiments or analyses are executed outside of the controlled environment, then provenance is no longer recorded.

In this paper, we suggest that provenance is merely a particular type of meta-data, that the operating system should be responsible for the collection of provenance, and that the storage system should be responsible for provenance management. Integrating provenance with the storage system provides several benefits:

- By making the system responsible for the collection and management of provenance, we are able to generate provenance automatically, thereby freeing users from having to manually track provenance and tool designers from having to construct application-specific provenance solutions.

- Provenance collection and management happen transparently. That is, users do not have to take any special actions or execute special commands to have the provenance of their data collected and maintained. A corollary to this is that unless users take extraordinary action, the correct provenance *will* be maintained on all objects residing on the storage system.

- The tight coupling provided by building provenance management into the system means that provenance cannot be lost or modified separately from the data it describes.
- By capturing the complete environment in which a process runs, we can provide a degree of reproducibility that is difficult, if not impossible, to achieve with application-level solutions.
- Properly collected and maintained, provenance can provide records required by today's business laws (*e.g.,* Sarbanes-Oxley).
- By placing provenance and data together in the storage system, we can ensure that provenance is not lost during normal management procedures such as backup, restoration, or data migration.

In this paper we define a new class of storage system, called a provenance-aware storage system (PASS), that supports the automatic collection and maintenance of provenance including the features described above. A PASS collects provenance as new objects are created in the system and maintains that provenance just as it maintains conventional file system meta-data. In addition to collecting and maintaining provenance, a PASS must also support queries upon the provenance.

After describing the general problems associated with a PASS, we introduce our PASS implementation. Our PASS collects provenance automatically and maintains it in a simple indexed data management facility that we have integrated with the kernel. It introduces reasonable overhead in disk space, memory usage, and performance. We provide a simple provenance schema on top of which arbitrarily complicated query or data management tools can be constructed. Thus, the storage system need not include sophisticated data management capabilities (*e.g.,* an RDBMS), but the system is flexible enough that one could still use such tools on top of it at user-level.

The rest of this paper is organized as follows. In Section 2 we outline the challenges that must be addressed in building a provenance-aware storage system. In Section 3 we describe our implementation of a PASS and discuss how it addresses the challenges discussed in Section 2. In Section 4 we present an evaluation of our system, demonstrating that the provenance collection and management does not induce unreasonable overhead and reporting on how the system is perceived by a computational biologist. In Section 5 we discuss related work and conclude in Section 6.

2 Trials and Tribulations of a PASS

Although many domain-specific provenance solutions exist, provenance is a kind of meta-data, and meta-data has traditionally been handled by the storage system. We consider a storage system provenance-aware if it:

- assumes responsibility for the maintenance of provenance;
- supports or enables automatic provenance collection; and
- provides provenance query capabilities.

The following sections explore each of these requirements in more detail.

2.1 Maintaining Provenance in the Storage System

Recall that a primary design goal for a PASS is to avoid the separation of provenance and data that accompanies domain-specific approaches to provenance. Thus in a PASS, provenance must be stored in such a way that during normal use, data and its provenance are tightly bound. In most systems this suggests that the provenance must be stored as part of the file system, on the same logical device, so that the loss of one logical device does not render data on another logical device unprovenanced.

We state no requirements about the storage structure of provenance in a PASS. For example, a simple architecture would be to store provenance attributes as extended attributes in a POSIX file system and use the `find` utility as a query mechanism. Such a query mechanism is likely to be slow as `find` neither requires nor knows how to take advantage of indexing. Other designs might store provenance in some form of in-kernel indexed storage system to facilitate fast, keyed lookup. At the other extreme, provenance could be stored in a relational database, so long as that database was guaranteed to contain the provenance of all data created on the file system. (Note that we do not suggest that it is a good idea to embed a full-featured database in a storage system – just that we consider it an acceptable possibility.)

2.2 Automatic Provenance Collection

In addition to storing provenance and data in a tightly coupled fashion, a PASS must also facilitate the automatic collection of provenance.

2.2.1 Different kinds of provenance

Data on a storage system can be divided into two categories: *original data* and *derived data*. Original data is created directly by a user or is delivered to a storage system from a monitoring device (*e.g.,* a telescope, a particle accelerator, a microarray). Derived data results from processing other data. For example, in a source code control system, source files are original data while object files are derived data. A PASS must provide means for collecting and maintaining provenance for derived data, but should also provide mechanisms that ease provenance collection for original data.

We view derived data as the result of a transformation. A transformation is the result of a computation that takes place in some particular environment, using some specific hardware and software configuration. It takes as input one or more objects and produces one or more new output objects. Note that not all processes are transformations; a process that produces no output is not a transformation as there is no new object created. Similarly, a process with no inputs is not a transformation as it is creating a kind of original data. The provenance of a transformation must include:

- A unique reference to the particular instance of a program that created it.
- The complete collection of all input objects.
- A complete description of the hardware platform on which the object was derived.

- A complete description of the operating system and system libraries running at the time the object was derived.
- The command line.
- The environment.
- Parameters to the process (frequently encapsulated in the command line or input files)
- The value of any random number generator seeds.

Data that is not the result of a transformation constitutes original data.

Original data introduces a separate set of challenges, and in some cases, automatically generating provenance is impossible. We identify four sources of original data: (1) a human being, (2) an external computational device (*e.g.,* a scientific instrument), (3) a networked source, and (4) application software that obfuscates provenance from the system. Data that truly originates from the mental output of an individual cannot have its provenance tracked automatically by a system, so we must rely on manual intervention. Fortunately, because humans generate data at slower rates than machines, networks, and programs, focusing on non-human provenance encompasses much of the provenance collection problem.

Data that arrives at a storage system from a computational device can be augmented using a device-specific adapter that provides the data's provenance with the data. For example, in astronomy, data from a telescope is typically accompanied by the position of the telescope (*i.e.,* the region of the sky being captured), the date, time of day, version of the hardware and software used to capture the data, and other forms of device-specific provenance. So long as a PASS provides a mechanism for these device specific adapters to specify provenance, collecting and tracking provenance for this class of derived data is only marginally more difficult than tracking it for derived data.

Data that arrives at a storage system over the network (*e.g.,* via `ftp` or `http`) is somewhat more challenging. If the originating source is provenance-aware, we might expect the provenance to arrive with the data. However, in today's world, few if any systems are provenance-aware. At a minimum, a PASS must record the source of the data (e.g., a URN), the time at which the data was obtained, and ideally a hash or other unique identifier so that future attempts to obtain the same data can be verified. A PASS might use an approach similar to the device-specific adapter discussed above. A filter mapped to a network connection could be used to automatically generate provenance for networked data.

Finally, applications may hide provenance from the system. For example, consider a GUI application that permits users to select analysis algorithms. The identity of each algorithm chosen ought to be part of the provenance of any output, but the operating system has no way of identifying them or even recognizing that they exist. Thus, a PASS must enable applications to add attributes to the system's provenance.

2.2.2 Network Attached Provenance

On a local file system, automatic collection of provenance is conceptually straightforward. The system controlling the storage also controls execution of processes and can therefore capture whatever information it needs while processes run. However, in many cases, local file systems are the exception and most important data is stored in network-attached storage systems. A network-attached PASS requires (1) a provenance-aware network protocol and (2) client-side extensions to transmit provenance.

Until provenance becomes mandatory in storage systems, we must adopt a solution that permits optional provenance. This can be achieved today by using the extended attribute capabilities of network storage protocols such as NFS [19] and CIFS [12]. In order to deploy a provenance-aware network-attached storage system, a vendor would either need to rely on client software to provide provenance or would need to supply client-side extensions that perform the provenance collection described above. Although this provides an obstacle to adoption, it is a small price to pay for the additional guarantees that integrated data and provenance management provides. (It is also no different from deploying any other client-side file system extension.)

In the long term, we firmly believe that provenance will become standard in storage systems and that protocols will be developed to address these issues. We merely bring up the challenges associated with network-attached storage to point out that they can be addressed in the short-term, even if the solution is less than ideal.

For the remainder of this paper, we assume a local file system. The majority of the issues are similar for network-attached storage, except we do not address the network protocol issues; they are beyond the scope of this paper.

2.2.3 Cycles and Versioning

Simple transformations lend themselves to simple provenance and simple ancestry relationships between objects (*i.e.,* object A is derived from object B which is derived from object C). Under such transformations, provenance can be expressed in a tree-structure, like ancestry, and new versions are created as the result of a transformation.

Since a process can create multiple output files and a process can read many input files long before it writes any output file, we find it convenient to think about the provenance of processes. A process's provenance is the concatenation of the provenance of its input files plus information about the process itself. A process's provenance is then passed along to any objects created or written by that process.

Multiple cooperating processes can generate ancestry structures that are not trees: they may contain cycles. Consider the example below.

	process P	process Q
1		read A
2		write B
3	read B	
4	write A	

These operations generate the following object-ancestry provenance records where $\rightarrow$ means "depends on":

1. $Q \rightarrow A$
2. $B \rightarrow Q$, hence $B \rightarrow Q \rightarrow A$
3. $P \rightarrow B$, hence $P \rightarrow B \rightarrow Q \rightarrow A$

4. $A \rightarrow P$, hence $A \rightarrow P \rightarrow B \rightarrow Q \rightarrow A$

Step 4 creates a cycle.

Cyclical ancestry is incorrect and should not be allowed. In this example, and in fact in all cases that do not violate causality, the actual ancestry is acyclic and the cycles are an artifact of the data collection: the granularity is either too fine or too coarse. If B is a temporary file and P is a subprocess, a high-level description of events might correctly omit it entirely; or, conversely, if the A written in step 4 is construed as a new version and thus a different object, the cycle disappears.

When provenance is entered manually, people will automatically and instinctively pick a recording granularity that makes the most sense in their environment, and the resulting ancestry information will be acyclic. However, when provenance management is automated, and the processes that act upon data are not simple transformations, selecting this intuitive granularity is more challenging.

First, provenance implies some form of versioning. If a transformation reads file A and B and then writes file A, the provenance of A before that transformation and the provenance of A after that transformation are different. That difference is what is traditionally referred to as a version. Even if a storage system does not support versioning of its files, once it becomes provenance-aware, it must become cognizant of the fact that a file may have different provenance at different points in time, and that these different collections of provenance represent versions of the provenanced object.

Second, at its lowest level, a system observes provenance events as a sequence of low-level read and write operations. A simple solution is to track provenance at this low level; that is, create new versions of objects and provenance those for every write. Not surprisingly, this level of detail generates unacceptable overhead: the size of the meta-data far exceeds the size of the actual data. In addition, this level of detail is not useful to end users. End users are interested in answering application-level queries and these do not correspond to per-write objects. Such fine-grained provenance recording might still be useful for security auditing and is left as a subject of future work.

The challenge in provenance collection is to recreate a high-level view of the transformations. As part of doing so, it must either avoid cycles or detect and break cycles that it encounters. Both cycle avoidance and cycle elimination require that the storage system carefully select points at which to cease adding provenance to an object, and instead, declare a new version of it. In this paper we refer to these points as "freeze" points. "Frozen" objects are those that have had no provenance (or data) added to them since a freeze point. When a frozen object is subsequently modified, we call that object "thawed", and a new version is materialized.

In section 3.2, we discuss our implementation of cycle detection and removal and the corresponding version management algorithms that we have developed.

2.2.4 Philosophical Issues

All the issues that we have discussed so far in this section are requirements for any PASS system. There is another class of features that raise questions whose answers are less clear.

Reproducibility Our initial goal for PASS was to enable the reproduction of specific objects, but not necessarily to provide direct support. However, early feedback from users suggested that the ability to generate scripts to recreate objects or capture sequences of experiments or analysis was highly desirable. Therefore, we wish to capture sufficient information in the provenance to support automatic regeneration. That said, in order to exploit provenance for reproducibility, it is essential that one can construct an identical execution environment. For example, if I run a program today on a particular release of the operating system and standard libraries, in order to reproduce it, I must construct an identical platform on which to run the process. This is not always practical, so we define the minimum requirement for a PASS that it be able to identify the complete environment in which a process must be run in order to generate identical output. As such, it is not a requirement that the storage system (or associated tools) be able to recreate that environment in its entirety.

Non-determinism The question of reproducibility introduces the question of how to handle non-determinism. Non-determinism that results from random number generation can be addressed simply by capturing the seed of that generator. (The particular generator is captured in the computing environment and is addressed as part of the software provenance discussed above.) However, not all non-determinism is so easily captured. For example, a multi-threaded process whose output is a function of the decisions of the thread scheduler can produce different outputs that are different, but whose provenance is indistinguishable. (Except perhaps for information identifying the specific invocation of the process.) In this case, exact reproducibility is simply not possible. The fundamental provenance question is whether different invocations of a process should be uniquely identified. For non-deterministic processes, the invocation is what makes two subsequent runs distinguishable. However, for deterministic processes, distinguishing outputs this way makes no sense. To address this issue, we define the process invocation as a provenanced element that can optionally be ignored.

Cryptographic Keys Processes that emit encrypted data generally use a cryptographic key to do so. This key is undoubtedly part of the provenance of the output; however, recording it would defeat the cryptography. Thus, a PASS is *not* required to record cryptographic keys for the data it stores. We understand that this limits the applicability of a PASS to certain domains, but view this as a lesser evil than retaining cryptographic keys.

Security Model Provenance is unlike other meta-data in its security properties. Most meta-data requires the same level of security or access control as the data it describes. However, this is not sufficient for provenance. Consider the following example from Cao *et al.* [13]: A manager is preparing a review for one of her employees. She receives several pieces of email from colleagues, segments of which are cut and pasted into the review. The manager would not want the person being reviewed to have access to the provenance of the review, because that would relinquish the anonymity she wishes to provide to colleagues. In this example, we need stronger access controls for the provenance than we do for the data. In other instances, *e.g.,* a classified user creates a classified document, the access controls for the

data need to be at least as strong if not stronger than those of the provenance. Thus, a PASS must provide separate access controls for data and provenance.

2.3 Provenance Queries and Indexing

Setting aside methods of presentation, a PASS must be able to answer certain fundamental queries about the provenance of an object. Some of these queries are similar to what current file access tools offer and others require more thorough indexing than the file system currently does to make them useful on an interactive basis.

Queries on provenance stored in a PASS fall into three main categories: (1) a file's immediate provenance, (2) a more comprehensive query over the ancestry graph, and (3) queries about other provenance-specific meta-data. These categories are analogous to `fstat`, recursive `find`, and a `find` starting at the root of the file system with parameters like `-size`. The first provides detailed information about a particular file, the second probes a graph recursively, and the third scans all files looking for matches.

The simplest provenance query asks for the immediate ancestors of a file: what executable created the file and what files the process read. A minor elaboration on this type of query might return any parameters used or the complete command line that created the file. File systems could support simple provenance queries through an augmented inode structure that includes references to parent inodes and the command line or other executable information. An augmented `fstat` could then return sufficient information to answer queries about immediate predecessors of a file. At the cost of some redundancy, inodes could also include pointers to their immediate children, allowing for simple descendant queries.

While finding a file's immediate ancestors and what files it has created are akin to standard lookups performed on meta-data today – *e.g.,* finding files in a directory – the true power for a user of a PASS comes in terms of full provenance ancestor and descendant queries. These queries are inherently *recursive*: they traverse up or down the ancestry graph. Therefore, in order to be efficient, they require some form of auxiliary data structure beyond the immediate parent and child references that suffice for simpler queries.

Examples of these more comprehensive provenance queries are of the forms "How do I recreate this file?" or "How did this file come into being?" Note that these queries are fundamentally distinct because it may be possible to describe how a file came into existence without being able to recreate it exactly. Like the simple query, both of these might return results augmented with parameters or whole command lines expressing the file's origin as a shell script.

Being able to perform the full provenance query is the essential motivation of this work. It is what makes PASS an enabling technology for businesses complying with Sarbanes-Oxley and scientists striving to reproduce their own or others' results.

Queries down the ancestry graph show the usage and impact of a file, which is useful in at least two forms. First, they find dependencies. This is useful when newer versions of objects become available; for example, if a code generation tool was found to be buggy and a fix was issued. The person in charge of such an object far up the dependency chain can find its users in order to notify them that an improved version is available. Of course, on a smaller scale, long chains of dependencies often exist within an individual's working environment, including those of the scientists we interviewed. Second, descendant queries show who has used a file. This answers the question "How much impact has my work had?". For example, if an astronomer has a particularly interesting snapshot of a galaxy, processed with her tools, she could find who had used the file.

The final type of query requires being able to search on attributes. As we conducted user studies, it became clear that the ability to query on particular attributes was important. For example, users wanted to be able to ask: "For which files did I use this set of parameters?" While this type of information could be accessed with `fstat` or within a parameterized `find`, such queries do not operate at interactive speeds on a storage system of any realistic size, unless there is indexing support.

The requirements of these three types of queries suggest a more robust architecture than simple tweaks to the file system can provide. Instead, a PASS must make comprehensive queries viable for an interactive user. The simplest provenance query, the immediate ancestry query, could work with an augmented inode. The more complex recursive and attribute queries, which might contain thousands of dependencies, however, require the storage system to include data structures specialized for searching.

3 Implementation

Our prototype implementation was developed with scientific users in mind. This target user community has guided our database design and our choice of events to record. Nonetheless, we believe that most of the decisions we have made are reasonably general and will be necessary in other domains as well. That is, PASS is *not* a domain-specific provenance solution.

Our system has three components: first, a storage system, described in Section 3.1; second, an automatic provenance collector, described in Section 3.2; and finally, query tools, described in Section 3.3.

The provenance collector records pertinent system activity and keeps track of provenance meta-data for in-memory objects. When these objects are flushed to disk, the provenance meta-data is handed to the storage system, which stores it in a simple indexed store. Later, users may extract information of interest from the database using the query tools. An illustration of how the parts fit together is shown in Figure 1.

We now examine each of these components in more detail.

3.1 Pasta: The Provenance And STorAge Layer

The storage system itself consists of a stackable file system, called Pasta, and a kernel-level database engine.

Pasta uses the FiST [22] toolkit for Linux to create layers on top of any conventional file system; in our case we use `ext2fs`. This underlying file system then holds both the regular file data and the provenance.

We use an in-kernel port of the Berkeley DB engine [17], called KBDB [10] to store provenance. Berkeley DB is a high-

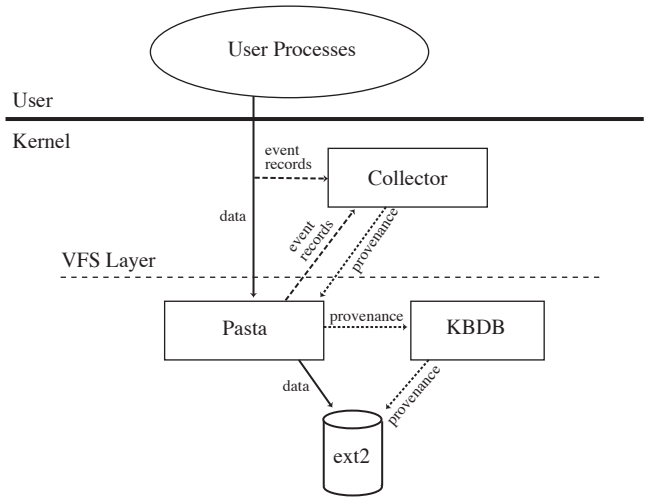


Figure 1: Architecture of our prototype PASS implementation. The query tool accesses the databases at user level.

performance transactional storage system that traditionally embeds in an application's address space. KBDB retains the high-performance transactional capabilities of Berkeley DB and runs in the kernel's address space. This allows Pasta to call directly into the database library to store provenance. Using a user-level database engine would incur extra domain-crossing costs.

Pasta is not a versioning file system in the conventional sense, in that it retains only the most recent version of each file's data. However, it is versioned in the provenance sense in that it retains distinct provenance information for every file version created by the system. These "provenance nodes," or "pnodes," are uniquely numbered and are not removed or recycled even when a file is deleted. This is necessary to ensure that full provenance information is retained for descendants.

Berkeley DB provides no schema capabilities; it simply stores key/value pairs on top of which applications (or in this case, the kernel) impose their own schema. Each collection of related key/data pairs is called a database in Berkeley DB terminology (in a relational database, it would be called a table; we use the Berkeley DB terminology here). Pasta uses four Berkeley DB databases, organized as follows:

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

The MAP database records the pnode number for each inode; this changes over time as files, described by the same inode, are frozen and thawed. The PROVENANCE database holds the provenance data: cross-references to other files, references to previous versions of the same file, generating process information, etc.

The ARGDATA database holds the command line and environment strings; this is a space-saving measure. Command lines and particularly environments are large, relative to the other provenance data, and are often repeated; offloading these to their own database allows us to avoid storing multiple identical copies in the PROVENANCE database. The record id number is an integer key assigned by Berkeley DB.

The ARGINDEX database is a secondary reverse mapping that serves as a text index of the command lines and environment lists. This mapping is maintained to accelerate queries based on specific command line arguments, command names, or environment settings.

The record types found in the PROVENANCE database are as follows:

`NAME`	full pathname of file within volume
`INPUT_FILE`	pnode number of an input file
`PREV_VERSION`	pnode number of a previous file version
`ARGUMENTS`	ARGDATA record id for arguments
`ENVIRONMENT`	ARGDATA record id for environment
`PROC_NAME`	name of generating application
`PID`	pid of generating process

This database permits duplicate data values for keys, so each input file appears in its own key/data pair. This database is stored sorted first by pnode and then by string so records are clustered by pnode.

In order to show how the schema is used, we describe a query that finds all the ancestors of a given file. We begin by using `stat` to retrieve the file's inode number. We then lookup the inode number in the MAP database; this lookup returns a pnode number. A pnode number provides access to the PROVENANCE database. Combining this number with each of the `PROC_NAME`, `ENVIRONMENT`, and `ARGUMENTS` strings returns exactly one entry. A fourth query using `INPUT_FILE` produces one entry for each input file. These values provide enough information to perform recursive lookups for each file whose version is less than our own.

3.2 Collector

The role of the collector is to record provenance data or events as they happen within the kernel and then to ensure that these records are passed to the file system for attachment to the appropriate output files.

There are several parts to this task: dataflow, data recording, and cycle eradication.

3.2.1 Dataflow

Provenance is meta-data, and the meta-data must flow through the kernel in parallel with the data it describes. In principle, each data transfer operation (`read`, `write`, etc.) must have a corresponding provenance record that describes it. Additional records describing actions of later interest may also be generated.

In our system, all files and all processes are considered provenanced kernel objects. We retain provenance in memory even for files that are not on provenance-aware volumes; this allows them to participate in the ancestry of files that are on such volumes. Processes, or, more precisely, the virtual memory spaces of processes, are collections of data similar in some ways to files; tracking the provenance of the data in each process mem-

ory space seems the most natural model, as it best matches the flow of data in the system.

Each provenanced kernel object has a virtual provenance node, or vpnode, associated with it. Each of these accumulates a list of provenance records, which, together, form the provenance of the kernel object.

Our system supports file-level, not record-level, provenance. So when data flows through the system, it is sufficient to record from where it came, that is, from which other provenanced kernel object it was derived; there is no need to record precisely which part of the object was accessed or the precise ancestry of that particular data.

There are two ways to store dataflow information: either as a cross-reference to the provenance of another object, or by directly copying the complete provenance of the referenced object. Within the kernel, for space and general efficiency reasons, we always create cross-references. However, when provenance is sent to the file system for storage on disk, some copying is always required: in general the provenance of an output file will always refer to the provenance of at least one process and occasionally more. It may also refer to non-PASS files or pipes and other file-like objects. At freeze time, when provenance is written, the collector logic traverses these other objects and copies their provenance records into the file system as part of the target file's provenance.

Recording ancestry at the file level also means that many duplicate records are introduced. A simple file copy operation that copies a large file in buffer-sized chunks will generate many cross-references from the source to the target, but we only need one. In our implementation these duplicates are eliminated as early as possible to reduce memory usage.

Note that provenance is written only at file freeze time, that is, when we decide we must stop adding to one version of the file and start a new one. This occurs on the last close, at `fsync` time, and potentially at other times as required by cycle aversion algorithms or other internal considerations.

3.2.2 Data Recording

Data recording refers to the creation of individual provenance records as events happen. Data recording occurs at various places in the kernel. In our current implementation, almost all events are recorded in the provenance of the current process; write operations are the only exception and are recorded in the provenance of the target file. This makes the recording of information reasonably straightforward.

As a matter of implementation convenience, certain provenance-related events are recorded by the storage system, that is, below the VFS layer, and then handed to the collector. Other events are recorded in the system call layer or wherever else seems most accurate.

Handling the `mmap` call poses a complication. The ordering of dataflow events into and out of a process is important, because it determines which input files are ancestors of which output files. Tracking `read` and `write` operations is easy; the kernel controls the data flow and can easily take notes as it happens. When a file is mapped into memory, however, the kernel is taken out of the loop of the primary data flow; it only knows when pages are moved in and out of main memory, which introduces "fuzz" into the relationships. Worse, implementation issues make it relatively difficult to record these flows. In our implementation, we treat `mmap` system calls as `read`/`write` system calls to the file and take the same action as we do for normal `read`/`write` calls. Treating `mmap` calls in this manner may introduce some false provenance if a region is mapped in and never accessed or written. The false read provenance that arises in this case can also arise from `read` calls as well, although it is significantly less likely (i.e., data can be read from a file, but that does not guarantee that the data read is actually used to generate the data written). The false write provenance is unique to `mmap` in that we assume the process modifies the mapped region, when, in fact, it might not. We can avoid this false provenance by initially mapping all segments read-only and then remapping them for writing if/when we take a fault.

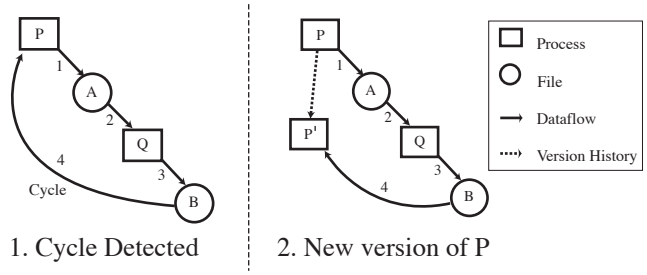


Figure 2: Cycle breaking by versioning. Files and processes frozen during cycle breaking.

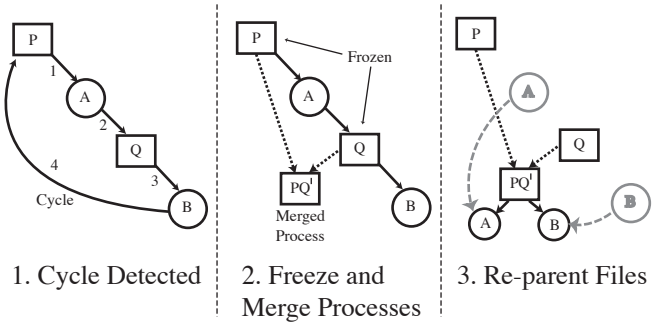


Figure 3: Cycle breaking by node merging.

3.2.3 Cycle Eradication

The collector takes responsibility for making sure no cycles appear in the provenance. The simple way to avoid cycles is to declare a new file version on every write operation; but as already mentioned, this is prohibitively expensive. Our solution detects and then breaks cycles.

To detect cycles we maintain, in memory, a directed graph holding the ancestry information for all the currently live vpnodes. This graph can be kept reasonably sized by pruning: an object that is frozen and has no unfrozen ancestors cannot participate in any future cycle and can be dropped from the graph. Likewise, objects newly loaded from disk are always frozen and need not be added to the graph in the first place. If written, they are thawed, creating a new version; only that version need be

considered.

When cross-reference relationships are recorded, the graph is checked; if a cycle would be introduced, the cycle-breaking algorithm is invoked. We considered two different cycle-breaking algorithms. The first, shown in Figure 2, breaks cycles by versioning. It bumps the version on one of the participants of the would-be cycle; this creates a new object with no descendants, which cannot be part of a cycle, so the cycle goes away. However, this method performs poorly with workloads that have cyclic dataflow: each iteration generates a whole new series of file versions, generating significant overhead and a complex data mining task for the query tools to sort out later.

Instead, we use a second algorithm based on node merging. It works on the principle that a set of processes moving data around in a circle are really a single cooperating entity, and instead of being individually provenanced, they should share a single common set of provenance.

When a cycle is detected, the algorithm is invoked and works as follows: first, freeze all the process vpnodes (but not the files) participating in the cycle. Then, create a new process vpnode, a descendant of these, to be shared in the future by all the participating processes. Finally, all files participating in the cycle become descendants of the merged process. We illustrate the algorithm in Figure 3.

We chose this algorithm because it generates the least "noise" in the resulting output; by modeling the high-level organization of the process activity it gives a better match to the application-level types of queries that users will want to perform later. It also has the advantage that while it freezes processes, thus generating new versions, it does not freeze files on disk. Process versions exist only in memory, incur almost no overhead, and are invisible to query tools afterwords; on-disk versions exist forever and furthermore complicate the data for the query tools.

3.2.4 Code

The implementation of the collector logic is about 3800 lines of centralized code, along with about 250 lines more of small patches at about a dozen critical points in the kernel (fork, exec, exit, iget, etc.)

The collector communicates with the Pasta file system via five new inode operations added to the VFS layer:

1. `getpnid` - returns the on-disk pnode number for this file
2. `thaw` - bump the version for this file: allocate a new pnode number and prepare for writes
3. `freeze` - notification on file freeze (does nothing in Pasta)
4. `write_prov_string` - write a "flat" (non-cross-reference) provenance record
5. `write_prov_xref` - write a provenance record that cross-references (by pnode number) another file on the same volume

3.2.5 Drawbacks

There are two primary drawbacks to our collector architecture: first, because it operates above the VFS layer, it does not know what events pertain to PASS file systems. It ends up collecting data for all files on all volumes and then discarding it.

Second, because provenance is not (and cannot easily be) sent to disk before freeze time, after a crash, partially written objects may be unprovenanced.

```
sort -n A > A.sort
sort -n B > B.sort

./multiply -x 1 -y 4 A.sort B > AB
./multiply -x 2 -y 5 B.sort A > BA

uniq AB > AB.uniq
uniq BA > BA.uniq
```

Figure 4: The set of commands used to generate a file. `A`, `B`, and `multiply` originated in `demo.tar`.

```
# prov -m BA.uniq

tar xf demo.tar
sort -n B > B.sort
./multiply -x 2 -y 5 B.sort A > BA
uniq BA > BA.uniq
```

Figure 5: The output from the query tool in shell script mode.

3.3 Query Tool

In our prototype, the provenance database is made accessible at user level and exists as a standard set of Berkeley DB databases. A wide variety of tools can be used to handle queries on provenance data. For example, BDB can be accessed with Python, Perl, Tcl, Java and other tools. Our query tool prototype, written in C, uses a user-level process to access the databases, read-only. It presents results to the user on the command line or through a web interface.

This is, strictly speaking, an architecture suitable only for a prototype; security considerations dictate that access to the provenance database be mediated by the kernel. Since both the user-level tool and the kernel can use the same Berkeley DB interfaces, adding kernel mediation is a straightforward procedure. We assign access controls on a per-pnode basis so kernel mediation requires only that the kernel verify that the user process requesting information have appropriate permissions to read the pnodes that are accessed during a query.

The command line version of the tool takes a single filename as input and a collection of parameters. The default operation is to print the full ancestry of the file. The parameters specialize the output into one of the following two forms:

- Prune the provenance after a specified level, *e.g.*, a level of one would show only those files upon which this file immediately depends; a level of three would show ancestry up to the "great-grandparent" level.
- Shell script mode outputs a list of commands to reproduce the file. Figure 4 shows some sample commands we ran, and Figure 5 shows the shell script mode output of the query tool when asked how to reproduce one of the files involved.

These modes can be combined to print shell scripts starting with a particular ancestor. The tool currently outputs full command

lines; the mechanism to query parameters individually is not yet in our prototype. The web version supplies the same options through an easier (but less scriptable) interface. Because we are simply accessing a database, additional basic query functionality required of a full PASS system is not fundamentally different from what we have currently. Instead, we have focused on simple tools that have allowed us to get early-use feedback as described in Section 4.5.

In addition to the simple text format shown in Figure 5, our query tool can output in the DOT graph representation format [7]. This allows easy generation of visual representations of the ancestry graph. An example of the graph produced by a BLAST [1] workload appears in Figure 7. (BLAST, the Basic Local Alignment Search Tool, is a program that assigns similarity scores to sets of nucleotides and amino acids based on their relative proximity.)

4 Evaluation

The benefit of PASS is increased functionality, not performance. However, new functionality is not useful if it affects performance so dramatically that becomes impossible to get real work done. Thus, our evaluation has two parts. First, we demonstrate that our PASS implementation does not introduce undue overhead in the system. Then we report on experiences that a colleague in computational biology reported when she conducted her daily work activities on a PASS.

4.1 Evaluation Platform

We evaluated our PASS prototype on a 500Mhz Pentium III machine with 768MB of RAM. All experiments were run on a 40GB 7200 RPM Maxtor DiamondMax Plus 8 ATA disk drive. The machine ran Red Hat 7.3, with a Linux 2.4.29 kernel. The Linux kernel was patched to enable provenance collection. We chose Red Hat 7.3 and Linux 2.4.x because two groups of our early users currently run their tools on this platform.

To ensure a cold cache, we reformatted the file system on which the experiments took place between test runs. We measured system, user, and wall clock times, and also the amount of disk space utilized for recording provenance. We also recorded the wait times for all tests; wait time is mostly I/O time, but other factors like scheduling time can also affect it. Wait time is computed as the difference between the elapsed time and the sum of system and user times. We ran each experiment 10 times. For each of our results, the standard deviation was less than 5%. We do not discuss the user time in the results as our code is in the kernel and does not affect the user time.

4.1.1 Overhead Evaluation

We ran two benchmarks on our system: a real workload from the Computational Genomics lab at our university and a CPU-intensive benchmark.

The first workload, from the Genomics lab, takes two input files and produces one result file at the end of the process. Each of the two input files contains protein sequences from different species of bacteria. The output file contains a list of proteins in the two species that may be related to each other evolutionarily. The workload consists of a series of commands that produce intermediary output files that are fed as input to the next command. Starting from input and configuration files, fifteen files are produced, including one result file. The scientists at our Genomics center would find PASS useful to easily "recollect" the input files from which the output was derived, two months after the fact.

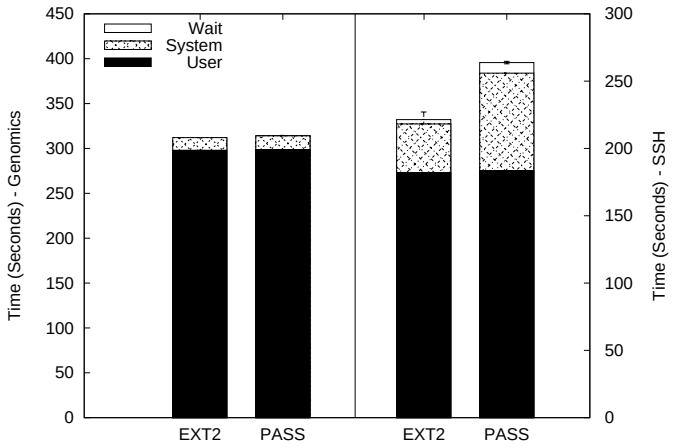


Figure 6: **Overhead for Genomics workload and SSH Compile. The first half of the graph is the Genomics workload result and uses the left scale. The second half of the graph is the SSH Compile result and uses the right scale.**

The second workload was a build of OpenSSH. We used OpenSSH 4.1p1; it contains 65,921 lines of C code in 158 files. The build process begins by running several configuration tests to detect system features. It then builds the binaries and documentation: a total of 194 new files and 11.8MB of data. Though the OpenSSH compile is CPU intensive, it contains a mixture of file system operations. This workload approximates the performance impact a user would see when using PASS under a normal workload.

For each workload, we evaluate the performance and space overhead and then compare the number of provenance records recorded by PASS to other similar systems. We found it challenging to precisely quantify the memory overhead that our system introduces, but several *ad hoc* measurements confirmed that the in-memory overhead is roughly comparable to the size of the provenance for objects being created/written.

4.1.2 Configurations

We used the following configurations for evaluation:

- **EXT2:** Vanilla Ext2, used as baseline for performance.
- **PASS:** PASS stacked on top of Ext2.

4.2 Performance Overhead

4.2.1 Genomics Workload

The left half of Figure 6 compares the overhead of PASS with EXT2 for the Genomics workload. The overhead on elapsed time is negligible (less than 1%) with both the system time and wait time increasing negligibly. At any point, there are at most three open files, hence the in-memory cycle breaking algorithm

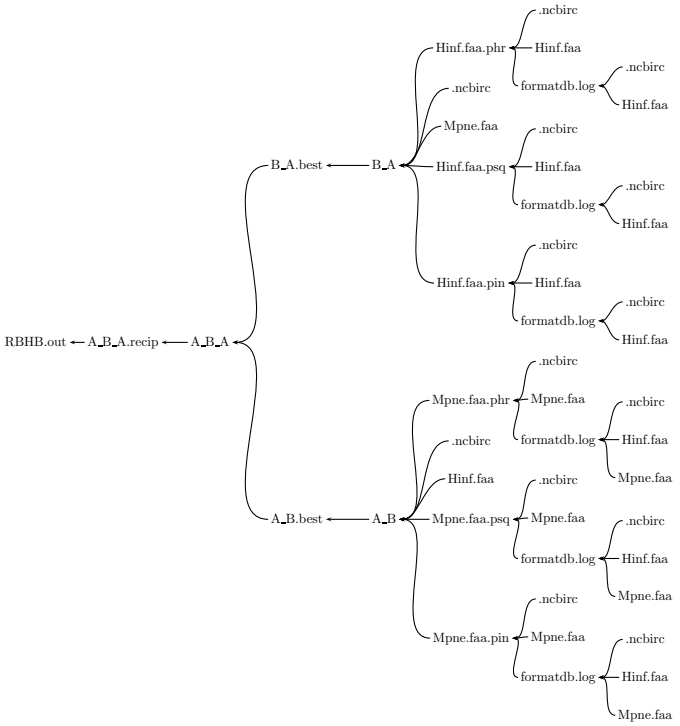


Figure 7: **Provenance tree for Genomics workload (Blast).**

does not measurably effect the system time. The amount of provenance generated is also quite small (see Section 4.3), hence the wait time is also not affected significantly.

Figure 7 shows the provenance tree for the Genomics workload. `Mpne.faa` and `Hinf.faa` are the two files containing the protein sequence, `.ncbirc` is a configuration file, and `RHRB.out` is the output file.

4.2.2 SSH Compile

The right half of Figure 6 compares the overhead of PASS with EXT2 for the SSH compile benchmark. Overall, there is a 19.2% increase in the elapsed time for PASS compared to EXT2. The increase in the elapsed time is mainly due to the system time doubling from 36.1s to 72.3s. The increase in system time can be attributed to the in-memory cycle checking. The number of nodes in the graph increases as the compile progresses (at the end of the compile the graph contains 12,025 nodes) increasing the cost of cycle checking. (The code in question is untuned and largely unoptimized at this point, so this overhead can be expected to be less in future versions.) The wait time also increases from 3.2s to 7.9s due to the extra provenance meta-data being recorded.

4.3 Comparison of records generated

Systems such as the Lineage File System [13] and Backtracker [11] log every `read` and `write` and later build the lineage from the log. Reading or writing a large file requires multiple `read`/`write` system calls on the file. In such cases, the Lineage File System and Backtracker record multiple entries where one would suffice. PASS, however, detects such duplicates producing fewer records. Table 1 shows the number of records that would have been recorded in systems like the Lineage File System and Backtracker and the number of records recorded by PASS. PASS records significantly fewer records. We estimated the number of records generated by the other systems by counting the number of `read`, `write` and `mmap` system calls executed by the benchmarks. The number of records generated by PASS is a count of the number of records in the provenance database at the end of the experiment. The last column in Table 1 shows the amount of reduction in the number of records due to the duplicate detection.

In scientific experiments, we expect there to be a small number of large files implying that a large number of read/write calls are needed to process them. Logging each call is inefficient. Duplicate detection reduces the storage space required to store provenance and hence the time required for building provenance trees.

4.4 Space Overhead

Table 2 shows the space overhead due to provenance. The space overhead for the Genomics workload is 2.8% and for SSH compile is 12.9%. Although we find these overheads acceptable, we expect that with pruning algorithms, we can reduce these if necessary.

4.5 User Evaluation

During the design of PASS, we met with researchers from astronomy, physics and computational biology to understand what functionality PASS needed to be practically useful to scientists. Once we had a working prototype, we sought out other scientists who knew nothing about PASS until being solicited for this study. Our goal was to generate early user feedback on PASS, in general, and on our query capabilities in particular. We successfully recruited a member of the Computational Genomics lab at our university. Our early adopter, who we call Suzy, was actually less UNIX-savvy than the scientists with whom we'd met earlier, but was willing to use our system and provide feedback.

According to Suzy, she exemplifies a fairly typical user in advanced computational biology: knowledgeable in her field, but not comfortable with the "black window" of the Unix command line. However, she did feel comfortable enough to run a small set of commands for which she had documentation. She performed all file operations (*e.g.*, copy and make directory) through a Windows GUI and did not use common Unix shell commands like `cp` or `mkdir`.

A typical computational biology experiment consists of sifting through gene expression data. Gene expression data is obtained by microarray experiments, which produce measurements that indicate what genes are "turned on" (expressed) in response to particular probes. Suzy first uses a microarray to gather the gene expression data for her organism (the fish *A. burtoni*). Then using BLAST [1], a tool for locating a particular gene sequence in the genomic sequence of known organisms, she creates hundreds of files from the parameter combinations used to generate the BLAST queries. Most of these queries would be "dead ends;" a small subset were compared against an dataset available on the Internet.

Suzy has a private system for keeping track of the parameters and inputs for each output file. Usually these were partially

Benchmark	Other systems	PASS	% Savings
Genomics Workload	28,499	160	99.4%
SSH compile	99,699	34,783	65.1%

Table 1: Comparison of number of records generated by other systems with number of records generated by PASS.

Benchmark	Data Size	Number of files	Size of Provenance	% Overhead
Genomics Workload	5.6MB	18	160KB	2.8%
SSH compile	33.1MB	595	4.3MB	12.9%

Table 2: Space overhead due to provenance.

reflected in the output file's name, but sometimes they were manually entered in her lab notebook.

Suzy tried out our system for approximately an hour. She found the system "transparent" because she was able to use her normal tools and features like output redirection as she normally does. The aspect she found most useful was the shell script creation. It showed the full set of parameters used to generate her output files. She said that if she were able to use the system full-time, she would create shell scripts both for documentation and to modify and simplify processing of other files. Currently, all of her typical procedures are captured using our PASS prototype, except for the final remote processing stage. We discussed how this stage might be augmented to allow for semi-automated provenance.

Suzy said two pieces of functionality would be most useful for short term use. First, she would like to annotate the outputs of her experiments with information on why she chose certain parameters and why, in particular, certain experiments did *not* work. She said it would be most useful if these annotations were closely associated with each file so that they would be either printed along with the provenance information or added into the shell script that described how the file was generated. She also wanted to be able to search these annotations, which, if they were stored as a separate file, could be indexed with existing tools (*e.g.,* Glimpse [15]). These user annotations are distinct from provenance, but could be linked with a file through our existing database mechanisms; in fact, this functionality has been part of our design from the outset and is scheduled for a future release. Second, she wanted to search the parameters for each experiment. For example, which outputs did I create yesterday using a BLAST distance of X? This is simply a query on the command-line records stored in our provenance database and could be constructed using Perl or Python, but is not yet implemented in our query tool. Finally, she asked when it would be possible to install PASS for full-time use because, she said, it would greatly simplify her and her labmates' work environment at no apparent cost.

Although this reflects only a single user's experience, we find it a powerful validation of PASS. With no training, a non-programmer was able to instantly collect and query provenance, without having to change her work habits at all or learn new tools. The human factors evaluation of PASS is outside the scope of our current work, but we intend to conduct a thorough user-study that describes and, where possible, quantifies the benefits it provides.

5 Related Work

There are three main areas of work that overlap with the research areas relating to provenance-aware storage: domain-specific provenance solutions (which have already been discussed in Section 1), source code control and build systems, and other general purpose provenance systems. In the remainder of this section, we focus on these latter two categories.

5.1 Source Code Control and Build Systems

As mentioned earlier, source code control and build systems are actually provenance systems. Although source code control systems store provenance, their primary purpose is to provide versioning and building capabilities. As discussed in Section 2, data versioning and reconstruction of derived objects is a secondary goal for a PASS. The primary goal is maintaining the complete provenance to facilitate responding to queries about an object's origin. This difference in emphasis leads to very different design decisions in source code control systems and in PASS, even though there is also a significant overlap in functionality. For example, a build systems typically maintains some sort of user-created description of how the derived objects are created (e.g., Makefiles) and to not necessarily explicitly track the dependencies that exist between objects. In a PASS, no such "recipes" exist; rather, the recipes are created each time a new object is created.

Although a number of source code control systems exist and manage provenance, those that are most similar to PASS are those that create environments in which the build process is automatically monitored and maintained. ClearCase (and its predecessor DSEE) and Vesta are the two systems most closely related.

ClearCase [3] is a source code control system, now owned and sold by IBM. It includes a custom file system that acts as the source code repository. The build system relies on the standard `make` utility, but the custom file system tracks and maintains system dependencies to avoid work in future builds and to trigger re-evaluation of the build process. These dependencies are part of the provenance that a PASS captures. However, as is the case with all build systems of which we are aware, the means by which derived files are created is specified *a priori*. In

a PASS, these derivations are obtained by observation and must be obtained as any process runs, not simply those processes under control of the source code control system.

Vesta [9] is a second generation build system developed at DEC Silicon Valley 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 in order to extract dependencies and record the complete environment information that will facilitate repeatable builds. And like DSEE and other source code control systems, it relies on an *a priori* description of the derivation process. As a result, it ignores the central PASS challenge: automatically generating the derivation rules as a system runs.

5.1.1 Source Code Control System Summary

In summary, source code control systems focus on versioning and facilitating builds. A PASS's primary goal is recording derivations, as they happen to facilitate responding to queries. Versioning is something that is a necessity in PASS, but only because it facilitates accurate provenance accumulation. Similarly, building is a desirable side effect, not a central design point. More fundamentally, source code control systems require explicit use; PASS is transparent and implicit. Users need not do anything special to track provenance on a PASS, yet they can recreate their actions and trace the derivation of the files they create.

5.2 Provenance Systems

The only two systems of which we are aware that address provenance in a general manner are Trio and the Lineage File System.

Trio [21] focuses on providing provenance for data in a database system. The Trio project is a centralized database system that manages not only data, but also the provenance and accuracy of the data. Trio focuses on the data model and extending SQL to support lineage and accuracy information. Provenance tracking is initially off, and must be activated – on a per relation basis – by the application. When active, lineage tracking occurs at the tuple level. Tuples are not modified in place or deleted; instead new versions of the data are created. Like PASS, Trio is interested in recording and querying provenance. Unlike PASS, Trio has initially focused on formalizing a query language and data model, and, to the best of our knowledge, does not yet have an implementation.

Trio and PASS are likely to ultimately be complementary. Since PASS tracks provenance at a file-level granularity, it will not be useful for tracking database updates. However, it is likely to be more efficient and useful for tracking provenance of file system objects, as observed by Suzy.

The Lineage File System [13] is an instance of a PASS, but differs in several significant ways from the implementation described here. Like PASS, the Lineage File System focuses on executables, command lines and input files as the source of provenance. Unlike PASS, it ignores the hardware and software environment in which such processes run. A second, and perhaps more important, difference is that provenance collection is delayed in the Lineage File System and it is performed by a user-level thread that writes the lineage data to an external database. As a result, the tight coupling that we require of a true PASS is lost, as is a significant part of the benefit. Since the Lineage File System stores its lineage records in a relational database, the query language is SQL. In our implementation, we use a simple key/value storage schema so that a variety of schema layers can be provided.

6 Conclusion

In this paper, we have defined a new class of storage systems that track and maintain provenance automatically. Our goal is to help users — currently, scientists — gain more control over their own data and work environment. We have done so by providing an environment in which users can identify the origin of the files they were working on yesterday, last week, or last year.

These new *provenance-aware* storage systems must provide: tight-coupling of provenance and files, automatic collection where possible, methods to tackle cycles and versioning, and robust query tools. Our prototype implementation includes a *collector* to record relevant system activity, a file system layer, Pasta, to capture this activity and work with the collector to store provenance in an in-kernel database, and a query tool to answer user requests about a file's provenance. In response to the positive feedback we received from the early user we report on and others, we will be distributing be running beta versions of our software in several labs around our university.

There is much future work to be done on PASS, in general, and on our prototype, in particular. Some of the open questions for PASS are:

- How do our user experiences generalize? Do other scientists gain the same benefit as Suzy? Does PASS provide a useful tool for those responsible for complying with Sarbanes-Oxley or for archivists storing digital data?
- How do we extend PASS to the wide-area? Do we need a new network protocol? If not, how do we extend existing network file system protocols?
- Is provenance simply an instance of a more general problem? Should we be investigating PASS-like approaches for other kinds of meta-data, such as DRM information?

Some of the short term enhancements to our prototype include:

- Introduce kernel mediation into provenance queries.
- Provide enhanced query capabilities, in particular, user-friendly interfaces to query command line parameters.
- Integrate our automatic provenance with that produced by a work environment, such as GenePattern.
- Tune the performance and storage used by our implementation.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Molecular Biology*, 215:403–410, 1990.

[2] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *International Conference on Database Theory*, London, UK, Jan. 2001.

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

[4] Collaboratory for Multi-scale Chemical Science. `http://cmcs.org`.

[5] Committee on Digital Archiving and the National Archives and Records Administration. R. Sproull and J. Eisenberg, eds. *Building an Electronic Records Archive at the National Archives and Records Administration: Recommendations for a Long-Term Strategy*. The National Academies Press, Washington, D.C., 2005.

[6] W. Deelman, G. Singh, M. Atkinson, A. Chervenak, N. Hong, C. Kesselman, S. Patil, L. Peakrlan, and M. Su. Grid-based metadata services. In *Scientific and Statistical Database Management (SSDBM)*, June 2004.

[7] E. R. Gansner and S. C. North. An Open Graph Visualization System and Its Applications to Software Engineering. *Software: Practice and Experience*, 30(11), 2000.

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

[9] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, March 2001.

[10] 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.

[11] S. T. King and P. M. Chen. Backtracking Intrusions. In *SOSP*, Bolton Landing, NY, October 2003.

[12] P. Leach and D. Naik. A Common Internet File System (CIFS/1.0) Protocol, IETF Internet-Draft, March 1997.

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

[14] GNU Make. `http://www.gnu.org/software/make`.

[15] U. Manber and S. Wu. GLIMPSE: a tool to search through entire file systems. In *Winter USENIX Technical Conference*, San Francisco, CA, January 1994.

[16] B. Mann. Some Provenance and Data Derivation Issues in Astronomy. In *Workshop on Data Derivation and Provenance*, Chicago, IL, Oct. 2002.

[17] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, Monterey, CA, June 1999.

[18] C. Pancerella et al. Metadata in the Collaboratory for Multi-scale Chemical Science. In *Dublin Core Conference*, Seattle, WA, 2003.

[19] R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Summer USENIX Technical Conference*, Atlanta, GA, June 1986.

[20] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A metadata catalog service for data intensive applications. In *Supercomputing*, Phoenix, AZ, November 2003.

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

[22] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *USENIX Technical Conference*, Monterey, CA, June 1999.