A Distributed Provenance Aware Storage System

Mihai Bădoiu [*] Kiran-Kumar Muniswamy-Reddy [†] Anastasios Sidiropoulos [‡]
Mythili Vutukuru [§]

Abstract

The provenance of a file represents the origin and history of the file data. A Distributed Provenance Aware Storage System (DPASS) tracks the provenance of files in a distributed file system. The provenance information can be used to identify potential dependencies between files in a filesystem. Some applications of provenance tracking include (i) tracking the transformations applied to process raw data in scientific communities and (ii) intrusion detection and forensic analysis of computer systems. In this report we present the design and implementation of a provenance aware storage system, which efficiently stores and retrieves provenance information for files in a distributed file system, while incurring minimal space and time overheads.

1 Introduction

Provenance, from the French word for "source or origin", refers to a complete history or lineage of a document. In computer terms, it consists of information about the objects that a particular object is based on, the process of creation/modification of an object, etc. For example, consider a process P that reads from files A and B, performs some computation, and writes to a file C. Then the provenance of C consists of, the input files A and B, the application P that modified the file, the command line arguments and environment of process P, the processor type on which P is running, etc.

Provenance is particularly useful for scientific communities like Physics, Chemistry and Astronomy. Raw data, generated by scientific experiments is further processed and transformed multiple times, before it is published. Before using the published data in their experiments, scientists need to know whether they can trust its source. At this end, they need to know where the data came from, and the transformations it went through. Also, if it turns out that there was a flaw during the data generation and transformation process, the originators of a flawed data-set need to inform all users of the data of the flaw. Moreover, it is often desirable to keep track of enough meta-data, so the exact same experiment can be recreated. The provenance of a file can be useful in all of these scenarios.

Provenance can also be used for security purposes, to conduct forensic analysis after a break-in. Intruders gain access into systems by installing malicious worm *backdoors*, that can then corrupt files in the system. Upon detecting a suspicious file, we can examine its provenance, backtrack through the file system and locate the worm backdoor. We could also locate all the files in the system which depend on the worm backdoor and thus identify other possible corrupted files. BackTracker [5] is based on a similar mechanism of intrusion detection.

A provenance aware storage system (PASS) maintains the provenance information of a file in the file system, along with the other meta-data of the file. Complete provenance includes information about the applications that modified the data, the input data, and the environment under which the application was executed. For this project, we limit ourselves to capturing only the application that modified a file, the host on which the file was modified, together with the set of files that the process read before the modification. The files $A_1, \ldots, A_n$ that a process read before modifying a file B are the *ancestors* of B, and B is a *descendant* of each A_i. One of our main goals is to capture the ancestor-descendant dependencies between files.

It is common for users to have their data on a centralized file system, so that they can access it by logging into any machine. In order for the user to access the provenance from any machine, the provenance has to be stored along with the data in a centralized file server. A Distributed PASS (DPASS) is a distributed storage system that stores the provenance of a file along with the data, enabling the user to access the provenance remotely. Note that since the provenance depends on the processes that are running in the user's machine, the recording of the provenance must involve both the machine that the user is logged in, and the machine which stores the file

[*]E-mail: mihai@mit.edu
[†]E-mail: kiran@eecs.harvard.edu
[‡]E-mail: tasos@mit.edu
[§]E-mail: mythili@mit.edu

system.

1.1 Challenges in Building a DPASS

- **Automatic Provenance Generation:** In a primitive provenance tracking system, users who generate or modify a file, can be responsible for tracking its provenance. This solution however is unacceptable, since users might neglect entering provenance, might enter it incorrectly, or might find it cumbersome to enter the provenance manually. A PASS should automatically record provenance of files without human intervention, and without changing the existing applications, and programming interfaces.
- **Transporting Provenance:** DPASS requires that provenance be transmitted to the file server. It is desirable to transport provenance without inventing a new protocol or changing an existing protocol like NFS.
- **Storing minimal required information:** A naive approach to provenance recording, is to record every read and write by a process. This approach results in redundant dependencies, and incurs unacceptable storage and processing time overhead. Thus, it is critical that a PASS should store only the minimum required information that is sufficient to reconstruct all relevant dependencies between the files.
- **Querying Provenance Efficiently:** The provenance should be efficiently retrievable by applications. While a simple log containing all the writes and reads, is sufficient to capture any possible file dependencies, it cannot be queried efficiently.

The rest of the report is organized as follows. Section 2 describes the provenance tracking algorithm and the design of the database. Section 3 discusses the implementation details. Section 4 evaluates our system. Section 5 describes the related work. We conclude in Section 6, and discuss future directions.

2 Provenance Tracking Algorithms and Database Design

Our system captures all dependencies of the form $A \rightarrow B$ that exist between any two files A and B, denoting that the contents of B might have been derived from the contents of A. More precisely, $A \rightarrow B$ means that B was modified by a process that read A before modifying B.

In this section, we first describe a naive algorithm for tracking dependencies, followed by an improved algorithm that we use in our system. We also describe the format of the database used to store the provenance, and explain how we construct provenance trees.

2.1 A Naive Algorithm

The naive algorithm to capture provenance is as follows:

- Each time a process P reads a file A_i, record this event by appending a record to a buffer.
- Each time P writes to a file B, the data written to B could potentially depend on each file A_i that P has read. On every write to B, for every file A_i in the buffer of P, record the dependency $A_i \rightarrow B$.

Althought the above algorithm seems like a reasonable way to record provenance, we will next explain the main problems that render it inappropriate for our system.

- The naive approach results in a lot of redundant storage. For example, if the same file is read a second time by a process, this information should be captured only if the file has changed since the last read. Thus, optimizations are needed to avoid recording every single dependency on write.
- Since there are many redundant dependency entries, the time required to build dependency trees is increased.
- The naive algorithm could result in cycles while building the provenance tree of a file. For example, if process P reads file A and writes to file B (resulting in $A \rightarrow B$) and another process Q reads B and writes to A (resulting in $B \rightarrow A$), a cycle is formed. This can send a provenance tree building algorithm into a loop. It is desirable that dependencies of the form $A \rightarrow B \rightarrow A$ should be avoided. This dependency can be eliminated by noting that there are in fact *two different versions of* A involved here, and the file A that Q has written to is no longer the same as the file A that P read from. Hence, our system needs to store additional timestamps to recognize the different versions of a file.

We will show that by carefully recording the dependencies, the above problems can be avoided. In the next section, we present a *formal improvised algorithm* that

keeps track of a few timestamps with every read and write, in order to avoid capturing redundant dependencies and avoid cycles in the provenance tree of a file.

2.2 An Efficient Tracking Algorithm

2.2.1 Active file dependencies

For each file A_i that a process P has read or written, we store a tuple $(\text{inode}_i, \text{first-read}_i, \text{mtime}_i, \text{lp-write}_i)$, such that:

- i-node$_i$ is the i-node number of file A_i.
- first-read$_i$ is the time the first read system call was issued by P on file A_i.
- mtime$_i$ is the modification time of file A_i. This is updated every time A_i is read by P. The mtime$_i$ of A_i changes between two reads if and only if some other process has modified A_i between the two reads. Different mtime$_i$ of a file denote different version numbers of a file. mtime$_i$ is used to identify if a process is reading a different version of the same file. We denote a file A with mtime $= t$ by $A(t)$.
- lp-write$_i$ is the time when the provenance of A_i was last recorded in the database. This corresponds to the last write system call to A_i before which new files have been read by the process P.

We refer to the set of these tuples as the *active file dependencies* of the process. The active file dependencies of each process are stored in a separate buffer in memory.

Note that, when a process reads a file A_i, only the first 3 elements of the tuple are populated and when a process writes to file A_i, only the lp-write$_i$ field is updated.

As we next explain, these timestamps are used to eliminate redundant dependencies, and to avoid cycles during the construction of provenance trees.

2.2.2 Provenance recording rules

Recall that a dependency of the form $A_j(t_j) \rightarrow A_i(t_i)$ means that the version of the file A_i at time t_i depends on the version of file A_j at time t_j.

When a process P performs a write system call on a file A_i, the system scans the active file dependencies of P, extracts any new dependencies on which A_i depends on and records it to the database. The exact rules for recording provenance when a process P writes to A_i at time t_i are as follows:

- **Rule 1** If lp-write$_i$ = null, then this is the first write of P to A_i. No file dependencies for A_i have been recorded so far, and A_i depends on all the active file dependencies of P. All active dependencies of P are recorded in the database. For every file A_j, with $j \neq i$, that P has read from, we record the dependency $A_j(t_j) \rightarrow A_i(t_i)$, where t_j is the version number of A_j.
- **Rule 2** If lp-write$_i$ $\neq$ null, the provenance of A_i has been recorded before, and thus only some of the active file dependencies need to be recorded. For every file A_j, with $j \neq i$ in the active file dependencies of P, the dependence of $A_i(t_i)$ on $A_j(t_j)$ is recorded only if one of the following rules are satisfied:
  - **Rule 2.1** first-read$_j$ > lp-write$_i$, which means that A_j is a new file that P has read for the first time after the last write to A_i. Since A_j had not been read before the previous writes to A_i, $A_j(t_j) \rightarrow A_i(t_i)$ should be recorded.
  - **Rule 2.2** mtime$_j$ > lp-write$_i$, which means that the file A_j has been modified by some other process and process P has read the modified version. Since A_j has been read after it was modified, the write to A_i implies that A_i now depends on the new version of A_j. This dependency on the newer version of A_j should to be captured.

If any of the rules result in A_i's provenance being updated in the database, the lp-write$_i$ field of A_i is updated to t_i, indicating that the provenance of A_i was last updated at t_i.

2.3 The Database

The recorded dependencies are converted into key-value pairs and stored persistently in a centralized *provenance database*. Storing provenance in a database allows us to build provenance trees efficiently as we have don't have to scan the whole database. Since inodes are recycled, each file is assigned a unique *p-node number* when it is created. A p-node number is never recycled. The details of how p-node numbers are assigned and maintained will be explained in Section 3.

Let pn_r and pn_w denote the p-node numbers of A_r and A_w respectively. The dependency $A_r(t_r) \rightarrow A_w(t_w)$ generated by process P on host H is stored in the provenance database as a tuple (pn_w, t_w, pn_r, t_r, H, P), where pn_w is used as the primary key.

Additionally, two secondary indices are maintained to speed up certain kinds of queries. The *process database* is a secondary index on the process name to enable efficient retrieval of files that have been modified by a particular application. When a tuple of the form (pn_w, t_w, pn_r, t_r, H, P) is stored in the provenance database, a tuple of the form (P, pn_w) with P as the key is stored in the process database. The *descendant database* is another secondary index maintained to efficiently retrieve the descendants of a particular file. For a tuple (pn_w, t_w, pn_r, t_r, H, P) in the primary, the tuple (pn_r, t_r, pn_w, t_w) with pn_r as the key is stored in the descendant database.

2.4 Retrieving the Provenance Information

DPASS supports two primary queries on the provenance stored in the database.

- **Retrieving the provenance tree of a file:** This query returns all the files in the system that a particular file X depends on, by tracing the ancestors of a file to its foremost ancestors. Intuitively, this amounts to backtracking the origins of a file.

- **Retrieving the descendant tree of a file:** This query returns all the files in the file system that have a file X as their ancestor. For instance, if a file X is corrupted at time t, the descendant tree is useful to determine all the files that have been derived using this corrupted data.

Provenance tree building algorithm To build the provenance tree of file A with version time t, the query application starts by retrieving all the immediate ancestors of A before time t, i.e. tuples of the form $B(t_1) \rightarrow A(t_2)$, where $t_2 \leq t$. For each chosen ancestor B, the application recursively retrieves all immediate ancestors of B recorded before t_2. The recursion ends when no more provenance records can be found. Clearly, this recursive algorithm retrieves all the ancestors of a file.

Observe that since the mtime of B at time t_2 was t_1, it seems sufficient to query for the provenance of B up to time t_1, and not up to time t_2. We found however, that this was not the case and that the mtime of a file does not always indicate the time of the last write to a file. For example, `tar` while untar-ing a file, sets the mtime of a file to something much earlier than even the file's creation time using `utime`. Hence we need to use the time when the dependency was recorded rather than the mtime of the ancestor for building provenance trees.

Using timestamps also allows us to avoid cycles since at any point, only dependencies recorded before a particular time are retrieved. As the depth of recursion increases the time also decreases, ensuring that the recursion ends.

Descendant tree building algorithm To build the descendant tree of file A with version t, the query application retrieves all all tuples of the form $A(t_1) \rightarrow B(t_2)$ where $t_1 > t$, using the p-node number of A as the key. The application recursively query the database for descendants of B recorded after t_2. The recursion ends when no more descendent's can be found.

2.5 Example

Consider a process P on host H that reads from a file A, and writes to a file B, within a loop. Formally, P reads from A at times t_1, t_3, and t_7, and writes to B at times t_2, t_4, and t_8. Moreover, assume that a process P reads a file C at time t_5, and another process P' on host H' reads a file D and writes to A at time t_6. Finally P' reads from B at time t_9 and writes to A at time t_{10} where $t_1 < t_2 < \ldots < t_{10}$. The following diagram summarizes the above scenario:

$P(H)$	$P'(H')$	time
read(A)		t_1
write(B)		t_2
read(A)		t_3
write(B)		t_4
read(C)	read(D)	t_5
	write(A)	t_6
read(A)		t_7
write(B)		t_8
	read(B)	t_9
	write(A)	t_{10}

The provenance capturing algorithm proceeds as follows:

After time t_1, the active file dependencies of P on host H will contain a tuple (i-node$(A), t_A, t_1,$ null), for some $t_A < t_1$, t_A being the version of A at time t_1. When the write to B happens at t_2, the dependency $A(t_A) \rightarrow B(t_2)$ is recorded following Rule 1. Also, the tuple (i-node(B), null, null, t_2) is added to the active file dependencies of P to indicate that the provenance of B was recorded at t_2.

Observe that after P reads from A for the second time, the tuple in the active file dependencies of P that corresponds to A remains unchanged, since A has not been modified since the previous read. This implies that in

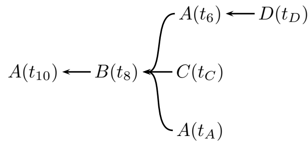

Figure 1: **An example provenance tree**

the second write to B, the provenance of B will not be updated.

When P reads file C at time t_5, a new active file dependency (i-node$(C), t_C, t_5,$ null) corresponding to file C is added to P's active file dependencies, where t_C is the version number (mtime) of C. When P writes to B at t_8, the first_read of C is greater than lpwrite of B ($t_5 > t_4$). Thus the dependency $C(t_C) \rightarrow B(t_8)$ is recorded following Rule 2.1, and the dependency $D(t_D) \rightarrow A(t_6)$ is recorded following Rule 1, where t_D is the mtime of D.

Next, when P reads from A at time t_7, the mtime of A has been changed, and thus the active file dependencies of P is updated to contain the tuple (i-node$(A), t_6, t_1,$ null). When P writes to B at time t_8, the mtime of A is greater than the time of the lpwrite to B (i.e. $t_6 > t_4$), and thus the provenance of B is updated by adding the dependency $A(t_6) \rightarrow B(t_8)$ following Rule 2.2. Finally, when P' reads B and writes to A, the H' records the dependency $B(t_8) \rightarrow A(t_{10})$.

Assuming that the p-node numbers of files A, B, C and D are p_A, p_B, p_C and p_D, the provenance for the dependencies generated above are stored in the database server as shown in the table below:

Dependency	Tuple
$A(t_A) \rightarrow B(t_2)$	$(pn_B, t_2, pn_A, t_A, P, H)$
$C(t_C) \rightarrow B(t_8)$	$(pn_B, t_8, pn_C, t_C, P, H)$
$D(t_D) \rightarrow A(t_6)$	$(pn_A, t_6, pn_D, t_D, P', H')$
$A(t_6) \rightarrow B(t_8)$	$(pn_B, t_8, pn_A, t_6, P, H)$
$B(t_8) \rightarrow A(t_{10})$	$(pn_A, t_{10}, pn_B, t_8, P', H')$

The provenance tree of A at time t_{10} is as shown in Figure 1.

2.6 Extensions to the provenance tracking mechanism

The above high-level description of the provenance tracking mechanism describes only the actions performed during read and write system calls. We now briefly outline the similar actions for other system calls and inter-process communication mechanisms.

Forks When a process P calls fork, the active file dependencies of P are copied to the child process.

Pipes The pipe data structure is extended to store a pointer to provenance information. On a write to a pipe, a pointer to a copy of the active file dependencies of the process is stored in the pipe data structure. On a read from a pipe, the active file dependencies recorded during the write to the pipe are removed and appended to the process reading from the pipe.

Mmaped files Tracking provenance in mmaped files is hard because writes translate to pages being marked dirty and by the time pages are synchronized to disk, the process could be long dead. So we treat an mmap system call as a read/write system to the file and record provenance as we do for normal reads/writes.

Note that, unlike the BackTracker [5], our system *does not* track dependencies between processes explicitly but uses processes only to implicitly capture dependencies between files. We claim that our system can still reconstruct all file-to-file dependencies that the BackTracker can capture, in spite of storing a smaller subset of the information that BackTracker stores. For example, suppose process P reads file A and later writes to process Q through a pipe. When process Q writes to file B, we record a dependency $A \rightarrow B$, since Q has indirectly read data from A (through process P). This dependency is captured as follows:

- When P reads A, the system adds the file A into its active file dependencies.
- When P writes to Q through a pipe, the active file dependencies of P are copied into the active file dependencies of Q.
- When Q writes B, the dependency $A \rightarrow B$ is recorded in the provenance of B.

The $A \rightarrow B$ dependency is recovered without explicitly tracking the $P \rightarrow Q$ or any other dependency between the processes in the system that BackTracker captures.

3 Implementation

In this section, we discuss the architecture of the DPASS system, the databases, the query application, and how we have overcome the challenges in building a DPASS.

3.1 Architecture

The overall Architecture of the system is shown in Figure 2. The system is composed of the following components:

- DPASS client
- BDB RPC Server

3.1.1 DPASS Client

The DPASS client is the component that is present in every host in a distributed file system. It consists of two components:

- DPASS Stacking File System
- A user level daemon called *provd*

DPASS Stacking File System A stackable file system is a file system layer placed between the VFS and a lower level native file system. It intercepts VFS operations enabling us to track the data/meta-data before passing it to the lower level file system. In our case, the lower level file system is NFS. Wrapfs, a wrapper stacking file system generated from FiST [10], was used as the starting point for building a Stacking file system for our needs. The DPASS stacking file system intercepts file system operations, runs the provenance tracking algorithm and updates the active file dependencies of the process. If the provenance tracking algorithm decides that a dependence should be recorded to the database, the DPASS stacking file system sends this information to *provd* via a netlink socket.

provd provd is a user level daemon that collects provenance sent out by the DPASS stacking file system and stores it in the database server. On receiving a record from the DPASS stacking file system, provd lookups the pnode number corresponding to the inode number of the record, and stores the record to the database.

3.1.2 BDB RPC server

The Berkeley DB (BDB) server is an embedded database [1], that provides an RPC interface to the Berkeley DB API. The provd daemon running on each client persistently stores provenance by executing the appropriate BDB API calls. The BDB API calls made by provd are converted to appropriate RPC calls by the BDB library, thus transmitting the data to the BDB server.

3.2 The Databases

The three primary databases are listed below. Note that all the clients operate on the same databases and share the databases. Updates made by provd on one client is accessible by provd on another client. Provenance generated by one provd can be used by another while recording new provenance.

- i-node → p-node map
- provenance database
- p-node → name map

Note that all the clients operate on the same databases and share the databases. Updates made by provd on one client is accessible by provd on another client. Provenance generated by one provd can be used by another while recording new provenance.

Every file is assigned a unique *p-node number* when it is created. The p-node number of a file, as described in Section 2.3, is used as the key to store and retrieve the provenance of a file from the provenance database.

The mapping from i-node numbers to p-node numbers is required to lookup the p-node number of the file since the provenance records that provd on receives from the DPASS stacking file system contain only i-node numbers. provd looks up the p-node number from the database and uses it as key for storing the record. The mapping from p-node numbers to filenames is useful to display provenance information in a more readable format.

The p-node number of a file is always unique. When a file is deleted, the associated provenance data is not deleted and the p-node number is not recycled, unlike the i-node number. To see why this property is necessary for tracking the provenance of a file, consider the following example: A process P_1 reads from a file A_1, and writes to a file A_2. Then, a process P_2 reads from A_2, and writes to a file A_3. Clearly, even if A_2 is deleted, the provenance records of A_2 need to be kept to be able to recover the dependence of A_3 on A_1. Since the p-node number of A_2 is used as a key for all these records, the same p-node number cannot be assigned to a new file.

When a file is created, the provd on the client that created the file allocates a new p-node number for the file and updates the (i-node number → p-node number), and (p-node number → filename) mappings. When a file is unlinked, the provd on the client that unlinked the file removes the (i-node number → p-node number) record. When a file is renamed, the (p-node number → filename) and (filename → p-node number) mappings are updated.

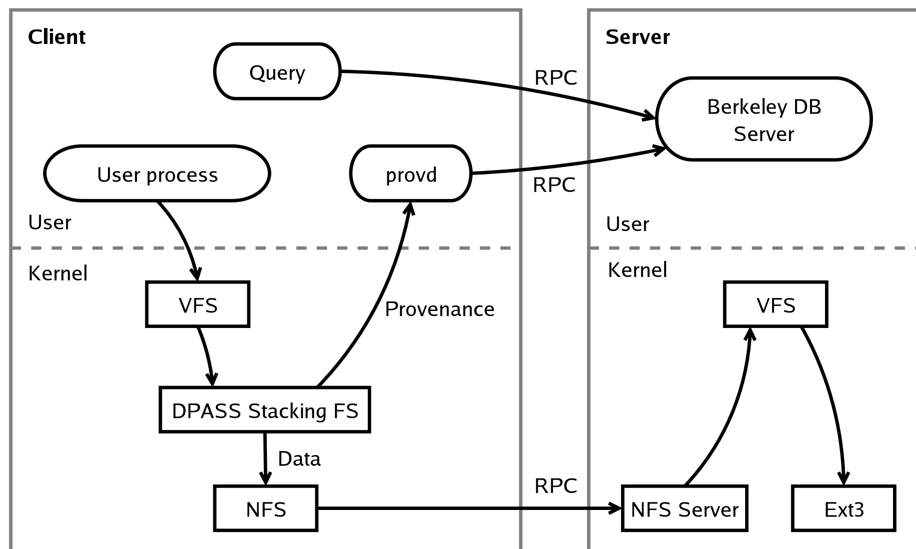

Figure 2: **Distributed PASS Architecture.**

Apart from the primary databases, provd also maintains the secondary indices.

3.3 Querying the Provenance

We built a query application (depicted as part of the client in in Figure 2), that implements the tree building algorithms described in Section 2.4. It interacts directly with the BDB RPC server using the BDB API, and uses the (i-node → p-node) mapping and the provenance database to construct the provenance tree of a given file.

3.4 Overcoming the Challenges

We summarize how we overcame the challenges in building our DPASS below:

- **Automatic Provenance Generation:** The DPASS stacking file system intercepts file system operations and runs the provenance tracking algorithm to generate provenance records, that are eventually stored in the database. Moreover, it does not require designing a new file system, or modifying an existing one.

- **Transporting Provenance:** DPASS stacking file system sends the provenance to provd via a netlink socket, and provd sends it to the BDB RPC server using BDB API calls. Provenance is thus transported over the network without designing a new protocol, or modifying an existing one.

- **Storing minimal required information:** The provenance tracking algorithm described in Section 2.2 ensures that DPASS stores only a minimal required set of dependencies.

- **Querying Provenance Efficiently:** The use of BDB databases together with the timestamps generated by the provenance tracking algorithm (Section 2.2), enables us to easily determine the relevant subset of the provenance records needed to build a provenance tree. A simple log on the other hand requires a sequential scan of the entire log starting from the last record.

4 Evaluation

We evaluated the performance of our system on 2 machines, one was configured to be an NFS server with file system operations being *sync*ed and the other machine was configured run an the NFS client with the DPASS stacking file system. The Berkeley DB RPC server was configured to run on the same machine as the NFS server.

The server is a 3GhZ Pentium 4 Machine with 512MB of RAM and a MAXTOR 6Y080M0 80GB Serial ATA 7200PRM HDD. The server runs Fedora Core 3, with a Linux 2.6.11-1.14_FC3 kernel. The client is a 500Mhz Pentium 3 Machine with 756MB of RAM running running RedHat 7.3, with a Linux 2.4.29 kernel. The Linux kernel on the client has a single line patch to store a pointer to the active dependencies of the files of the process. We set the receive buffer size for sockets to be

16MB at the client.

In all our evaluations, to ensure a cold cache, we unmounted the file systems on which the experiments took place between each run of a test. We recorded elapsed, system, and user times, and 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 system+user times. We ran each experiment at least 4 times. For each of our results, the standard deviation was less than 5%. We do not discuss the user time in the results as DPASS stacking fs is in the kernel and hence the user time remains unaffected.

4.1 Workloads

We ran two benchmarks on our system: a real workload from the Bauer Center for Genomics Research (CGR), Harvard University and a CPU-intensive benchmark.

The first workload, from CGR, takes 2 files and produces 1 result file at the end. Each of the 2 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 series of commands that produce output files that are used as input to the next command. Starting from 2 files and 1 configuration file, 15 more files are produced, with the 1 result file. The scientists at CGR would find DPASS useful to easily "recollect" the input files from which the output was derived, two months after the fact.

The second workload was a build of Am-Utils [6]. We used Am-Utils 6.0.9: it contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Though the Am-Utils compile is CPU intensive, it contains a fair mix of file system operations. This workload demonstrates the performance impact a user sees when using DPASS under a normal workload.

For each workload, we evaluate the performance overhead due to DPASS, the space overhead required to store provenance and the reduction in dependencies recorded due to the improvised provenance tracking algorithm.

4.1.1 Configurations

We used the following configurations on the client machine for evaluation:

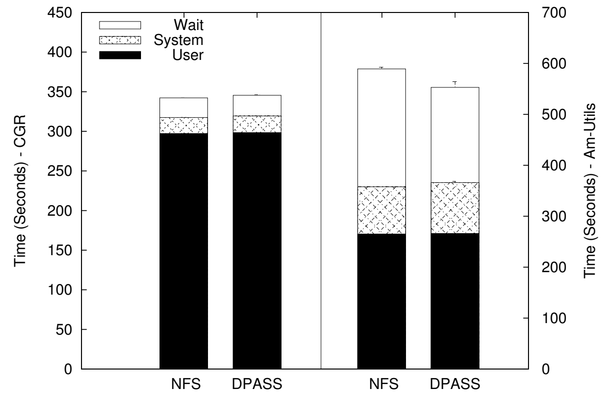

Figure 3: **Overhead for CGR workload and Am-Utils Compile. The first half of the graph is the CGR workload result and uses the left scale. The second half of the graph is the Am-Utils Compile result and uses the right scale.**

- **NFS:** Client Machine running NFS client without the DPASS stacking file system or provd.
- **DPASS:** Client Machine with provenance tracking enabled, i.e., NFS client with DPASS Stacking file system and provd.

4.2 Performance Overhead

4.2.1 CGR Workload

The left half of Figure 3 compares the overhead of DPASS with NFS for the CGR workload. The overhead is negligible (less than 1%). The system and wait time in DPASS increase, but are within the standard deviation and hence the increase can be attributed to noise. At any point, there are at best 3 files open, hence the provenance tracking algorithm will not have any effect on the system time. The amount of provenance generated is also (see Section 4.3) very small, hence the wait time is also unchanged.

Figure 4 shows the provenance tree for the CGR 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 Am-Utils Compile

The right half of Figure 3 compares the overhead of DPASS with NFS for the Am-utils compile benchmark. Overall, there is a 6.2% decrease in the elapsed time for DPASS compared to NFS. The decrease in overhead can be attributed to the 19.2% decrease in the wait time for

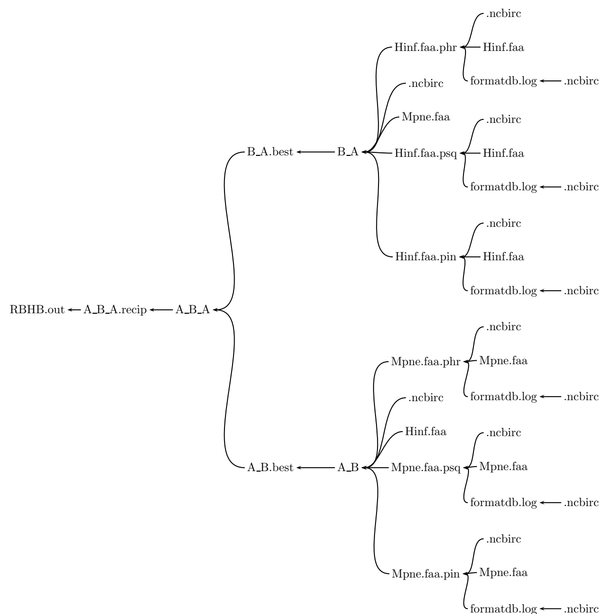

Figure 4: **Provenance tree for CGR workload.**

DPASS. We believe that this decrease in the wait time is mainly due to caching at the DPASS stacking file system layer . The system time for DPASS increases by 7.9%, due to the provenance tracking algorithm running in the DPASS stacking file system. The increase in system time is offset by the decrease in the wait time.

4.3 Reduction in dependencies due to Provenance Tracking Algorithm

Table 1 shows the number of `read` system calls, number of `write` system calls and the number of `mmap` system calls for each of the workload. Table 1 also shows the number of dependencies captured by DPASS. The number of dependencies captured by DPASS is drastically less than systems like backtracker [5] and lineage file system [9] which log every `read` and `write` and later build the dependencies from the log. The last column in Table 1 shows the amount of reduction in the number of dependencies due to the provenance tracking algorithm. The previous section has already shown that the cost of running provenance tracking algorithm, that reduces redundant dependencies, is very low.

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. While logging each call will prove to inefficient, using the provenance tracking algorithm should reduce the storage space required to store provenance and as a result, 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 CGR workload is 0.4% and for Am-utils compile is 3.3%. Clearly, the amount of space occupied is within admissable limits.

In summary, our performance evaluation demonstrates that DPASS has a low performance and space overhead, while also demonstrating that our provenance tracking algorithm is effective in reducing dependencies.

5 Related Work

The Lineage File System [9] is a system that logs each read/write syscall into a SQL database. The user then directly runs SQL queries to retrieve provenance. The disadvantage with this system is that it does not eliminate redundant data.

The Semantic File System (SFS) [4] is another system which uses provenance. The system allows users to access files based on their content. File type specific transducers automatically extract attributes (field-value pairs) from files and insert them to an index on file modification. These attributes are used for query based file retrieval. Queries are in the form of virtual directories. For example, to list all files that export the procedure lookup fault, the user can run `ls /sfs/exports:/lookup_fault`. This lists the files that export *lookup fault*. Although SFS is similar to DPASS in that it creates indices and provides a queriable interface, it is different from DPASS as SFS creates indices and allows for queries on the content of the files rather than the provenance.

Many Grid and workflow management systems like the Metadata Catalog Service (MCS) [8], the replica location service (RLS) [2], Chimera [3], and the provenance aware service oriented architecture (PASOA) [7] provide provenance tracking mechanisms for various applications. However these systems are very domain specific and cannot be used elsewhere.

There has been earlier work on tracking the flow of information in a filesystem to detect intrusions. For example, the BackTracker [5] is a system that logs every read and write and beginning with suspect log record for a file the BackTracker is able to track back and identify the files and processes that affected that file, and also to display chains of events in a dependency graph. Note that the BackTracker is limited to a non-distributed system,

Benchmark	**Number of reads**	**Number of writes**	**Number of mmap reads**	**Dependencies generated by DPASS**	**% Savings**
CGR Workload	251	8,522	18,688	245	99.1%
Am-Utils compile	27,230	70,607	1,040	6,062	93.9%

Table 1: Reduction in dependencies due to Provenance Tracking Algorithm.

Benchmark	**Data Size**	**Number of files**	**Size of Provenance**	**% Overhead**
CGR Workload	5.7MB	18	24KB	0.4%
Am-Utils compile	34.4MB	564	1.1MB	3.3%

Table 2: Space overhead due to provenance.

whereas our system works in a distributed environment. We also take care to avoid redundant dependencies.

6 Conclusions

In this project, we have designed and implemented a Distributed Provenance Aware Storage System that automatically captures and efficiently retrieves the provenance of files in a distributed file system. We have proposed a provenance tracking algorithm that reduces redundant dependencies significantly. Our system also incurs minimal space and processor overheads.

6.1 Future Work

The Berkeley RPC server does not support concurrent operations as it is currently unithreaded. Although this project does not focus on performance, making the Berkeley DB RPC server support concurrent operations will be useful.

Our current implementation does not capture provenance of input files that exist outside the mount point. Designing a system that is capable of capturing provenance from multiple mount points will be useful.

References

[1] Sleepycat Software. `http://www.sleepycat.com`.

[2] A. L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and Scalability of a Replica Location Service. In *Proceedings of the International Symposium on High Performance Distributed Computing Conference (HPDC-13)*, Honolulu, HI, June 2004.

[3] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR*, Asilomar, CA, Jan. 2003.

[4] D. Gifford, P. Jouvelot, M. Sheldon, and J. J. O'Toole. Semantic file systems. In *Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, Oct. 1991.

[5] S. T. King and P. M. Chen. Backtracking Intrusions. In *SOSP*, Bolton Landing, New York, Oct. 2003.

[6] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[7] Provenance aware service oriented architecture. `http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome`.

[8] 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 *Proceedings of SC2003 Conference*, November 2003.

[9] The lineage file system. `http://crypto.stanford.edu/~cao/lineage.html`.

[10] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.