PTFS: A Provenance-Tracking File System

6.033 Design Project I
Colleen Josephson
cjoseph@mit.edu
Shavit (2pm)
3/19/2012

1. Introduction

PTFS is a file system that tracks provenance, so users can trace the lineage of any file or file part. Each file has a hierarchy of *parts* associated with it. For example, a PowerPoint file contains sides, and a slide contains text and image boxes. Every part can be queried to find its parents and children.

Provenance data is stored on-disk in a kernel-level database, and propagated by reference. Only immediate ancestors and children are tracked. The Unix file system API have been modified and extended to properly manage provenance. The signatures of existing API procedures have not been modified, which ensures that the file system is compatible with provenance-unaware applications, such as *cp, mv, rm* or *make*. Provenance-aware applications can use the API to track provenance at a finer granularity. Files imported from outside the machine start out with no provenance information.

The goals of this design are to provide fast searches and minimize overhead, while retaining simplicity.

2. Design

2.1 Description

The system keeps all provenance information on-disk in a simple relational database that operates at the kernel level to prevent cross-domain costs [1]. PTFS is pseudo-versioning: only one version of a file part exists, but there can be multiple versions of its provenance data. Versioning prevents cycles [1], allowing the provenance data to be represented abstractly as a DAG (directed acyclic graph). A garbage collector runs in the background pruning provenance entries that are no longer of use (see Section 2.3).

Provenance data also has an intermediate representation in-memory called a *temp_prov* list that can be finalized to disk. This allows complex operations to be performed without invalidating the database (Section 2.2.2).

2.2 Representation

2.2.1 On-Disk

Each version of a part has a unique *partID.* The main provenance table, pictured in Figure 1, contains detailed provenance data for each *partID.* Each *partID* has a 64-byte freeform text *partType* field. Applications define their own part types. A basic file is indicated by *partType* of null. The *sub-part* field lists all sub-parts associated with a particular part. A basic file has no sub-parts. A user friendly description of a part is stored in the *Description* field. A basic file is described by its full pathname. Applications have their conventions for describing a particular part. For example, a slide on a PowerPoint may be described as *"Slide 1 in awesome.ppt".*

Provenance-aware applications are responsible for maintaining provenance of sub-parts, since it is not reasonable to expect the file system to know the part hierarchy associated with every possible

type of file. This also means that the application is responsible for adopting a versioning policy that will not introduce cycles.

To find out what *partID* is associated with a particular inode, the PTFS also provides a lookup table in the database that maps inode numbers to the latest-versioned *partIDs* (see Fig. 2). Both tables are hashed for rapid lookup.

partID	partType	Date	Description	Comment	Parents	Children	Subparts	Stale	Deleted
1234	null	3/12/12 15:47:22	/home/cjoseph/ index.html	Null	[222]	[82, 7, 4]	[ ]		
222	null	3/10/12 0:12:45	/home/cjoseph/ template/index.html	From www.css.com	[ ]	[1234]	[ ]		

Figure 1: Database schema for provenance table. *partID* is a 64-bit unsigned integer, *partType* is a 64-byte freeform text field, *Date* is 4-byte timestamp, *Description* and *Comment* are an unlimited text fields, *Parents, Children* and *Subparts* are unlimited lists of 64-bit *partIDs*, and the *Stale* and *Deleted* flags are 1 bit each.

Inode Number	partID
1111	1234

Figure 2: Database schema for inode-to-partID lookup table. An inode number is a 32-bit number, and the *partID* is a 64-bit unsigned integer. Only the most recent *partID* associated with an inode is listed.

2.2.2 In-Memory

An intermediate representation of the provenance data is kept in memory, called a *temp_prov* struct:

```
struct temp_prov{
        int provID;
        struct action ** actions;
}
```

The **WRITE_PROV** system call takes in a pointer to a *temp_prov* struct, then parses *actions*** into a single database operation on *partID* (if *partID* does not exist, it is created). The *actions*** array contains a set of *action* structs that indicate an operation to be performed on a particular column of a row:

```
struct actions{
        int type; //set, append or remove
        char * field; //the field to act upon
        int type; //the data type of the field
        void * data;
}
```

If an action is not given for a particular column, then that column is left unmodified.

A list of *temp_prov* structs is kept in memory, one for every *partID* to be modified. Provenance-aware applications maintain their own *temp_prov* data. **WRITE** is responsible for maintaining the *temp_prov* data for provenance-unaware applications.

2.3 Versioning

Versioning makes the DAG much easier to maintain since it prevents cycles [1] and makes the graph easier to maintain. For example, if a user copies file A to file B to file C, and then edits file B, we can keep the *stale* version of B's provenance data to provide a link between A and C, without requiring expensive tree surgery operations. An entry is stale if it does not describe the current version of a file.

The *stale* flag should make it obvious to the user that the link between A and C comes from a prior revision of the file, so the current version of B cannot be guaranteed to bear any similarity whatsoever to A or C. However, information about a past relationship could be very valuable—if a minor edit was performed on B, it would still share a lot of content with A and C.

When a stale entry becomes childless, it is removed from the database and its *partID* is recycled. **DELETE_PROV** checks for stale ancestors each time it is called (see Section 2.4.1).

2.4 API

PTFS implements the following system calls:

- **OPEN(name, flags, mode):** Opens an existing file *name* or creates a new one.
- **READ(fileID, buf, n):** Read data from *fileID,* as implemented in the Unix file system.
- **SET_PAWARE(fd, pointer):** Marks the file as provenance-aware in the kernel's file table, and sets where the kernel can look to find the *temp_prov* data.
- **WRITE(fileID, buf, n):** Writes to the file associated with *fileID* and maintains the *temp_prov* table in memory if *fileID* is not marked as provenance-aware in the kernel's file table. See 2.4.1 for more details.
- **CLOSE(fd):** Close the file descriptor, write *temp_prov* to disk. See 2.4.1 for more details.
- **UNLINK(name):** Removes *name* from directory and the provenance *Description* field, and decrement reference counter. If the number of references is zero, delete the file call **DELETE_PROV**. See 2.4.1 for more details.
- **GET_PARTID(inode_num):** Retrieve the latest-versioned provenance ID associated with *inode_num.*
- **RES_PARTID( ):** Reserves a valid unused provenance ID. A bit vector is used to keep track of free IDs, analogous to the way the UNIX file system tracks free inodes.

- **READ_PROV(partID, buf):** Retrieve the provenance data for *partID* from the database, package into a struct, and write it to *buf*.

- **(Kernel Only) DELETE_PROV(partID):** If *partID* has no children, provenance data (including references from parents and subparts) associated with *partID* is deleted. Otherwise, the deleted flag is turned on for *partID* and all of its subparts, but the entries are not actually deleted until they have no children. See 2.4.1 for more details.

- **(Kernel Only) WRITE_PROV(pointer):** Parses the *temp_prov* structure located at *pointer* and parses it into a database operation Modifies the provenance of *provID's* parents to add *partID* as a child. Returns -1 if there is an error.

2.4.1 Implementation Details

The implementations of **WRITE, CLOSE**, **UNLINK, DELETE_PROV** and **WRITE_PROV** are particularly important for ensuring that PTFS is compatible with provenance-unaware applications such as *cp, rm,* and *mv.*

WRITE:

- Checks the provenance-awareness flag of the file we are writing to. If the application is provenance aware, then the data is simply written to disk.
- Otherwise, calls **RES_PROV**, add a *temp_prov* with the new *partID* for the working file. (This step is only performed once before a file is closed).
- Look at process' open files (minus *stdin, stdout* and *stderr*). For each file:
    - Mark the open file as a parent in the working file's *temp_prov*
    - Copy all *partID*s from open file's the *Subparts* attribute into memory. Use that data to create new *temp_prov* structs marking the original subpart as the parent. Finally, the working file's *temp_prov* needs all the newly created subparts added to the *Subparts* attribute.
- Write the data to disk.

In the provenance-unaware case, since it is possible for a process to read data from a file, then discard it, false-positives can occur. Additionally, because a new *partID* is always requested for the working file, modifying an existing file automatically creates a new provenance version. Unfortunately, we have not discovered a way to mark previous version as the parent of a new version.

CLOSE acquires a lock on the database, then executes **WRITE_PROV** on every element in the list of *temp_prov* structs. Locking the entire database is an expensive but necessary measure to ensure that the on-disk provenance data is never in an invalid state. For example, if a user were to copy file A to file B, and then start copying file A to file C before file B was completed, file A's provenance entry could refer to children that have not yet been added to the database.

UNLINK decrements a reference counter in the file's inode. If the counter reaches zero, the inode is freed for use and **DELETE_PROV** is called on the main file's most recent provenance entry.

DELETE_PROV:

- Checks if *partID* or any of its subparts have children
    - If not, *partID* and all of its subparts are recursively deleted from the database, freeing their *partIDs* for future use.
        - Whenever a part is deleted, its parents are examined as well. If a parent is stale or flagged for deletion and *Children* = [ ], it is deleted.
    - Otherwise, *partID* and all of its subparts are flagged for deletion by turning the *Delete* bit for the provenance entry to 1.

Because provenance on the file level is associated with a particular inode, *mv* has no effect on the provenance other than changing the description while linking and unlinking the inode to pathnames. *rm* simply runs **UNLINK**, which uses **DELETE_PROV** to manage the deletion of provenance data appropriately. A detailed analysis of *cp* is carried out in Section 3.1.3.

Finally, it is worth noting that **WRITE_PROV** return as failed if an *action* struct has field="Children". Modifying the *Children* field is an invalid operation because **WRITE_PROV** is responsible for maintaining children. If **WRITE_PROV** did not manage this, then applications would have to write *temp_prov* structs for the parent *provIDs*. In addition to making things more complicated for the programmer, this policy would also blur abstraction barriers by requiring a process to modify provenance for parts it does not own.

3. Analysis

3.1 Usage

3.1.1 PowerPoint Slide Copying

When the PowerPoint application first opens a .ppt file, it calls **SET_PAWARE**. This disables **WRITE**'s automated creation of *temp_prov* structs, thus allowing the PowerPoint application to maintain its own *temp_prov* structs.

It is suggested that developers provide an application-specific way to look up the provenance of a file. For example, a user could right click on a slide and select "Get Provenance" to pull up a dialog that executes a search.

3.1.2 Compiling Software

A provenance-unaware compiler records every file open during **WRITE** as parents of the resulting binary, which should include all of the source files.

3.1.3 Copying Files

Copying file A to file B can be summarized as: OPEN A, OPEN B, READ block from A, WRITE block to B, CLOSE A, CLOSE B.

Every time **WRITE** is called, it records a process' open file descriptors as parents to the file in *temp_prov*. If a parent file has sub-parts, new subparts are created and added to the working file,

and linked to the parent subparts. This ensures that provenance information about sub-parts is not lost.

When **CLOSE** is called, it makes calls to **WRITE_PROV** to write the *temp_prov* structs to the database. **WRITE_PROV** also automatically maintains the parent-child duality by checking to make sure that each parent of a child contains information about the child in its *Children* field.

3.1.4 Zip Files

Tar performs file concatenation. **WRITE** ensures proper provenance of the resulting output file, since the zip program reads data from all of the files it needs to compress. Since provenance is stored separately from the physical files, the zip application must include provenance information in the zip file itself. One possibility would be recording provenance information in the zip file's header, and then re-instating the provenance data when the files are unzipped. The zip program should also prune out references to file parts not included in the .zip itself.

3.2 Performance

3.2.1 Space

Assuming that each file typically has no more than 5 ancestors, 5 children and 20 subparts, the average provenance entry is:

partID = 8 byte
partname = 64 bytes
children = 5*8 = 40 bytes
parents = 5*8 = 40 bytes
description = ~256 bytes
subparts = 20*8 = 160 bytes

A per-entry estimate is 568 bytes = 0.5kB.

Assuming that each file has, on average, 3 versions of provenance data active in the database, a file system with 1 million files has approximately 150 million parts. The total space taken up by provenance data is 9GB. For a 500GB hard drive, this is a total overhead of 1.8%.

Each partID is 64 bits, yielding 10^{19} possible part names, providing a namespace much larger than the number of parts likely to be present in a modern system.

3.2.2 Time

Reading a single provenance entry from the database is an O(1) operation, since it is essentially a hash table lookup. Writing a single provenance entry to disk is also a constant-time operation. Querying the entire ancestry of a particular *partID*, therefore, is proportional to the number of ancestors in the graph.

Because the each provenance entry tracks both parents and children, looking up which parts depend on a particular file can be done by merely reading the provenance entry from the database, an O(1) operation. Provenance of multiple levels of descendants, just as with ancestors, grows linearly with the number of descendant nodes.

We analyze the performance of continuous file copying using the I/O assumptions in Tables 1 and 2.

Table 1: Latency Assumptions

Operation	Time
1 DRAM Load	0.0001 msec
Random Disk I/O	10 ms

Table 2: Throughput Assumptions

Operation	Rate
DRAM Access	10,000 MB/s
Sequential Disk Access	100 MB/s
Random Disk Access	1 MB/s

We assume that the average user has 100,000 files in their file system, and each file is approximately 2MB. For a standard file system, copying a 2MB file takes 10ms to seek the source file, (2MB)/(100MB/s) = 20ms to sequentially read the data, 10ms to seek the destination, and another 20ms to sequentially write, totaling 60ms per file. This is a file transfer rate of about 16 files per second.

A provenance file system has some additional overhead because of the required provenance maintenance. Using the assumption from Section 3.2.1, we assume that each file has about 20 sub-parts. Each 0.5kB subpart needs to be read from the database once, and written twice. This is quite expensive, since it is random disk access. However, it may be possible to speed things up greatly by keeping parts of the database in memory.

An average file system has 100,000 files, and if the files are 20 parts each (with 3 versions per part), this would yield a database of ~400MB. At this size, it is possible to cache the entire database in memory, occasionally writing to disk for posterity. Thus, each read and write of provenance data would take 0.0001ms of latency plus (0.5kB)/(100,000MB/s) totals 100.61 ns.

Thus, the overhead per file would be 20*3*100.61ns = 0.006ms.

For a system whose provenance is small enough to be stored in memory, a time overhead of 0.01% is incurred. The file transfer rate will still be over 16 files per second.

4. Conclusion

This design stores provenance information persistently, while keeping overhead low because only immediate ancestors are tracked. The use of database software allows for compact storage and rapid indexing. Furthermore, bi-directional references makes finding children fast.

Problems that remain to be solved include eliminating false positives on provenance in provenance-unaware applications, and discovering a way to mark old provenance data as the parent of a new version when editing provenance-unaware files.

Acknowledgments

I want to thank Kyle Miller for suggesting the idea of structs as an intermediate provenance representation, and Kiran Bhattaram for proofreading.

Word Count: 2677 (excludes header text, internal diagram text, acknowledgments and references)

References

[1] M. Seltzer *et al.*, "Provenance Aware Storage Systems," Harvard University Computer Science Technical Report TR-18-05