ProFS: A lightweight provenance file system

Alan Wagner
abw333@mit.edu
Abelson (R05)
22 March 2012

1 Introduction

The Provenance File System (ProFS) is a modified version of the UNIX file system that keeps track of content flow between files. This design is intended to automatically generate provenance information by analyzing read and write patterns. Additionally, it provides an interface for applications to provide their own provenance data. Through this interface, application programmers are able to specify how the files used by their software are divided into parts and manage the provenance data of each part. The design of ProFS allows it to track content flow on applications such as Power Point, make, copy, zip, and tar, though it may require cooperation from application programmers in some cases.

1.1 Design Overview

Provenance information is maintained within special provenance files that are assigned to every regular file in the system. A regular file's provenance file indicates the direct children (file parts that have taken content) and parents (file parts that have contributed content) of each part of the regular file. By following references specified in provenance files, it is possible to construct a provenance graph that can be used to answer queries.

In order to ensure that provenance files are created, maintained, and deleted correctly, ProFS alters the UNIX implementations of the write, create, and delete functions. Additionally, ProFS implements an Application Programming Interface (API) that allows applications to read, write, delete, and search provenance files as well as create, rename, and delete file parts.

1.2 Trade-Offs and Design Decisions

The trade-offs made during the design of ProFS are based upon the assumption that provenance queries will not be made on a regular basis. This assumption is based upon the fact that, historically, provenance file systems have not dominated the market. This implies that users value the features of a traditional file system more than the ability to perform provenance queries. Therefore, a major design goal of ProFS is to support provenance queries while being lightweight in terms of memory and speed overheads, even if it means that provenance queries will not be as efficient as they could be.

Under this assumption, ProFS was not designed using a versioning file system (VFS). A ProFS implemented on a VFS is advantageous due to implementation simplicity, but it suffers from noticeable memory overhead. As will be explained in section 2, however, this ProFS design uses timestamps to maintain considerable simplicity while avoiding the overheads of a VFS.

This ProFS design propagates provenance by reference, not by value. When provenance is propagated by value, redundant provenance information is kept in memory, which results in unnecessary overhead. Furthermore, common operations such as deletions and provenance writes have poor performance because these could cause changes to an arbitrary number of provenance files. When propagating provenance data by reference, on the other hand, memory overhead is reduced because only direct provenance relations are stored. Moreover, deletions, and provenance writes take a much more limited amount of time. A negative consequence of

propagating provenance by reference is that many files might need to get fetched from disk during a provenance query, which could result in suboptimal performance. Due to the assumption that provenance queries will not be a common occurrence, however, this performance blow is acceptable for this ProFS design.

2 Design

ProFS uses *provenance files* to store provenance information that it gathers through an interface to users and applications and by automatically analyzing read and write patterns.

2.1 Provenance Files

In order to store provenance information, the UNIX inode is modified so that it can support the provenance file type. Furthermore, the inode is augmented with a COMPLEMENT_NUMBER field that, on provenance files, indicates the inode number of the corresponding regular file and vice-versa. Provenance files are moved between disk and main memory according to the same rules that apply to regular files.

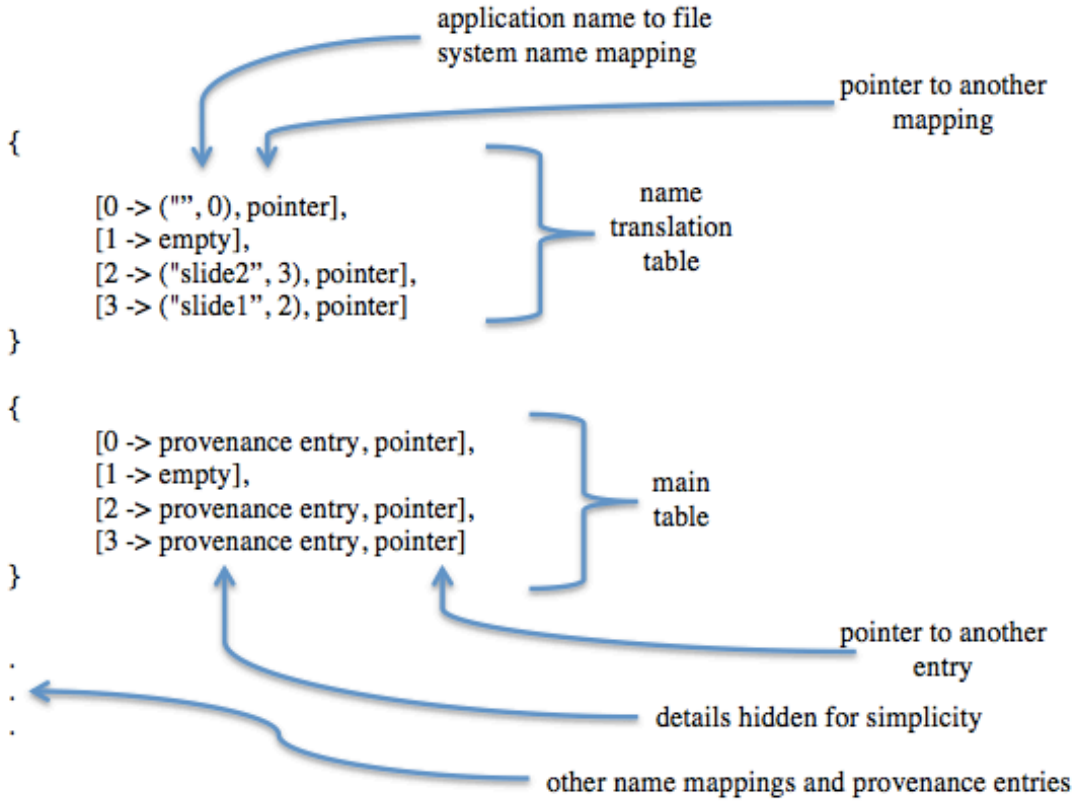

Figure 1 A simplified diagram of a provenance file.

Each provenance file, as shown in Figure 1, contains a hash table that stores *provenance entries*. Collisions are resolved by chaining because provenance entries should not be spontaneously

discarded. Provenance entries each contain provenance information about one part that was specified by applications. Additionally, there is a special entry that contains provenance information that is deduced automatically by ProFS and has no specific part associated with it. Provenance entries have two names: one used by applications and one used by ProFS. The application name is simply a non-empty, free form string and the file system name is a positive integer, with the exception of the special provenance entry, whose names are an empty string and 0.

Provenance files also contain a hash table that is used for name translation. It stores application name, file system name pairs and uses chaining so that no mapping is lost. Since the hash tables that make up the provenance file both use chaining, they can grow without having to move the entire table. Thus, it is possible to place each table at a constant offset from the beginning of the file so that they are easily found. The current ProFS design uses 128 slots for each hash table because most files will have fewer than 128 parts.

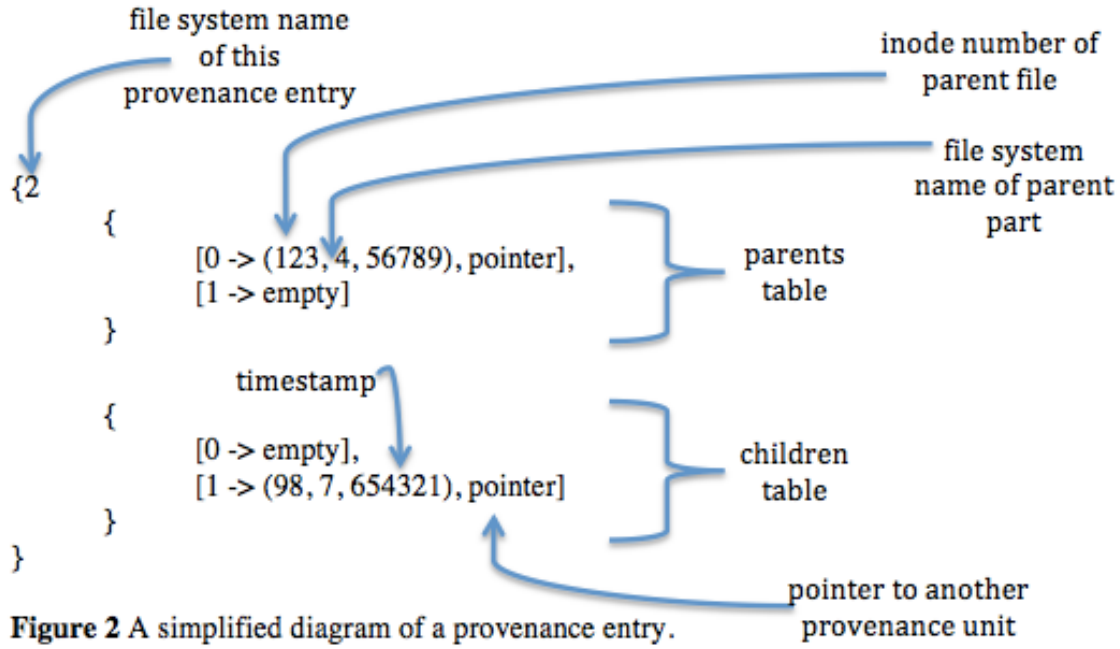

Figure 2 A simplified diagram of a provenance entry.

Each provenance entry, as shown in Figure 2, has two hash tables within it, one of which stores *provenance units* for children, and the other for parents. Since provenance units should not be spontaneously discarded, collisions are resolved by chaining. A unit represents a provenance relation; it contains the inode number of the parent or child file, the internal name of the parent or child part, and the timestamp at which the provenance relation was established. The current ProFS design uses 4 slots for each hash table because most parts will have fewer than 4 parents or children.

In order to correctly maintain provenance files, the UNIX implementations of create and delete are altered. Create must ensure that, whenever a regular file is created, a corresponding provenance file is also created and that the COMPLEMENT_NUMBER fields are populated correctly. Whenever a file is deleted, on the other hand, ProFS must ensure that all references to

the deleted file are either erased or altered to reflect a direct provenance relation that arose due to the deletion. In order to do this, delete obtains a list of all provenance entries using **`read_prov`** and then deletes them all using **`delete_part`**. Then, delete proceeds to actually delete the file and it's provenance file. A detailed explanation of **`delete_part`** and **`read_prov`** can be found in section 2.3.

2.2 Automatically Generated Provenance Information

ProFS automatically generates provenance information so that content flow can be tracked even when applications have not been modified to take advantage of ProFS. This is accomplished by assuming that files that a process has read are the parents of the files that that process is writing.

In order to utilize this assumption, ProFS will use a table in the OS kernel that tracks what files have been read by each process. Furthermore, the UNIX implementation of write must be altered so that it sets the read files as parents of the written file at the current timestamp.

2.3 Application Programming Interface

In addition to the provenance information that is generated automatically by ProFS, applications that are modified to manipulate provenance information can interface with ProFS to control custom provenance information. The API consists of:

- **`read_prov(file_name, part_name)`**
- **`write_prov(file_name, part_name, prov_info)`**
- **`delete_prov(file_name, part_name, prov_info)`**
- **`search_prov(file_name, part_name)`**
- **`create_part(file_name, part_name)`**
- **`rename_part(file_name, from_name, to_name)`**
- **`delete_part(file_name, part_name)`**

`read_prov`, as shown in Figure 3, returns the provenance data of the part specified by **`file_name`** and **`part_name`**. This function uses **`file_name`** to locate the relevant provenance file and **`part_name`** to hash to the correct entry. If no **`part_name`** parameter is provided, **`read_prov`** walks through the provenance file and returns all entries. Note that this function and all subsequent ones convert **`part_name`** to its file system name by using the name translation table before using it to hash to the main table. Additionally, inode numbers are converted to filenames before returning to the application.

```
read_prov(file_name, part_name):
      inode_number = file_name_to_inode_number(file_name)
      inode = inode_number_to_inode(inode_number)
      complement_number = inode.complement_number
      prov_file = inode_number_to_file_name(complement_number)
      linked_list = prov_file.translation_table[hash(part_name)]
      file_system_part_name = None

      for mapping in linked list:
            if mapping[0] == part_name:
                  file_system_part_name = mapping[1]
                  break

      if file_system_part_name == None:
            return "Error: part not found"

      main_linked_list =
      prov_file.main_table[hash(file_system_part_name)]

      prov_entry = None
      for entry in main_linked_list:
            if entry.name == file_system_part_name:
                  prov_entry = entry
                  break

      if prov_entry == None:
            return "Error: entry not found"

      return str(inode_numbers_to_file_names(prov_entry))
```

Figure 3 Pseudocode for **`read_prov`**.

`write_prov` creates a provenance relation between two file parts. First, it uses **`file_name`** and **`part_name`** to reach the relevant provenance entry and utilizes **`prov_info`** and the current timestamp to hash into the appropriate table and write a provenance unit. Next, it uses **`prov_info`** to write the correct unit in the appropriate entry of the other file's provenance file.

`delete_prov` erases the provenance relation between two file parts. It arrives at the relevant units in the same way that **`write_prov`** does and nullifies the information. Note that, in many cases, one of the provenance units will be absent, as is explained in section 3.4.

`search_prov` returns all of the descendants and ancestors of a file part. It finds them by doing a slightly modified depth first search on a directed graph. When searching for ancestors, the vertices of the graph are **`part_name`** from **`file_name`**, its ancestors, the ancestors of the ancestors, and so on, and the edges are all the provenance relations that point from each child to its parents. The search will begin at **`part_name`** and will follow paths as long as the timestamps associated with each edge are decreasing. The same algorithm is applied to find all of the descendants, except that the nodes are **`part_name`**, its children, its children's children, et cetera,

the edges now point from parents to their children, and paths are followed as long timestamps are increasing.

create_part adds a new part to a file. It starts by using **file_name** to locate the provenance file and walking the name translation hash table to find the smallest available file system name. Then, it adds an element to this table that contains **part_name** and the smallest name that was just found. Finally, a blank provenance entry is inserted at appropriate location in the hash table.

rename_part changes the application name of a part. It finds the provenance file using **file_name** and searches for **from_name** in the translation table. Then, it replaces **from_name** with **to_name**. Since no other provenance files depend on the application name, no other changes need to be made.

delete_part deletes the specified part and either erases all provenance references to this part or alters them to reflect a direct provenance relationship that arose due to the deletion. It does so by visiting the provenance entry of the part specified by **file_name** and **part_name** and finds all pairs of child and parent such that the relation with the child got established after the relation with the parent. Then, for all these pairs, the information in the provenance units is used to visit the parent's children and child's parents. Now, all instances of the deleted part are replaced with the parent or child part such as to reflect the direct provenance relation that arose between child and parent due to the deletion. Next, the provenance entries of the children and parents that have not yet been visited are visited and all references to the deleted part are removed. Finally, the provenance entry and the name translation entry of the deleted part are erased.

3 Analysis

The following sections present several use cases and performance metrics to determine how well ProFS performs.

3.1 Power Point Slide Copying

Suppose that a user copies slides from several Power Point presentations into a new presentation. Assuming that the application is modified to support ProFS, Power Point can call **create_part** to designate each slide. It can then call **write_prov** every time a slide is copied so that the flow of content is recorded. Finally, the user would be able to find the ancestors or descendants of any slide by calling **search_prov** on that slide. The application can now present the provenance information to the user, who can then proceed to open any file.

If Power Point had not been modified to support ProFS, the user would have had to make a direct call to the file system to query the provenance. In this case, the results produced by **search_prov** would contain only information that was automatically gathered by ProFS. This means that the new file's ancestors would be all the presentations from which a slide was copied.

3.2 Compiling Software

The make application does not need to be modified to produce optimal results on ProFS. When make is called, ProFS will automatically record the source files as the parents of the resulting binary because all of the sources were read before the last write to the binary. **`search_prov`** can be called on the resulting binary in order to track down the source files.

3.3 Copying Files

As in the use case regarding compiling software, the copy application does not need to be modified, but the resulting provenance information may be surprising. This is because copy reads from one file and writes to another, which results in ProFS listing the read file as the parent of the written file. This implicitly means that all ancestors of the original file are ancestors of the copy, but not that all descendants of the original are descendants of the copy. This happens because the descendants of the original never took content from the copy, so they cannot be considered descendants of the copy.

3.4 Handling Zip and Tar Files

Zip and Tar applications must be modified in order to work correctly with ProFS. When files are compressed, the application must ensure that the accompanying provenance files get compressed as well. Upon decompression, zip and tar must use **`read_prov`** to read the provenance units of the decompressed files. They must find all units that have references to parts that no longer exist call **`delete_prov`** in order to avoid dangling references. Note that it might be difficult to determine if a file still exists due to inode reuse. A clever implementation of zip and tar is necessary to minimize provenance errors on decompression. Lastly, the provenance of a zip or tar file will be defined by the information automatically generated by ProFS. It will thus be a child of all of the files that were compressed into it.

3.5 Continuous Copying

If the user copies file A to B, then B to C, deletes B, copies C to D, deletes C, and continues ad infinitum, ProFS would have no trouble because provenance files get garbage collected when their corresponding file gets deleted. Furthermore, ProFS maintains exactly two references to each provenance relation, one in each of the provenance files of the related files. Since this workload has at most two live provenance relations at a time, memory overhead will be negligible.

3.6 Search Workloads

Suppose that a file A has n levels of ancestors and n levels of descendants and that each ancestor has k parents and each descendant has k children until, of course, the highest and lowest level of the resulting graph. If **`search_prov`** is called on A, then $1 + 2k + 2k^2 + \cdots + 2k^n = \frac{2k^{n+1}-k-1}{k-1}$ provenance files are fetched from memory. If each memory access takes t, the reads take $\frac{(t)(2k^{n+1}-k-1)}{k-1}$. Substituting $t = 10ms, k = 3$, and $n = 5$, **`search_prov`** spends about

7.27s reading from disk. Reading is, by far, the slowest operation in the search, meaning that this relatively large provenance query takes no more than 8s, which is reasonable.

3.7 Memory Constraints

In order to obtain an estimate of the size s of a provenance file, assume that each file has n parts and that each part has k parents and k children. If each name mapping and provenance unit consumes approximately 20 bytes, then $s = (n)(20)(1 + 2k)$. To put this into perspective, consider a 100-page Word file with $k = 3$. The document takes up approximately 750kB and the provenance file takes up about 14kB, which translates to a mere 1.8% decrease in memory space, which reinforces the notion that ProFS is lightweight.

3.8 Effect on Traditional File System Functions

ProFS has a minimal negative impact on the traditional file system functions. Given the speed of modern processors, the bottleneck is clearly disk accesses. The modified create and write functions now access disk twice instead of once, but they will still be able to finish execution in about 20ms, meaning that users probably will not notice. The delete function is more complicated because it has to make k disk accesses, where k is the amount of children and parents that the deleted file has. In an extreme case, $k = 300$, meaning that delete will take around 3s. This will not be a problem because files rarely have this much provenance information, deletes are uncommon, and users will likely not wait for the delete to complete to continue with their other tasks.

4 Conclusion

ProFS is a remarkably lightweight addition to the UNIX file system that was designed to be simple and unobtrusive to traditional file system functions. Provenance files are uncomplicated and produce insignificant memory overhead. A modest pattern recognition system empowers ProFS to automatically produce accurate provenance data and an elegant API allows for effortless and time-efficient maintenance of custom provenance data. What makes ProFS a unique and innovative file system, however, is the fact that it is able to provide the aforementioned provenance services while minimizing its negative impact upon traditional file system functions.

5 References

[1] B. Bitdiddle, “6.033 2012 Design Project 1: A provenance-tracking file system,” [Online document], 2012 Feb 14, [cited 2012 Mar 22], Available HTTP: http://mit.edu/6.033/www/assignments/dp1.html

[2] Gokce, Yasemin (personal communication, March 22, 2012)

[3] Moll, Oscar (personal communication, March 21, 2012)

Word Count: 2766