The Parts-of-file File System

Yoann Padioleau, Olivier Ridoux

To cite this version:

Yoann Padioleau, Olivier Ridoux. The Parts-of-file File System. [Research Report] RR-4783, INRIA. 2003. inria-00071803

HAL Id: inria-00071803
https://hal.inria.fr/inria-00071803

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Parts-of-file File System

Yoann Padioleau — Olivier Ridoux

N° 4783

March 2003

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4783--FR+ENG

INRIA
RENNES

The Parts-of-file File System

Yoann Padioleau , Olivier Ridoux

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n° 4783 — March 2003 — 14 pages

Abstract: We present a new way of managing file contents, its implementation and preliminary experimental results. The goal is to permit simultaneous read/write accesses to different *views* on a file, in order to help in separating a user's concerns, even when they are not independent. Files are considered as mount point directories from which views on the files are accessible as subdirectories and files with read and write permissions. Views are designated with logical formulas describing desired properties of views. They contain those parts of the original file that satisfy the formulas. Properties are attached to parts of the file automatically via programs called *transducers*. A file system interface is used for querying views, navigating between views, and updating view contents.

Key-words: file system, view, update, query, navigation

Un système de gestion de vues de fichiers : *Parts-of-file File System*

Résumé : Nous présentons une façon nouvelle de gérer le contenu des fichiers, sa mise en œuvre et les premiers résultats expérimentaux. L'objectif est de permettre l'accès simultané à différentes *vues* d'un fichier, en lecture et en écriture, et ceci afin de contribuer à une véritable séparation des problèmes d'un utilisateur, même quand ceux-ci ne sont pas disjoints. Les fichiers sont considérés comme des répertoires points de montage, depuis lesquels des vues sur les fichiers sont accessibles comme des sous-répertoires et des fichiers accessibles en lecture et en écriture. Les vues sont spécifiées à l'aide de formules logiques, et elles contiennent les parties du fichier qui satisfont ces formules. Des propriétés sont attachées aux parties de fichiers via des programmes appelés transducteurs (*transducers*). Une interface de système de fichier standard est utilisée pour lister, explorer et mettre à jour les vues de façon cohérente.

Mots-clés : système de fichiers, vue, mise à jour, interrogation, navigation

Contents

1 Introduction **3**

2 Principles **4**
- 2.1 Viewed files, views, and view file . . . 4
- 2.2 A running example 4
- 2.3 Indexing 6
- 2.4 Querying a view 7
- 2.5 Navigating between views 7
- 2.6 Updating 7

3 Algorithms and data structures **7**
- 3.1 View files 8
- 3.2 On-the-fly indexing 8
- 3.3 Synchronizing views 9
- 3.4 Impacts on other mechanisms 9
- 3.5 File operations 10

4 Extensions **10**

5 Experimentation **10**
- 5.1 Implementation 10
- 5.2 Efficiency 11

6 Related work **12**

7 Applications and future directions **13**

8 Conclusion **13**

1 Introduction

Human-composed text files (e.g., program source files, reports, configuration files) have a life cycle that alternates text editing by a user, and processing by an application (e.g., (error checking; compilation)*, or (fault finding; testing)*). The text editing phase requires that the files clearly separate concerns, whereas the processing phase requires that the files are easily loadable. Moreover, separation of concerns cannot be realized by any single file format, because concerns often overlap.

We will take an example from the domain of software engineering, as we believe these questions are very important to this domain. Consider first procedural programming, *à la* C; what source files most clearly separate are operations (i.e., functions). Each operation appears as a separate concern. They are applied to objects that are not clearly separated at all, since every operation applies to several kinds of object. Consider now object-oriented programming, *à la* Java; what source files most clearly separate are (classes of) objects. Now, each class of object appears as a separate concern, but operations (i.e., methods) are not clearly separated. Indeed, methods that implement a given operation are scattered in several places. In the first case, separation of operation concerns makes it easy to add a new operation, but difficult to add a new type of object. In the second case, to add a new type of object is easy, but to add a new operation is cumbersome. In both cases, the file format that is the most convenient to the program processor is imposed to the user at the price that some concerns are not clearly separated. But there are more than two families of concerns in programming. Beside objects and operations, there are non-functional requirements like security, life cycle concerns like versions, etc. No single source file format can separate properly all these concerns.

This is only an example in one domain, but similar examples exist in many domain. Very often, no single file organization can satisfy all desired separations of concerns.

Separation of concerns is also an issue for human-readable files that are not human-composed, e.g., log files, where to isolate a single concern is difficult.

We propose a file management service that permits to store files of an arbitrary format, in such a way that the user can manipulate them through projections of various concerns. This raises issues of *querying* and *navigating* inside a file to specify *views*, of *updating* a file through its views, and of *concurrent* view updates. The file will be called the *viewed file*. The task of the proposed service is to maintain coherence between the *viewed file* and its updatable views, and between the views themselves. In order to offer a generic service, this is realized at the operating system level. So, the generic service is the management of composite operating system objects, their parts, and the coherence of the whole.

The crux of our proposition is to use for these mechanisms the same interface as for *file systems*. In short, a *viewed* file is somewhat like a mount-point of a file system. So, the file system offers the interface for querying views, navigating between views, and updating view contents. This provides a unifying framework for application level tools, allowing to combine them. This file system will be called the *Parts-of-file File System* (PofFS for short).

The plan of this article is as follows. We first present the principles of the *Parts-of-file File System* in Section 2. Then, we expose an implementation scheme in Section 3. We present additional features in Section 4. Section 5 describes experiments and their results. Section 6 presents related works. Finally, we present future search directions in Section 7 and conclude in Section 8.

2 Principles

2.1 Viewed files, views, and view file

A *view* is specified as a property of parts of a *viewed file*: e.g., to be a declaration in a program file, or to concern variable x. A *view* determines a *view file*, which contains exactly all the parts of the *viewed file* that satisfy the property: e.g., all declarations of a program file. We propose a scheme by which properties are attached to parts of a file automatically via programs called *transducers*. So, a view file is determined by comparing the properties attached to parts of file and the property that specifies the view.

The most general property, "to be a part of a file", determines a view file that is simply a copy of the viewed file. Too specific properties, e.g., "to be a declaration and a comment", determine view files where no part of the viewed file is present. This latter kind of view is called *empty*.

Useful properties are somewhere between the most general property, and the over-specific ones. Properties and the generality ordering form a directed acyclic graph (a DAG). Moreover, different properties may specify the same view, e.g., "to be a declaration" and "to be a declaration and not a comment" determine the same view file. We say that "no comment" is not a proper *increment* of "declaration". This notion is formally defined in Section 2.5. Note that "declaration" may be a proper *increment* of "no comment". The property "to concern variable x" is a proper increment of "declaration" only if x is not the only declared thing. Two different increments of the same property may specify the same view. The file system maintains the relations between properties, and can compute on demand all possible increments to any property, that specify different views.

In summary, a *view* on a *viewed file* is specified by a property, it determines a *view file* that contains all parts of the viewed file that satisfy the property, and it determines also a collection of proper increments. In standard file system words, a *view* on a *viewed file* is a directory that is accessible via a path (the property). It contains a *view file* formed of all parts of the viewed file satisfying the property, and subdirectories that can be accessed via the proper increments.

Parts are represented by their coordinates in the viewed file: line numbers if the syntax is line-oriented (e.g., like many configuration files and log files), or stream intervals if the syntax is more structured (e.g., many source files). Coordinates are only used in the tables of PofFS; the user never sees them. Instead, the user designates views using the properties. The properties form the path to the directory where the view file resides.

Navigation among views adapts principles of hierarchical navigation. In the same way as hierarchical navigation starts from the most general directory (e.g., the root, or a homedir), and goes down the hierarchy following subdirectories, navigation among views starts from the most general view, and goes down the DAG following proper increments.

All this involves several mechanisms:

1. The *indexing mechanism* by which transducers decorate parts of file with properties.

2. The *querying mechanism* that compares the properties specifying views, and the properties that decorate parts of file.

3. The *navigating mechanism* by which proper increments of view properties are proposed as subdirectories.

4. The *updating mechanism*, as modifying a view must modify the viewed file and the other views.

The first three mechanisms inherit from mechanisms we have proposed for *a Logic File System* (LISFS [8]). The essence of LISFS it to mix querying and navigation for files decorated with logical properties. The main difference between LISFS and PofFS is that the terminal objects of the file system are files for LISFS, and parts of file for PofFS. So, PofFS can be seen as a Logic File System for file contents.

In the following sections, we will first describe the usage of the *Parts-of-file File System* with a simple example, then we will describe more precisely the different mechanisms of PofFS.

2.2 A running example

A typical shell sequence using PofFS would be:

```
$1> cat -n foo.c
     1  int f(int x) {
     2  int y;
     3  assert(x > 1);
     4  y = x;
     5  fprintf(stderr, "x = %d", x);
     6  return y * 2
     7  }
     8  int f2(int z) {
     9  return z * 4
    10  }
```

Command 1 shows the content of the viewed C file `foo.c`.

```
$2> poffsmount foo.c /poffs
    --transducers=c_transducer,
       /home/pad/my_transducer
    --meta=/tmp/poffs_tmp
```

Command 2 mounts it under `/poffs`, using a general transducer `c_transducer`, and the user defined `my_transducer`, to associate properties to parts of the file. In this example, the parts will be lines. Properties are: to which function belongs this line

line numbers \ properties	function:f	function:f2	var:x	var:y	var:z	debugging	specification
1	×		×				
2	×			×			
3	×		×				×
4	×		×	×			
5	×		×			×	
6	×			×			
7	×						
8		×			×		
9		×			×		
10		×					

Figure 1: A file context

(*function*:*f*), what are the variables involved in this line (*var*:*x*), or does this line have a *debugging* aspect, or a *specification* aspect. We can represent such associations by a matrix *lines*×*properties* which forms the *file context* (see Figure 1 for an illustration). A place (`/tmp/poffs_tmp`) is given to program `poffsmount` in order to store the file context on disk with other meta-data such as view file contents.

```
$3> cd /poffs
$4> ls
debugging/ specification/
function:f/ function:f2/
var:x/ var:y/ var:z/
foo.c
```

Command 4 has two effects. First, it creates a *view file* `foo.c` which contains the parts of file corresponding to this directory. As the directory is the root of the mounted file system (`/poffs` in the example), no properties have been selected yet and the view will have exactly the same contents as the viewed file. Second, it presents navigation increments to the user as subdirectories (as `function:f`, `debugging`, ...). Those subdirectories correspond to properties that actually refine the current view, without making it empty.

```
$5> cd function:f
```

Command 5 refines the view, selecting parts of the view file having the *function*:*f* property. The current directory changes to `/poffs/function:f`, and this path corresponds internally to a *logical query*, in this case *function*:*f*.

```
$6> ls
debugging/ specification/
var:x/ var:y/
foo.c
```

line numbers \ properties	function:f	function:f2	var:x	var:y	var:z	debugging	specification
1	×		×				
2	×			×			
3	×		×				×
4	×		×	×			
5	×		×			×	
6	×			×			
7	×						
8		×			×		
9		×			×		
10		×					

```
> cd function:f
> ls
var:x/  var:y/
debugging/
specification/
```

Figure 2: Navigation in a file context

Again, command 6 shows how a new view file `foo.c` has been created, containing this time less lines that the viewed file, and shows how the property increments are related to the current query. Indeed, under `poffs/function:f`, `var:z` is no longer listed as an increment, as the current view file (which contains only the code of the `f` function) contains no line using variable `z` (see Figure 2 for an illustration). Moreover, `function:f` is also not listed because this subdirectory does not refine the view, but just generates exactly the same view as the current one.

```
$7> cd !(debugging|specification)
```

Command 7 illustrates the possibilities of the query language, combining negation (written `!`) and disjunction (written `|`). The slash can be read as a conjunction, so the path `/poffs/function:f/!(debugging|specification)` corresponds logically to
$function{:}f \wedge \neg(debugging \vee specification)$.

```
$8> ls
var:x/ var:y/
foo.c
$9> cat foo.c
int f(int x) {
int y;
................:1
y = x;
................:2
return y * 2
}
................:3
```

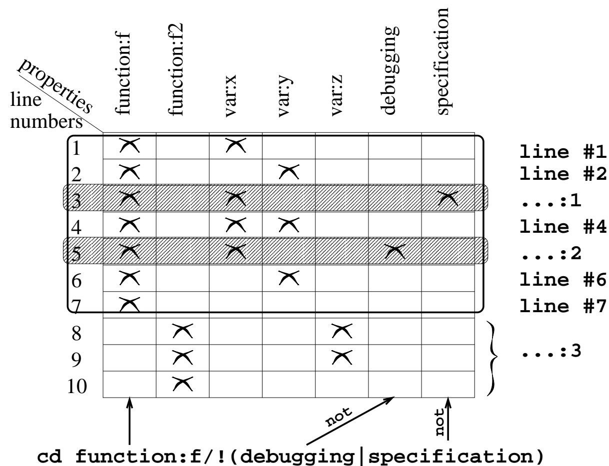

Figure 3: Creation of a view

Command 8 shows that the list of subdirectories has been reduced to `var:x` and `var:y` (see also Figure 3). Command 9 shows the content of the view file. Lines not satisfying the current query have been filtered out and replaced with special *marks*, like `...........:1`

```
$10> cat foo.c | sed -e s/y/z > foo.c
```

Command 10 shows that the views are *updatable views*, and can be modified by any tools. The effect of this command is to replace all occurrences of `y`, *in the current view file*, with `z`.

```
$11> ls
var:x/ var:z/
foo.c
```

Command 11 shows how modifying the view affects the property increment (compare with result of command 8).

```
$12> pwd
/poffs/function:f/!(debugging|specification)/
$13> cd /poffs
$14> cat foo.c
int f(int x) {
int z;
assert(x > 1);
z = x;
fprintf(stderr, "x = %d", x);
return z * 2
}
inf f2(int z) { return z * 4 }
```

Finally, command 14 shows how modifying the view affects the other views, by propagating the modification.

2.3 Indexing

A file transducer for PofFS attaches a property to every part of a file. The definition of a part can be line-oriented; i.e., parts are lines. Alternatively, it can be structure-oriented; i.e., parts correspond to nodes of the abstract syntax tree of the viewed file. For the sake of simplicity, we will only develop the line-oriented case in this article. In fact, operating system issues do not depend on this orientation.

Several transducers can be given to the `poffsmount` command; PofFS will cascade them. This makes the system easily extensible. For example to define a memory management aspect, one just has to write a script such as:

```
for each line
 if line match
        (malloc|calloc|new|delete|free)
 then print "memory_managment\n"
 else print "\n"
```

and pass it to the the `poffsmount` command, without taking into account other transducers.

A transducer is a finite state machine, so that the indexing of a part may depend on previous lines. For example, remembering in what function body a line lies, permits to add the property *function:f* to the line. This shows that even with the line-oriented approach structured data can be analyzed.

Attached properties are writen in a logic language, which is used for querying and navigating. A logic is defined by a language F, i.e., its formulas, and an entailment relation that is usually written $\models$. $f1 \models f2$ means that if $f1$ is true, then $f2$ must be true too. For instance, $a \wedge b \models a$ holds in propositional logic. Let $\mathcal{O}$ be the set of all objects in the file (objects are parts of file)), and $c(o)$ be the content of the object o, transducers implement a function d that associates to each object o a formula $d(o)$ describing the object.

In the current prototype, the description of an object is limited to a conjunction of atomic properties (but queries can be any formulas). So, transducers actually return for each line (or token) a set of properties. This association can be represented as in Figure 1 by a matrix *lines* $\times$ *properties*.

Indexing is costly, so that we have tried to limit it. At some points, a transducer may come back to its initial state. This means that parts before that point do not influence parts beyond that point. These points are recorded as *synchronization points*. If a file is modified at some point, and must be re-indexed, it is enough to re-index the file starting from the last synchronization point before the modified point, and ending at the first synchronization point of the modified file that is also a synchronization point of the old file.

2.4 Querying a view

Views are designated with formulas that serve as paths. Each directory contains a unique *view file*, which contains the set of lines satisfying the path.

In the current prototype, the logic is the logic of proposition. Queries are either simple atoms (e.g., a), negations of formulas (e.g., $\neg a$), disjunctions of formulas (e.g., $a1 \vee a2 \vee a3$), or conjunctions of formulas (e.g., $(a1 \vee a2) \wedge a3 \wedge (\neg a4)$). The entailment relation is that of usual propositional logic. These formulas are written `a`, `!a`, `a1|a2|a3`, and `(a1|a2)&(a3)&(!a4)` in the concrete syntax. The slash is read as a conjunction. Simple atoms can be valued attributes. The query language supports comparison and matching operations on attributes valued by integers or strings: e.g., `cd "function:^f.*"` (i.e., select all functions whose name starts with an 'f'). Descriptions must be conjunctions of atomic formulas, but any formula can serve as a query.

The query mechanism checks if a conjunction of properties satisfies a query. In a directory p, the view file contains $\{c(o) \mid o \in \mathcal{O}, d(o) \models p\}$. E.g., in the context of `foo.c`, line 2 belongs to the view file of directory `function:f` because $function{:}f \wedge var{:}y \models function{:}f$. The $c(o)$ are always displayed in the order of the viewed file.

2.5 Navigating between views

Property increments, which allow to refine views, are proposed to the user as subdirectories. More formally, let $\mathcal{F}$ be the set of all properties, let $ext(p) = \{o \in \mathcal{O} \mid d(o) \models p\}$ (the extension of p), then the set of subdirectories Sub in a directory p is a finite subset of $\mathcal{I} = \{f \in \mathcal{F} \mid \emptyset \subset ext(f \wedge p) \subset ext(p)\}$. The subset is chosen to cover all different extensions $ext(f \wedge p)$. E.g., at step 6 of the running example, `var:x` is a subdirectory of `function:f` because

$$ext(var{:}x \wedge function{:}f)$$
$$= ext(var{:}x) \cap ext(function{:}f) = \{1, 3, 4, 5\}$$
$$\subset ext(function{:}f).$$

Note that

$$ext(f \wedge g) = ext(f) \cap ext(g),$$
$$ext(f \vee g) = ext(f) \cup ext(g),$$

and $ext(\neg f) = complement(ext(f))$.

PofFS provides also mechanisms to group related properties together, in order to reduce the number of answers in `ls`. For instance, directories `function:f/` and `function:f2/` can be grouped together under the directory `function/*`, making the navigation process far more convenient. Directories can also be grouped by the user in taxonomies. As these mechanisms are quite complex and inherited from LISFS, we refer to [8] for more information.

2.6 Updating

As we want to be able to update views, PofFS needs to remember what has been filtered out for back-propagating an update to the viewed file. We also need to make the missing parts explicit in the view file. Indeed, it must be visible if a new line is added before or after a missing part. For instance, assuming a schematic view `a ... b`, where `...` represents the missing part, `a c ... b` can not be distinguished from `a ... c b` if the position of the missing part is not shown. So, PofFS inserts special marks in the view file everywhere parts have been filtered out. In order to not pollute too much the file with marks, only one mark is generated for consecutive missing parts. In order to designate missing parts (e.g., to move them), a unique number is associated to each mark. We will also see in section 4 another use for this number.

So, a view file is composed of a set of lines satisfying the query, and a set of marks, internally referring to lines in the viewed file. Updating a view file involves an update of the viewed file and of the properties of each part of file. The updated viewed file is composed of a new set of lines, which are the lines that are hidden by a mark in the updated view file, and the visible lines in the updated view file. The properties of each line are updated by re-applying the transducers on this updated viewed file. Remember that, thanks to the synchronization points, re-indexing is actually done only on a section of the viewed file.

3 Algorithms and data structures

Each line of the viewed file is represented by an internal object identifier `oi`. The data structure is essentially a matrix `object->properties` stored on disk. It represents the meta-data *object* $\times$ *properties* information. It is often large, but sparse. The query and navigation mechanisms use this matrix; the indexing mechanism fills it in. To make the query and navigation algorithms more efficient, PofFS also stores on disk an inverted table `property->objects`. Transducers initialize the matrix, allocating fresh new objects for each line. There is also a table `object->contents` that associates to each object its content, and a table `line->object` that associates to each line number of the viewed file, its internal object identifier. Indeed, the line order in the viewed file, and the line order in the internal representation need not be the same. E.g., if a line is inserted in a view file, it will be added to the end of its representation. These two tables allow to reconstitute the content of the viewed file. Figure 4 illustrates these data-structures with `foo.c` (see Section 2.2), where we assume that function `f2` is created first, then func-

tion `f`, then the `fprintf` to `stderr`, and finally the `assert`. This scenario explains the numbering of objects in the file context.

3.1 View files

Each directory corresponds to a logical formula f. Using table `object->properties`, the query algorithm computes the set of all objects `ois` that satisfy f. Every time the user reads the content of a directory, e.g., with the `ls` command, a new *view file* is created. A new inode `i` is allocated, but no data is created. Then its data blocks `i.data` and a table `i.marks` (associating to each mark a list of objects) are filled in on demand (e.g., command `open`) according to the following algorithm:

```
mark   = 0
inmark = false
foreach l in line->objects
 let o = line->object[l] in
 if (o is in ois) & not inmark
   then add object->contents[o] to i.data;
 if (o is in ois) & inmark
   then inmark = false;
        add "........:" and mark
        and object->contents[o] to i.data;
 if (o is not in ois) & not inmark
   then mark++; inmark = true;
        i.marks[mark] = o;
 if (o is not in ois) & inmark
   then add o to i.marks[mark]

if inmark
   then add "........:" and mark
            to i.data
```

3.2 On-the-fly indexing

When the user modifies a view file, PofFS updates the content of the viewed file, and also updates the table `object->properties` as the properties may have changed. This work is done when the user commits a view file (e.g., every time the user saves a view file with its text editor).

Recalculating from scratch table `object->properties` is expensive, because it also requires to calculate the inverted table `property->objects`. So, we prefer to "patch" this table according to the modifications. Every time a modification is committed, PofFS will change only a few lines of table `object->properties`, instead of erasing it from disk and creating a new table.

When the user commits (in fact, a `release` operation) an updated view file whose inode is `i`, PofFS first computes the new content of the viewed file, using `i.data` and `i.marks` according to the following algorithm:

The viewed file

inode(foo.c)

data

```
int f(int x) {
int y;
assert(x > 1)
y = x;
fprintf(stderr, "x = %d", x);
return y * 2
}
int f2(int z) {
return z* 4
}
```

The file context

	line->object
l1	o4
l2	o5
l3	o10
l4	o6
l5	o9
l6	o7
l7	o8
l8	o1
l9	o2
l10	o3

	object->contents
o1	int f2(int z) {
o2	return z * 4
o3	}
o4	int f(int x) {
o5	int y;
o6	y = x;
o7	return y * 2
o8	}
o9	fprintf(stderr, "x = %d", x)
o10	assert(x > 1);

	object->properties
o1	#function:f2, #var:z
o2	#function:f2, #var:z
o3	#function:f2
o4	#function:f, #var:x
o5	#function:f, #var:y
o6	#function:f, #var:x, #var:y
o7	#function:f, #var:y
o8	#function:f
o9	#function:f, #var:x, #debugging
o10	#function:f, #var:x, #specification

	property->object
#function:f2	o1, o2, o3
#function:f	o4, o5, o6, o7, o8, o9, o10
#var:x	o4, o6, o9, o10
#var:y	o5, o6, o7
#var:z	o1, o2
#debugging	o9
#specification	o10

A view file

cd function:f/!(debugging|specification)

i

	marks
m1	o3
m2	o5
m3	o8, o9, o10

data

```
int f(int x) {
int y;
................:1
y = x;
................:2
return y * 2
}
................:3
```

Figure 4: Representation of a view

```
new_content = empty content
foreach line l in i.data
 if l contains a mark with number j
 then
  foreach o in i.mark[j]
   add object->contents[o] to new_content
 else
  add l to new_content
```

This yields a new content `new_content`. Then, PofFS builds a new table `new_properties` which associates properties to each line of `new_content`. It builds it by applying the transducers on `new_content` starting from the appropriate synchronization point, and by copying the old properties in `object->properties` for sections of the viewed file that contain no modified part and are delimited by synchronization points. Then, PofFS computes the difference between the old properties and the new ones, modifying the tables according to the following algorithm:

```
done = empty array;
foreach l in line->object do
 done[line->object[l]] = 0
foreach l in new_properties do
  find in object->properties an object o
    such that (object->properties[o]
             == new_properties[l])
            && done[o] == 0
  if an object a is found
   line->object[l] = a
   object->contents[a] = new_content[l]
   done[a] = 1
  else
   allocate a new object o
   object->properties[o]
          = new_properties[l]
   line->object[l] = o
   object->contents[o] = new_content[l]
   foreach p in new_properties[l]
    add o to property->objects[p]

foreach o in done
 if done[o] = 0 then
   free o in object->properties
```

This schema saves calls to the transducers that experiments have shown to be costly.

Note that the algorithm compares the properties and not the contents of the lines, as two lines may have the same content but not the same properties (e.g., the closing brackets in `foo.c`:7 et `foo.c`:10 form identical lines, but one will have the $function$:f property, and the other one $function$:$f2$).

3.3 Synchronizing views

The user can open different view files in different directories simultaneously, which introduces the concurrent view update problem because updating one view file modifies the viewed file, and so may invalidate the contents of the other view files. We choose to delegate resolution of the concurrent view update problem to applications that use PofFS. For instance, `emacs` already checks off concurrent updates. This requires some support from the operating system to indicate to an application that the view file it manipulates is not "fresh" enough. It also requires the assurance that the application will check this indication.

PofFS support to synchronization is to have a timestamp in each directory inode, and to increment a global timestamp every time a view file is successfully committed. Every time the user uses `ls` (in fact, a `readdir` operation), PofFS checks if the timestamp of this directory is equal to the global timestamp. If 'yes', then the view is fresh enough and nothing has to be done, if 'no' then a new view file, with a new inode, is created, and the timestamp of this directory is set to the global timestamp.

Moreover, as some applications may keep a view file open without checking if the view file is fresh enough, we simply forbid to update out-of-date view files. We achieve this by associating to each inode representing a view file a timestamp. When an application commits a view file, PofFS checks if the timestamp of this view file is equal to the global timestamp. If 'yes', then the global timestamp is incremented, and tables are updated as seen in section 3.2; if 'no', an error code is returned.

Basically, this system can be seen as a classic multiple readers and single writer system.

3.4 Impacts on traditional file handling mechanisms

A user can add a line in a view where this line does not belong; e.g., adding a comment in a view file which resides in a directory with the property `!comment`. Saving this view file calls the indexing mechanism (described in Section 3.2), and generates an up-to-date view file in this directory (see Section 3.3). As text editors are not prepared to the fact that the content of a file after a save differs from the content just before the save, they often do not refresh the buffer with the new content. That means that the comment will remain in the view file, and in particular that a second save operation will have for effect to reinsert this line a second time in the viewed file. To avoid this problem, the text editor must be configured so that every save operation refreshes the buffer with the new content. This is possible for `emacs` by defining a macro.

3.5 File operations

We switch now from application level operations to file system operations to enter in more details. We use Linux *VFS* terminology (*Virtual File System* [6]).

Operation `read_super` is called via the user program `poffsmount`. It takes as parameters a file name, and a list of transducers. It creates the tables of the file context, and fills them in by passing the file through the transducers. It also creates the root inode. Operation `put_super` (shell command `umount`) needs not `sync` the file context tables; it can free them. So, all the tables that form the file context can be seen as temporary data. However, as the construction of a file context is costly, one may also keep the internal tables on disk, to reuse them when re-mounting.

Operation `readdir` is called via the user program `ls`. It takes as a parameter an inode that represents a view, and returns a list of pairs `(name,inode)` for every subdirectory of the view, and for the view file. Command `readdir` may build the representation of the view file, though it is preferable to operate lazily, and leave it to file operation `open`. Operation `lookup` can be called via the command `cd keyword`. It takes as a parameter an inode (the current view) and a string, which is the plain name of a file or of a property, and it returns the corresponding inode, or an error condition.

Operations `lseek`, `read`, `write`, and `truncate` are standard. For example, operation `read` takes as a parameter an inode, and a buffer to be filled in. It gets the block addresses of the contents of the file and fills in appropriately the buffer passed in parameter. Operation `open` actually builds the representation of the view file, if it does not exist yet, then it behaves as usual. Operation `release` causes the construction of the new contents of the viewed file, and its re-indexing, which yields a new file context. Operations `mkdir`, `rename`, `create`, and `unlink` are forbidden, because all objects in view directories are there for logical reasons: property increments, and view file. Moving them, or creating new things would lead to incoherence.

VFS includes a cache mechanism for name `lookup`, which makes it possible to avoid to call the concrete operation of a file system. As VFS is used to work with traditional hierarchic file systems, the existing cache handling strategy is not prepared to the fact that modifying the content of a file has "side effects" on the contents of other files. However, this is exactly what happens when an update in a view file causes updates on other view files, and so on the inodes associated to those view files, as described in Section 3.3. So, PofFS invalidates the cache handling mechanism by forcing the VFS to call every time the concrete operation.

4 Extensions

Missing parts in a view file are like hidden. Being able to hide some parts of a file is a useful feature, but a user may want to see what is inside one hidden part. So, "to be a hidden part with number x" is treated as a property. The user is allowed to specify a view by a property *mark*:x provided x is a mark number in the view file of the current view. E.g., in the example of Section 2.2, another shell sequence using PofFS would be:

```
$10> cd mark:1
$11> ls
foo.c
$12> cat foo.c
................:1
fprintf(stderr, "x = %d", x);
................:2
$13> cd mark:1
$14> ls
specification/
var:x/ var:y/
foo.c
$15> cat foo.c
int f(int x) {
int y;
assert(x > 1);
y = x;
................:1
```

We implement this feature by using `i.marks` to compute the set of objects `ois` that belong to the new view file, bypassing the query mechanism. This allows to really navigate in the content of a file, adding to PofFS the advantages of *hypertext* systems. Note that the directory `mark:1` is not special; it specifies a view, in which increments and a view file are computed as usual.

Some tools require that information remains in separate files, e.g., Java compilers force the programmer to map all classes on different files. So, we added to PofFS the ability to mount a group of files. This just requires to insert special marks in the view file to represent the file boundaries.

5 Experimentation

5.1 Implementation

The *Parts-of-file File System* is implemented as an extension to the *Logic File System*. It consists in a user-level file system based on PerlFS. EXT2 is used as an underlying file system to store file contents and meta-data. This implementation style is very convenient for prototyping, but it yields a rather slow file system. The time penalty ratio of PerlFS is about two

	BibTeX	Article	Program
size of specific transducer	23 LoC	29 LoC	61 LoC
size of file	269 Kb	78 Kb	92 Kb
number of lines	8055	1783	2651
mount time (1st/others)	122s /0.216s	22s /0.058s	29s /0.078s
Total number of different attributes	7061	2132	1498
Average number of attributes per line	16	15	14
size of context file	7800Kb	1960Kb	2164Kb
save time	0.956s	0.256s	0.331s
`ls` time	0.253s	0.100s	0.170s

Table 1: Summary of experiments (1)

	BibTeX	Article	Program
Space overhead per line	0.9 Kb	1.0 Kb	0.81 Kb
Space overhead	28×	25×	23×
Space overhead per attribute	1.1 Kb	0.9 Kb	1.4 Kb

Table 2: Summary of experiments (2)

w.r.t. EXT2, for equal functionality. The added functionalities in PofFS/LISFS augment the time penalty. Experiments on a prototype LISFS show that navigation and querying is rather efficient, especially when compared with their application level counter-parts like command `find`. However, we observed that creation commands, like `create`, are slow. We consider this a current state of affair that gives indications on where to refine the implementation. Both LISFS and PofFS are prototypes that are in permanent evolution.

Transducers are not proper parts of either LISFS or PofFS. Instead, both file systems offer hooks for calling user-defined transducers. Our transducers are either ad-hoc scripts or calls to indexing tools like `etags`. In our experiments, we always have used two transducers for each application. The first one performs a full-text indexing, and is used in every application. It consists of a 2 lines long script that extracts one attribute per word in a viewed file. The second one is specific to the application, and will be described with the application.

We ran several experiments to assess the efficiency of PofFS, both in speed and disk space, and in usability. The platform for all experiments was a Linux box running kernel 2.4, with a 2Ghz Pentium 4, 750Mb RAM, and a 40 Gb IDE disk. In the following section, all the experiments are line-oriented.

5.2 Efficiency

A first experiment is to mount PofFS on a BibTeX file. The BibTeX file is about 8000 lines long, and contains more than 900 entries. The specific transducer extracts properties that correspond to the most frequent BibTeX fields: title, author, year, etc. The two transducers produce a total of 7061 different attributes, and an average of 16 attributes per line. Note that attributes of a line of a given BibTeX entry may be replicated on all lines of the same entry to handle the fact that the really useful unit is the entry and not the line. Then, it becomes very easy to navigate in the BibTeX file. Navigation proposes subdirectories in a way that reminds of text-mining: e.g., who are the most frequent co-authors of author A? What were the most prominent years for subject S? PofFS forms a file context of about 8000 objects × 7000 attributes in 122 seconds. This is done at mount time, but since internal tables are persistent, subsequent mounts only cost 0.216 seconds. The size of the file context is 7800 Kbytes. Once the file context is created, one can navigate in it, which is fast, and modify view files. Finding all co-authors of a given author (i.e., `cd author:Jones; ls author:*`) takes 0.25 seconds. Modifying an entry takes about 1 seconds. Tools exist for manipulating and editing BibTeX files, but they do not offer as many possibilities as simply mounting PofFS on a BibTeX file. The only thing that may be missing is a graphical user interface. A file browser or a windowed text editor like `emacs` do a part of the job. This demonstrates the interest of an operating system approach; existing interfaces already works.

Our second experiment is the edition of this article. It is composed in LaTeX as a single file of about 1700 lines. The specific transducer extracts attributes like section, subsection, comment, etc. This produces a total of 2132 attributes, and an average of 15 attributes per line. The file context is built from scratch in 22 seconds, and loaded on subsequent mounts in 0.058 seconds. It occupies 1960 Kbytes. Again, navigation and querying are fast: finding all subsections that talk about synchronization (i.e., `cd contains:synchro.*; ls subsection:*`) costs about 0.1 seconds. Modification takes about 0.25 seconds. LaTeX permits to split a text into several files, but it is rather inconvenient because many operations do not cross file boundaries: e.g., searching, spell-checking, query-replacing. So, we

believe it is better to keep it all in one text file, and cut slices in it at will. Since views are operating system level objects, they can be accessed by different applications that do not know it others, and PofFS will maintain their coherence. For instance, one may `detex` a LaTeX file to strip off its commands, and then pass the result through a spell-checker, but it will not correct the LaTeX file. If instead, it is a transducer that hides the LaTeX commands, the view file can be passed through the spell-checker, and it will correct the viewed file.

The last real size experiment is the LISFS/PofFS program. It is a single Perl program of about 2600 lines. It follows a very elaborate coding discipline, in which several aspects of the final product are identified and interleaved: e.g., debugging and assertions as in Section 2.2, different aspects of the file system like security, and several versions of the same operations. The specific transducer extracts attributes that correspond to these aspects. This produces a total of 1498 attributes, and an average of 14 attributes per line. The file context is built in 29 seconds, reloaded in 0.078 seconds, and it occupies 2164 Kbytes. Modification takes about 0.3 seconds. Navigation is fast: 0.170 seconds for listing functions that use a given variable (i.e., `cd var:transact; ls function:*`). Using navigation and querying, one may select a slice of the source file (i.e., a configuration), and edit it or execute it. One may also consider the different facets of an aspect across the whole program. This makes it easier to make a coherent change in this aspect. A change in any view is back-propagated into the source file. No single tool offer all this services, and no split of the program into several files fits all needs.

Tables 1 and 2 summarize these three experiments. These experiments on real data are encouraging. They show that PofFS can be used in practice, as its response times are compatible with interactive usage. Only mounting from scratch is too slow. This is why it is important to keep the internal tables on disk, and reload them when re-mounting. In the latter case, mounting is fast enough. Note also that the transducer used in the three experiments permits full-text indexing, but costs one attribute per word. If it is disabled, keeping only the specific transducers, mounting is four times faster.

6 Related work

On the application side, modern text editors such as `emacs` or *syntactic editors*, and *Integrated Development Environments* (IDEs) as in `Smalltalk`, ease the manipulation of file contents. They provide query and navigation tools, and often allow to hide some parts of the program, compressing the code under an icon, on which the user can click to expand the content. The contribution of PofFS is to generalize those ideas, allowing powerful query, navigation and updating of arbitrary text data. Furthermore, these ideas are supported in a generic way at the operating system level.

There is an active research area in software engineering for manipulating programs via views. However, proposed tools in this area lack of at least one of PofFS three facets, navigation, querying, and updating, and they often are very specific to a task (e.g., navigating in Java class hierarchies).

Still at the application level, but general purpose, archive tools (e.g., `tar`) provide a relation between (parts of) a file system and a single file. However, the relation is merely duplication, so that updating an archived file does not update the archive. Moreover, their navigation and querying capabilities are limited, and they do not offer an operating system API.

The view update problem is an important issue in data-base management [4]. However, PofFS case is much simpler than for data-bases. Indeed, our view files are sets of actual parts of the viewed file. So, back-propagating updates to the viewed file only requires to know the coordinates of the parts that form the updated view file. In data-bases, views are relations, i.e., sets of items, that are not made of items of the viewed relations. Instead, the items in the views are *computed* from the viewed relations. So, one would have to invert the computation to be able to back-propagate updates. Not all computations can be inverted, so updating views is only possible in restricted cases. The fact that a view update may create items that do not belong to the view (see Section 3.4) is also a data-base issue. In our context, it is solved by re-indexing. Note also that data-bases distinguish *virtual views* from *materialized views* (or *snapshot*). With PofFS, views are just materialized enough so that applications can see them as ordinary files, but they remain synchronized with the viewed file.

On the operating system side, many works on file organization have been made. SFS [2], HAC [3], or Nebula [1], mix querying and navigation in a file system. For instance, SFS has transducers that extract automatically properties from file contents, such as the names of the functions in a C file. This makes it easy to search for a file using a query such as `cd function:foo`. The contribution of PofFS is to go deeper than the file level, allowing to go inside files, no longer selecting sets of files, but sets of parts of files.

7 Applications and future directions

An important research direction is to study the impact of PofFS in larger applications than our experiments. Another one is to make it more efficient.

In the domain of software engineering, being able to manipulate comments, documentation, debugging, specification or platform dependent code or other *aspects* such as security or memory management, to filter them out or to focus on them, provides a great improvement for managing a project. Using PofFS, the programmer can write more commentaries or defensive code without being afraid to pollute the program, as those information can be hidden. The programmer can also keep in source files the naive preliminary version of an optimized function, which makes the understanding of the optimized one easier.

Software engineering has developed rich notions of views which either are extracted from programs (e.g., slicing [10]) or guide the production of programs (e.g., UML, Aspect Oriented Programming [7, 5]). All these views can be designated by properties and serve as an effective way of manipulating programs. For example, UML diagrams could serve as guides for refinement steps that lead to a concrete program. The whole thing would be kept coherent by accessing it only through view files.

Another direction is to add more transducers to PofFS. Many services provided traditionally by `etags`, class browsers, call graphs, `javadoc`, literate programming, versioning, ... can be put in PofFS easily. Moreover, the unified interface for these services, the file system, makes new fruitful combinations possible.

System administration often incurs the management of table files, for users, services, etc. These tables obey precise formats, and sometimes have a system API. One could take advantage of PofFS to offer a secure and uniform handling of these tables. This requires that every type of files comes with its dedicated transducer (as for `emacs` modes). For example, a user could mount PofFS on file `/etc/passwd`, and simply use `cd` and `ls`, or `getdents` to get the information he needs, making the use of function `getpwent` useless. An interesting aspect of this API, is that it has an interactive interface via a shell. So, a programmer can first test his query under a shell, and then put it in his program. Similar techniques could be used for analysing log files.

Our perspectives for improving the efficiency of PofFS are to make the construction of the internal data-structure more lazy, and to use a more direct implementation style than PerlFS, at least for file operations that are similar in PofFS and in usual file systems.

8 Conclusion

Our contribution is to have identified a conflict between applications that handle complex and structured information stored in single files, and the need to manipulate (i.e., navigate, query, and update) more elementary units. The management of file contents in general purpose file systems has not evolved that much since the development of file systems; files are considered as units, and navigation and querying are only defined at the file level. We propose to consider files as mount points to be able to navigate *inside* them. This raises the problem of what to do if a part of a file is updated in such a file system. We have proposed a notion of *updatable views* to solve this problem. It provides a unifying framework of operations like indexing, querying, navigating and updating, and a unifying system level support, under a standard interface, the file system. This is implemented as the *Parts-of-file File System.*

PofFS is easily extensible, and allows to freely combine querying and navigation, which makes it possible to combine services that were kept separate in application level tools. Such a file system gives at a system level services that are useful in many applications. We have used it in real size applications like text editing and programming.

References

[1] C.M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. A File System for Information Management. In *ISMM Int. Conf. Intelligent Information Management Systems*, 1994.

[2] D.K. Gifford, P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr. Semantic file systems. In *13th ACM Symp. on Operating Systems Principles*, pages 16–25. ACM SIGOPS, 1991.

[3] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *3rd ACM Symp. Operating Systems Design and Implementation*, pages 265–278, 1999.

[4] A. Keller. Algorithms for translating view updates into database updates for views involving selections, projections, and joins. In *4t-f ACM Symp. Principles of Database Systems*, pages 154–163, 1985.

[5] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4):154, 1996.

[6] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer*, pages 238–247, 1986.

[7] P.B. Kruchten. The $4+1$ view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[8] Y. Padioleau and O. Ridoux. A logic file system. In *USENIX Annual Technical Conference*, 2003. Long version in [9].

[9] Y. Padioleau and O. Ridoux. A logic file system. Rapport de recherche 4656, Inria, 2003.

[10] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399