A Parts-of-file File System

Yoann Padioleau and Olivier Ridoux
IRISA / University of Rennes 1
Campus universitaire de Beaulieu
35042 RENNES cedex, FRANCE
{padiolea,ridoux}@irisa.fr, http://www.irisa.fr/lande

Abstract

The *Parts-of-file File System* (PofFS) allows read-write accesses to *views* of a given file or set of files. This helps a user separate and manipulate different concerns. The set of files is considered as a mount point, and each view is made of the parts of the mounted files that satisfy a property expressed as a path to the view. In short, a view belongs to a directory specified by a property. Thus, the entailment relation between properties correspond to the nesting of directories. It is possible to open simultaneously several non-disjoint views using several non-exclusive properties. These views share common material from the mounted files, but special care is taken to permit efficient and concurrent view updates. This service is offered generically at the file system level, but application-specific details are handled by plug-in operators. Special plug-ins called *transducers* automatically attach properties to parts. Performances are encouraging; files of more than 100 000 lines are handled efficiently.

1 Introduction

With the advent of UNIX, files are seen as unstructured data stream. Quoting Ritchie and Thompson [17], "*No particular structuring is expected by the system [. . .] the structure of files is controlled by the programs that use them, not by the system.*" This is convenient because the operating system treats input-outputs homogeneously, but it delegates the care for file structures at the application level. However, applications are designed independently, and are not compatible with each others. This is clearly inconvenient, because they share requirements like being able to navigate in a structure and update the structure. Our goal is to handle shared requirements at the system level, and to handle application specific requirements at the application level. Thus, it will be possible to combine structure-oriented applications as easily as the unstructured data-stream model has made the combination of filters easy (see for instance Bentley's accounts of McIlroy spelling checker [1]).

The previous paragraph exposes the operating system point of view; next paragraphs expose the user point of view.

A typical user working on a structured document performs alternatively one of the three following operations: *searching* for a desired piece of data, *understanding* a piece of data and possibly its relationship with other pieces of the document, and *updating* coherently related pieces. Examples of this situation can be found with textual documents, such as source programs or reports, text databases such as BibTeX files, agenda, or more recently, Web pages and XML documents. Hopefully, tools can exploit the structure of these documents in order to help the user.

For searching, tools such as class browsers or hypertext tables of contents, and `grep` or the search button of an editor provide navigation and querying methods. However, these tools suffer an important limitation in that they cannot be combined with each other to make search more efficient. For instance, a text file can be `grep`'ed for a searched string or navigated using a table of contents. However, there is no way to combine the `grep` program and the table of contents program so that one can get the smallest subset of the table of contents that covers the result of a `grep`.

For understanding a document, a commonly accepted practice is to build incomplete but simpler *views* of the document. A popular family of views is obtained by seeing the document at different depths. For instance, a table of contents offers a superficial view, but helps in understanding a document by giving a bird-eye's view on it. At a given depth, many views can also be defined. For instance, showing only the specifications of a program, hiding the debugging code, the comments, or showing only the functions sharing a given variable are possible views on a program file. Each such view helps in understanding one aspect of a program, and also helps in focusing on one task as the user is not visually bothered by irrelevant details. However, usual tools, like say a class browser or tools that support literate programming, provide only a few of these views, whereas all these views are conceptually simple to describe. The user ability to express what he wants to see in a view and what he wants to hide is often

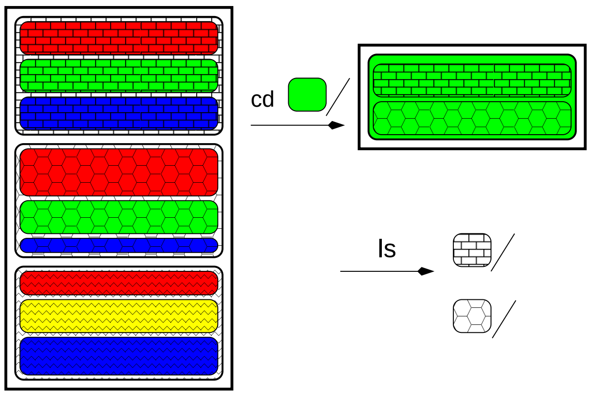


Figure 1: Symbolic representation of a file content

very limited.

For updating a document, views are also helpful. Indeed, an appropriate view can bring together related parts of a document that are distant in the original. For instance, gathering all conclusions of a book may help in updating them coherently. However, tools supporting views seldom support view update because it causes coherence problems between views, and between views and the original document.

The problem with all those tools is that they lack of shared general principles that would make it possible to incorporate new tools, supporting new kinds of navigation, query, views and updates, and that would make it possible to combine them fruitfully.

We can draw a parallel between the management of file contents and the management of file directories. Directories offer one rigid classification of files, in the same way as files offer one rigid organization of data. The possibility to associate several *properties* to a file and then to combine navigation and querying in *virtual directories* has been proposed in the past to help in the management of file directories [6, 16]. We propose in this article to associate several properties to *parts* of a file, and to consider views as *virtual files* built of selected parts to help in the management of file contents.

Figure 1 shows an iconic representation of the problem. The patterns represent the rigid structure of a file, and the grey levels represent different concerns. For instance, patterns correspond to functions in a C program, and grey levels correspond to heading, local declaration, body, or comments. As usual, concerns are scattered across structural boundaries. Hierarchical navigation must respect these boundaries. However, property-based navigation can focus on a concern across boundaries. Note that brick-pattern and dark gray form an example of two properties that overlap. So, updating the brick-pattern view should update the dark-grey view.

Views must be first-class citizens, and they must be updated without restriction. We propose this new service at the file system level, under a standard interface, so that its impact is maximum without rewriting applications. So, several applications or users, with different requirements, can read and write the same file under their own visions. This new file system is called *Parts-of-file File System* (PofFS). Since criteria for defining a view are application specific, we propose that the file system operations only offer the generic background mechanisms, and that plug-ins could be defined to handle the application-level details.

We present the principles of the *Parts-of-file File System* in Section 2. Then, we expose the great lines of an implementation in Section 3, and several other features of PofFS in Section 4. PofFS has been implemented and used as an actual file system. Section 5 describes experiments and their results. Section 6 compares PofFS with other works in the literature. Finally, we present future search directions in Section 7 and conclude in Section 8.

The reader is invited to play with the material presented in this article at the following URL:

`http://www.irisa.fr/LIS/PofFS/`

2 Principles

2.1 Viewed files and views

We propose to navigate in selected parts of a given set of files, the *viewed files*. For the sake of simplicity of the explanation we will restrict ourselves to a unique viewed file (see Section 4 for more details on the handling of multiple files). A *view* is a selection of some parts of the viewed file that share a property; e.g., "to be a declaration", or "to use variable x" are possible properties for parts of a program file, and the listing of every declaration of a program file is a view. Properties are attached automatically to parts of files by special programs called *transducers*. Several properties can be attached to the same part, thus allowing many search criteria and many kinds of views.

In terms of an ordinary file system, directories will be the way to query and navigate; path will be formulas built from properties (e.g., `cd` "uses variable x OR is a comment"), and sub-directories are possible *refinements* to queries. Each directory contains a file, the view, which contain the parts of file that satisfy the properties expressed in the path. These notions are formally defined in Sections 2.4 and 2.5.

All this requires several mechanisms: (1) *indexing*, where transducers attach properties to parts of file, (2) *querying*, where properties that specify views are compared with properties attached to parts of file, (3) *navigation*, which computes refinements, and (4) *updating*, by which a modification in a view is back-propagated to the viewed file. The first three mechanisms are inherited from the *Logic File System* (LISFS [16]), the essence of which is to combine querying and navigation among files decorated with logic properties. The contribution of this article is to adapt LISFS operations to the context of navigation inside files, and to propose a solution for the fourth mechanism, updating. In particular, unrestricted view update is managed efficiently. In the following sections, we describe PofFS through a use case, and then we go deeper in its mechanisms.

2.2 An example

Consider a C program file, `foo.c`.

```
[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  }
[2] % poffsmount foo.c /poffs
```

Command `1` displays the content of `foo.c`, and command `2` mounts `foo.c` on `/poffs`, using transducers for attaching properties to parts of file `foo.c`. Thus, `foo.c` is the *viewed file* for this introductory example. The transducers are not mentioned in the command line because they are selected automatically using a mechanism similar to MIME types. In this example, parts of files are lines. Properties are "this line belongs to the definition of f" (`function`:f), "this line mentions variable x" (`var`:x), "this line contains a trace instruction" (`debugging`), and "this line is an assertion" (`specification`). The attachment of properties to lines can be represented as a $lines \times properties$ matrix which forms the *file context* (see Figure 2). In real-life, the $lines \times properties$ matrix can be very large, e.g., 100 000×10 000, but it is also very sparse, e.g., an average of 10 properties per line. Note that indexing is not local to parts. For instance, line 7 has property `function:f` because a declaration of function `f` has been found 6 lines above. A change at line 1 may affect properties of lines 2 to 7.

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

Command 4 has two effects. First, it creates a *view* which contains all the parts of the file that correspond to this directory. As this directory is the mount point, the view has the same content as the viewed file. Second, it computes possible *refinements* to the current directory, and presents them to the user as sub-directories (`function:f/`, `debugging/`, . . .).

```
[5] % cd function:f
[6] % ls
foo.c
debugging/ specification/ var:x/ var:y/
```

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

Figure 2: A file context

Command `5` *refines* the current view by selecting parts of file that have property `function:f`. Command `6` shows how refinements are related to the current property. In directory `poffs/function:f`, property `var:z/` is no longer a refinement. This can be checked in the current view which contains only the code of function `f`, and contains no line related to variable `z` (see Figure 3). Moreover, `function:f/` is no longer a refinement, since it yields exactly the same view as the current one.

```
[7] % cd !(debugging|specification)
```

Command `7` illustrates the possibilities of the querying language. Negation is written `!`, and disjunction is written `|`. Character *slash* can be read as a conjunction, so that path `/poffs/function:f /!(debugging|specification)` corresponds to the logic formula

$$function{:}f \wedge \neg(debugging \vee specification) \ .$$

More sophisticated logics than propositional logic can be used by plugging *logic solvers* in the file system. For instance, program functions could be indexed by their types, and the types be compared using a type logic implemented as a pluggable module. So, *valued attributes* can be compared and filtered: e.g., `cd "type:?bool"` selects all functions with a boolean parameter. taking in parameter a boolean. Similarly, `cd "function:^f.*"` selects all functions whose name starts with an '`f`'. In this case a logic of strings (regexp) is used. PofFS also offers mechanisms for grouping resembling refinements; this reduces the size of answers to `ls`. For instance, sub-directories `function:f/` and `function:f2/` can be grouped in a directory `function:/`. Properties can also be grouped by the user in taxonomies, thus permitting to focus on a subset of the properties and making navigation easier.

```
[8] % ls
foo.c
var:x/ var:y/
[9] % cat foo.c
```

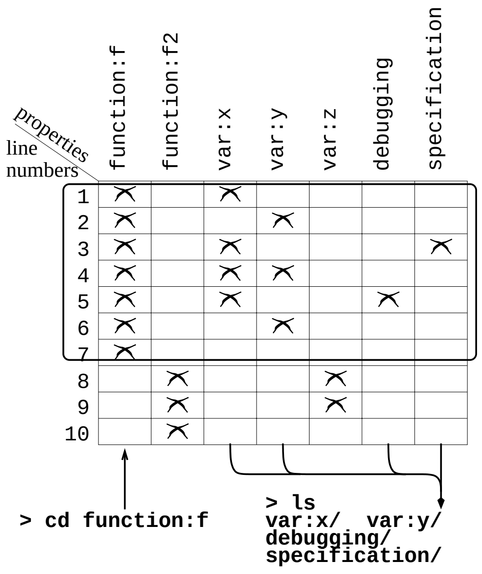


Figure 3: Navigation in a file context

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

Command 8 shows a list of sub-directories reduced to `var:x/` and `var:y/` (check on Figure 4). Command 9 displays the content of the current view. Lines that do not satisfy the current property are replaced by *marks of absence*, e.g., `...........:1`. These marks will make it possible to back-propagate updates to the viewed file.

```
[10] % cat foo.c | sed -e s/y/z > foo.c
```

Command `10` demonstrates that views can be *updated*, and can be so by any kind of tool. The effect of this command is to replace all occurrences of `y` by `z`, *of course only in parts that belong to the current view*.

```
[11] % ls
foo.c
var:x/ var:z/
```

Command `11` shows that updating a view affects property refinements (compare with results 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);
```

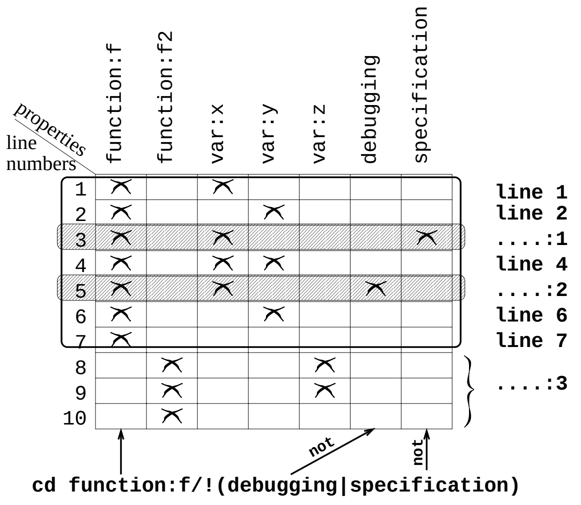


Figure 4: Creation of a view

```
return z * 2
}
inf f2(int z) {
return z * 4
}
```

Finally, command `14` shows that updating a view affects the viewed file and other views. PofFS maintains the coherence of all views and the viewed file by back-propagating view updates to the viewed file, and then to the other views.

This example shows how properties of very different natures are combined in PofFS. It is kept deliberately simple, but the same principles apply to any kind of property. For instance, we have experimented with Java package, using Javadoc to extract types of methods, keywords, name of authors, etc, and logics for comparing types, strings, etc. This yields automatically a powerful environment for retrieving software components and editing them. Properties and logics cooperate fruitfully to offer advanced queries, navigations and views. For instance, a programmer can ask who are the authors who have programmed code using the global variable `foo` with the command `ls use:foo/author:`. The user can combine different query methods, can navigate in more than one classification, and can even use a query method as a navigation one. For instance, the user can either do `cd author:jones` for query, and `ls author:/` for navigation.

The example manipulates a toy program, but the proposed file system works as well for large collections of programs like, say, an operating system kernel. It works also for files that are not intended to be read directly by an end-user like, say, HTML files.

2.3 Indexing

We now switch to a more formal description of the principles of PofFS.

A file transducer for PofFS attaches a property to every part of a viewed file. The definition of what is a part can be *line-oriented* or *structure-oriented*. In the first case, parts are entire lines, while in the latter case, parts are made of a range of tokens that build a structured component, e.g., an expression in a program file. When structured components always fit on sequences of entire lines both orientations are equivalent. We will develop in this article only the case of line-orientation; however, principles of PofFS do not depend on this orientation.

Transducers can be passed manually to program `poffsmount`, or they can be stored in a pre-defined repository. In the latter case, they are automatically selected according to the type of the file. Transducers are called in sequence, and their results accumulated. For instance, if a part receives property a from a transducer, and property b from another one, it will altogether receive property $a \wedge b$. This makes the system easily extensible. For instance, the following transducer defines a "memory management aspect" of C programs:

```
foreach line
  if line matches
          (malloc|calloc|new|delete|free)
    then print "memory_managment\n"
    else print "\n"
```

This can be passed to program `poffsmount`, without bothering for the other transducers.

More formally, a transducer has type:

$$\mathit{list}(\mathit{partcontent}) \rightarrow \mathit{list}(\mathit{set}(\mathit{property})) \; .$$

It takes a list of parts and returns a set of property for each part. The interface convention is that list elements are separated with *new-line* characters, and set elements are separated with a *slash*.

A transducer can be a state machine, so that the property attached to a part may depend on other parts. For instance, every line of the definition of a function f may receive property `function`:f if the transducer remembers the name given to the function in its declaration. The property attached to a part may also depend on subsequent parts. For instance, assuming a property `calls:`f ("this function calls function f"), the property of a function declaration depends on whether there is a call to function f in the subsequent definition. Note that states in transducers allow for handling parts that are made of several lines, e.g., function bodies or BibTeX items. So, even if line-oriented, our scheme is not line-restricted. On the opposite, popular tools like `grep` are line-restricted because they cannot handle patterns that do not fit in one line. For instance, there is no way with `grep` or with a search button in an editor to search for paragraphs that contain two given words, whereas with PofFS a 5-line Perl transducer and a simple query like `cd contains:foo/contains:bar/` do the job.

Building a transducer is a technical task, but the experience of Emacs modes, and more generally of software plug-ins, shows that such a task can be factorized so that end-users never need to write an Emacs mode or a plug-in. Moreover, the example above, and the transducers we already have developed, which form really small programs, show that this is not such a difficult task. They can handle Prolog, O'Caml [14], Perl, C and Java programs, HTML, LaTeX and BibTex files, administration files such as `/etc/passwd`, and also plain text. In some cases, existing tools like Javadoc provide a significant help for programming complex transducers.

2.4 Querying a view

Properties attached to parts are written in the same logic language as is used for querying and navigating. A logic is defined by a formula language $\mathcal{F}$, and by an entailment relation written $\models$. Formula $f_1 \models f_2$ means that if f_1 is true, then f_2 is also true. For instance, $a \wedge b \models a$ is true in propositional logic. Let us call $\mathcal{O}$ the set of all parts of a viewed file f, and $c(o)$ the content of part o; a transducer computes a function d which associates to every part o a formula $d(o)$ that describes the part in the context of f. The description of a part must be a conjunction of atomic properties (though queries can be any formula).

Views are designated by logic formulas that serve as access paths. Every directory contains a single file which contains the view corresponding to the formula denoted by the access path. Formulas can be atoms (e.g., a), negations (e.g., $\neg a$), disjunctions (e.g., $a_1 \vee a_2 \vee a_3$), or conjunctions (e.g., $(a_1 \vee a_2) \wedge a_3 \wedge (\neg a_4)$). The entailment relation is that of propositional logic. These formulas are written `a`, `!a`, `a1|a2|a3`, and `(a1|a2)&(a3)&(!a4)` in the concrete syntax. More sophisticated logics than propositional logic can be used by plugging logic tools in the file system. Atomic formulas can be valued attributes and the user can plug in special logic solvers for handling those values. Several different modules can be plugged in simultaneously, thus producing a rich combination of services.

The querying mechanism checks for every part of file whether its property entails the access path formula. In a directory pwd, the view contains all parts of file whose property entails pwd:

$$\mathit{View}(pwd) = \{c(o) \mid o \in \mathcal{O}, d(o) \models pwd\} \; .$$

For instance, in the context of file `foo.c`, line 2 belongs to the view of directory `function:f` because $\mathit{function}{:}f \wedge \mathit{var}{:}y \models \mathit{function}{:}f$. The $c(o)$'s are presented in the view in the same order as in the viewed file.

2.5 Navigating between views

Property refinements are presented as sub-directories. Formally, let $\mathcal{P}$ be the set of all properties, let $ext(pwd) = \{o \in \mathcal{O} \mid d(o) \models pwd\}$ (the extension of pwd), then the set of all sub-directories, $Dirs$, of directory pwd is

$$Dirs(pwd) = \\ max_{\models}\{p \in \mathcal{P} \mid \emptyset \subset ext(p \wedge pwd) \subset ext(pwd)\} \quad .$$

For instance, at command `6` of Section 2.2, `var:x` is a sub-directory of `function:f` because $ext(var{:}x \wedge function{:}f)$ is equal to $ext(var{:}x) \cap ext(function{:}f)$ which is equal to $\{1, 3, 4, 5\}$ which is a subset of $ext(function{:}f)$.

PofFS also offers mechanisms for grouping resembling refinements; this reduces the size of answers to `ls`. Properties can also be grouped by the user in taxonomies. So, the role of $max_{\models}$ is to limit refinements to most general formulas. This makes answer shorter and navigation more progressive. This is important because the range of properties used in applications is actually very large, but not all properties are relevant at a given time. So, PofFS offers way to hide these other properties. These mechanisms are inherited from LISFS [16].

2.6 Updating

In order to back-propagate view updates to the viewed file, PofFS must combine the parts that are present in the view (modified or not) and the parts that are missing. The result must be foreseeable, and coherent with the intent of the user. To do so, missing parts must be made explicit in the view. Assume for instance a schematic viewed file `abc` (where `a`, `b` and `c` are groups of lines), and a view that hides `b`. This view is displayed as `a....c`, where `....` is a mark of absence. If the missing parts were not marked (view displayed as `ac`), inserting an `x` between `a` and `c` could be interpreted as inserting an `x` either before or after the missing part, i.e., either `axbc` or `abxc`. Similarly, changing `a` into `a'` without knowing where the missing parts are could be interpreted as forming `a'bc`, `ba'c`, and even `a'c`. Thus, it would be impossible to infer the intended effect of a view update on the viewed file. To solve this problem, PofFS inserts marks of absence in views everywhere parts are missing. A single mark is inserted for several consecutive missing parts in order to not pollute too much the view. A unique number is associated to every mark of absence. This makes it possible to delete or move missing parts in a view, and back-propagate the result to the viewed file.

Figure 5 illustrates the ambiguity of inserting a new piece in a file where missing parts are not marked.

In summary, a view is composed of all parts of a file that satisfy a query. Views can be updated and view updates are back-propagated to the viewed file. The updated viewed file is composed of all parts missing in the view and of all visible parts (possibly updated) of the view. Then transducers

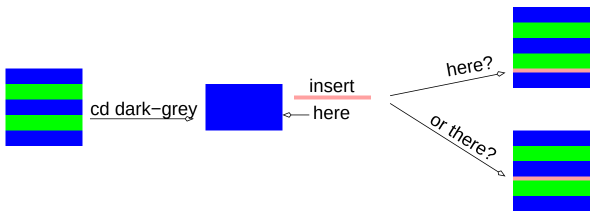


Figure 5: Why do we need marks of absence ?

are applied to the viewed file to re-index it, because properties of some parts may have changed. Consider, for instance, the renaming of a function in a declaration; this changes the `function` property for every line of the function definition.

This suggests that updating causes a complete re-indexing of the viewed file, but we will see that this can be avoided.

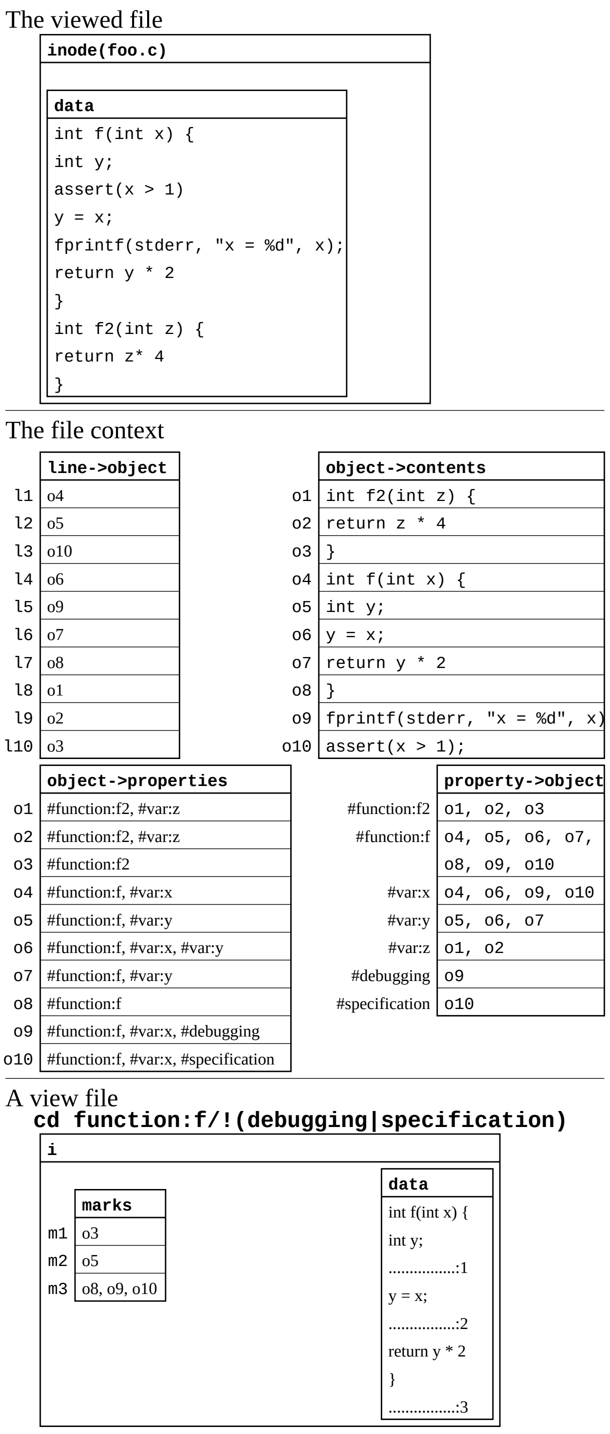
3 Algorithms and data-structures

Every part of a viewed file is represented by an internal object identifier `oi`. Basically, the main data-structure on disk is the *objects* $\times$ *properties* matrix. It is often very large, but sparse Typically, the number of “crosses” in every line is a constant, a few dozens, so that the total number of crosses is linear with the number of parts, instead of being linear with the product of the number of parts by the number of properties. This makes it amenable to an efficient representation.

Querying and navigation mechanisms use the matrix, and the indexing mechanism fills it in. In order to make these mechanisms more efficient, PofFS stores an `object->properties` table and the inverse table `property->objects`. Transducers fill in the matrix by allocating a new internal object for every part of file. PofFS also manages a table `obj->contents` which associates to every internal object its content, and a table `line->obj` which associates to every part designated by its coordinates its internal object. Remind that in this article, parts are always lines. Note that the ordering of parts in the viewed file is not necessarily respected in the internal representation; if a part is inserted in a view, it will be appended at the end of the internal representation, in order to not relocate the existing parts. The last two tables are used to put back together the contents of a viewed file, especially when it has been updated *via* a view. Figure 6 illustrates these data-structures for file `foo.c` (see Section 2.2), in which we assume that function `f2` was created first, then function `f`, then the `fprintf` for tracing, and finally the `assert` instruction.

3.1 Views

Every directory corresponds to a logic formula f. The querying algorithm uses table `property->objects` for computing efficiently the set of internal objects that satisfy f (the


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 6: Representation of a view

extension of f). Every first time the content of a directory is read, e.g., with command `ls`, a new view is built. A new *inode* `i` is allocated, but no data is created. Then, data blocks `i.data`, and a table `i.marks`, which associates to every mark of absence a set of internal objects, are filled in on demand (e.g., when opening the view) using the following algorithm:

Algorithm: `compute_view`

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

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

3.2 On-the-fly indexing

When a user updates a view, PofFS updates the viewed file and the $objects \times properties$ matrix because properties may have changed. This means recomputing table `object->properties` and also the inverse table `property->objects`; this is costly if done naively.

Observation 1: *a user often changes only a few parts between two commits.*

So, PofFS needs not recompute the tables entirely, but only *patches* them according to the update. Every time an update is committed to the disk, PofFS will only change a few lines of these tables. These parts are detected using a `diff`-like algorithm [15]. However, one must not consider only modified parts. This is because properties of other parts may depend on these modified parts. For instance, changing the name of a function at some line also changes the properties of all the subsequent lines that contain the function definition. So, even knowing that the changes are few, one should still apply the transducers to the whole viewed file. We propose a scheme by which transducers give more information so that PofFS can avoid executing transducers on entire files.

Observation 2: *a file is often made of relatively independent units whose properties do not depend on each other; e.g., functions in a C file or a BibTeX entry.*

In principle, a transducer must go through all these units. When a transducer enters a new independent unit, it reset-

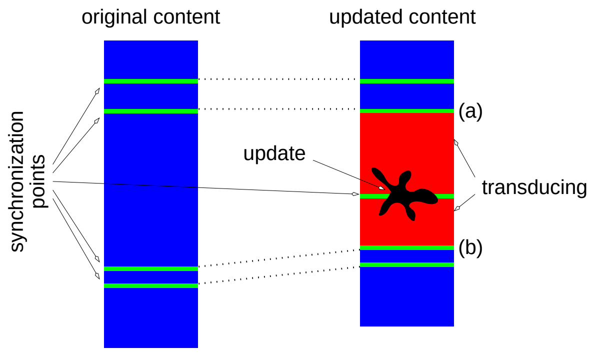


Figure 7: Re-indexing between synchronization points

s its state to an initial state, because the new unit does not depend on the former one. This means that the contents of parts before this point do not influence the indexing of parts after this point. These points play a crucial role in keeping indexing costs to a minimum; we call them *synchronization points*. Note that synchronization points have nothing to do with marks of absence, because boundaries of missing parts need not fall exactly on synchronization points. A new table, `synchro`, contains all synchronization points of a viewed file. If a file is updated, and must be re-indexed, it is enough to re-index it between the last synchronization point before the updated parts (marked (a) in Figure 7), and the first synchronization point after the updated parts that is also a synchronization point in the viewed file (marked (b) in Figure 7).

So, the actual specification of a transducer is as follows. It is a state machine, with an initial state. It associates to every part of file properties that may depend on previous and subsequent parts. Every part where a transducer comes back to its initial state is called a synchronization point; it has property `synchro` which is recognized by PofFS. A correct transducer must not make a part depend on other parts beyond a synchronization point.

Requirement: *the fact that a part is a synchronization point must not depend on other parts.*

This requirement is very strong, but it allows for the efficient algorithms we present later. Moreover, it ensures that effects of an update are as in watertight compartments; like in submarines, where a breach in a compartment will not destroy the whole ship. In our case, an unmatched open bracket in a function declaration does not cause the costly re-indexing of the other functions.

We now present the actual algorithms for reindexing. When a user commits an update in a view with inode `i` (operation `release` of the file system), PofFS first computes an image of the new content of the viewed file (let us call it `new_content`) using `i.data` and `i.marks` as follows:

Algorithm: `new_content`

```
new_content(i) =
  new_content = empty_string;
  foreach line l in i.data
    if l contains a mark with number j
     then
      foreach o in i.mark[j]
       add obj->contents[o] to new_content;
     else add l to new_content;
```

Then PofFS computes the difference between the old and new contents. Then, it applies transducers to segments of `new_content` delimited by the synchronization points that surround the updated parts, and it updates all the tables. To compute the difference, PofFS uses the output of the `diff` tool [8]. Different actions are performed according to the result of comparing lines of the two contents. In the following algorithm, these actions are represented by a function `f` that is passed to `diff` and called after every comparison of two lines. The function takes as parameters the line numbers (`newi` and `oldi` in the algorithm), and the result of comparing the corresponding content lines (either `Match`, `Old_not_in_New` or `New_not_in_Old`). The algorithm is as follows:

Algorithm: `reindex`

```
reindex(new_content, old_content) =
  newline->obj = empty_array;
  newsynchro   = empty_set;
  must_index  = empty_list;
  to_del     = empty_list;
  is_in_clean = true;
  may_index     = empty_list;

  let f(newi, oldi, comparison) =
   if comparison = Match then
    if oldi is in synchro then
     is_in_clean = true;
     may_index = empty;
     add newi to newsynchro;
    if is_in_clean then
     newline->obj[newi] = line->obj[oldi];
     add newi to may_index;
    else
     add oldi to to_del;
     add newi to must_index;
   else   // not Match
    add may_index to must_index;
    is_in_clean = false;
    may_index = empty_list;
    if comparison = New_not_in_Old
     then add newi to must_index;
    if comparison = Old_not_in_New
     then add oldi to to_del;
  in diff(new_content, old_content, f);

  let (stdin, stdout) =
    pipe with transducer;

  foreach i in must_index
    write to stdin new_contents[i];
  foreach i in must_index
    let l = read_line from stdout in
    let props = parse l in
    let o = free_object() in
    foreach p in props
     add o in property->object[p];
    object->properties[o] = props;
    newline->obj[i] = o;
    obj->contents[o] = l;
    if 'synchro' is in props
      then add i to newsynchro;

  foreach i in to_del
   let o = line->obj[i] in
   foreach p in object->properties[o]
     del o from property->object[p]
   free object->properties[o]
   free obj->contents[o]

  line->obj = newline->obj
  synchro = newsynchro
```

In order to be more efficient, the actual algorithm is slightly more complicated. It does not apply `diff` to the whole file contents, but only to the old and new contents of the current view[1]. So doing, the cost of an update is roughly proportional with what the user sees.

3.3 Synchronizing views

A user, or several users, may open simultaneously several views on the same viewed file; this introduces a *consistent view update problem*. Updating a view causes the update of the viewed file, and finally the update of other views that see the updated parts. We have chosen to delegate the solution of this problem to applications that use PofFS. This is coherent with the current usage for file systems. For instance, a text editor like Emacs already checks the coherence of different views (aka. *buffers*) on the same file. This requires that the file system indicates when a view is no longer *fresh*. This requires also that applications check this indication. So, PofFS stores a *time-stamp* in every directory inode, and increments a global time-stamp every time a view is committed to disk. Every time a user consults a directory (command `ls`, or operation `readdir`), PofFS checks that the time-stamp of the corresponding directory is not older than the global time-stamp. If it is not older, the view of this directory is fresh, otherwise a new view is built from the viewed file, with a new inode whose time-stamp is a copy of the global time-stamp. Applications may not read the freshness indication, and ignore that a new view has been built. PofFS manages these applications by simply forbidding updating from non-fresh views. We achieve this by associating to each inode representing a view a time-stamp. When an application commits a view, PofFS checks if the time-stamp of this view is equal to the global time-stamp. If 'yes', then the global time-stamp is incremented, and tables are updated as seen in section 3.2; if 'no', an error code is returned.

[1]A difficulty of this variant of the re-indexing algorithm is that a mark of absence may hide a synchronization point, so that transducers must also go through hidden parts.

3.4 Impact on traditional file handling mechanisms

The design of PofFS has several unusual consequences that a user must be aware of. Users may insert a line in a view to which it does not belong; e.g., to insert a comment in the view of directory `!comment`. Committing the updated view will cause a re-indexing of the viewed file, and produce an up-to-date version of the view, without the new line (i.e., the comment line in our example) because it does not belong to the view! So, an application must not ignore the fact that committing a view may change its visible contents, so that it must re-read the view. If it does ignore this, inconsistency will follow. The requirement that applications refresh their buffers after a commit is not excessive. For instance, the Emacs text editor can be configured to reload its buffers after a commit.

3.5 File system operations

We now switch the description of PofFS operations from the application level to the file system level. To this end we redefine the operations of the *VFS* interface of Linux (*Virtual File System* [12]).

Operation `read_super` is called by program `poffsmount`. Its parameters are a file name, and a list of transducers. It creates the tables of the file context, and fills them in using the transducers in sequence. It also create the *root* inode.

Operation `put_super` (called by program `umount`) may discard the tables of the file context. However, it is sometime better to save them in order to reuse them when remounting the same file.

Operation `readdir` is called by program `ls` or, more directly, by the system call `getdents`. Its parameter is an inode of a directory. It returns a list of pairs `(name,inode)` for every sub-directory, and for the view. Sub-directories are computed according to the definition of $Dirs$ (see Section 2.5). Operation `readdir` could build the content of the view, but it is preferable to delay until the content is actually used, e.g., until operation `open`. Indeed, a user may infer from the names of the sub-directories that he should navigate at least one step further, without actually reading the view.

Operation `lookup` is called every time a program interprets an access path. Its parameters are an inode of a directory, and a string, which is intended to be the name of a file or of a sub-directory. It returns either the inode of the file (which is always the view of the directory) or sub-directory, or an error code if the string is an unknown name.

Operations `lseek`, `read`, `write`, and `truncate` behave as usual. For instance, the parameter of operation `read` is a file inode, and the address of a buffer to fill in with file contents.

Operation `open` computes the content of a view, if it does not exist yet. It calls the `compute_view` algorithm. The view is stored as a temporary file of a standard file system. Then `open` works as usual.

Operation `release` commits a view on disk. It calls the `reindex` algorithm. Updates are then back-propagated to the viewed file.

Operations `mkdir`, `rename`, `create`, and `unlink` are disabled, because every object that is visible in a PofFS directory is there for the consistency of the design.

4 Extensions

Several other features make a complete PofFS system; namely, to navigate in hidden parts, and to navigate inside several files at a time.

To build a view in which non-desired parts of file are hidden is useful, but sometimes one wants to discover what is hidden. To this end, "to be a missing part with number x" is considered as a property; a user may specify a view using property `mark`:x, provided that x is actually the number of a missing part. So, in the example of Section 2.2, another scenario could be:

```
[10] cd mark:3
[11] ls
foo.c var:z/
[12] cat foo.c
................:1
int f2(int z) {
return z * 4
}
```

This allows one to navigate in a file contents as in an hypertext document. A directory `mark:`x is nothing special; it contains refinements and a view computed as usual.

Some applications require that information resides in distinct files. For instance, this is the case for Java environments where different classes must belong to different files. So, we have added to PofFS the possibility to mount a group of files. There are two options here: (1) to have as many views as there are viewed files, (2) to concatenate all the files, and use the single file scheme. The first mode favors the discovery of the roles of the different files; e.g., who defines what, who uses what? In this mode, command `ls` list refinements as usual, and non-empty views of the viewed files. Second mode favors the global manipulation of a set of files along concerns that cross file boundaries. In this mode, refinements are as for the single file scheme, and there is also a single view, but it contains all parts of the concatenation file that match the query. A new kind of mark is needed, for separating file views. This feature goes as far as possible in the direction of virtualizing files since it abolishes file boundaries. In Emacs-style words, it permits to navigate in several files using a single buffer. This gathers information that were scattered in several files. For instance, it permits to get in a single buffer the declaration and all calls to some function, and thus makes it easier to modify consistently the function signature.

5 Experiments

The *Parts-of-file File System* consists in a user-level program to which file system requests are redirected by PerlFS [3]. EXT3 is used as the underlying file system for storing views and meta-data. This style of implementation is very convenient for prototyping, but it introduces new costs: mirroring system level operations to a process in user space using a pipe. The specific services of PofFS/LISFS still augment the overhead. Nevertheless, experiments with LISFS show that navigation and querying are rather efficient, especially considering the costs of related tools like `find`.

The system is mainly written in O'Caml [14]. This is a functional programming language that is strongly typed and can be compiled either to byte-code or to native code. It is unusual to program system level components in such a language (C is much more usual), but we contend that this is a good choice for PofFS. Indeed, PofFS operations implement complex algorithms working on data-structures whose life-length is uncertain. O'Caml ensures a good level of security because it has a strong type-checker and an automated garbage collector. This approach is similar to other approaches where a safer programming language is used for system programming (Cyclone and Vault [9, 5]).

Figure 8 shows the architecture of the implementation of PofFS. PofFS is an assembly of several components living in kernel-level and user-level. The core of PofFS is programmed in user-level, and is interfaced with VFS *via* the PerlFS system. PofFS communicates directly with user-defined programs called plug-ins. These programs permit to specialize PofFS for the user context. There are two kinds of plug-ins: transducers and logic solvers. Transducers are used for extracting properties from viewed files. Logic solvers permit to reason on those properties. In order to be recognized by PofFS, plug-ins must be copied in a pre-defined directory of PofFS. The selection of the appropriate transducers is done after the suffix of the filename of the viewed file (a.k.a its extension). Plug-ins can be programmed in any programming language provided the program interface conforms to a simple protocol that we have defined. Finally, user applications can be anything that uses files. Applications such as shells, editors, browsers, compilers or players, have been programmed for hierarchical file systems but they adapt very well to this new paradigm, and can even offer new services for free (e.g., compiling a slice of a program). Really new applications that use PofFS as a repository and that actually requires PofFS services can also be defined (e.g., experiments on a programming environment, a geographic information system, and a time-tabling system have already been

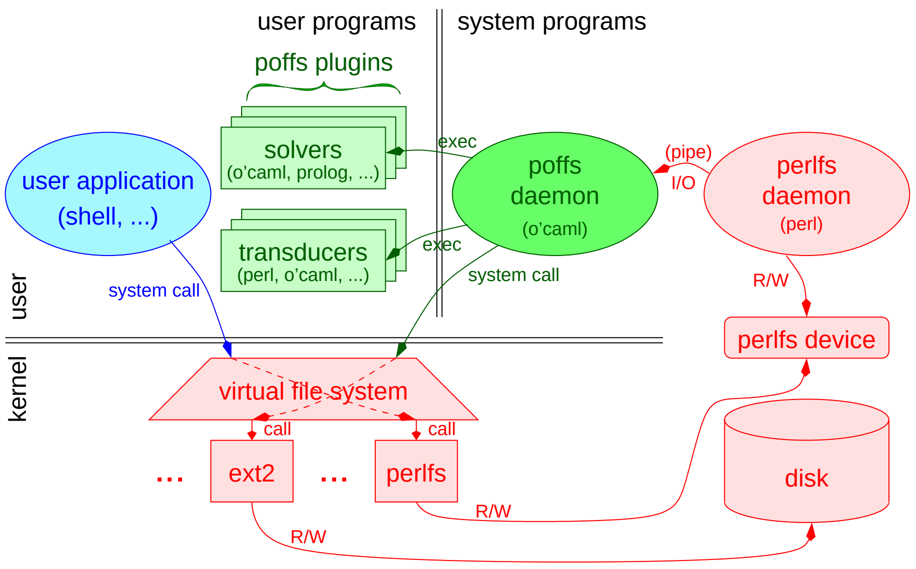


Figure 8: PofFS software architecture

done). In those applications PofFS is a key element of the software architecture.

Transducers do not belong properly to PofFS; they are user-defined *plug-ins* of the file system. However, they play an important role for the efficiency of the overall system; every transducer sees every part of file several times during the life-length of a document. In the experiments presented in this section, we always use 2 transducers per application. The first one performs a full-text indexing of the viewed file, and it is used by every application. It extracts an attribute for every word in the viewed file. The second transducer is specific to each experiment, and is described with the experiments.

All experiments described in this article have been done on a Linux 2.4, Pentium 4 at 2 GHz, with 750 Mbytes RAM, and 40 Gbytes IDE hard-disk.

First experiment is to mount PofFS on BibTeX files. The specific transducer extracts properties that correspond to BibTeX fields: title, authors, year, domain, etc. The specific and full-text indexing transducers yield an average of 9 attributes per line. Attributes are duplicated on every lines of each BibTeX entry to handle the fact that the actual notion of a part is the entry and not the line. Navigating in a BibTeX file yields a vision of the file that resembles what can be obtained by *data-mining*; e.g., most frequent co-authors of author A, most important dates for subject S. Tools exist for manipulating BibTeX files, but they do not offer as much possibilities as simply mounting PofFS on a BibTeX file. The only thing that is missing is a graphical user interface. A *file browser* or a text editor may be used instead. This shows the interest of a system level solution; existing applications run unmodified on PofFS, but offer more services.

Follows a use-case scenario for BibTeX files.

```
[1] % poffsmount ref.bib /poffs
```

```
[2] % cd /poffs; ls
title:/ type:/ ref:/
author:/ year:/ title:/
institution:/ contains:/
domain:/
ref.bib
```

PofFS is mounted on BibTeX file `ref.bib`. Command `ls` yields the most general refinements and the current view, `ref.bib`. At this stage, refinements form a catalog of available properties.

```
[3] % cd author:/; ls
author:A.Aho/ author:K.Thompson/
author:O.Ridoux/ author:S.Ferre/
...
year:/ title:/ type:/ ref:/ ...
ref.bib
```

A selection by authors is chosen. The result is a list of authors, plus a list of refinements that have not been used yet, but are still relevant.

```
[4] % cd author:O.Ridoux/year:>1990/; ls
author:S.Ferre/ author:Y.Padioleau/
author:P.Brisset/ author:Y.Bekkers/
...
year:2003/ year:2002/ year:2000/
year:1998/ year:1996/
...
title:/ type:/ ref:/ ...
ref.bib
```

An author is chosen, and the range of the year attribute have been restricted. The result is a list of relevant authors (i.e., co-authors), relevant dates (i.e., dates of publication), and other relevant refinements. Answering the query `year:>1990` needs to use a user-defined logic solver for reasoning on ordered integers. This plug-in and several others (e.g., a logic of program types, a logic of strings) are released with the PofFS software.

```
[5] % ls domain:/
domain:logic/ domain:compilation/
domain:langage/ domain:formal-methods/
domain:filesystem/
...
```

A selection by domain is chosen. It shows the domains of the selected author.

```
[6] % cat author:Y.Padioleau/ref.bib
.......................:1
@inproceedings{lfs-article,
author = "Y. Padioleau and O. Ridoux",
title = "A Logic File System",
year = 2003,
booktitle = "USENIX Annual Technical Conf.",
keywords = "filesystem, logic",
}
.......................:2
```

Until this point the view has never been actually computed because it has never been opened. Command `[6]` selects a second author and actually opens the view for the first time. So, the view is actually computed, but at this stage it is very small. Like any other view, this view can also be modified and the modification will be back-propagated to the entire viewed file.

Our second experiment is the edition of this article. It consists in a LaTeX file of about 1800 lines. The specific transducer for this experiment extracts attributes such as section names and paragraph type. The specific and full-text indexing transducers yield a total of 2038 attributes, for an average of 8 attributes per line. PofFS forms the file context in a few seconds. LaTeX allows one to split a text into several files, but it is often inconvenient because several useful tools do not cross well file boundaries: e.g., search tools, and spell-checkers. Using the multi-file extension of PofFS solves this problem. As views are standard system objects (files) they can be used by different applications that do not know each others. For instance, a bilingual text can have its one language view spell-checked by an appropriate tool, and the other language view spell-checked by another tool.

Last experiment is the management of programs. We have considered PofFS source files, and a collection of C files (the Andrew benchmark and the core kernel of Linux, without drivers, and for Intel x86 only). The PofFS source files include the PofFS core files (O'Caml), interfaces to PerlFS (Perl), and various transducers and logic solvers programmed in different languages (Perl, Prolog, O'Caml). They obey a specific coding discipline that permits to identify and interleave different aspects of the prototype: e.g., trace and assertion as in Section 2.2, security, optimizations, and versioning.

	# files	Length	Size	LoC	Prop	Prop/ line
small BibTeX	1	8184	280	33	7015	9
large BibTeX	1	100001	3716	33	37077	10
Andrew	76	12841	560	38	2259	3
core kernel	214	127912	3620	38	26467	4
PofFS article	1	1856	88	114	2038	8
PofFS source	69	13451	972	174	8003	6

Length: Number of lines; **Size**: Size of viewed file (Kb); **LoC**: Size of specific transducer (line of code); **Prop**: Number of different properties; **Prop/line**: Average number of properties per line.

Table 1: Summary of experiment contexts

The specific transducer extracts attributes that correspond to the different aspects. This yields a total of about 8000 attributes, with an average of 6 attributes per line. Using navigation and querying, one may select slices of the source program (e.g., a configuration), and edit it or execute it. It is also possible to select all parts that are impacted by a given aspect. It makes it easier to update coherently one aspect, even if it is programmed in different files and different programming languages. No single tool offers all these services, and no single splitting of a program into several source files offers them either.

Table 1 summarizes the characteristics of the experiment contexts. Table 2 displays results of several experiments. For both kinds of contexts, BibTeX files and source files, a large context and a small one are considered.

'Ls' times are measured in several different situations. For instance, very unfocused queries (e.g., `ls author:` for BibTeX files) yield many answers (more than 4000 in the large BibTeX case). In another situation, more focused queries (e.g., `ls year:1998/author:/`) yield much less answers (less than 40, in this case). Table 2 displays the average performance of a mix of all these situations in all kinds of context.

'Upd' times are also measured in several different situations. First situation is when the committed view is not modified. Note that this does not mean that PofFS has nothing to do; it is up to the update algorithm to find out that committed view is not modified (see Section 3.2). For instance, `diff` must be called anyway. Second situation is when the committed view differs locally. For instance, for BibTeX files only one entry is modified. Last situation is when a global modification is committed. For instance, when replacing type `book` by type `article` for all the entries of a view in BibTeX files. Table 2 displays the average performance of a mix of all these situations in all kinds of context.

Experiments show that mount time remains high. Note that the part of transducers in mount time is important; e.g., suppressing the full-text indexing transducer divides mount times by 3 to 10. Note that re-mounting from a saved file context only costs a few milliseconds; so, saving file contexts is an

	FileCont	Mount	Upd	Ls
small BibTeX	5	9.7	0.3	0.1
large BibTeX	35	328.2	2.8	0.9
Andrew	3	12.3	0.2	0.1
core kernel	30	213.1	1.1	1.9
PofFS article	1	2.8	0.3	0.1
PofFS source	6	30.0	1.3	0.1

FileCont: Size of file context (Mb); **Mount**: Mount time (sec); **Upd**: Average re-indexing time (sec). **Ls**: Average `ls` time (sec);

Table 2: Summary of experiments performances

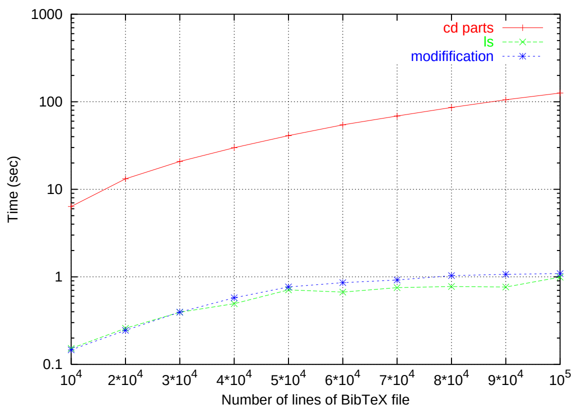


Figure 9: number of objects/time correlation (sec)

option worth considering.

The impact of our sophisticated re-indexing algorithm (see Section 3.2) can be measured by comparing mount times (complete indexing) and update times (partial re-indexing). Partial re-indexing is often 10 to 100 times faster than complete indexing.

Figure 9 shows how performance is correlated with the number of objects in the viewed file; execution time augments linearly with the number of objects.

We believe that these results are encouraging. PofFS handles large size documents in a reasonable time. We are actually using it on a daily basis for working on texts and programs.

6 Related works

The service of manipulating file contents is already offered by several kinds of application: e.g., text editors like Emacs, CIA [4], and IDEs (*Integrated Development Environments*) like Eclipse. These applications offer means for querying and navigation, and they also allow to hide parts of file. The novelty of PofFS is to isolate the generic background of these ideas, and to implement them at the operating system level. So doing, they are accessible to every existing application, and they permit combining logics and transducers in order to offer a powerful environment. In particular, it is easy to handle several different programming languages in the same context.

There is a domain of software engineering [13, 11] that studies the manipulation of programs *via* views, for separating concerns, but proposals in that domain lack of generality. For instance, navigating in a hierarchy of Java classes is an issue for this domain, but it is specific to a programming language, and to a particular organization of programs.

The view-update problem is very important in the database domain [10], where it cannot be solved in all generality. It can be solved in PofFS because this instance is simpler than for data-bases. Indeed, views of PofFS are sets of parts of the viewed file. To back-propagate updates of a view only requires to know the coordinates of the parts in the viewed file. With data-bases, views are not made of parts of the data-base, but they are computed. To back-propagate an update requires to inverse a computation. However, not all data-base operations are invertible, so that back-propagating view-updates is only possible in restricted cases. With data-bases, one distinguishes *virtual views* and *materialized views*. The former are always coherent because they are always recomputed, and the latter are computed only once but may become incoherent. With PofFS, views are materialized but remain coherent with the viewed file.

At the operating system level, there have been many proposals for alternative file systems. SFS [6], HAC [7], or Nebula [2], combine querying and navigation in a file system. For instance, SFS uses transducers for extracting properties like the name of functions in a C file. Then, it is easy to search for a file using a query such as `cd function:foo`. However, navigation is poor (SFS yields only an extension) and it does not go inside files. The first point is solved with refinements, the second with views which select parts of file, instead of files. Moreover, every view can be updated.

7 Future direction

A future work is to extend PofFS to handle parts that are not exactly lines. This is useful for highly structured files like programs and for files that are not made of lines, like sound and video files. For instance, the MPEG7 standard describes how meta-data can be attached to segments of a video stream. Assuming a segmentation tool that we could use as a transducer, one could navigate in a movie by expressing simple queries such as `cd scene:kung-fu ; ls character:/` to get only characters appearing in kung-fu scenes, and `mplayer character:neo/matrix.wmv` to play selected scenes of the movie.

Another important research direction is to study the impact of PofFS on more complex applications. For instance, software engineering has defined sophisticated notions of views, which either are extracted from programs (e.g., program slicing [18]), or guides the production of programs (e.g., UM-

L, Aspect Oriented Programming [13, 11]). All these views could serve as properties for designating parts of program.

8 Conclusion

The management of file contents at the system level has not evolved much since the first systems with stream files; files are units, querying and navigating work at the file level. Only read and write go inside files. We have proposed to consider files as possible mount-points, to navigate in them, to extract views, and to update them. This raises the consistent view update problem between the views and the mounted file. We have proposed a mechanism of updatable views that solves this problem efficiently. The *Parts-of-file File System* (PofFS) implements these ideas at the file system level. It makes the structure of files virtual, and less tightly related to the physical model of a stream of characters.

PofFS combines at the system level services that were often offered separately in user-level applications. We have used it in applications like text edition and programming, and also in trace and log analysis. In all cases, performances are encouraging. The most critical operation in PofFS is updating a view. It requires a rather sophisticated algorithm for performing this operation efficiently.

In the past, considering files as flat streams have been a fruitful abstraction to permit the combination of tools *via* pipes, redirection, etc. What PofFS does is to recover structure in files and still permit the fruitful combination of tools.

A prototype PofFS and more information on this project can be down-loaded at the following URL:

`http://www.irisa.fr/LIS/PofFS/`

This page also displays all demos and experiments presented in this article.

References

[1] J. L. Bentley. Programming Pearls: A Spelling Checker. *Communications of the ACM*, 28(5):456–462, May 1985.

[2] 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.

[3] Claudio Calvelli. PerlFS, 2001. `http://perlfs.sourceforge.net/`.

[4] Y.-F. Chen, M. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[5] Robert DeLine and Manuel Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 20–22, 2001.

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

[7] 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.

[8] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey, June 1976.

[9] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *General Track: USENIX Annual Technical Conference*, pages 275–288, June 2002.

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

[11] G. Kiczales. Aspect-Oriented Programming. *ACM Computing Surveys*, 28(4):154, 1996.

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

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

[14] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.08, 2004. `http://caml.inria.fr/ocaml/htmlman/index.html`.

[15] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–256, 1986.

[16] Y. Padioleau and O. Ridoux. A Logic File System. In *USENIX Annual Technical Conference*, 2003.

[17] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

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