Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 12, Number 1, pp. 79–92. http://www.scpe.org © 2011 SCPE

FLEXIBLE ORGANIZATION IN THE *ORCFS* RELATIONAL FILE SYSTEM FOR EFFICIENT FILE SEARCHING

ADRIAN COLEŞA[*], ALEXANDRA COLDEA[†] AND IOSIF IGNAT[‡]

Abstract. The need for efficient organization of files grows with the computer storage capabilities. However, a classical hierarchical file system offers little help in this matter, excepting maybe the case of links and shortcuts. OrcFS proposes a solution to this problem. By redefining several file system concepts, it allows the user to set custom metadata, in the form of "$(property, value)$" pairs, to express relationships between files. Using it, the system automatically creates a classified view of its components, in which those having similar characteristics are grouped together. A virtual hierarchy is generated, which provides multiple access paths to the same file. This assures that the data can be classified in a more flexible way and the navigation can be done more intuitively. The traditional concept of directory is extended, to accommodate the user-defined properties. In OrcFS, both classical navigation and query interrogation are possible. The enhanced system is compliant with the Linux's VFS interface, thus no changes need to be made to existing applications and they may be used in OrcFS. A prototype of the project was implemented in user-space using the FUSE library to reimplement system calls. The performed tests proved that even if introducing new data in the OrcFS implies some overhead, this is negligible compared with the gain obtained when searching for files in an immense file tree structure.

Key words: file-system; file relationships; metadata; fast retrieval; compatibility

1. Introduction. In the last years notable enhancements have been obtained in the field of data storage. The most modern computers are a collection of impressive numbers. The disk capacity increased up to the order of terabytes, the data can now be stored more efficiently and retrieved faster. However, these improvements at a low-level bring along with them the necessity of a higher-level data organization revolution.

The classical operating system offers only basic help for data organization at user level. Moreover, there is almost no flexibility in expressing relationships between files.

A relational file system′s objective is to find a manner to overcome the fore mentioned limitations. Due to the extensive research in this domain several directions have been outlined. All projects associate semantic information with files [11] in order to describe their contents and interrelationships. Some solutions, like [15, 12, 3] store this metadata in databases, while others [10] use the already existent i-node and directory structures. While in most implementations a file is presented to the user as an atomic element, there are some [14, 12, 3] which split this concept and work with different parts of the file. The latter extract from file contents different pieces of information, interpret and manage them to provide the user the possibility to see and interact with the stored data at a more abstract and complex level. Regarding the user capabilities, the projects can be classified in two categories: some, [11, 3], simply index the metadata extracted from different file formats while others [15, 12, 10, 9, 8, 7] allow the definition of custom properties.

We propose a system compatible with the already existent applications, that provides a classification in which files and file containers are treated similarly. From the point of view of the direction taken, our project belongs to the path that treats the file as an indivisible item. The focus is not on analyzing the file content, but on extending the organization and search methods. In this respect, the project does not head towards becoming a data storage enhancer, like [12], but towards a relational file system that introduces new file container concepts in order to obtain the desired flexibility. The two concepts are not mutually exclusive, the proof of which is [15], and thus the analysis of the file content can be seen as a possible extension of OrcFS.

Our system allows the user to define metadata, in order to describe files and file containers on two levels. Firstly, properties specify the criteria by which the files in the current location will be classified. Next, each file may or may not assign values to these criteria. Using this information the system will dynamically create virtual directories to reflect the classification. Moving further, the user will be able do define his own virtual containers, which are the result of boolean queries on properties and values. Because this implementation needs flexibility in manipulating relationships, we will be using a relational database to store the metadata, similar to [15].

An important characteristic of our system is that it runs over the classical file system, respecting the VFS [5] interface and maintaining an apparent hierarchical view.

[*]Computer Science Department, Technical University of Cluj-Napoca, Romania (adrian.colesa@cs.utcluj.ro).
[†]Computer Science Department, Technical University of Cluj-Napoca, Romania (alexandra.coldea@student.utcluj.ro).
[‡]Computer Science Department, Technical University of Cluj-Napoca, Romania (iosif.ignat@cs.utcluj.ro).

The following sections describe in greater details the implementation and testing of the system. As such, Section 2 and Section 3 present how several concepts of the classical file system have been redefined in order to serve the needs of OrcFS. Section 4 analyzes the project architecture and individual modules. Section 5 presents an example of how the system can be used. Section 6 contains a description of the tests performed and an examination of their outcomes. Section 7 summarizes the approach taken by other projects in this area. Section 8 contains several conclusions.

2. Redefinition of File System Concepts. The integration of the proposed relational model in the classical file system is done by redefining the old concepts and defining new ones. Due to the fact that the most affected is the logical organization layer, the main concept which was extended is that of a file container, or directory. The following sections describe how each concept has been altered.

2.1. Category. Categories are the only physical containers in the file system. Every other type of container presented from here on is virtual. They are the true file holders in the sense that every file must belong to a category. The notion remains from the hierarchical organization and its functionality is similar to that of a directory. Therefore, a category and a directory both define a file container, specified by the user by certain logical criteria. In the case of the traditional directory these logical criteria are the name and, indirectly, the hierarchy created. However, for the category, the classification criteria will be more detailed. The next sections will explain these criteria.

Similarly to directories, each category may contain several subcategories. However, the difference between subcategories is not only based on their names, like for subdirectories, but also on specific properties each item possesses.

Some examples are the category of books, papers, projects.

The notion of absolute and relative path maintain their meaning from the classical operating systems. However, the difference is that a file is no longer uniquely identified by its absolute path, but by a combination between it and properties. The following section will detail on this change.

2.2. Property and Value. In order to obtain more flexibility in organizing the categories, properties (tags, attributes) are associated to each of them. Every subcategory or file has the properties of the category it belongs to. Therefore, inside a category, the significance of properties is classification criteria. In order to classify a file by a certain criteria the user must assign a value to the corresponding property. Hence, the elements of a category are described using pairs "$(property, value)$". An item may have several pairs "$(property, value)$" associated with it and it may associate several values to the same property.

Thus, a property is associated to a category, while a value is associated to a file or a subcategory. Each category contains several properties and each of its files or subcategories, may or may not assign one or more values to any property associated with that category. In this organization a file is uniquely identified by its absolute path — indicated by the sequence of categories from root to the file location concatenated with file name— and a list of "$(property, value)$" pairs.

Let's assume the example of the *Articles* category. Some properties (classification criteria) which may be defined are: *author*, *year*, *keyWord*. Different files in this category may assign different values to the same property — they may have different authors. At the same time, the same file may assign different values to the same property — for example an article may be associated with several key words. The user may search through the files of a category using different classification criteria. Thus, he is able to search for all the articles written by a certain author, or all the articles written by several authors. Moreover he may ask for all the files "*having the authors X and Y and the year Z*". This leads to a very flexible and intuitive way of navigation through the file system.

Both properties and values are seen by the user as directories. However, they are only virtual containers because their content is not stored on disk, it is generated. In order to improve the performance and not be compelled to create every time the content of these types of directories, indexing will be used.

The difference between these types of containers and a category is that in the latter the user groups together files that, according to him, have a logical connection and a common set of properties. On the other hand, in property and value containers, the system groups together files that have the same subset of metadata fields.

2.3. The *Classification* Directory. In order to link the categories with the property and value directories, the *Classification* directory was introduced. It represents an isolation of the classified view from the non-classified view. This distinction is introduced in order to make the system and part of its facilities accessible through the interface of existing applications.

Each category contains a virtual directory called $Classification$ which consists of the files in the current location in a classified hierarchy. At the first level, this directory contains all properties associated with the current category.

At the next level, each property will contain directories corresponding to the values each of them may take. A value subdirectory contains all files with which it is associated but also, another $Classification$ directory. The latter contains a list of all the properties of the category that are not in the path of the current location (all properties not visited yet). Therefore, this organization treats the navigation through the classified view as a query composed of property and value constraints imposed on the files of the current category. By going deeper in the tree, one permanently refines the search, by adding more constraints to the query.

With this organization it may seem that the same file is present in several places. In fact, what this implementation of a relational file system does is provide several virtual paths to the same file.

Take for example a category *Articles* which contains the properties *author* and *year*. The file *File1* gives the values "*Author1*" and "*Author2*" to the property *author* and "*Year1*" to the property *year*. The generated $Classification$ directory contains two virtual directories *author* and *year*, corresponding to the two properties. The directory *author* contains two subdirectories called *Author1* and *Author2*, corresponding to the two values the property author may take. The directory *year* contains one subdirectory called *Year1*. The directory *Author1* contains the file called *File1* and another $Classification$ subdirectory. The latter contains the subdirectory *year*, corresponding to the property on which no filtering has been done yet on this path. In this way, *File1* can be found by taking any of the following paths:

- `/Articles/File1`
- `/Articles/Classification/Author/Author1/File1`
- `/Articles/Classification/Author/Author1/Classification/Year/Year1/File1`
- `/Articles/Classification/Author/Author2/File1`
- `/Articles/Classification/Author/Author2/Classification/Year/Year1/File1`
- `/Articles/Classification/Year/Year1/File1`
- `/Articles/Classification/Year/Year1/Classification/Author/Author1/File1`
- `/Articles/Classification/Year/Year1/Classification/Author/Author2/File1`

The structure that is created is very useful for retrieving files on systems with large amount of data and on which multiple users operate. Because this entire structure is generated, it does not take up additional disk space.

2.4. Query. The notion of query represents a formula used for locating files that satisfy certain conditions.

A query places constraints on the files in the current location, and only those items which satisfy the restrictions will be part of the result. A constraint has the form: '$p1 = v1$' and it is translated in natural language as: "*Find all files for which property* p1 *has value* v1". The individual constraints are connected using boolean operators: & (and), | (or), ! (not).

In this way, the file system can be queried like a relational database. A concrete example of such a query is: "$(p1 = v1)\&(p2 = v2)$", which will return all the items which give value 'v1' to property 'p1' and value 'v2' to property 'p2'. This result can also be obtained by classical navigation and an example of such a location is `Classification/p1/v1/Classification/p2/v2`. However, the result of the query "$(p1 = v1)|(p2 = v2)$" may not be obtained by navigating through the file system because it corresponds to the reunion of two directories (`Classification/p1/v1` and `Classification/p2/v2`). The query "$(p1 = v1)!(p2 = v2)$" can be translated in natural language as "*Find all files which give value* v1 *to property* p1 *but which do not give value* v2 *to property* p2". This is another example of query which may not be simulated by classical navigation.

The only difference between a query directory and a value directory is that the former represents a more complex way of expressing constraints. In fact, a query directory must be seen and manipulated like any location on the disk. This is why queries do not have to be implemented as new system calls, but can be integrated in the operating system by using the existing ones and changing their in-kernel implementation. Therefore, the `chdir` system call was used (its correspondent in shell terms is the `cd` command) and did not interpret anymore its parameter as a simple path, but as a query.

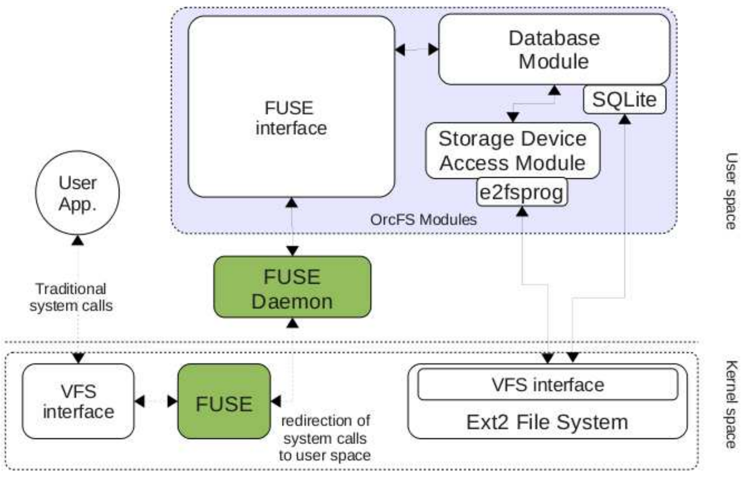


FIG. 3.1. *OrcFS user-space architecture*

Therefore, one will be able to write "`cd p1=v1`" in a terminal and the system will display all files for which this condition holds. Going further with the idea of increasing the navigability through the file system, the result will also contain a *Classification* directory which will provide further options for search refinement.

The way our system integrates the queries into the operating system provides compatibility with the exiting applications, allowing them to use our system as a normal one, but providing many different paths to the same file.

2.5. File. The concept of file has not changed relative to its meaning from the classical file systems. All the changes made are at a higher level, thus maintaining the file implementation unaltered. However, what has changed is how files are being manipulated. A user is now able to associate with them properties and values which convey meaning. This way, it is possible to treat files like entities of a database and use queries to group them into virtual directories. The same are treated subcategories. From now on, the term file system element will represent either a file or a subcategory.

3. System Architecture. The system is divided into three modules that communicate with each other in order to perform an action on the file system, required by the user or an application.

At the highest level, there is the VFS interface module. This receives requests from the user space, delegates them and then returns the result. At the lower level it is placed the module for storage device access. It can use an exiting file system storage strategy like Ext2 for example, provided in Linux kernel by the *e2fsprogs* library [1]. The metadata management module connects the other two modules. It manages the properties, values and relationships between them. It is implemented using an in-kernel database engine like in [13]. We used for that the *SqLite* [4] library. We chose SqLite because it is fast and not based on the client-server model and small enough to be easily integrated in kernel. The metadata management module stores and loads the database using the storage access module and answers metadata requests received from the VFS interface. All the commands that do not involve metadata operations are forwarded directly to the storage interface, without interference from the database module.

In a final implementation, all three modules will have been placed in the operating system's kernel. However, in order to ease the development process, a testing prototype using the FUSE [2] library was used. The latter is an application that runs both in the kernel — as a module which implements the VFS interface — and in the user space — as a daemon. It allows a developer to rewrite system calls in the user space and test them without recompiling of the kernel.

Figure 3.1 illustrates the way the system is placed in user space and how different parts of it interact with each other.

A user application issues a request which must me resolved by the VFS. The FUSE in-kernel module captures the request and generates a FUSE event. The FUSE daemon in the user space listens for such events and when one occurs, it delegates it to the OrcFS implementation. This application contains a database module (i.e. the metadata management module) and a storage device access module, both of which communicate with the underlying file system using specific libraries.

4. Transparent Integration of OrcFS in an OS. Our file system can be integrated in an OS completely transparent for the existent applications. This is because we do not change the traditional file-system interface adding new system calls, but just extend the functionality of the system calls already belonging to it. This way, an existing application can access our file system with no modification of its code. Nevertheless, if the application is aware of the fact that our file system provides extended functionality, it can take advantage of them. We see how this can be done in the following subsection.

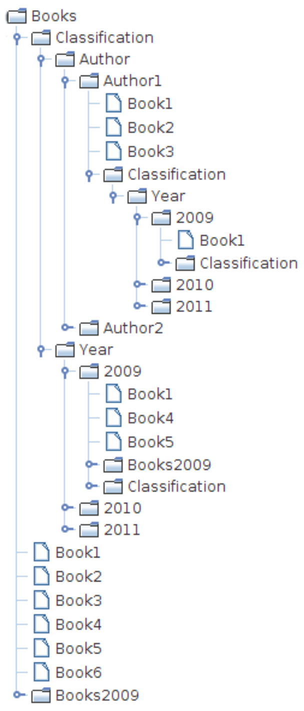
4.1. Backward compatibility: Browsing a Category Using a Traditional File-System Explorer. Assume the file system contains a category named *Books*, which we want to browse using a file-system explorer. Table 4.1 describes its contents in terms of "$(property, value)$" pairs. There are six files, one subcategory and two properties, i.e. *Author* and *Year*. Each file or subcategory is given zero or one values to each category's property. Based on these associations, the category's contents will be classified and viewed in different ways. The multiple ways a category's element can be found can be easily seen in Figure 4.1, which illustrates the file tree generated for the *Book* category. Still, there could be files given no value to any property. We call such files unclassified files. Consequently, they belong only to the unclassified view of the analyzed category and can be found in just a single way and place. We also note that files and subcategories are treated similarly.

	Type	Property *Author*	Property $Year$
Book1	file	Author1	2009
Book2	file	Author1	2010
Book3	file	Author1	2011
Book4	file	Author2	2009
Book5	file	Author2	2009
Book6	file	Author2	
Book7	file		
Books2009	category		2009

TABLE 4.1
Books category description

Let us start the navigation through the *Book* category from its root directory. It corresponds to the *unclassified area*. This is the place where all category's elements are visible like in any other traditional directory. Yet, we can also find here the special virtual directory $Classification$, which gives us access to the *classified area*. Entering the *Classification* directory, we can find two subdirectories *Author* and $Year$, corresponding to the two properties of the category. They are also virtual directories, while they do not correspond to physical containers. Their contents is generated when the user explicitly asks it or, like in our example, he accesses them. The $Year$ subdirectory gives the user the possibility to see all its books classified based on the year when they were published. The *Author* subdirectory is where books are classified based on their author. The two directories' contents consists only in subdirectories, each one corresponding to a different value given to those properties by different category's elements. Entering one of it, the user can find category elements having associated the corresponding "$(property, value)$" pair. For example, following the path `Books/Classification/Author/Author1`, the user can find files *Book1*, *Book2* and *Book3*. Besides the elements belonging to a value subdirectory, our system also generates a new *Classification* subdirectory, to be used to further classify the current directory contents. This lower level $Classification$ subdirectory contains as subdirectories only the properties not considered on the path to it. Thus, we can find only the $Year$ subdirectory in the `Books/Classification/Author/Author1/ Classification` directory. The $Year$ subdirectory contains subdirectories corresponding to its possible values. Entering, for example, the 2009 subdirectory, the user can find again the *Book1* file. This will be in fact the only file found here, because it is the only one having associated both "$(Author, Author1)$" and "$(Year, 2009)$" pairs. The other files can be found on different paths, while they give different values to the $Year$ property.

The $Classification$ subdirectory also belonging to the `Books/Classification/Author/Author1/Classification/Year/2009/` is empty, because there is no other property not yet considered on that path, i.e. there is no further classification criterion.



```
Books
  Classification
    Author
      Author1
        Book1
        Book2
        Book3
        Classification
          Year
            2009
              Book1
              Classification
            2010
            2011
      Author2
    Year
      2009
        Book1
        Book4
        Book5
        Books2009
        Classification
      2010
      2011
  Book1
  Book2
  Book3
  Book4
  Book5
  Book6
  Books2009
```

FIG. 4.1. *Generated structure of the Books category*

A similar way of finding the *Book1* file could be the one starting with the other property in the first $Classification$ directory, which is `Books/Classification/Year/ 2009/Classification/Author/Author1/`. This is because the queries corresponding to the two different paths, i.e. "*Which are all the books written by* Author1 *in* 2009?" and "*Which are all the books written in* 2009 *by* Author1?", are logically equivalent and must return the same answer. The logical equivalence of the two queries results from the fact that the two conditions they include are linked by a logical `AND` operator, which is commutative. This gives the user the possibility to find the same set of files following different paths in the file tree. Although, we must note that these navigating alternatives correspond to queries containing only the `AND` operator. Still, the real queries can also be expressed using some other different logical operators, like `OR` and `NOT`, and actually there are files corresponding to such queries. For instance, the answer of the "*Which are the books written by* Author1 *in* 2009 *or* 2010?" query, would contain both *Book1* and *Book2* files. As we can easily observe, this file set cannot be located in any directory in the file tree illustrated in Figure 4.1. We eliminated this limitation, giving the user applications the possibility to directly jump to virtual directories containing the answers of questions formed by different logical operators, directories which do not correspond to any node in the file tree, still being located on a sort of a file path. This remaining limitation is that such directories cannot be touched navigating down in the file tree level by level. The way we did this is described in the following subsections.

4.2. Reuse and reinterpret the existent file-system system calls. We have reimplemented the system calls related to directory access, but left unchanged the ones providing access to files' contents. This is because we gave a new interpretation of the directory concept — related to data organization, but kept that of the file concept — related to storing of data. Still, the traditional effect of the system calls we reinterpreted is changed only in the classification area, not in the unclassified area. Thus, if a file is removed from the root directory of a category, it is completely removed from that category, different from removing the same file from one of the subdirectories in classification area, which has another effect. Let us see exactly how each change on a directory is interpreted.

Creating a file in the classification has also the additional effect of classifying the new file based on the "$(property, value)$" pairs included in the path specified at file creation. For example, creating a new file *Book7* in the directory `Books/Classification/ Author/Author1/Year/2011` will create the file in the *Books* category, but also associate to it the "$(Author, Author2)$" and "$(Year, 2011)$" pairs. We must note that the same effect is obtained using the similar path `Books/Classification/Year/2011/ Author/Author1`. If a file with the same name already exists in the category, the file contents is truncated and its metadata is replaced with the one specified in the creation path. If the file cannot be truncated, because of permission rights restrictions, the user is reported an error. This can be confusing if the existent file is not visible in the path specified at file creation. In order to avoid such situations, it is recommended to create new files only in the root directory of a category, where all its files are visible. After that, the file can be associated explicitly "$(property, value)$" pairs, by performing other operations on it, interpreted as file classification operations.

Removing a file from a directory located in the classification area will not result in the physical removal of that file, but just in removing some of its associated metadata. More precisely, the "$(property, value)$" pairs included in the path of the directory the file is removed from will be the one removed. For example, removing `Books/Classification/Author/Author1/Book3` results in removing the "$(Author, Author1)$" pair from those associated to file $Book3$. A file remains in its category until it is removed from the root directory of that category.

Creating a directory in the unclassified area just creates a new element, i.e. subcategory, in the corresponding category. The result of creating a new subdirectory in the classification area depends on the exact directory (i.e. parent directory) the operation is performed. There exist the following cases:

- If the parent directory is one of the $Classification$ directories, then the new directory is interpreted as a new property of the category. For example, creating the directory `Books/Classification/Publisher/` creates a new property, named *Publisher*, in category *Books*.
- If the parent directory is property directory, then the new directory is taken as a new value of that property. For example, creating the directory `Books/ Classification/Year/2003` creates the new value 2003 for the existent property $Year$. Obviously, there will be at that moment no file having associated the "$(Year, 2003)$" pair, but this can be done later, using other file operations.
- If the parent directory is a value directory, the new directory is an element of the category, classified based on the properties and values in the path used on its creation.

We must note that a problem similar with that from file creation can also occur in the case of directory creation: using directories names already existent in a category. We already described the reason they are not visible on any level of the category file tree. So, in order to avoid confusion, it is recommended for the user to create properties or values on the highest levels of the category. Thus, new properties should be created in the top level $Classification$ directory, where all category's properties are visible.

Removing a directory is very similar with the directory creation operation, in the way its parameters are interpreted. There are more cases:

- If the removed directory is a $Classification$ one, an error is reported, since it does not correspond to any data in the file system, but it is used just to provide specialized access to files.
- If the removed directory is a property one, then the files that could be found on the subtree having as root that directory are removed their associations with the removed property. Although, the property itself is not removed unless it is removed from the $Classification$ directory on the top level.
- Removing a value directory is similar with removing its corresponding property directory, but its effect is reduced only to that value, not to all the values of that property, like in the previous case.

Creating a hard link to a file in the same category classifies that file, i.e associates to it new $(property, value)$ pairs extracted from the path of the new hard link. There are two restrictions when a new hard link is created: the same name must be used for the new link and it cannot be created in a property directory, where only value directories are allowed.

Creating a symbolic link is identical to creating a new file, since a symbolic link is actually a new file, different by the referred one.

Another system call we reinterpreted is *chdir*. It is not really a file operation. Yet, we used it in order to give the user access to sets of files that cannot necessarily be found just navigating in a category's file tree. It also allows the user "jumps" in physically non-locatable directories, corresponding to queries including logical operators like "OR" and "NOT".

4.3. The Query Language. We have seen that both the navigation through the file tree going down level by level and the direct "jump" to a virtual directory are made using the *chdir* system call. We interpreted the path specified for it as a request (query) addressed to our system to generate the contents of a directory, being it an immediate subdirectory in the file tree, relative to the current directory, or even one having no corresponding node in the file tree. The difference between the two cases is that in the former, the query contains only the "AND" logical operator, while in the latter, any logical operator could be used. We illustrate in the following examples, executed in the *Books* directory, the possible forms of such queries.

Navigation through the category's file tree can be done in the classical way, by moving up or down the tree, searching for some files. Moving down a new level means refining the result of the previous query, by classifying the files in the current directory by the remaining properties. This is very useful in cases the user does not remember from the beginning all (or many of) the properties and values associated to a file.

```
> cd Books/Classification
> cd Author
> ls
Author1 Author2
> cd Author1
> ls
Book1 Book2 Book3 Classification
> cd Classification
> ls
Book6 Year
> cd Year
> ls
2007 2008 2009
> cd 2007
> ls
Book1 Classification
> cd Classification
> ls
>                # no file
```

In case the user remembers from the beginning more information about the needed files, he can specify directly more complex queries corresponding to paths in the category's file tree. The advantage provided by our system is that the user must not specify the "$(property, value)$" pairs in a fixed order, having more alternatives to do that, as illustrated below.

```
> cd Books/Classification/
> cd Author/Author1/Year/2009
> ls
> Book1
> cd -  # change to the previous location
> cd Year/2009/Author/Author1
> ls
> Book1
```

Although, a more complex way to search files is by using the specialized query syntax supported in the path specified to *chdir* system call or *cd* command in the command line. The following examples illustrate several possible queries together with their outcome.

```
> cd Books
> cd 'Author=Author1&Year=2009'
> ls
Book1  Classification
> cd 'Author=Author1&Year=2009|Year=2010'
> ls
Book1  Book2 Classification
> cd 'Author=Author1!Year=2009'
> ls
Book2  Book3 Classification
```

The examples described has proven that in our enhanced file system, both files and categories can be manipulated easier and located faster.

5. Tests and Results. The previous sections have proven the utility of having a metadata improved file system that automatically constructs a classified organization of files and categories. However, in order to prove that such a system can be used without introducing significant overhead, several usage tests were performed. They measured the time of completing different tasks in a classical file system, i.e. Linux Ext2, compared to the same operations in our enhanced file system.

We run each test using the Linux *time* command. This command takes as parameter the name of the testing application, runs it and returns at the end of that test's execution the following time information: (1) the *real time*, which is the wall-clock time elapsed from the start of the testing application until its termination, (2) the *user time*, which is the time the testing application spent using the processor in user mode to execute its own code, and (3) the *system time*, which is the time the processor was used in kernel mode executing system code on behalf of the testing application. We named the sum of the user and system times *effective time* and the difference between the real time and the effective time *waiting time*. The waiting time is the time spent by the testing application waiting for processors to become available and is highly dependent on the system's scheduler and load. Because the OrcFS testing prototype was implemented in user space using Fuse, most of what should have otherwise been run as system code in kernel mode, was actually run in user mode in an user application, whose execution could have been interleaved with the execution of other running applications. Consequently, the execution of our file system's code could have been suspended many times by the operation system's scheduler in order to also let other applications run. That is why we observed, especially for long running tests, that their waiting time accounts for a large percentage from their real time. We did not consider however the waiting time to be either important or relevant for our tests, as long as our system is ultimately intended for the kernel space implementation. Thus, the overhead introduced by our system is mainly indicated by the comparison of effective times of the tests. Although, we also illustrated in the result tables the overhead at the real time level, just to see the disadvantage of having a user space implementation of our system.

The tests performed can be divided into several categories which are presented next.

5.1. Creation Tests. The first two tests performed concern the creation of files and directories.

In **Test1** we created N subcategories in the same category of OrcFS, opposed to creating N subdirectories in the same directory in the classical file system.

The results are depicted in Table 5.1. The last column contains the mean percentage the OrcFS time represents from the Ext2 time, for different types of times we illustrated.

It must be emphasized that when N is rather small (100 or 1000 directories) the difference between the real creation times is not very big. The only overhead here is the one introduced by the FUSE indirection and by the extra logic. However, as N increases and reaches the values of 10.000 or 50.000, another factor must be taken into consideration, which is, as we already mentioned, the Linux scheduler. The problem that occurs is that, by using FUSE the system calls are redirected in user space. Therefore, they are now run in user processes that have a smaller priority. Even more, the Linux scheduler decreases the priority of a process if that process uses intensively the processor. It can be seen from the Table 5.1 that while creating 50.000 directories, the process spends the majority of time waiting for processor. In fact, while performing the test in the classical file system there was a noticeable decrease of the speed of the other processes during the execution of system calls in kernel, while in OrcFS no such event was observed, because most of the code was executed in user space. On the other hand, taking into account only the effective time, we can observe that the overhead is rather small, i.e. up to about 31% for 50.000 directories.

This is why, after a certain number of directories created, the real times obtained are no longer relevant. As a consequence, the following tests will no longer use very large values. We must also take into account that in practice new elements are very rarely introduced simultaneously in the system in such a huge number.

In **Test2** we created N files in the same category in OrcFS, compared to creating N files in the same directory in the classical file system. The result is depicted in Table 5.2.

In the case of file creation the situation is similar to that of directory creation. It is expected that once the project is introduced inside the kernel, the difference will decrease.

No. of directories	Time type	Ext2 time	OrcFS time	Mean percentage
100	real	0m0.408s	0m0.499s	122%
	user	0m0.156s	0m0.176s	
	sys	0m0.144s	0m0.152s	
	effective	0m0.300s	0m0.328s	109%
1000	real	0m3.194s	0m6.315s	197%
	user	0m1.116s	0m1.212s	
	sys	0m1.168s	0m1.388s	
	effective	0m2.284s	0m2.600s	113%
10000	real	0m38.723s	3m4.346s	485%
	user	0m10.969s	0m12.545s	
	sys	0m10.817s	0m14.121s	
	effective	0m21.786s	0m26.666s	122%
50000	real	3m11.378s	68m4.267s	2136%
	user	0m50.863s	1m03.220s	
	sys	0m52.711s	1m13.473s	
	effective	1m43.574s	2m16.693s	131%

Table 5.1
Test1 results

No. of files	Time type	Ext2 time	OrcFS time	Mean percentage
100	real	0m0.355s	0m0.631s	177%
	user	0m0.132s	0m0.148s	
	sys	0m0.112s	0m0.116s	
	effective	0m0.244s	0m0.264s	108%
1000	real	0m3.144s	0m7.048s	224%
	user	0m1.048s	0m1.312s	
	sys	0m1.164s	0m1.364s	
	effective	0m2.212s	0m2.676s	121%
5000	real	0m16.202s	0m57.611s	356%
	user	0m05.120s	0m06.228s	
	sys	0m05.192s	0m06.656s	
	effective	0m10.312s	0m12.884s	125%

Table 5.2
Test2 results

5.2. Directory Opening Tests. **Test3** opens a category containing N subcategories, opposed to opening a directory containing N subdirectories. Table 5.3 presents the result of these tests.

Test4 opens a category containing N files, opposed to opening a directory containing N files. Table 5.4 presents the result of these tests.

Test5 opens a *Classification* directory containing N properties, compared to opening a directory with N subdirectories. Table 5.5 presents the result of these tests.

Test6 opens a property directory with N values, versus opening a directory with N subdirectories. Table 5.6 presents the result of these tests.

5.3. System Simulation Test. In **Test7** we created a category with N properties and M values associated with each property, compared to creating in the classical file system (Ext2) the entire structure generated by OrcFS in the *Classification* directory in the same situation. We measure only the real time, as it was fully relevant. Table 5.7 illustrates the results.

For the first part of Test 7 there was created a category with 3 properties each of which contain 3 values. In

No. of directories	**Time type**	**Ext2 time**	**OrcFS time**	**Mean percentage**
100	real	0m0.008s	0m0.025s	315%
	user	0m0.000s	0m0.000s	
	sys	0m0.008s	0m0.008s	
	effective	0m0.008s	0m0.008s	100%
1000	real	0m0.032s	0m0.164s	512%
	user	0m0.004s	0m0.004s	
	sys	0m0.012s	0m0.016s	
	effective	0m0.016s	0m0.020s	125%

Table 5.3
Test3 results

No. of files	**Time type**	**Ext2 time**	**OrcFS time**	**Mean percentage**
100	real	0m0.008s	0m0.014s	175%
	user	0m0.008s	0m0.006s	
	sys	0m0.004s	0m0.008s	
	effective	0m0.012s	0m0.014s	117%
1000	real	0m0.020s	0m0.081s	202%
	user	0m0.012s	0m0.008s	
	sys	0m0.008s	0m0.016s	
	effective	0m0.020s	0m0.024s	120%

Table 5.4
Test4 results

No. of directories	**Time type**	**Ext2 time**	**OrcFS time**	**Mean percentage**
100	real	0m0.008s	0m0.060s	750%
	user	0m0.000s	0m0.004s	
	sys	0m0.008s	0m0.008s	
	effective	0m0.008s	0m0.012s	150%
1000	real	0m0.022s	0m0.107s	486%
	user	0m0.004s	0m0.010s	
	sys	0m0.012s	0m0.018s	
	effective	0m0.016s	0m0.028s	175%

Table 5.5
Test5 results

No. of directories	**Time type**	**Ext2 time**	**OrcFS time**	**Mean percentage**
100	real	0m0.008s	0m0.055s	687%
	user	0m0.000s	0m0.000s	
	sys	0m0.008s	0m0.004s	
	effective	0m0.008s	0m0.004s	150%
1000	real	0m0.022s	0m0.107s	486%
	user	0m0.004s	0m0.006s	
	sys	0m0.012s	0m0.020s	
	effective	0m0.016s	0m0.026s	163%

Table 5.6
Test6 results

No. of properties	**No. of values**	**Ext2 time**	**OrcFS time**	**Mean percentage**
$N = 3$	$M = 3$	0m2.502s (526$dirs$)	0m0.079s (10$dirs$)	2.65%
$N = 4$	$M = 4$	1m18.218s (17.749$dirs$)	0m0.176s (17$dirs$)	0.22%

TABLE 5.7
Test7 results

the OrcFS, this means creating 10 directories. However, simulating the classified structure that is automatically created means generating 526 directories in the classical file system. Therefore, the resulting time difference between these two operations is great. The second part emphasizes even more this difference, since in the case of 4 properties and 4 values for each of them means creating 17 directories in OrcFS and 17,749 directories in the classical file system. This is the test that shows the full power of the OrcFS. A single file may be found by taking several paths and thus, the risk of taking a wrong path while knowing a few characteristics of the desired file is significantly reduced.

In conclusion, while operations of introducing new data in the file system are delayed in the OrcFS, these time differences are not as important as the gain for creating an immense structure with only a few number of operations.

6. Related Work. The first project that approached this issue is the Semantic File System [11]. It was designed to provide associative attribute-based access to the contents of an information storage system. The attributes are automatically extracted from the files stored on disk with the help of transducers. A transducer is a filter that takes as an input the contents of a file and outputs the files entities and their corresponding attributes. A user does not have the capability to define attributes, he may only write queries, whose result are virtual directories. The solutions implemented in [11] represent the starting point of OrcFS. Some of the concepts proposed by the Semantic File System, such as virtual directories or automatic classification were adapted by OrcFS as the base of the entire project. However, what was not implemented is the automatic attribute extraction, which was left as a future improvement.

An improvement of the Semantic File System is Nebula [6], in which the user is able to perform CRUD (create, read, update, delete) operations on attributes. Nebula implements files as sets of attributes. Each attribute describes some property, such as owner, protection or file type. The content of a file is represented by a special text attribute. The traditional directories are replaced with database views. When a file is created, it is placed in an index, not a directory. When the file system is refreshed, the file will appear in the appropriate views. This is how views dynamically classify files. As opposed to this approach, OrcFS treats the file as an indivisible item.

A further improvement was made by Be File System (BFS) [10]. It stores the list of attributes associated with a file as a directory, the address of which is stored in the attributes field of the BFS i-node structure. The attribute directory of a file is similar to a regular directory. Programs can open it and iterate through it to enumerate all the attributes of a file. The attribute directory solution is rather slow in the case of having several small attributes. This it why, the spare area of the i-node disk block is used to store small attributes. Because the focus of OrcFS is to increase the flexibility of expressing relationships, it does not employ the i-node structure, but a database to store the file attributes.

A notable approach to the relational organization of the file system is LisFS (Logical File System) [15]. In this approach the storage structure, called formal context, holds objects that are described by logical formulas. These objects represent files or parts of files. Both paths and queries are seen as formulas. LisFS allows boolean querying. There are two types of proprieties: intrinsic, which are computed from the content of the object and extrinsic, which are assigned by the user. A key feature is the fact that LisFS runs over the Linux kernel, respecting the VFS (Virtual File System) interface. This assures compatibility with the current applications. This project is the one that has the most elements in common with OrcFS. They both use a database to store metadata, work in user space and enable query navigation. The main difference is that in OrcFS treats both files and directories in the same way.

WinFS (Windows Future Storage) [12] was first presented in 2003 as an advanced storage subsystem for the Microsoft Windows operating system. It breaks down the concept of files into subdivisions, extending the possibility of data processing at a lower level. Furthermore, it treats all data as objects. The user can create relationships or sets of objects through an application and reuse them in other applications and define functions on objects.

Although it is not a file system, Google Desktop [3] is worth mentioning. It is a desktop search application that uses indexes to manage the information on the computer. Once installed, it starts indexing the email, files and Web history stored. This operation runs only once, when the computer is first idle for 30 seconds. Google Desktop also ensures that the index stays up to date, adding new e-mail as it is received, files as they are updated them and web pages as they are accessed.

The most similar projects with OrcFS are those in [9] and [8] on whose architecture the OrcFS is based on, but which we extended to homogeneously treat files and subcategories in a category as generic file system elements. We also developed the query command line syntax to support complex interrogation and navigation in the OrcFS file system using existing shells and file browsers.

7. Conclusions. An architecture of the file system enhancing the organizing and searching capabilities has been developed. It is based on allowing the user to associate descriptive metadata in the form of property-value pairs to both files and directories. The system creates virtual file containers forming a classified organization, in order to provide multiple access paths to the same file. Navigation down this tree structure represents a permanent query refinement, in which the set of results is diminished until the user either finds the desired file or an empty directory is obtained.

The usability of OrcFS has been proven at a subjective level, by creating a specific system and navigating through the classification. It was also proven at an objective level, by measuring and comparing the times of performing different tasks both in the classical file system and this enhanced system.

Several steps that will be taken in the future for improving even further the navigation, search and overall accessibility are:

- Automatically extract property-value pairs from different file formats.
- Allow the user to define category templates with specific properties. This would shorten the initial job of environment creation. Therefore, the user will be able to use the same set of properties in different locations of the system, without having to specify them every time.
- Maintaining a record of the paths where a template has been applied, so that the user is able to formulate queries such as "Find all locations for the template where certain restraining conditions are valid".
- Develop a specialized file browser which takes advantage of the enhancements of OrcFS, in order to present a richer file system. This application must clearly differentiate between property, value and category directory types. It must allow the user to specify property-value pairs using a form and not through file operations like in the classical browser.
- Integrate OrcFS in the Linux kernel, in order to decrease the overhead introduced by FUSE redirection.
- Develop a distributed relational file system. A client should be able to make queries and navigate like in OrcFS but the answer will come from different servers, transparently to the user. The individual responses of each server will be merged, in order to present the user with a uniform view of the entire system.

REFERENCES

[1] Ext2 file system utilities. Internet page last accesed on July 2010.
[2] Fuse. Internet page last accesed on July 2010.
[3] Google desktop. Internet page last accesed on July 2010.
[4] Sqlite. Internet page last accesed on July 2010.
[5] Daniel Bovet and Cesati Marco. *Understanding the Linux Kernel.* O'Reilly Media, 3 edition, November 2005.
[6] Mic Bowman, A. Dharap, Mrinal Baruah, Bill Camargo, and Sunil Potti. A file system for information management. Technical report, The Pennsylvania State University, 1994.
[7] Alexandra Coldea, Adrian Coleşa, and Iosif Ignat. Orcfs: Organized relationships between components of the file system for efficient file retrieval. In *Proceedings of The 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '10)*, pages 434–441. IEEE Computer Society, 2010. (DBPL, IEEExplore).
[8] Adrian Coleşa, Victor Cionca, Alexandru Ţaţa, and Iosif Ignat. A meta-data enhanced file system. In *IEEE International Conference on Intelligent Computer Communication and Processing (ICCP'07)*, 2009.

[9] Adrian Coleşa, Iosif Ignat, Zoltán Majó, and Victor Cionca. A meta-data enhanced file system using the classical interface. In *Proceedings of The Tenth International Conference on Applied Mathematics, Computer Science and Mechanics*, Cluj-Napoca/Băişoara, Romania, June 2006.

[10] Dominic Giampaolo. *Practical File System Design with the Be File System.* Morgan Kaufmann Publishers, Inc.

[11] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 16–25, October 1991.

[12] Richard Grimes. Revolutionary file storage system lets users search and manage files based on content. *MSDN Magazine*, January 2004.

[13] Aditya Kashyap. File system extensibility and reliability using an in-kernel database. Technical Report FSL-04-06, Stony Brook University, December 2004.

[14] Yoann Padioleau and Olivier Ridoux. A parts-of-file file system. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05)*, page 17. USENIX Association, 2005.

[15] Yoann Padioleau, Benjamin Sigonneau, and Olivier Ridoux. Lisfs: a logical information system as a file system. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 803–806. ACM, 2006.

Edited by: Dana Petcu and Alex Galis
Received: March 1, 2011
Accepted: March 31, 2011