A Case for Dynamic File Attributes

Randy Smith, Joe Meehean
University of Wisconsin—Madison
{smithr, jmeehean}@cs.wisc.edu

Abstract

A file attribute is a user defined <key, value> pair associated with a file. We explore the idea of extending a file system to contain dynamic per-file attributes. This paper has two goals: first, it describes our implementation of an attribute file system overlay, in which the attribute functionality is provided as a user-level library and the system calls are modified to provide a seamless environment to users and applications. We compare the performance of our implementation under various scenarios, and we modify several applications to demonstrate the implications and benefits of customizable file attributes.

Second, we explore the use of file attributes as a mechanism for approximating full file-content search. Our approach is to define a file-type-independent attribute format in which searchable data is stored, allowing for simple, type-agnostic indexing and search tools to be used. Preliminary performance and functionality tests indicate that this technique shows promise: searchable indexes may be built and searched with small disk-space and cpu-time overhead.

1. Introduction

Files in an operating system are not stand-alone, isolated constructs. Users of files—both humans and applications—often need to store information about files without necessarily wanting to change the files themselves. For example, revision control tools store file version numbers, executable files may have associated icons for display in GUI systems, or documents may be associated with cryptographic keys or digital signatures.

One solution to this problem is to associate dynamic, user-defined attributes with individual files. The concept of customizable file attributes is not new, although their use is currently not widespread. This paper describes our efforts to investigate the feasibility and usefulness of providing dynamic attribute support for files. To test our ideas, we have implemented a user-level library and have modified the necessary system calls to provide a seamless prototype of an attribute-supported file system.

Of particular interest to us is the role that attributes may play in approximating file-content search. Accurately querying file content in an application-independent manner has been an elusive goal [5]. We present an approach that uses file attributes to approximate file content search. In our technique, we provide generic indexing and searching tools, and we place the burden of creating searchable file content on applications. Preliminary results indicate that this technique may provide an acceptable approximation of file content searching.

This paper is organized as follows: Section 2 provides the design goals and implementation details for our attribute system. In Section 3 we describe our mechanisms for performing attribute searches and approximating file-content search, and we report the results of two case studies. Section 4 presents several attribute-aware applications as a means of describing the convenience and advantages that file attributes provide. Section 5 describes the relevant related work, and Section 6 concludes.

2. Attributes

Conceptually, an attribute is simply a tuple of data in which the first element contains an identifying key and the second contains the value corresponding to that key. For example, a typical attribute for a version-controlled file might be represented by the tuple <version, 3.1.2>. An individual attribute is identified by its key, and attributes associated with files may be freely created, modified, and deleted over time.

2.1 Design and Implementation of File Attribute

In our implementation, the attributes for a file *f* are stored in a separate, distinct hidden file contained in the same directory as *f*, with the name of the attribute file produced by prefixing the filename of *f* with ".attr__". Since attribute files are stored as distinct file system entities, we were required to modify selected system calls so that changes to *f*, such as moving and renaming, are also applied to the attribute file for *f*. The result is a seamless attribute implementation: file attributes are not implemented in the file system, but this cannot be discerned operationally either by users or by applications.

File attributes are exposed to users and user-level applications via function library calls and executable tools. In our implementation, applications access a file's attributes through library functions `set_attr` `get_attr`, `erase_attr`, and `enum_attr`. In addition, we have written utilities *setattr*, *getattr*, *rmattr*, and *lsattr* that wrap these functions to provide attribute manipulation capabilities from within a shell.

In this section we present the details of our attribute system. We first give our assumptions and goals, followed by our design and implementation. We then provide some performance comparisons of the library. Finally, we discuss the system call changes we made.

2.1.1 Assumptions and Goals

The design and implementation of our attribute system follows from several assumptions and goals. Our purpose in stating these assumptions is to identify the common cases and develop a design that optimizes those scenarios while still providing acceptable behavior in other less-frequent cases. We have the following assumptions:

1. Attributes are used primarily to store extra data about a file rather than file data itself. We expect the common case to be that individual files have ten or fewer attributes associated with them.

2. Attributes tend to follow a write-once read-many paradigm. Once a specific attribute is associated with a file, it frequently becomes a relatively static entity that does not change over the course of the file's life.
3. Attributes are fixed size, structured entities, and most attribute data associated with files will be in a structured form. For attribute values that do change, successive writes of different values can be made in-place and should not require disk space to be reallocated.

Obviously, these assumptions do not always hold. However, they do state our common cases and provide a backdrop upon which we can describe the aims of our design. In light of these assumptions, our attribute system has the following goals:

1. Ease of use. Programmers should be able manipulate attributes easily, using simple, clean interfaces. Users should feel that attributes are a natural extension of files themselves—viewing file attributes should be relatively easy.
2. Fast look-up and in-place overwrite. If read operations are the most common, then attribute access functions that do not change an attribute file structurally should be fast and efficient.
3. Unconstrained attribute value sizes. The maximum size of a key-value pair on disk should not restrict the types of attributes that can be stored. Users and programmers should be free to store attributes without being constrained by size restrictions.
4. Small space overhead. Indexing and maintenance data used to maintain attribute integrity should be a small percentage of total attribute file size.

2.1.2 Implementation

Our attribute library implements the <key, value> abstraction described above, and additionally associates a specific type for each attribute value. Currently, the library recognizes bytes, ints (4 bytes), booleans, character strings, and byte streams, which are used as a generic type.

We employ a linear attribute storage scheme that lays out key-value pairs sequentially in the attribute file. Figure 1 depicts the on-disk structure of a single attribute entry. As the figure shows, keys and values in an attribute entry are each prefixed by a small header. The key header contains the length of the key, the total allocated space for the entry, and some miscellaneous housekeeping flags. The value header contains the value's type along with the actual size of the value (which may be less than the allocated space).

This layout of key-value pairs is designed to increase the efficiency of attribute traversal when setting, retrieving, or enumerating attributes. When scanning for a specific key, the key header and the key itself are retrieved from the file. If the key matches, the value and value header fields of the entry are both retrieved. If the key does not match, the offset of the next attribute entry is determined using the total allocated space field contained in the key header.

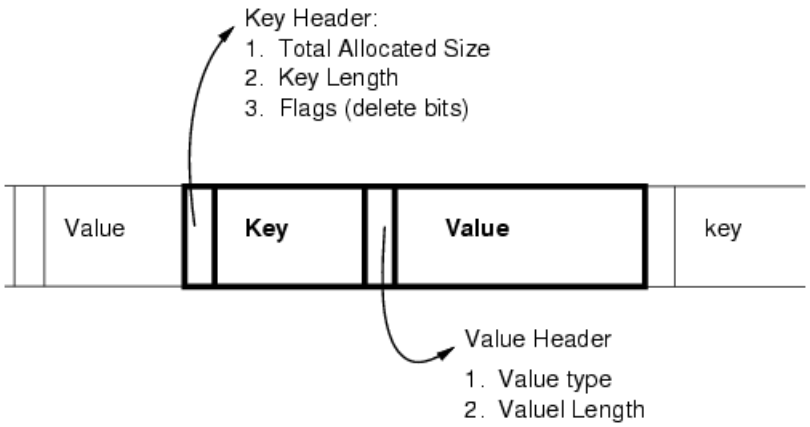

Figure 1 Structure of a single attribute entry in an attribute file

Attribute entries maintain both a total allocated size and a current value size to increase overwriting efficiency. When an attribute value is to be overwritten with a new value, the new value is written in place if the size of the new value is less than or equal to the space allocated for the existing value. Otherwise, the existing attribute entry is deleted and a new attribute entry is appended to the end of the attribute file. This behavior follows from our assumption that attribute values are generally fixed size, structured entities so that in-place overwriting is the most common case.

Attribute deletion is the slowest of all the attribute manipulators, reflecting our assumption that it is the least frequently used operation. Attributes are deleted when either `erase_attr` is called or an existing attribute needs to be overwritten with a value that is too large to fit in the allocated space. At a minimum, deleted attribute space can be reclaimed by simply compacting the remaining attributes whenever an attribute is deleted. While functionally correct, this technique is inefficient and does not scale well to large numbers of attributes [3].

A more sophisticated approach is to provide a delete bit for each entry that is set when an attribute is erased. In this case, the header block that prefixes the attribute file is updated to contain the number of deleted attribute entries and their sizes. The attribute file is then periodically compacted whenever the relative amount of deleted data surpasses a specified threshold. Currently, our implementation defines a delete bit in the housekeeping flags, but compacts on every delete request.

With regard to the programming interface, the attribute functions `set_attr`, `get_attr`, and `erase_attr` are all stateless—the programmer calls these functions without needing to perform any explicit setup or needing to maintain any state for subsequent calls. The exception to this is `enum_attr`, which iteratively walks through an attribute file. Between successive calls of `enum_attr`, an application program must hold the position of the "cursor" in the attribute file (implemented as an opaque `char **`). To protect against file corruption due to potentially simultaneous access by multiple processes, all accesses to an attribute file (both reads and writes) are wrapped with `flock` system calls.

2.1.3 Comparison and Performance

To better understand the performance characteristics of our implementation, we also implemented a version of our attribute library that uses Sleepycat's Berkeley DB library [7] to store and manage file attributes. For this implementation, our library interface effectively became a lightweight wrapper around the Berkeley-DB library package. To preserve consistency for our tests, the attribute library interface is identical for both our implementation and the Berkeley DB version—applications may switch between implementations simply by linking the desired library. Since the Berkeley DB software performs its own insertion, deletion, and retrieval, wrapping it inside our attribute interface is largely a matter of calling the appropriate Berkeley DB function.

Our tests[1] have shown that our linear model has lower disk space and access time costs compared to the Berkeley-DB based implementation. With regard to disk-based costs, small attribute files result in under-utilized data blocks when using the Berkeley DB library. Further, for large attribute files, the structure of B-tree indexes themselves lead to partially empty blocks, resulting in an average of 50% unused space per attribute file. While under-utilized space is not generally an issue for a single attribute file, the cumulative lost space in a file system full of attribute files can be quite large. With the linear model, on the other hand, only eight bytes per attribute are used to maintain the attributes associated with a file. In our experiments, we have observed that the linear model's attribute file size averages one fourth the size of the Berkeley-DB model.

To evaluate the relative time costs, we performed tests comparing the execution times of the two implementations with regard to reading, writing, deleting, and enumerating large numbers of attributes. Figure 2 shows the results of one of these tests, which measured the time required to write several string-valued attributes in succession followed by the time to read all of the attributes written. In this test, each string-valued attribute was given a random string varying uniformly in size between 1 and 1024 bytes. Although we anticipate the common scenario to associate fewer than ten attributes with a file, we repeated this test for attribute counts of up to 2000 attributes per file to stress the scalability of the linear design. Table 1 summarizes the cumulative reading and writing times for small and large numbers of string-valued attributes, showing that even for 2000 attributes per file the linear model outperforms the Berkeley DB-based model.

As these results show, our linear model performs attribute reads (gets) and writes (sets) several times faster than our Berkeley DB-based implementation. These results seem counterintuitive at first, considering that the Berkeley DB implementation uses a constant-time B-Tree indexing method for looking up attributes. Our analysis of this phenomenon shows that the biggest single factor affecting the access time of our Berkeley-DB is the cost associated with opening and closing the database. Retrieving a single key-value pair from a B-Tree based attribute file requires on average 3800 microseconds to complete in our 1GHz Pentium III-based test environment. However, experimental results show that approximately 80 microseconds is spent performing the lookup, with the remaining 3700 microseconds used to open and close the database.

1 Our tests showed comparable results for both Hash table and B-tree indexes. In this discussion, we present only the B-tree results for comparison.

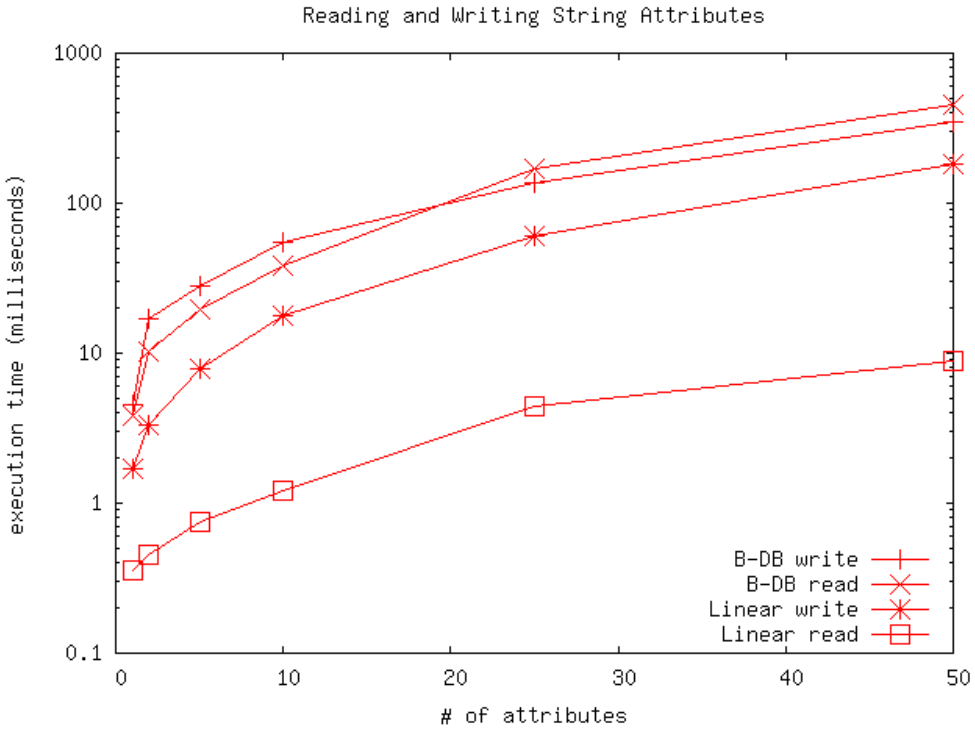

Figure 2 Comparing cumulative reading and writing times for increasing numbers of string-valued attributes.

# of attributes	**1**	**2**	**10**	**1000**	**2000**	**1/2000**	**2000/2000**
Berkeley set (berk-set)	4.514	16.822	54.381	6939.101	21947.125	--	--
Berkeley get (berk-get)	3.858	10.266	38.676	11279.323	33219.092	29.773	23.008
Linear set (lin-set)	1.697	3.300	17.814	3647.491	17588.211	--	--
Linear get (lin-get)	0.357	0.451	1.202	919.281	3416.864	0.233	9.256

Table 1 Cumulative reading and writing times (in milliseconds) for small and large numbers of string-valued attributes per file are shown in the left-most columns. The right most columns show the elapsed time required to access the first and last attributes, respectively, from a collection of 2000 attributes. In all cases, our linear library outperforms our Berkeley DB-based library.

Were it feasible to avoid an explicit open and close operation for each file access, a Berkeley DB-based implementation would provide faster access. However, our assumption that attributes are accessed individually and independently combined with our design goal that the attribute interface be stateless (e.g., no explicit setup or teardown is required when working with file attributes) effectively mandates that an open and close operation is required for each invocation of the interface.

2.2 Integration into Linux Systems

File attribute inconsistencies may occur because our implementation of file attributes is library-based as opposed to a fundamental part of the file system. Some attribute inconsistencies include:

- a file may be moved to a different directory than its attributes
- a user may own a file but not the associated attributes
- a user may have different permissions for a file than for the file's attributes
- a file is deleted but its attributes remain
- a file is hard or soft linked but its attributes are not
- a file may be renamed, disconnecting it from its attributes

These attribute inconsistencies may be caused by directly interacting with the file system through the file system utilities, which are not aware of the associated attribute file. For example, executing the *mv* command on a file with attributes would only move the file leaving the associated attribute file behind. Programs implemented to be explicitly attribute aware could avoid attribute inconsistencies by performing each necessary action on both the file and the associated attributes. However, rewriting all programs to be attribute aware is impractical, and impossible in most cases because the source code is not readily available. To prevent attribute inconsistencies and provide a seamless presentation of attributes to the user, we integrated a portion of the attribute library into the Linux kernel.

To integrate the attribute library with the Linux kernel we developed a kernel module that provides a layer between the system call interface and the actual system calls. In this discussion we refer to this kernel module as the attribute module. The attribute module intercepts the appropriate file system calls and splits the original call into two system calls, one to manipulate the file and one to manipulate the associated attributes. Figure 3 depicts this interaction. For example, when the attribute module intercepts `chmod` it will determine the attribute file associated with the file name parameter and call the non-attribute aware `chmod` system call for both the file and the attributes. In this way the file will not become inconsistent with or disconnected from its attributes. Analysis of the Linux kernel system calls revealed nine file related calls that required intercepting and splitting to prevent attribute inconsistencies.

We compared the run times of the attribute-aware system calls to their standard counterparts to ensure that the cost of our changes is not prohibitive. The results are shown in Table 2. This comparison shows that kernel module prevention of attribute inconsistencies is in general only slightly worse than performing the system call twice. Two exceptions, `fchmod` and `fchown`, are over three times as slow as their standard counterparts. This occurs because these systems calls take an open file descriptor, whereas our attribute library requires file names. The extra computation needed to determine the file name from the file descriptor accounts for the longer times of `fchmod` and `fchown`. While at first glance it may seem that the new costs of `fchmod` and `fchown` are prohibitive, they are still slightly faster than their counterparts `chmod` and `chown`, which perform the same

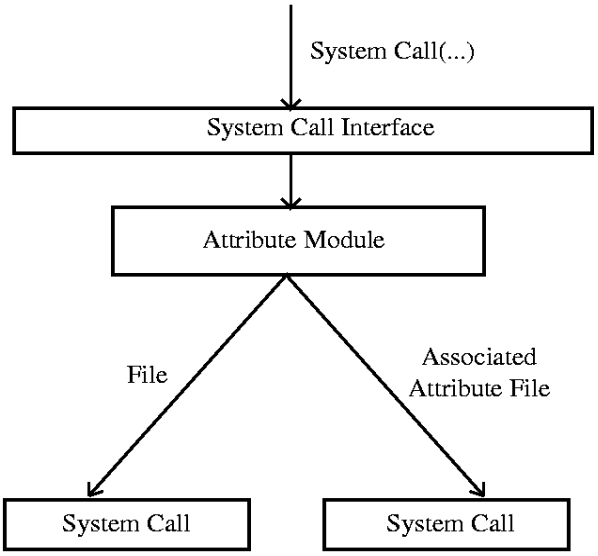

Figure 3 Attribute Module splitting a system call to prevent file attribute inconsistencies.

functions but take the file name as a parameter. This means that in our attribute-aware system there is not a significant advantage to opening a file before changing its mode or owner.

There are some programs that the attribute module cannot prevent from causing attribute inconsistencies. These programs execute a series of system calls that in and of themselves do not cause inconsistencies, but specific combinations of these system calls can create attribute file inconsistencies. An example of this type of program is *cp* -- executing *cp* will copy a file but not its attributes. *cp* opens the source file, reads the data from the source file, creates the destination file, and writes the data from the source file to the destination file. It is difficult for the attribute module to distinguish the series of system calls generated by *cp* from any other series of innocuous opens, reads, and writes. Therefore, the attribute module is unable to split the series of operations correctly to prevent attribute inconsistencies. Since *cp* is an important, heavily-used shell program with available source code, we modified it to prevent attribute inconsistencies associated with our implementation of file attributes. The *cp* source code was changed to perform a copy operation on the file and a similar copy operation on the associated attributes. With the modification of *cp* and the insertion of the attribute module, the standard file utility programs are unable to cause attribute inconsistencies.

In practice, programs that the attribute module cannot prevent from causing attribute inconsistencies are not a significant problem. One of the system call these programs must execute in order to cause an inconsistency is `open` with a parameter to create a new file. The inconsistency occurs because the new file does not have attributes associated with it. We feel that in the common case the user would not want attributes implicitly associated with a new file. In our implementation, a program must explicitly associate attributes with a new file by setting the new file's attributes through the attribute library interface.

System Calls	*Normal*	*Attribute Aware (no attributes)*	*Attrib. Aware*	*System Calls*	*Normal*	*Attribute Aware (no attributes)*	*Attribute Aware*
link	18762	25009	39526	*lchown*	8057	13442	17984
symlink	24358	52553	53617	*unlink*	17000	22379	34384
rename	27668	59076	59054	*fchown*	5270	12373	17168
chmod	10936	16069	20723	*fchmod*	5063	12589	17335
chown	9001	13928	17980				

Table 2: All times are in nanoseconds. Normal column represents the times for a single system call with the unmodified Linux kernel using the ext3 file system. Attribute Aware (no attributes) column represents the run-time of these system calls when the attribute module is inserted but a file has no attributes associated with it. Attribute Aware column shows the run-times for system calls performed when the attribute module is inserted and the file has attributes associated with it.

An advantage of implementing file attributes with a library that depends on a kernel module is that any Linux system can become attribute aware simply by installing the library and inserting the kernel module, regardless of the underlying file system. Similarly, an attribute aware Linux system can revert to normal operation simply by removing the kernel module and deleting the library.

3. Attributes and Search

We would like to search file content in a type-independent manner. More specifically, we would like to search for information in a file without regard to its underlying structure and encoding. In reality, there are many difficult, unsolved problems that prevent this from being feasible. For example, application-independent searching requires that indexing and searching tools have semantic understanding of the files being indexed. In addition, a search system needs to translate a query given by a user into an appropriate form that can accurately be searched for among the various types of data being indexed. In this section, we describe an approach that avoids these problems by using attributes to provide an approximation of file-content searching.

Ideally, an attribute-based search system would provide a mechanism for several types of searches to be performed. Some examples are as follows:

1. A client program or user can specify an attribute name and/or value and search for all files containing an attribute with the specified name and/or value. For example, searching for the attribute "author" would give all documents having the an attribute named "author". Searching for the attribute “author=Abraham Lincoln” would return all files containing the attribute "author" that has value "Abraham Lincoln".

2. A client program or user can provide only a specific value and find all <file,attribute-name> pairs that have the requested value. Searching for the integer value "2000", for example, would return all files and associated attribute names that have as a value the integer 2000.
3. Applications can store portions of the files they manipulate in attributes, which can then be independently indexed and searched. For example, a word processing program may store some or all of a document's text in an attribute, providing search capability if the document itself is stored in a binary format.[2]
4. Some files, such as graphic images, do not have readily searchable formats. If annotations such as captions or descriptive text are stored in attributes, then searching for image contents can be approximated by searching among the annotations.

In this section, we discuss our approach to designing and building an attribute-based search tool that enables many of the queries listed above to be performed. We first describe the issues and design trade-offs we faced, followed by a description of our implementation. Finally, we present two cases studies that illustrate the capabilities of our prototype solution.

3.1 Motivation and Analysis

Searching for data in an information retrieval system is a two step process. First, indexes are constructed over the search data, if they do not already exist, in a manner that can be efficiently searched. Next, the search is performed by accessing the indexes to find the queried data. A file-content search system implements these same two steps, although the process is complicated by the need to index and search files of arbitrary content and format. In general, placing the burden of semantic file understanding on the indexing and searching mechanisms is troublesome, as it requires the indexer and search tools to be extended to have syntactic and semantic knowledge for every type of file to be searched.

Our approach is to approximate file-content searching by fixing a standard “searchable format” and placing the burden of file content understanding on applications that manipulate files, where it exists already. In other words, we define a generic data format readable by the indexer, and we require applications to provide the searchable data in this generic format for the files they manipulate. We use a file's attributes as the storage repository of its searchable data, and we perform indexing and searching on attributes that satisfy our data format. Thus, operationally, we have reduced the problem of file content searching to file attribute searching.

This partitioning of responsibility follows naturally. Syntactic and semantic file knowledge is naturally present in the applications that create and modify the files we are indexing—after all, they are the producers of the files and give them their structure and content. Applications understand the meaning of the files they produce, whereas in principle an indexer sees only a delineated sequence of bytes. Furthermore, this approach allows the

2 Giampaolo [4] reversed this idea and took it to the limit, suggesting that a word processing application store the document text as a plain text file with all the markup and formatting stored in attributes.

indexing and searching tools to remain simple and type-agnostic. Searching for information in a new file type simply requires that the producing application be modified (or equivalently, a type-specific tool be written) to place the searchable information in the file's attributes. Since the attribute information is stored in a format that an indexer and search tool can parse, no changes to the indexer or search tool are needed.

Obviously, there are disadvantages to this scheme. First, file contents are not necessarily being searched. Instead, applications write the searchable data into attributes, which may not capture the information that is being searched for in all cases. Second, not all file contents can be faithfully represented using these techniques, resulting in loss of accuracy during searches. Nevertheless, this technique does provide a reasonable approximation to file content search that may not readily exist otherwise.

We illustrate these ideas with two examples. First, one can write a tool to read an Adobe PDF document and store its Table of Contents (or the content itself) in searchable form as an attribute. Once the attribute is populated with the this data, the searching tools can query this content as easily as any other attribute content, without requiring any specific PDF format knowledge. Second, a spreadsheet application may elect to store column or row heading for its data in a searchable-format attribute. As with the previous example, this data can be searched without requiring any modifications to the indexing and searching tools.

3.2 Design and Implementation

We implemented an indexer and search tool that work together to provide attribute search capability. Our prototype creates distinct indexes for each of the primitive types: byte, int, boolean, and string. In the case of byte, int, and boolean types, the indexes are essentially secondary indexes on the value fields of the attributes. Given a specific value for a search term, all the attributes (regardless of their names) that contain that value can be efficiently computed and returned along with the their associated file names.

String data is indexed differently. String indexes are implemented as an inverted index of each of the words in the string. Inverted index entries contain <file, attribute> entries for each file and string-valued attribute that contains the indexed word. For a given string-valued attribute, the string is tokenized into a sequence of white-space delineated substrings, and each substring becomes an entry into the inverted-index. Figure 4 shows a sample portion of an index for the string "Four score and seven" in an attribute of file f.txt named “attr”. In this example, there are four entries added to the inverted index--one for each word.

String-valued attributes are the "searchable format" described above that can be indexed and searched. Applications create searchable attributes by writing the data in string-valued attributes. The indexer understands this format and indexes each of the terms in the string as described above.

Attribute indexes are created and managed on a per-directory basis. Each directory has its own indexes that index only the attribute files in that directory. Thus, performing a search in a directory tree requires a recursive search in which each subdirectory's indexes are accessed independently. We chose this design for several reasons. First, we wanted local changes to attribute contents to be reflected locally, so that changes to the searchable

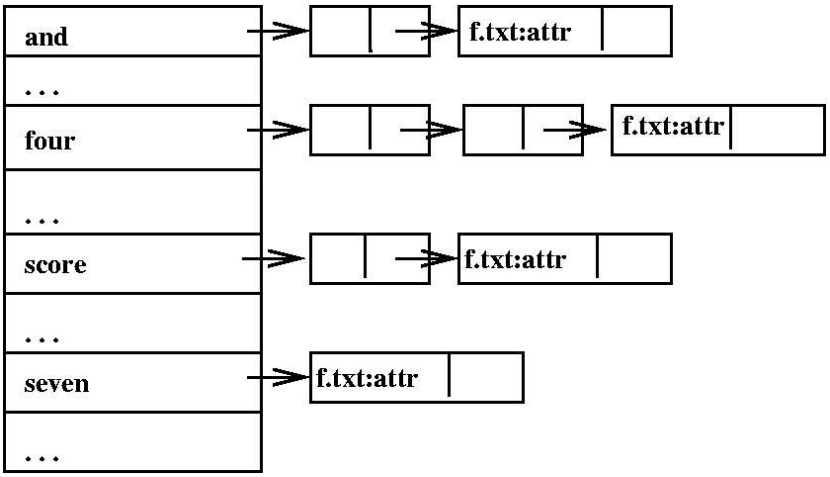

Figure 4 Portion of an inverted index showing entries from the attribute attr with value "Four score and seven", associated with file *f.txt*.

attributes in a directory are restricted to that directory. This provides compartmentalization, as indexes are not responsible for any content outside of their enclosing directory. Second, as discussed below, this scheme greatly simplifies index management in the face of additions and removals from the index. Individual directory indexes can be created and destroyed without affecting the indexes in other directories. Third, faults such as index corruption are limited to the affected directory's index, and consistency is restored simply by rebuilding that index.

Index updates are performed on-demand. When searching a directory, our search tool rebuilds a directory's indexes on-the-fly if it discovers that either (1) the indexes for the directory do not exist, or (2) attribute files in the directory have been created or modified since the indexes were constructed. By so doing, we avoid the need to remove entries from a master index whenever an attribute is deleted, which cannot always be determined in our implementation. This meshes nicely with our per-directory index scheme: during a recursive search, indexes are rebuilt only when necessary and only in the subdirectories required.

As a side note, since our system indexes words, we should apply case folding and stemming techniques to increase search accuracy and flexibility [8]. Currently, our indexer case folds all indexed terms to lower case. It does not perform stemming, although adding a stemmer would be straightforward. Partial-term searching (e.g., searching for “Wash*”) is supported, and our indexing tool indexes attribute names as well, so that all attributes with a specified name can be efficiently retrieved.

3.2 Case Studies

3.3.1 Approximating Image Search

We tested our indexing and search tools on an image library of 950 images, with the goal of assessing the performance of our system and (subjectively) evaluating the effectiveness of our approach. The images in this library form a genealogical photo archive and are stored in a directory tree containing 23 subdirectories. Originally,

```
>query_attrs -r iowa “laura*”
1.match term = iowa
    file = .../2213echodaledriveiowa.jpg
    attr = (caption) “2213 Echodale Drive, Iowa. Babcock Family home.”

2.match term = iowa
    file = .../christmas77.jpg
    attr = (caption) “Christmas 1977, Bettendorf, Iowa.”

3.match term = laurajane
    file = .../davidlaurajohn.jpg
    attr = (people)
             “David Babcock, Laurajane (Babcock) Smith, John Babcock”

4.match term = laura's
    file = .../familymay80.jpg
    attr = (caption)
           “Laura's birthday, May 1980, right before move from ...”
```

Figure 5 Sample image library query using the terms '`iowa laura*`'. The term “`laura*`” is quoted to avoid shell expansion of the wildcard “`*`” symbol.

each image was annotated (with annotations stored in a separate file) with a list of the people in the image and a historical description of it. To make this library suitable for our purposes, we wrote a small tool to recursively traverse the directory structure and store each image's annotations as “people” and “caption” attributes, respectively. Figure 5 gives a portion of the output for a query of the terms '`iowa laura*`'.

Table 3 Gives execution times and file sizes for various facets of this test case. For the query “`washington* john`”, the first row shows the total elapsed time necessary to build all 23 indexes (one for each subdirectory in the library), the second row shows the elapsed query execution time when all indexes are up to date, and the fourth row shows the elapsed time when 25% of the indexes are outdated and must be rebuilt. Even in the worst case, in which all indexes must be rebuilt, the query completes in less than one second.

3.3.2 Searching File Content

In the second case study, we used our attribute and search libraries to perform a full-text search of a library of PDF- and postscript-formatted research papers. To some extent, this case study highlights the file-type independent nature of our tools—we can create searchable indexes for any document type provided the document contains attributes in a searchable format. We wrote tools to extract the textual content from each of the file types and store each page of a file in a distinct attribute of that file, so that a 15 page paper will have 15 associated attributes. This case represents the opposite extreme from the image library, since we are peforming full-text searches rather than searches of annotations. Table 3 contrasts the query execution, index construction, and file sizes of the two case studies.

Search time	*Image library*	*PDF and PS library*
All indexes rebuilt	0.63 sec	38.14 sec
No indexes rebuilt	0.05 sec	0.02 sec
Index construction time (%)	92.64%	99.95%
25% of indexes rebuilt	0.29 sec	6.60 sec
Attribute/Index size		
Total number of indexed files	950	42
Total Index size	1.23 Mbytes	11.32 Mbytes
Total collection size (including index)	239.91 Mbytes	73.73 Mbytes
Index size as % of total	0.51%	15.35%
Attribute file size: avg (std. dev)	180 bytes (116.96)	58122 bytes (27165.06)

Table 3 Index construction, query times, and file size statistics for the image library and pdf library case studies. The image library query is 'washington* john', and the PDF library query is 'synch*'. The first row gives the total query time when all involved indexes are rebuilt, and the second row gives the total query time when none of the indexes are rebuilt. The fourth row lists the query time when 25% of the affected indexes must be rebuilt, illustrating the time savings of providing an index distributed among the subdirectories. All results were collected on a 1GHz Pentium III processor-based machine.

It is interesting to compare the index construction and query execution times of the two cases. Attributes in the document library are on average 300 times larger than attributes in the image library, with roughly 13 attributes per document and only 2 attributes per image. Consequently, our attribute files and index sizes are considerably larger, and index construction takes considerably longer for the document library. However, once the indexes are created, queries proceed rapidly, independent of the size or number of source attributes, as row 2 of the table suggests.

Row 4 of Table 3 highlights the performance advantages of placing distinct indexes in each directory. For this test, we randomly deleted 25% of the indexes in the test case directories and re-ran the queries. During query execution, only the indexes that had been eliminated needed to be rebuilt, avoiding the cost of full index reconstruction.

4. Applications of File Attribute

In this section we present several applications that have been developed or modified to illustrate the increased functionality and convenience that can be gained with per-file attributes.

4.1 Per-File Application Association

Attributes allow users and programs to define relationships between files. More specifically, attributes can associate a file with an application that opens the file. To illustrate this feature of attributes, we implemented *visit*, a program that opens files with an attribute specified application. *visit* reads the *visit-default-app* attribute for a given file then

calls `execv`, passing the value of *visit-default-app* and the file name as arguments. *setvisit* is a simple program that wraps the `set_attr` attribute library function and allows the user to manually associate a file with an application.

While allowing the user to set a file-by-file application preference is convenient, it may be more intuitive to simply open the file with the last application that wrote to it. To this end we modified Emacs to set itself as the *visit-default-app* every time it saves an open file. Emacs encourages customization and exports several function hooks where users can attach their own custom Emacs-Lisp functions. The visit-aware customization of Emacs attaches a function that executes the *setattr* attribute program with the name of the file and the key-value pair <visit-default-app, emacs> every time a file is saved. The visit-aware customization function is less than 20 lines of Emacs-Lisp code, and the added delay of setting the *visit-default-app* attribute is not noticeable to the user.

4.2 Storing File-specific Session Information

Attributes provide the ability to associate extra information about a file with the file itself, including information about how this file should be edited. Some examples of this type of information include: language-specific editing mode, cursor position from the last session, undo stack from the last session, and the location of errors from the last compile.

Emacs performs language specific editing by default. Some of its language-specific features include setting the TAB width, highlighting keywords, and distinguishing comments from code. Emacs determines the language of a file by examining its suffix. Unfortunately, some programming languages share file suffixes: both Prolog and Perl files have the suffix “pl”. A user can change their Emacs settings to associate “pl” with Prolog or Perl, but not both. A user that edits both Perl and Prolog files must set the language specific editing mode manually after opening the file. To address this ambiguity, Emacs allows the user to specify the language type in a commented header at the top of each file. However, were this source code file distributed, a *vi* user may take offense at having an Emacs specific comment cluttering the source code. File attributes provide a simple solution to the problem of determining the language of a file: store the language in an attribute.

We modified Emacs to save the language-specific mode that it was editing a file with at the time it saves the file. In addition, we added functionality to Emacs to check the language of a file every time it is opened. As mentioned above, Emacs exports a rich customization interface. Using this interface, we attached a function that uses the *setattr* program to set the *emacs-mode* attribute to the current Emacs language-specific editing mode every time a file is saved. In addition, we attached a function that allows Emacs to determine the language-specific editing mode for a file by executing the *getattr* program to get the *emacs-mode* attribute every time a file is opened. This customized version of Emacs defaults to determining the language-specific editing mode by the file suffix if there is no *emacs-mode* attribute associated with a file. The Emacs-Lisp code for getting and setting the *emacs-mode* attribute is less than 60 lines of code, and opening and saving a file is not noticeably slower when using the attribute-customized Emacs.

4.3 Application-specific environment customization

Our next example illustrates the ability of attributes to readily associate custom environments with individual applications. There are may situations in which it makes sense to associate custom environment extensions with particular files, rather than making them globally available in the *.login* or other initialization scripts. We describe two here. First, *preloading* allows a custom library to override any function call in a library at will. When a program is first executed, the library specified by the LD_PRELOAD variable is loaded after all other dynamic libraries, allowing it to override any library functions previously imported. This technique can be used to perform dynamic instrumentation and memory debugging, for example. By associating this environment variable as a custom environment extension, we can ensure that this library affects only the intended applications.

Second, with the introduction of the Native POSIX Thread Library [2], older applications that depend on prior pthread semantics may break when executed. This problem is avoided with the LD_ASSUME_KERNEL environment variable, which specifies the thread behavior that should be used. By binding this variable as an environment extension attribute, we provide a way for thread-sensitive applications to run in the required environment without imposing this environment on other applications.

To test this idea, we modified the *bash* shell source code to extend the environment passed to an application with the values contained in the *shell-env-list* attribute for that application, if it exists. The following is a sample shell-extension:

```
<shell-env-list,"LD_PRELOAD=./LeakTracer.so;TRACE_LEVEL=DEBUG;PATH%=/u/s/m/smithr/apps">
```

This attribute provides three environment variable customizations: specific values for environment variables LD_PRELOAD and TRACE_LEVEL are set, and the PATH variable is augmented with the provided path (the '%' marker is used to indicate that an existing environment variable should be extended rather than replaced).

Whenever *bash* executes a disk-command, as it is called, the shell forks and calls `execve`, passing in the executable name, its arguments, and the exported environment. Our modifications extend the exported environment after the call to `fork` but before the call to `execve`, ensuring that the environment modifications apply to the executing disk-command only.

4.4 Approximating File Content Searching Using Attribute Search

Performing an effective search over binary data is a very difficult problem, and while attributes do not solve this problem, they can help alleviate it. File attributes allow us to approximate binary data searches by searching over the attributes that are associated with a binary data file. Essentially, attributes can describe the data in the binary file, and our search library can search effectively over attributes. To provide a specific example of this approximate file-content search capability we implemented a photo album creation application as a wrapper around our attribute and attribute search libraries.

The Photo Album Maker is a suite of programs for annotating and searching over photographs. Each photo has four attributes: *caption*, *people*, *place*, and *year*. The photos are annotated using the *pamscaption*, *pamspeople*, *pamsplace*, and *pamsyear* programs for the respective attributes. These annotating programs are just simple wrappers around the attribute library function `set_attr` and each program consists of no more than 20 lines of C++ code.

The Photo Album Maker's primary function is picture searching, or photo album creation. A photo album consists of a directory containing photographic files. The album creation program, *pam*, uses the attribute search library to search a repository of annotated photos, and creates a directory filled with symbolically linked photos that match the search criteria. Search criteria for *pam* include values for all of the above mentioned annotations, and in addition, *pam* takes advantage of the attribute search library's ability to search on partial values. An example of a partial search criteria is *year* = 197*, this criteria would cover any photo annotated with a year from the 1970's. *pam* is also capable of creating an album from any combination of annotated search criteria. For example, *pam* can create an album of photos containing Ted from the year 1989, by combining the search criteria *year* = 1989 and *people* = Ted. Within *pam***,** multiple criteria searches are created from the intersection of sets returned from executing the `query_db` function of the attribute search library for each individual criteria.

pam was implemented using the attribute search library and consists of less than 400 lines of C++ code. A substantial portion of the code written for *pam* is to handle recursively searching the subdirectories of the photo repository and creating the intersection set of the single criteria searches. The remaining code parses the command line arguments, and creates the album directory and its symbolic links.

5. Related Work

NTFS[6] supports user-defined alternative file data streams that can be used as file attributes; a given file may have several of these data streams. These user-defined data streams are named by concatenating the stream's name to the file name. For an user-defined stream *author* associated with a file named “file.txt” the alternative data stream would be named “file.txt:author”. NTFS provides no special functions for accessing alternative data streams; they are opened, closed, read, and written like a file.

In NTFS, every file has fixed sized record in the Master File Table that indexes the file's data streams. To reduce a file's on-disk size, NTFS stores a file's data streams within the file record if there is enough space. Any data stream that does not fit within the file record is assigned one or more sector clusters that are located outside the MFT. A single sector cluster is at least 512 bytes long.

BeFS, [4] the file system for BeOS, has built-in file attribute support. Small attributes are stored in a special small data area of the inode, while attributes that do not fit in the small data area are stored in an invisible directory structure that “hangs-off” the inode. Attributes are stored in the invisible directory as individual files, and each attribute has its own inode and data blocks. A minimum of four blocks are needed to store an attribute that does not fit within the small data area of the inode. One block is needed for the invisible attribute directory inode.

At least one block is needed for the attribute directory's data. A third block is needed for the attribute's inode, and at least one more block is needed to store the attribute value.

BeFS provides access to attributes through special system calls for reading and writing attributes. These system calls take as parameters an open file descriptor and the attribute name. BeFS also provides file attribute indexes and a search mechanism over the indices. The user must specify which attributes to index, with the exception of file size, name, and last modification time, which are indexed automatically by BeOS. Indexing is over the entire file system with one index file for each indexed attribute. The attribute index files are stored in a single invisible directory whose position is known by the super block. The index file for a given attribute is updated every time an instance of the attribute is written.

XFS [1] is a Linux and IRIX based file system that has file attribute support. File attributes are stored within the inode in place of file data extent pointers if both the attribute and the file size are small enough. All attributes associated with a file that do not fit in the inode are stored in a single alternate data stream that is indexed from the inode. XFS provides system calls to get, set, delete and list the attributes of a file, given either the file name or an open file descriptor. In addition, XFS provides two name spaces for file attributes: a system name space writable only by the super user, and a user name space that has the same permissions as the associated file. For example, the ACL for each file is stored as an attribute in the system name space, while an application-defined *author* attribute would be stored in the user name space.

6. Conclusion

This paper has explored the potential benefits of extending file systems to support per-file dynamic attributes. Even with a simple <key, value> pair abstraction, this capability provides a level of convenience that allows users and programmers to readily define and manage custom file meta-data. Experience with the sample applications described above indicates that the uses of attributes are many, and files and applications can very naturally become attribute-aware. Additional applications of file attributes include:

- organizing ROM games into categories based on search criteria
- extending a web browser to store the URL as an attribute of every file downloaded
- extending a GUI desktop environment such as KDE to implement the functionality of *visit*
- storing resource requirements such as memory and processor needs with applications executing in a distributed environment
- storing change histories with files, so that previous versions can be retrieved as necessary

In some cases, our lightweight <key, value> pair abstraction is not rich enough, and a stronger model may be required to accurately represent an attribute. Storing a file's version along with its history illustrates this problem, in which each entry must store three pieces of information: the key, the version number, and the changes from the previous version. Although it is possible to squeeze these items into our abstraction, doing so is unnatural

and gets messy as more and more versions are squeezed into a single attribute value. Alternative solutions may be to provide multivalued keys or to use a relational model. These solutions need to be explored.

With regard to file content searching, initial results suggest that searching file content may be reasonably approximated using attributes to store searchable data. Our case studies have focused on two extremes—approximating image data content and performing full-text searching—showing that our approach is technically feasible. However, our biggest assumption, that any given set of data can be effectively represented in a searchable format and meaningfully searched, is qualitative and largely untested. It would be interesting to test this idea on a larger scale, with more and varied file types.

One of our purposes has been to demonstrate the usefulness of an attribute-supported file system. To the extent that file systems in use do not contain such support, such usefulness cannot be realized. To overcome this issue, our attribute implementation could be modified to match the interface for XFS's attribute implementation. By using a common interface, application developers can bundle our attribute implementation with their applications for Linux systems not using XFS, and supply the same functionality minus the library for systems with XFS. Our library may be able to provide a stop gap until more Linux file systems add attribute functionality.

7. References

[1] C. Anderson, “xFS Attribute Manager Design.” *SGI Developer Central Open Source*, [Online document], (1993 October), Available at HTTP://oss.sgi.com/projects/xfs/design_docs/xfsdocs93_pdf/attributes.pdf

[2] U. Drepper and I. Molnar, " The Native POSIX Thread Library for Linux," *RedHat*, [Online document], (2003 January), Available at HTTP: http://people.redhat.com/drepper/nptl-design.pdf

[3] D. Embley, Object Database Development, Reading, Massachusetts: Addison Wesley Longman, Inc., 1998.

[4] D. Giampaolo, Practical File System Design: With the Be File System, San Francisco, California: Morgan Kaufmann Publishers Inc. , 1999.

[5] R. Grimes, "Code Name WinFS: Revolutionary File Storage System Lets Users Search and Manage Files Based on Content," MSDN Magazine, vol. 19, no. 1, January 2004.

[6] “NTFS Basics,” [Online document], [2004 April 28], Available at HTTP: http://www.ntfs.com

[7] M. Olson, K. Bostic, and M. Seltzer, " Berkeley DB," *SleepyCat*, [Online document], (2000 June), Available at HTTP: http://www.sleepycat.com/company/technical/whitepaper.pdf

[8] I. Witten, A. Moffat, and T. Bell, , Managing Gigabytes, New York: Van Nostrand Reinhold, 1994.