LiFS: An Attribute-Rich File System for Storage Class Memories

Sasha Ames Nikhil Bobb Kevin M. Greenan Owen S. Hofmann
Mark W. Storer Carlos Maltzahn Ethan L. Miller Scott A. Brandt

Storage Systems Research Center
University of California, Santa Cruz

Abstract

As the number and variety of files stored and accessed by a typical user has dramatically increased, existing file system structures have begun to fail as a mechanism for managing all of the information contained in those files. Many applications—email clients, multimedia management applications, and desktop search engines are examples—have been forced to develop their own richer metadata infrastructures. While effective, these solutions are generally non-standard, non-portable, non-sharable across applications, users or platforms, proprietary, and potentially inefficient. In the interest of providing a rich, efficient, shared file system metadata infrastructure, we have developed the Linking File System (LiFS). Taking advantage of non-volatile storage class memories, LiFS supports a wide variety of user and application metadata needs while efficiently supporting traditional file system operations.

1. Introduction

File system interfaces have changed relatively little in the three decades since the UNIX file system was first introduced. Metadata in standard file systems includes directory hierarchies and some fixed per-file attributes including file name, permissions, size, and access/modification times. While primitive, these interfaces have served well.

In the same time frame, demands on the storage subsystem have increased both quantitatively and qualitatively. Storage systems have grown, the amount of storage accessed by individual users has increased, and the variety of data stored has grown dramatically. General-purpose file systems are now used to store tremendous volumes of text documents, web pages, application programs, email files, calendars, contacts, music files, movies, and many other types of data. Although current file systems are relatively effective at reliably storing the data, the increasing size and complexity of the information stored has made management and retrieval of the information problematic. Simply stated, with so much information, it is difficult to find what one really wants. This problem has been addressed on the web with the development of search engines, and it is now often harder to find information on one's own hard drive than on the web.

To address this shortcoming, application developers have been forced to develop their own metadata infrastructures. Email applications, digital photo albums, digital music applications, desktop search applications, and many others have their own file system metadata to enable the organizing, searching, browsing, viewing/playing, annotating, and generally working with specific types of files. Many of these applications have special-purpose code for dealing with the specific properties of the type of data they manage, but they also include code that is not data-specific for organizing, annotating, browsing, etc. Because this code was developed as part of an application, it rarely adheres to any standard, is often not portable, it is difficult to share between applications, users, or platforms, it is typically owned by the company that developed it, and it is potentially inefficient.

Key functions of the operating system are to efficiently provide services that are used by a variety of applications, to abstract away low-level details by providing a useful high-level API, and to facilitate sharing of resources used by multiple users and applications. The wide variety of applications developing their own file system metadata infrastructure shows the need for this infrastructure to be provided by the file system. Recently, researchers (ourselves included) have taken steps in that direction by including additional per-file metadata in the form of $\langle key, value \rangle$ pairs. While useful, this is inadequate to support the needs of the wide variety of applications described above.

We present the Linking File System (LiFS). In addition to the standard file system operations, LiFS provides searchable application-defined file attributes and attributed links between files. File attributes are in the

This paper was published in the *23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2006)*, College Park, MD, May 2006.

form of application-defined $\langle key, value \rangle$ pairs. Links are explicit relationships between files that can themselves have attributes expressed as $\langle key, value \rangle$ pairs to express the nature of the relationship created by the link. These simple additions dramatically change the nature of file systems, enabling a wide variety of operations, providing a rich, shared metadata infrastructure, and allowing applications to focus on managing application-specific data instead of managing things that are best managed by the file system.

In LiFS, all files may contain data and links to other files. Thus, a traditional data file is one with contents and no links, and a traditional directory is one with no contents and directory containment links to other files. Many more interesting examples are possible. For example, a `.c` file can contain links to the `.h` files it includes. An executable can contain links to its source `.c` files, the compiler used to generate it, and the library files on which it depends. A document can contain a link to the application used to edit and view it. With respect to the application-specific infrastructures mentioned above, an email client, a digital photo album, and a music player now all need only provide a GUI for managing their specific type of data and manipulating the attributes of and relationships among the files they each manage, and the file system can take care of storing the attributes and relationships and efficiently supporting their manipulation, search, and other actions.

LiFS is enabled by storage class memories—nonvolatile, byte-addressable RAM—by making the reading, writing, indexing, and searching of such rich metadata fast and efficient. Our prototype is designed with MRAM in mind, but is implemented in Linux using standard DRAM. Our results demonstrate that the performance of LiFS is comparable to, if not better than, that of other Linux file systems, while providing far richer metadata semantics. As a proof-of-concept we have implemented a simple browser that functions alternatively as a file system browser, an email browser, a digital photo browser, or a music browser, depending upon the files and links contained in the file system hierarchy it is exploring. The following sections examine some motivating examples in more detail, discuss the LiFS design, present the details of our LiFS implementation, and show the performance of LiFS in various scenarios.

2. Motivating Examples

Relational links and attributes are surprisingly useful. In this section we will illustrate their utility in finding and organizing information and in coping with change in computing infrastructure. These areas are of particular interest in the context of rapid growth of personal and enterprise-level data and the emergence of utility computing requiring scalable IT management technologies.

2.1. Information Management

2.1.1. Searching Finding data on the vast Worldwide Web is often easier than finding the same content on a local hard drive. The same search engine technology that does well on the Internet typically performs poorly when applied to enterprise-level file systems. To our knowledge both of these statements are only based on anecdotal albeit common evidence but can be plausibly explained: web links convey relevance and semantic information that turns out to be very useful for searching and presenting search results [36]. However, traditional file systems only convey relationships among files through the hierarchical directory system.

Hierarchical directories are actually a compromise between the need of users to organize their data on one hand and file system designers who aim to reduce the cost of maintaining metadata on the other. This dearth of relationships between files is the primary reason that finding data in enterprise-wide file systems or even in local file systems appears to be harder than on the Internet.

There is in fact a wealth of explicit and implicit relationships among files. Because hierarchical directories make it difficult to express explicit relationships, this information often ends up being stored in application-specific files and obscured by proprietary file formats. There are also implicit relationships such as provenance, application and data dependencies, as well as contexts that may span applications; this information is typically not recorded at all. However, maintaining provenance and context relationships enables powerful search capabilities. For example, two downloaded files that are in some way related on the web, *e. g.*, originating from the same web site, normally are stripped from their context when stored on a local file system. Provenance preserves their relationships and provides important clues to context-sensitive searches.

The history of Internet search engines shows a progression of increasingly sophisticated ways of mining relationships, extending successful searching even to documents with content obscured by proprietary formats. The introduction of rich relationships for files will allow the successful use of advanced search technologies within personal or enterprise-wide file systems.

2.1.2. Repository Sharing Because applications are often the sole maintainer of meaningful relationships among files, repositories are often partitioned. As a result, each repository can only be accessed by

one application and relationships that span repositories are hard to represent. A good example is the common fact that notes, email, calendar entries, and instant message conversations each have their own application-specific repositories with no mechanism to represent relationships between, for example, an email and a calendar entry. Some commercial personal information systems such as Microsoft Outlook [31], and other systems currently under development such as Chandler [34] and Haystack [38] try to alleviate this problem by combining traditionally separate repositories into one application-specific repository. However, this approach only alleviates repository partitioning rather than solving the problem.

Another disadvantage of maintaining relationships on the application level is that it makes integration among applications unnecessarily hard: To maintain data relationships across applications, each application has to know how to communicate with another application's API to access and manipulate data in the other application's repository. Some alleviation of this $n : m$ scaling problem is offered by "glue" languages such as Apple's AppleScript [4] and more recent Automator [5], which are designed to enable end users to automate common tasks spanning multiple applications. However, the resulting scripts still interact with data repositories via the API of applications and quickly become obsolete due to API changes from new versions or substitutions of applications.

LiFS, on the other hand, provides a file-centric (as opposed to application-centric) infrastructure that allows applications to not only store files but to insert relationship information directly into the file system's metadata. All applications that take advantage of LiFS relational links and attributes automatically share one repository and integration among applications only needs to interact with the file system's API.

2.1.3. Navigation Rich relationships between files require more sophisticated navigation tools than a traditional file system browser. File system browsers are specialized for traversing hierarchies but are ill-suited for navigating non-hierarchical graph structures with different kinds of links. Web browsers, on the other hand, are designed to navigate hypertext graphs and have developed mechanisms to handle a variety of link types. For example, a web page can contain image source links that are immediately resolved to inline images while anchors are not resolved but displayed as clickable text. Other examples of link types are references to frames, style sheets, or more complicated Javascript constructs that allow asynchronous requests for updates without reloading the web page, such as those used by Google's Gmail, Maps, and Suggest.

Recent web browser designs offer a high degree of extensibility and the ability to render sophisticated user interfaces. We believe that this evolution of web browsers is not a coincidence but was both possible and necessary because of the Web's complex relational linking structure. There is no one good way to display complex structures. Instead, more or less specialized interfaces for interacting with these structures are adopted as these complex structures become commonplace. We are faced with a similar situation when designing file system browsers once we introduce complex linking structures as in LiFS.

The need to render a wide variety of structures has led to powerful architectures, culminating in Mozilla [45], that offer a very flexible and fast layout engine (Gecko [44]) and extensible component architecture that loads the entire specification of a user interface from files. This user interface specification framework is referred to as XML User interface Language (XUL [47]). It is powerful enough to fully specify the user interface of Firefox and Thunderbird, two instances of Mozilla, one for browsing the web and one for managing email. Furthermore, XUL's sister eXtensible Bindings Language (XBL [46]), allows dynamic modification of parts of the user interface.

Significant in the context of LiFS is the fact that XUL presents a fully implemented framework that allows the linking of file structures to specify how to display these structures. It is now conceivable to integrate formerly segregated navigation and management activities regarding notes, email, instant messages, calendar entries, and software development projects into one "file system browser".

Interestingly, the relationship between XUL components (UI content, skin, and scripts) is specified in RDF files [48]. These RDF files can be directly translated into LiFS linking structures. Furthermore, XUL populates the user interface by specifying queries to one or more RDF data sources. The Mozilla component architecture allows the creation of new RDF data sources. We are in the process of implementing a component that provides LiFS linking structures and file attributes as RDF data source.

In summary, by providing relational links we can leverage file system navigation in LiFS with Web browser technologies and use these technologies to combine file system content in ways that were not possible before because of repository partitioning and applications that are hard to integrate.

2.2. Infrastructure Change Management

Infrastructure change can, and often does, destroy the usefulness of data. However, infrastructures change

all the time: software and hardware get upgraded or replaced and a change as small as a single upgrade of an application can render data useless. This process of continual change is the primary reason for data obsolescence and the number one threat to digital preservation [11]. A special case of infrastructure change occurs when data is migrated. In the following section we will illustrate how relational links can make infrastructure change more manageable.

2.2.1. Obsolescence Users often discover data obsolescence when it is too late. A typical example is someone who has to amend the tax return for the year 2001 while filing for 2003 and has changed computing platforms sometime in 2002. While changing platforms, the user copies the file for the 2001 tax return to the new platform not realizing that each tax year and each computing platform requires a separate application. Two years later the user is unable to read the 2001 file because the retroactively purchased application for 2001 cannot access 2001 files generated on the old platform. The user is left with trying to reconstruct the former infrastructure which might be more difficult than recreating the 2001 file. Thus, the 2001 data was essentially lost.

The example illustrates that even well within typical record life cycles it is difficult to anticipate the consequences of infrastructure change *in time* unless the dependencies of data are made explicit. One way to address the rapid rate of change is to introduce another application for managing the context of data by keeping track of which application version created what data. However, such an application would be as exposed to obsolescence as any other application. File systems, on the other hand, have historically enjoyed low obsolescence and are therefore better suited to manage dependencies over longer time periods.

In LiFS we can represent the dependencies in this example with relational links. This allows the generation of a detailed list of consequences *before* deleting an application or changing the infrastructure in other ways that might cause damage.

2.2.2. Migration To continue with the tax return example, recall that the key failure that led to obsolete data was migrating tax data from one platform to another without realizing (for two years) that not all dependencies on the target platform were satisfied. To solve this problem, the source platform has to communicate data dependencies to the target platform and the target platform needs to figure out how to satisfy these dependencies. This functionality is similar to popular open source package managers which package recognize what other packages need to be downloaded for a given package in order for the software to function.

Import and export of LiFS metadata for communication of dependencies between file systems is still ongoing research. We anticipate that metadata will be exported as RDF, and that the import of RDF into metadata will involve a resolution process that either generates requests for missing files or generates alerts for non-satisfiable dependencies.

3. File System Design

3.1. Basic Goals

We have designed LiFS with the goal of providing attributed relationships between files and enhanced metadata with no perceptible performance overhead compared to traditional file systems. We assume that the storage system LiFS runs on will include a high-bandwidth, higher latency component with large amounts of storage (such as a hard disk), and a lower bandwidth, low-latency component with a small amount of storage proportional to the size of the larger storage. This lower bandwidth, low-latency component could be any type of byte addressable, non-volatile memory such as magnetic RAM [12] or any other storage class memory technology. The larger capacity, higher latency storage is used to hold the user data stored in the file system, while traditional and enhanced metadata resides on the smaller, faster storage class memory.

Three specific features of the file system that allow us to reach our design goals are named links between files, attributes on files, and attributes on links. Attributes are composed of $\langle key, value \rangle$ pairs such as $\langle author, john \rangle$. Files are named according to the name of the link by which they are accessed. For instance if there is a link named bar from source file foo, the target file is accessed as bar. Multiple links between two files may exist as long as they are disambiguated by a unique set of attributes. Multiple links are useful when there are multiple users of on a system, and more than one user or application wishes to have a link between two files with their own set of attributes.

In LiFS, the concept of a set of links from a source file replaces that of a directory. For example, if a file work/document.txt links to a file picture.jpeg, a user can change directories to work/document.txt and find picture.jpeg in the listing. Traditional directories are emulated via zero byte files with links to all member files. For backwards compatibility, a legacy application may set a special STAT attribute on a file in order to access that file as a directory, or vice versa.

System call	Function
rellink	Create a new relational link between files
rmlink	Remove a relational link between files
setlinkattr	Set attributes on an existing link between files
openlinkset	Returns an identifier for a set of links from a source file
readlinkset	Fills in standard directory entry structure with link name and attributes for the next link in a set

Table 1: New file system calls

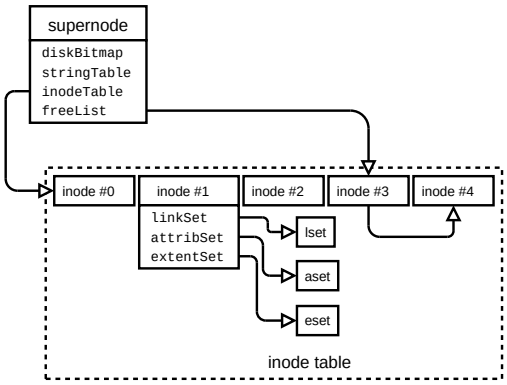


Figure 2: Structure of the LiFS supernode

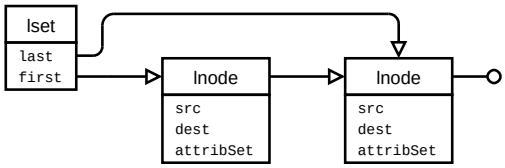


Figure 3: A set of links from a file in LiFS

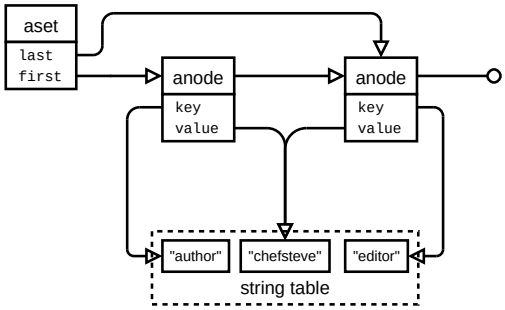


Figure 4: A set of attributes in LiFS

To offer this new functionality, we propose several new system calls, shown in Table 1, to manipulate links and attributes on links. The new system calls have a syntax very similar to that of current calls to manipulate and get information about directories, links, and extended attributes. For file attributes, we implement the standard getxattr and setxattr extended attribute calls.

3.2. In-Memory Data Structures

Traditional file system data structures are optimized for accessing both data and metadata on a high-latency disk. In contrast, we designed the file system data structures in LiFS to take advantage of low latency storage class memories. Traditional inodes have been augmented by link nodes (*lnodes*), attribute nodes (*anodes*) and extent nodes (*enodes*). Because memory access is very cheap, lnodes, anodes and enodes are arranged in linked lists, allowing for trivial insertion and deletion.

Our file system uses a hash table which eliminates duplicate storage of strings. The first time a string is used in LiFS, it is added to the table. When an identical string is later used, a lookup returns a pointer to the string in the string table. A reference count is kept for each string so that unused strings do not remain in memory. The string table is used to optimize string comparisons and searches in the file system. Strings for which there are entries in the string table can be tested for equality with a pointer comparison rather than comparing string data. Additionally, if a string is not in the string table we know that any search on that string in the metadata will not be satisfied.

All data structures necessary for file system usage are referenced from the supernode, shown in Figure 2. The supernode is stored at the beginning of non-volatile memory. The supernode references the string table, a bitmap of allocated blocks, the inode table, and the first free inode. The list of free inodes is embedded within the inode table itself, with each free inode pointing to the next. Each entry in the inode table stores traditional metadata about a file such as mode and size as well as a pointer to three linked lists: an lset, aset, and an eset.

Links in LiFS are maintained in an lnode structure. As shown in Figure 3, an *lset* contains a linked list of lnodes, each of which contains a source inode, destination inode and a set of attributes. The set of attributes is a pointer to an attribute set, or *aset*, shown in Figure 4. An aset contains a linked list of anodes, each of which has a pointer to the string table entry for the key and value of that attribute.

LiFS attempts to allocate disk space in extents—sequential series of blocks. An extent set, or *eset*, shown in Figure 5, contains a linked list of enodes, each of which specifies the limits of a single extent. When LiFS grows a file, it attempts to grow the last extent if possible. If this is not possible, LiFS allocates a new extent and corresponding enode. Free space is found by scan-

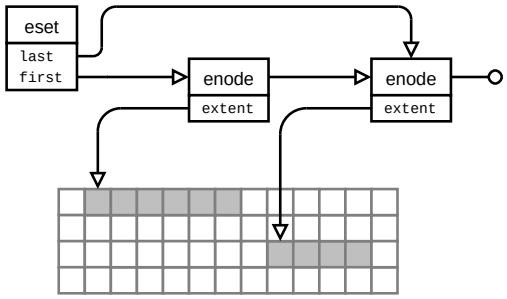


Figure 5: LiFS extent system for disk allocation

ning the bitmap of allocated blocks contained in the supernode using first fit.

4. Implementation

The novel features of LiFS such as relational links require modifications to the Linux Kernel. In Linux, the VFS (Virtual File System) layer provides access to file systems for user mode programs. We added both kernel system calls and VFS functions as required for our new functionality. All new system calls were based on the syntax of similar existing operations.

Our development process was made easier by use of FUSE (File system in User Space). FUSE is a userspace library and Linux kernel module that directs VFS calls to a userspace daemon. Implementing LiFS in userspace through FUSE freed us from the complexities of kernel development, albeit with considerable overhead. We modified FUSE to match our changes to the Linux VFS in order to support the new functionality in LiFS. Once again, the similarity in syntax and function to existing interfaces aided in development. As Figure 6 shows, LiFS resides in a userspace daemon that communicates with the FUSE kernel module through a file in the Linux proc file system. Both our new calls and standard file system calls are passed to the kernel via this channel.

For this paper we were not able to procure sufficient quantities of a non-volatile, byte-writable storage class memory; instead we used system DRAM. The major consequence of this approach is that many operations will run faster than they would in storage class memories. In the future, we will compensate for this speed up with artificial delays. The ability to model slower storage class memories using DRAM will allow us to simulate the performance of LiFS across a wide variety of non-volatile storage.

We assume that a storage class memory would be mapped into an arbitrary segment of the system address space. To imitate this mapping, we allocated several hundred megabytes of system memory exclusively for LiFS data structures. This memory is locked to prevent

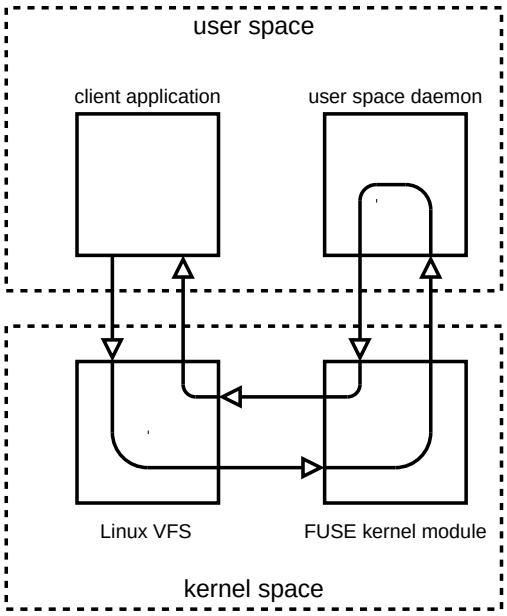


Figure 6: The relationship between FUSE and the FUSE/LiFS userspace daemon

it from being swapped out, based on the assumption that LiFS will have enough storage class memory to hold all the file system's metadata. All memory within this space is allocated using a custom allocator. The allocator allows relocation to arbitrary address spaces, anticipating the possibility that a storage class memory could be mapped into a different part of the system memory address space, perhaps after the system was rebooted or the memory was moved to a new machine. Since our metadata data structures are each a fixed size, we exploit this characteristic by optimizing our custom allocator to preallocate several pools of different constant size chunks. On memory allocation, a chunk of memory is retrieved from a pool of the appropriate size. On deallocation, the chunk is returned to its pool. This allows for very quick turnaround times.

Lookup operations in LiFS require resolution over all links as opposed to solely being resolved over directory structures. Consequently, lookups involve traversing a series of inodes and lnodes beginning at the root inode. Lookup is an iterative process which scans all links originating at its current node. If a link matches the corresponding part of the input pathname, then the target of that link becomes the current node. If no matching link is found, an error is returned.

We have implemented two optimizations to speed up lookups. The first optimization checks that the current path component is in the LiFS string table; if it is not there then it can be safely concluded that the lookup will not be successful. The second optimization is a lookup cache which stores full pathname to inode number map-

File System	Create Files	Read Files
LiFS with FUSE	1.043	0.720
ext2 with FUSE	1.445	1.012
XFS with FUSE	3.998	1.050

Table 7: Create and read time, in seconds, for 15,620 zero-byte files ($k = 5$, $d = 5$, $n = 4$). Times shown are the arithmetic mean of five test runs.

File System	Create Files	Read Files
LiFS with FUSE	12.195	2.027
ext2 with FUSE	2.613	1.679
XFS with FUSE	4.871	1.836

Table 8: Create and read time, in seconds, for 15,620 files ($k = 5$, $d = 5$, $n = 4$) of size 384 bytes. Times shown are the arithmetic mean of five test runs.

pings. Because there is no need for persistence in this cache, it is stored in DRAM as opposed to a storage class memory which stores all other LiFS data structures. The lookup cache is filled on successful lookups. Conversely, before the lookup operation begins, a request is made to the cache for the appropiate pathname to inode mapping.

The core LiFS code has 3,400 lines of C code. Modifications to the FUSE kernel module and userspace libraries required additional code, but these changes were necessitated by FUSE and would not be part of an in-kernel implementation. The small size of our implementation may be attributed to the simplicity of storing all file metadata as in-memory data structures. That the speed of such a simple implementation can compete with more optimized file systems such as ext2 (5,500 lines of code) demonstrates the power of storing metadata in a fast byte writable memory.

The simplicity of our implementation and its speed suggest the potential for significant performance improvements with additional optimization. We already optimize performance and storage requirements using the string table and other data structures which are not feasible in traditional disk-based file systems. We also plan to improve scalability of LiFS by replacing the linked lists prevalent in our implementation with balanced trees.

5. Results and Performance

LiFS was implemented and evaluated on a pair of Sun workstations running the Linux kernel 2.6.9-ac11. Each system was configured with an AMD Opteron 150 processor running at 2400 MHz and one gigabyte of RAM. For testing, LiFS was compared with the XFS and ext2 file system versions included with the Linux kernel. Benchmark testing was automated using Python version 2.3.4, gawk 3.1.3 and GCC 3.4.2. All of the following tests are run on freshly created file systems since the focus of this paper is on the performance of in-memory data structures and not on disk sub-system performance.

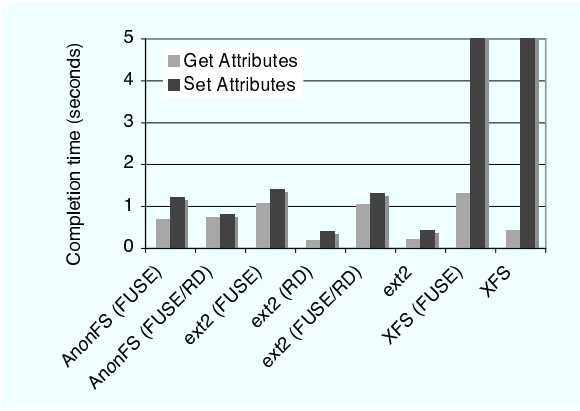
5.1. Standard File System Operations

In order to compare the baseline performance of LiFS, ext2 and XFS, we used a set of six standard file system operations: creating directories, creating files, reading files, setting extended attributes on files, retrieving extended attributes from files and removing directories. Systematic tests of these operations on each file system enabled a fair comparison between LiFS, ext2 and XFS. Each of our tests was run in kernel, through FUSE, on RAM-disk and through FUSE on RAM-disk. LiFS is currently not running in the kernel and XFS cannot be created on a RAM-disk, thus these three scenarios (LiFS in kernel, XFS on a RAM-disk, and XFS via FUSE on a RAM-disk) were omitted. To compare these file systems, we created complete k-ary trees of depth d, with n files per directory each containing a extended attributes.

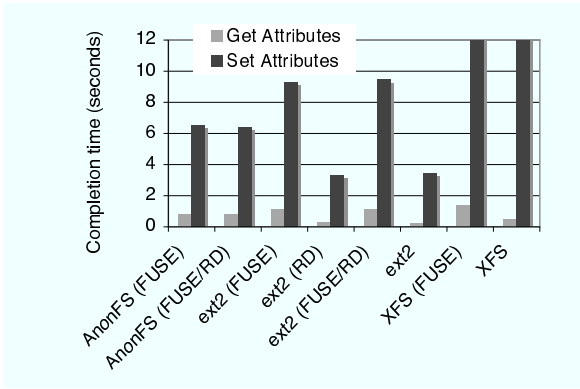
Table 7 shows LiFS performance for creating and reading zero byte files. It is clear that LiFS is competitive with ext2 through FUSE and XFS through FUSE in both tests. Table 8 shows the same tests for files 384 bytes in size. These results show that LiFS is still competitive with ext2 and XFS in the area of file reads but falls behind in file creation. This is due to the extent allocation scheme used in the current implementation of LiFS. For each extent allocation, the current implementation searches the list of free blocks sequentially, starting from the beginning. The results from these two tests indicate that a significant performance gain can be expected from an optimized extent allocation algorithm.

Figures 9(a) and 9(b) show the performance of LiFS, ext2, and XFS creating and retrieving extended attributes on 15,620 files ($k = 5$, $d = 5$, $n = 4$). Figure 9(a) shows the running time for two attributes set on each file, while figure 9(b) shows the running times for twenty attributes per file. In both cases, LiFS performs better than ext2 and XFS through FUSE. When setting twenty attributes per file, LiFS takes roughly 70% of the time required by ext2 under FUSE, while requiring about 73% of the time needed by ext2 to retrieve attributes. These figures also show that the running times for setting and getting file attributes scale well in LiFS as compared to ext2 and XFS.

Directory tree creation performance was tested by



(a) Setting and getting file attributes on 15,620 files ($k = 5$, $d = 5$, $n = 4$, $a = 2$). The times are in seconds and averaged over 5 runs. RD indicates file system on RAM-disk. Note that the XFS runs to set attributes both took over 21 seconds; the graph is cut off at 5 seconds to show details for the other runs.



(b) Setting and getting file attributes on 15,620 files ($k = 5$, $d = 5$, $n = 4$, $a = 20$). The times are in seconds and averaged over 5 runs. RD indicates file system on RAM-disk. Again, XFS, both with and without FUSE, was much slower, requiring over 109 seconds to complete the set attributes benchmark. The graph was cut off to show detail for the other file systems.

Figure 9: Time required to set and retrieve attributes in different file systems.

	Number of Directories		
File System	3,905	19,607	111,110
LiFS with FUSE	0.140	0.7882	4.292
ext2 with FUSE	0.226	1.2315	8.550
XFS with FUSE	0.710	3.3193	25.285

Table 10: Create time, in seconds, for directory trees of various sizes. Times shown are the arithmetic mean of five test runs.

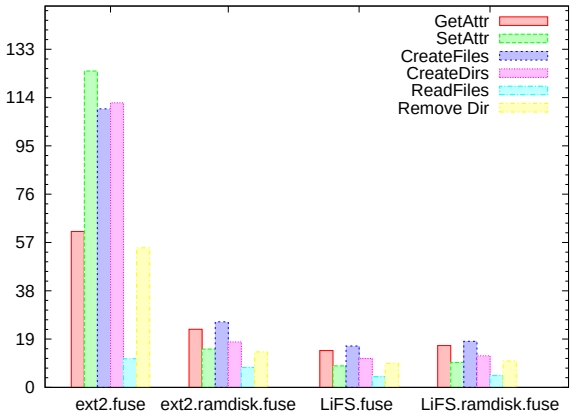


Figure 11: Running times of the 6 file system operations for ext2 and LiFS through FUSE. Results are averaged over 4 runs on a 111,110 directory tree with 1 zero-byte file per directory ($k = 10$, $d = 6$, $n = 1$).

generating directory trees for various values of k and d. Table 10 shows the results of the test, demonstrating that LiFS is comparable to ext2 and XFS. The running time of directory creation scales linearly with the number of directories. This result is comparable to ext2 and XFS. In addition, LiFS is faster than both ext2 through FUSE and XFS through FUSE.

Figure 11 shows the performance of ext2 and LiFS under FUSE and FUSE+RAM-disk. We were unable to create and mount an XFS file system on a RAM-disk, thus XFS is omitted. Since the file system is recreated for each test, the first iteration was working with a cold cache. To avoid extreme shifts in our averages, we discarded the first iteration and averaged over the remaining four. The results displayed in Figure 11 were taken from 4 runs of the 6 file system operations over 111,110 directories with 1 zero-byte file per directory ($k = 10$, $d = 6$, $n = 1$). It is clear from the tests of ext2 that a great deal of overhead is incurred when all of the operations are performed on disk. Because the operations performed in figure 11 use metadata exclusively, LiFS performs better than ext2 on RAM-disk in all cases.

5.2. LiFS Specific Operations

Performance testing of the LiFS specific operations focused on three operations: creating a link between two files, creating attributes for links and removing links. To test the performance of these operations, we created a directory tree and constructed a random graph, where the vertices are files and the edges are attributed links. A directory tree is built in the same manner as in our other tests. The links are then created using a ran-

Operation	Attributes per Link 2	30
Create Links	1.073	1.054
Create Attributes	1.148	2.751
Remove Links	1.086	1.288

Table 12: Time in seconds to create 15,620 random links over a directory tree ($k = 5$, $d = 5$, $n = 4$) with 2 and 30 attributes on each link.

dom number generator and a list of all of the files in the directory tree.

We tested the performance of link and link attribute creation on a directory tree having 15,620 files ($k = 5$, $d = 5$, $n = 4$). After creating the tree, 15,620 links are created between randomly selected files. Table 12 shows the time required to perform a variety of link operations for different numbers of link attributes. The running times shown in Table 12 are averaged over five runs. Link creation is a separate operation from the attribute creation and thus the average running time is not affected by the number of attributes.

The time required to remove links, shown in Table 12, demonstrates the results of a data-structure design choice in the version of LiFS used for testing. When removing a link between two files, a $\langle key, value \rangle$ pair is specified to unambiguously identify a link. This is due to the fact that LiFS allows for multiple links to be made between the same two files. The $\langle key, value \rangle$ pairs are currently stored in a linked list data structure and thus attribute retrieval must, on average, search through half of the link's $\langle key, value \rangle$ pairs. This results in the increased time required to delete a link with 30 attributes versus a link with only 2 attributes.

5.3. SSH Compile Performance

To demonstrate the performance of LiFS in a practical scenario, we compiled OpenSSH version 4.1 using GCC 3.4.2. As Table 13 demonstrates, LiFS running through FUSE is competitive with other file systems in tests both with and without a RAM-disk. Compiling OpenSSH on a LiFS file system running though FUSE took 24.28 seconds. Ext2 through FUSE took 23.40 seconds and XFS through FUSE required 23.49. In a complex series of operations, such as compiling a non-trivial source tree, completion time is not strictly bounded by the file system. Our results show that in such a scenario the extended capabilities of LiFS do not incur an overhead that introduces a new bottleneck to the system.

The OpenSSH compile times show a relatively minor performance hit when running through FUSE. This is to be expected, as compiling a source tree is not bound by

File System	Avg(sec)
ext2	22.52
ext2 through FUSE	23.40
ext2 through FUSE on RAM-disk	23.95
LiFS through FUSE	24.28
LiFS through FUSE on RAM-disk	23.75
XFS	23.20
XFS through FUSE	23.49

Table 13: Compile times for OpenSSH 4.1 (in seconds) on a variety of file systems. Times reflect the arithmetic mean of five trial runs.

Operation	ext2-FUSE	ext2	Speedup
Create Tree	0.2262	0.080	2.828
Create Files	2.613	1.146	2.280
Read Files	1.679	0.474	3.542
Operation	XFS-FUSE	XFS	Speedup
Create Tree	0.709	0.650	1.091
Create Files	4.872	4.282	1.138
Read Files	1.837	0.541	3.396

Table 14: Speedup ratios for selected operations comparing file systems through FUSE versus the same file system without a FUSE layer. Times shown are are the arithmetic mean of five test runs.

disk performance. A more dramatic example of performance gained by eliminating the FUSE layer is shown in Table 14. Reading files without the FUSE layer was over 3.3 times faster for both XFS and ext2. For file and directory tree creation the ext2 file system showed a much more dramatic performance gain compared to XFS. It is therefore difficult to speculate on the exact performance gain that can be expected by eliminating the FUSE layer in LiFS. However, we can assume based on our test results that the gain would be nontrivial.

Another source of potential performance increases for LiFS involves the data structures utilized in the current implementation. The version of LiFS utilized in the testing presented here makes extensive use of linked lists. This includes frequently accessed information such as attribute lists. This will obviously incur a performance penalty when accessing an attribute as, on average, half of the list must be traversed. This behavior is demonstrated in the amount of time required to remove a link as shown in Table 12. There is no design restriction which prevents the use of balanced trees in place of linked lists. Thus a performance increase would be expected by transitioning from linked list data structures to balanced tree data structures.

5.4. Discussion

We have shown that the running times for metadata-based operations in LiFS are faster than ext2 and XFS, while those requiring disk allocation are slower. These results are to be expected because our metadata structures reside in memory and the disk-based data structures are not yet optimized. The LiFS-specific operations regarding link and link-attribute creation have also shown to scale well and run in a reasonable amount of time despite our unoptimized implementation. We have shown that LiFS performs well in a real-world situation, by comparing the running times of a OpenSSH build on LiFS, ext2 and XFS. These results indicate that LiFS can be used in place of most existing file systems without incurring additional overhead. In this section we have also discussed the overhead associated with FUSE. We expect that an implementation of LiFS running in-kernel with optimized disk allocation algorithms be significantly faster than traditional file systems. In addition, the new operations introduced by LiFS should be similar in speed to existing operations.

6. Related Work

The concepts we use in the current and previous designs for LiFS borrow from various research areas ranging from semantic file systems to databases and the Web [2]. We first look at file systems with queryable metadata, such as semantic file systems, and file systems designed to run in nonvolatile memory. We then touch upon upcoming advanced commercial systems. Finally we look at the Semantic Web and archiving, and how they try to convey knowledge more accurately and for the long term.

6.1. Semantic and Other Queryable File Systems

The Semantic File System [21] was originally designed to provide flexible associative access to files. File attributes, expressed as $\langle key, value \rangle$ pairs, are extracted automatically with file type-specific transducers. A major feature of this work is the concept of virtual directories, in which a user makes an attribute-based query and the system creates a set of symbolic links to the files in the result set, providing access that crosses the directory hierarchy. A similar file system, Sedar [29], is a peer-to-peer archival file system with semantic retrieval. Sedar introduces the idea of semantic hashing to facilitate semantic searching and reduce storage and performance costs.

The Inversion File System [33] uses a database to store both file data and metadata. The database also provides transaction protection, fine-grained time travel, and instantaneous crash recovery. Each file is identified by a unique ID, but also has a name and directory associated with it. Moreover, Gupta, *et al.* [22] cite the difficulty of managing different but related sets of files as motivation for their fan-out unification file system, in which fan-out unification refers to merging two directories, and implicitly, entries in a directory are treated as members of a set.

The Logic File System (LISFS) uses a database to support queries for sets of files in the system [35]. Database tables are composed of mappings from keywords to objects. The contents of a directory is the set of objects that meets the criteria of the relation in a query.

Like the above mentioned file systems, the use of attributes in LiFS allows a user to perform expressive queries to locate files. However, these file systems all use secondary storage for metadata, either with or without a database, and must operate under such performance constraints. Additionally, none contain a linking mechanism that supports attributes, thus allowing inter-file relationships that can express the structure of the Web or establish data provenance and history.

6.2. In-Memory File Systems

Douglis, *et al.* compared various storage alternatives for mobile computers and found that flash memory exhibited low power consumption while still providing good read and acceptable write performance [16]. Their focus however was on power consumption and assumed traditional file system functionality.

Conquest [50] utilizes persistent RAM for storage of metadata and small files. Unlike HeRMES and LiFS, which plan to utilize MRAM, Conquest has explored the use of battery-backed DRAM as its form of persistent RAM. The eNVy system and the work of Kawaguchi *et al.* explored the use of flash memory as a primary non-volatile memory storage system [52, 24]. All of these these systems, however, present a traditional file system that lacks the advanced file system features facilitated by utilizing persistent RAM. The HeRMES system [32] proposed that magnetic RAM (MRAM) be used to store file system metadata, and proposed different metadata structures to take advantage of MRAM; however, HeRMES was not implemented, so many of the ideas remained untested.

File system on persistent memory presents a new set of challenges compared to magnetic disk storage. To this end much research has been performed to overcome these challenges. David Lowell's Rio Vista project provides atomicity for memory access operations in the form of transactions [27] while the Journaling Flash File System (JFFS) takes a slightly different approach with a

log-structured file system for flash memory [51]. Edel, *et al.* [17] noted that metadata overhead for a file system is 1–2 percent and suggested the space requirements could be reduced using various compression schemes and algorithms, demonstrating that inode storage space could be reduced by an order of magnitude. They also found that there was no significant difference in performance compared to non-compressed metadata.

Database performance has also been an area that researchers have hoped to improve through persistent memory. Molina and Salem have explored various ways that memory residence optimization could improve data access performance [20].

6.3. Advanced Commercial File Systems

WinFS is Microsoft's in-development file system, for which they have published a preliminary description of planned features[1]. WinFS appears to be a marriage of a database for metadata and NTFS for file stream performance. WinFS treats the contents of the file system as *Items* that cover a full range of granularity from simple descriptions to collections such as folders. The database backing allows SQL-type queries, XPATH searches [14], and the use of Microsoft's OPATH, a query language designed for a directed acyclic graphs [39].

Apple Computer's Spotlight is a metadata and content indexing system integrated into the HFS+ file system [6]. As with WinFS, metadata is stored in a database; Spotlight indexes file content and includes the results in the database as well. Apple's approach could benefit from a LiFS-like linking mechanism with metadata allowing relationships between content to be expressed; Spotlight currently only allows indexing on files, not the links between them. The Linux counterpart of Spotlight is Beagle [1] which provides an API and a plugin infrastructure for new file types and takes advantage of Inotify [18], a file system event service recently merged into the Linux kernel.

Sun Microsystem's ZFS allows administrators to configure individual file systems for users or application, all allocated from a single pool of storage [3]. Like ZFS, LiFS allows for the definition of unique and dynamic file systems per user or per application by virtue of links with attributes. Thus, it is possible to create file systems on demand utilizing either system. However, ZFS does not contain the rich metadata constructs present in LiFS.

[1]This comes with the caveat that all is subject to change. Further information is available at http://longhorn.msdn.microsoft.com/portal_nav.htm

6.4. The Semantic Web

The original World Wide Web has expanded upon the ability of traditional documents to convey knowledge by adding links. Within the World Wide Web and hypertext documents in general, links allow readers to automatically traverse from one document to another when the document refers to the other. The Semantic Web expands upon this by allowing the links themselves to contain information about that particular relationship from one document to another. On top of that basic framework, one may devise ontologies to further convey knowledge in ways not possible through previous means. [10].

To make the Semantic Web possible, authors at the W3C have been developing various standards for its implementations in a way analogous to the standardized HTML and HTTP for the World Wide Web. The group of Semantic Web standards fall into layers, with URI and Unicode on the bottom, XML, name spaces, and schemas making up the self-descriptive document layer in the middle, and the RDF layer on top, which provides a common framework for metadata across applications. Atop the three bottom layers are additional layers for ontology vocabularies, logic, proof, and trust [9, 25]. The ontology layer has room for different attempts to devise languages in which to describe ontologies, such as OIL [19].

We envision three potential expectations for the Semantic Web. For humans, it may be a readily accessible universal library. Moreover, and in line with the ideas of its inventors, the Semantic Web has increased potential for machine processing of its contents, and this introduces the other two perspectives: the knowledge navigator and the federated knowledge or database [30].

Whereas the Semantic Web allows for the addition of richer metadata for the World Wide Web, it does so on a global scale. LiFS allows for the same depth of knowledge representation through its links and attributes. It accomplishes this within the scale of the local file system, which to many users, contains the data on which the semantics of relationships matter most. Additionally, the Semantic Web RDF format can be basically broken into tuples of a subject, property, and object. Links within LiFS likewise contain a source, attributes and a target. Thus, we can express the same relationships locally that are possible given the richness of the Semantic Web. Based on their similarities, LiFS could make an excellent file system or storage layer for Semantic Web data.

6.5. Digital Preservation

Digital objects do not survive unless one makes a conscious effort to preserve them. This is in contrast to artifacts such as books, papyrus, and cuneiform tablets which exist until someone or something actively destroys them. One reason for the ephemeral quality of digital media is unreliable physical media: the shelf life of magnetic tape, hard drives, and CD-Rs can be less than a decade [13, 37, 7]. However, the loss of data due to unreliable media pales in comparison to the loss of data due to the rapid obsolescence caused by technological change. Moreover, digital objects are often related to other digital objects that might change names or disappear entirely [11]. There are now large national and international efforts to address these issues; these efforts aim to provide standards for an exhaustive list of aspects for digital preservation in museums and libraries [49, 26, 43].

Trusted digital repositories [40] adhering to the now dominant international standard of the OAIS Reference Model [15] are an important component of digital preservation. Many systems have been explored which utilize this model [42, 23, 41]. Common challenges of these digital repositories are scalability, interoperability with other repositories, and efficient workflow support for the entry of large numbers of digital objects.

All these digital repositories are designed for use on an institutional level. However, the combination of unreliable storage and obsolescence unintentionally destroys much of digital media long before it can be considered for digital libraries. Personal correspondence and images that survived from earlier times form a significant part of our cultural heritage [28]. Today, personal correspondence in the form of email and chats as well as personal photos and movies are largely kept on home computers that neither meet standards nor follow practices of national digital libraries and are therefore unlikely to survive.

The design of LiFS provides the infrastructure to make digital preservation an integral part of file systems. Links and attributes can be used to explicitly represent the dependencies of digital objects on the software infrastructure thereby preventing accidental obsolescence or at least alert users to obsolescence events introduced by a particular change in the software infrastructure. Our hope is that this will make it easier for users to maintain good digital preservation practices.

7. Future Work

Moving forward, we hope to improve LiFS' performance, storage overhead and availability. We plan to replace many of the linked lists in our data structures with faster balanced trees, as well as implement LiFS in the Linux kernel. It has been shown that inode storage space could be reduced by an order of magnitude without sacrificing performance by compressing inodes [17]. We are planning to add such capabilities to LiFS, and also to investigate how file compression can be achieved efficiently. We are also looking at how we can use fault tolerant data structures[8] to increase the reliability and availability of LiFS. Implementing an online file system consistency checker is another area we are investigating in which availability can be improved. Finally, we are investigating ways of providing LiFS rich metadata structure and performanc without the use of storage-class memories.

8. Conclusions

LiFS makes two important contributions. First, it supports far richer file system metadata via file attributes and attributed links between files. Second, it provides a common, high-performance metadata store for applications, further facilitating interactions between disparate applications through the file system.

LiFS' three major advantages over other file systems are its performance, simplicity, and expressiveness. LiFS clearly demonstrates the performance advantages of storing metadata in a storage class memory. Our tests showed that LiFS is faster than ext2 and XFS in it's metadata performance, even when they are running on a RAM-disk. However, our on-disk data storage is currently not comparable to these file system. We are planning to implement a more advanced extent based approach, such as that used by XFS, which we believe will produce significant performance improvements. As previously mentioned, implementing LiFS in the kernel without FUSE should also provide a further performance boost. LiFS also offers a simplicity not approachable by other file systems via our use of simple data structures. This simplicity is evident in the relatively small code base of LiFS compared to other file systems. We believe that this results in fewer bugs as a function of LiFS having fewer lines of code.

The most important reason for using LiFS, however, is its ability to provide a rich environment for expressing inter-file relationships. Just as flat files and UNIX pipes enabled the creation of small, targeted applications such as awk, eqn, and pic that could work together to perform complex tasks, we believe that the addition of highly flexible attributed links to the file system will enable many new domains for applications to cooperate. Individual application developers can take advantage of a common substrate that supports extensive use of linking and attributes to all applications,

allowing them to develop and enhance applications to provide and manage an increasingly interlinked single repository of data.

Acknowledgments

We would like to thank the faculty and students in the Storage Systems Research Center for their help and comments. This research was funded in part by National Science Foundation grant 0306650. Additional support for the Storage Systems Research Center was provided by Hewlett Packard Laboratories, Hitachi Global Storage Technologies, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Symantec, and Yahoo.

References

[1] Main page - beagle. http://beaglewiki.org/Main_Page.

[2] A. Ames, N. Bobb, S. A. Brandt, A. Hiatt, C. Maltzahn, E. L. Miller, A. Neeman, and D. Tuteja. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Monterey, CA, Apr. 2005.

[3] Anonymous. In a class by itself - the Solaris 10 operating system. Technical report, Sun Microsystems, Nov. 2004.

[4] Apple Computer, Inc. AppleScript scripting language for Mac OS X. http://www.apple.com/macosx/features/applescript/, 2005.

[5] Apple Computer, Inc. Automator automation tool for Mac OS X. http://www.apple.com/macosx/features/automator/, 2005.

[6] Apple Developer Connection. Working with Spotlight. http://developer.apple.com/macosx/tiger/spotlight.html, 2004.

[7] Associated Press. CDs, DVDs not so immortal. In *CNN.com*, May 6 2004. last viewed on Jan 9, 2005 at http://www.longnow.org/10klibrary/darkarticles/ArtCDROT.htm.

[8] Y. Aumann and M. A. Bender. Fault tolerant data structures. In *FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 580, Washington, DC, USA, 1996. IEEE Computer Society.

[9] T. Berners-Lee. Semantic web roadmap, Sept. 1998. Available at http://www.w3.org/DesignIssues/Semantic.html.

[10] T. Berners-Lee and E. Miller. The semantic web lifts off. *ERCIM News*, 51, Oct. 2002.

[11] H. Besser. Digital longevity. In M. Sitts, editor, *Handbook for Digital Projects: A Management Tool for Preservation and Access*, chapter 9, pages 164–176. Andover: Northeast Document Conservation Center, 2000.

[12] H. Boeve, C. Bruynseraede, J. Das, K. Dessein, G. Borghs, J. De Boeck, R. C. Sousa, L. V. Melo, and P. P. Freitas. Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures. *IEEE Transactions on Magnetics*, 35(5):2820–2825, Sept. 1999.

[13] F. R. Byers. Care and handling of CDs and DVDs: A guide for librarians and archivists. Report 121, Council on Library and Information Resources and National Institute of Standards and Technology, October 2003. Last viewed on Jan 5, 2005 at http://www.clir.org/pubs/reports/pub121/contents.html.

[14] J. Clark and S. DeRose. XML path language (xpath), 1999.

[15] Consultative Committee for Space Data Systems. Reference model for an open archival information system (oais). Standards Recommendation 650.0-B-1 (Bluebook, Issue 1), CCSDS, January 2002. This Recommendation has been adopted as ISO 14721:2003.

[16] F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 25–37, Monterey, CA, Nov. 1994.

[17] N. K. Edel, D. Tuteja, E. L. Mlller, and S. A. Brandt. MRAMFS: a compressing file system for non-volatile RAM. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 596–603, Oct. 2004.

[18] I. Edoceo. inotify for linux. http://www.edoceo.com/creo/inotify/.

[19] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. Patel-Schneider. Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–44, 2001.

[20] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, Dec. 1992.

[21] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25. ACM, Oct. 1991.

[22] P. Gupta, H. Krishnan, C. P. Wright, M. Zubair, J. Dave, and E. Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01, Computer Science Department, Stony Brook University, January 2004.

[23] K. J. Jon, D. Bainbridge, and I. H. Witten. The design of greenstone 3: An agent based dynamic digital library. Technical report, Department of Computer Science, University of Waikato, Hamilton New Zealand, December 2002. last viewed on Jan 9, 2005 at http://www.sadl.uleth.ca/greenstone3/gs3design.pdf.

[24] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 155–164, New Orleans, LA, Jan. 1995. USENIX.

[25] M.-R. Koivunen and E. Miller. W3C semantic web activity, Nov. 2001.

[26] Library of Congress. Library of Congress announces awards of $15 million to begin building a network of partners for digital preservation. http://www.digitalpreservation.gov/about/pr_093004.html, September, 30 2004.

[27] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 92–101, Dec. 1997.

[28] C. Lynch. The battle to define the future of the book in the digital world. *First Monday*, 6(6), June 2001. available at http://firstmonday.org/issues/issue6_6/lynch/index.html.

[29] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. Technical Report HPL-2002-199, HP Laboratories, Palo Alto, July 2002.

[30] C. C. Marshall and F. M. Shipman. Which semantic web? In *HYPERTEXT '03: Proceedings of the fourteenth ACM conference on Hypertext and hypermedia*, pages 57–66. ACM Press, 2003.

[31] Microsoft Corporation. Outlook software for Microsoft Windows. http://www.microsoft.com/office/outlook/prodinfo/default.mspx, 2003.

[32] E. L. Miller, S. A. Brandt, and D. D. E. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 83–87, Schloss Elmau, Germany, May 2001.

[33] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, San Diego, California, USA, Jan. 1993.

[34] Open Source Application Foundation. What's compelling about Chandler: A current perspective. http://www.osafoundation.org/Chandler_Compelling_Vision.htm.

[35] Y. Padioleau and O. Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[36] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford, Nov. 1998.

[37] M. Pollitt. Ever decreasing circles. In *The Independent*, 21 April 2004. Last viewed on Jan 5, 2005 at http://www.fbia.org/print.asp?ID=27963.

[38] D. Quan, D. Huynh, and D. R. Karger. Haystack: A platform for authoring end user semantic web applications. In *2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, FL, USA, Oct 20-23 2003. Last viewed on Jan 10, 2005 at http://haystack.lcs.mit.edu/papers/iswc2003-haystack.pdf.

[39] T. Rizzo and S. Grimaldi. Data access and storage developer center: An introduction to "WinFS" OPath, 2004. http://msdn.microsoft.com/data/default.aspx?pull=/library/en-us/dnwinfs/html/winfs10182004.asp.

[40] RLG. Trusted digital repositories: Attributes and responsibilities. Report, RLG-OCLC, Mountain View, CA, May 2002. Last viewed on Jan 9, 2005 at http://www.rlg.org/en/page.php?Page_ID=583.

[41] T. Staples, R. Wayland, and S. Payette. The Fedora Project: An open-source digital object repository management system. *D-Lib Magazine*, 9(4), April 2003. Last viewed on Jan 9, 2005 at http://www.dlib.org/dlib/april03/staples/04staples.html.

[42] R. Tansley, M. Bass, M. Branschofsky, G. Carpenter, G. McClellan, and D. Stuve. DSpace system documentation. Documentation, MIT Libraries, May 04 2005.

[43] The Joint Information Systems Committee. Supporting digital preservation and asset management in institutions. http://www.jisc.ac.uk/index.cfm?name=programme_404, October 2004.

[44] The Mozilla Organization. Gecko layout engine. http://www.mozilla.org/newlayout/, 2005.

[45] The Mozilla Organization. Mozilla internet application suite. http://www.mozilla.org/products/mozilla1.x/, 2005.

[46] The Mozilla Organization. XBL - extensible binding language 1.0. http://www.mozilla.org/projects/xbl/xbl.html, 2005.

[47] The Mozilla Organization. XML user interface language (XUL). http://www.mozilla.org/projects/xul/, 2005.

[48] The World Wide Web Consortium. Resource description framework RDF. http://www.w3.org/RDF/, 2005.

[49] U.S. National Archives & Records Administration. National Archives names two companies to design an electronic archives. http://www.archives.gov/media_desk/press_releases/nr04-74.html, August 3 2004.

[50] A.-I. A. Wang, G. H. Kuenning, P. Reiher, and G. J. Popek. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[51] D. Woodhouse. The journalling flash file system. In *Ottawa Linux Symposium*, Ottawa, ON, Canada, July 2001.

[52] M. Wu and W. Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 86–97. ACM, Oct. 1994.