JOINFS: a Semantic File System for Embedded Systems

Matthew Harlan Gabriel Parmer

Computer Science Department
The George Washington University
Washington, DC
{mharlan,gparmer}@gwu.edu

Abstract

Hierarchical file systems are the de-facto abstraction for storing information on modern computing systems. Though they are useful for providing structure for categorical data, their failings are most pronounced on consumer embedded devices. The limited interface for organizing a directory hierarchy, and abundant structured, network-accessible data complicate the management of the hierarchy.

JOINFS *attempts to enhance the tradition hierarchical file system abstraction by marrying it with an efficient query system. Special* dynamic directories *are populated not by a rigid hierarchy, but by an active matching of query terms to meta-data associated with the files in the system. Hierarchical dynamic directories provide the novel concept of categorizing semantics, rather than the data itself. These dynamic directories are exposed as normal filesystem directories, thus enabling applications and scripts to harness the querying power of* JOINFS *with negligible effort.* JOINFS *maintains file-system performance on normal files and directories, has a small footprint, and is implemented as a modular addition to traditional file-systems.*

1 Introduction

File systems provide a hierarchical name-space enabling the organization of system data into separate directories. Though pervasive, this abstraction is increasingly inappropriate with the steady increases in hard-drive/SSD size, amount of data stored, and the constrained interfaces used to access and organize embedded systems and consumer electronics. All of these trends make it more difficult to organize larger amounts of data. For example, cell phones and other personal embedded systems often do not provide a natural file-system interface for users to use to organize their data, yet the amount of data available to such systems is vast. Current mobile systems rely on individual applications to manually manage their own data. The static organization of directories seems unable to scale to vast amounts of complex data a system is exposed to, even on devices as small a cell-phones. The web is a collection of data with little (global) coordination and classification, and is naturally organized by *search*. Fundamentally, complex data often has no single natural category, and instead has many attributes, any one of which might be useful as a discriminator for a user or program.

Others have claimed that "hierarchical file systems are dead"[6], and that search-based data-access should be the dominant form of organization. Indeed, commodity operating systems have integrated search functionality in the form of Windows search [9] and Apple Spotlight [7]. Such concepts are not new: twenty years ago, the semantic filesystem [3] was proposed in which directories are organized by attributes, and in '93 the Inversion FS implemented a file system on top of a data-base [4] while exposing querying functionality. JOINFS is motivated by this shared goal of enabling semantically-aware access (i.e. that is aware of the meaning of the contents of files) to vast amounts of data. However, we also focus on the implementation of a practical system with low resource requirements, that is fully backwards compatible, and study its performance impact on modern hardware.

To provide a semantic, search-based interface to traditional FSs, JOINFS provides *dynamic directories*. Dynamic directories appear as file system directories, but are populated with either files or directories that semantically match some a predicate associated with the directory. Each file has a set of meta-data associated with it, and dynamic directory predicates are used to match this meta-data, thus presenting only those files relevant to the search. Importantly, we maintain the hierarchical structure even in dynamic directories: JOINFS enables dynamic directories to nest inside other dynamic directories, effectively returning the intersection of the result set for the parent and child dynamic directories. Thus, the normal manual organization possible in the hierarchical name-space can also be used in dynamic directories.

JOINFS design goals. We designed JOINFS with a number of goals in mind. These include:

• *Dynamic directories.* Fundamental to JOINFS is the concept of dynamic directories that are populated dynamically with content with meta-data that matches a predicate associated with the directory. This enables search at the

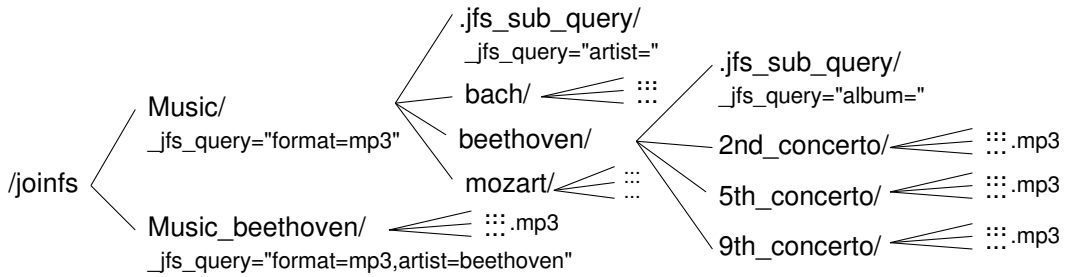

Figure 1. Example dynamic directory hierarchies created by specific search terms to demonstrate JOINFS usage. Though we use music in this example, dynamic directory hierarchies can be applied to any files and directories with any type of meta-data.

file-system level. Additionally, we marry this search capability with the hierarchical model by enabling *hierarchical dynamic directories*, thus enabling the manual categorization of the desired semantics of data, rather than the data itself.

• *Transparent Application Enhancement.* As JOINFS is implemented at the actual file-system level, and not in the shell or as a separate indexing program, any application can benefit from its functionality. For example, an image viewing program can get a succinct list of all images in the system by simply creating (or using an existing) an appropriately configured dynamic directory. Indeed this increases the power of even lowly shell scripts.

• *Backwards compatibility.* We strive for not only functional compatibility with traditional hierarchical systems, but also architect JOINFS as an *extension* on a normal file system. This means that a traditional FS is still used, and we maintain performance compatibility with previous systems.

• *Low resource requirements.* As appropriate for embedded devices including consumer electronics such as cell phones, we focus on minimal resource utilization including both CPU and memory. JOINFS uses only technologies and libraries that are appropriate and commonly found in these environments, and does not require – through also does not preclude – a kernel-level implementation.

Contributions. This paper makes the following contributions: 1) it introduces the concept of dynamic directories that marry semantic search and hierarchical categorization to enable *hierarchical categorization on semantics, not data*, 2) it details a practical user-level implementation of JOINFS that builds on commonly available technologies, and 3) empirically evaluates the system to measure its performance characteristics. JOINFS provides a practical and powerful solution to the data management problem and is tailored for use not only on conventional desktops and server, but also embedded and mobile systems.

The rest of this paper is organized as follows: Section 2 discusses how JOINFS is harnessed by applications and users. Section 3 outlines JOINFS's implementation, and Section 4 discusses its experimental validation. Section 5 covers the related work while Section 6 concludes.

2 JOINFS Interface and Use

Much of the novelty of JOINFS is in the interface of the semantic search with the file system, and in hierarchical dynamic directories. JOINFS must associate queries with directories to populate dynamic directories. For this purpose, and to store the meta-data associated with each file, we use POSIX extended attributes. Many modern file-systems enable the storage of meta-data in the form of untyped strings in extended attributes associated with a file.

Dynamic directories. The simplest example usage scenario for JOINFS use is to create a dynamic directory holding file matching a search. We use the `setfattr` command to associate a named attribute with its value. For this paper, we will describe setting attribute `n` to value `v` on file `f` as `(f,n,v)`. Given a file system with a number of file types, we can search for all of the `mp3` files by creating a directory `Music`, and setting a fixed attribute name `_jfs_query` to a list of `key=value` pairs: `(Music,"format=mp3")`. The directory will be populated with all mp3 music files. Queries can be more specific, for example, `(Music_beethoven,"format=mp3, artist=beethoven")`, in `Music_beethoven` will contain all `mp3`s with music by Beethoven. Dynamic directories is dynamically updated with changes in the file system. If new `mp3`s are added to the file system, or existing ones are removed, they will appear or disappear from the dynamic directory, respectively.

Hierarchical dynamic directories. JOINFS not only adds semantic search to the file system interface; it also maintains a hierarchical file system interface. This enables the simple categorization of search data. To enable hierarchical dynamic directories, each dynamic directory can optionally include a `.jfs_sub_query` directory. In this case, the parent dynamic directory's query will result in *directory* results, rather than files. The `.jfs_sub_query` directory can have queries associated with it similar to the parent dynamic directory: `(Music/.jfs_sub_query,"artist=")`. Intuitively, this says that the dynamic directory `Music` should be populated with a number of directories, each with the name

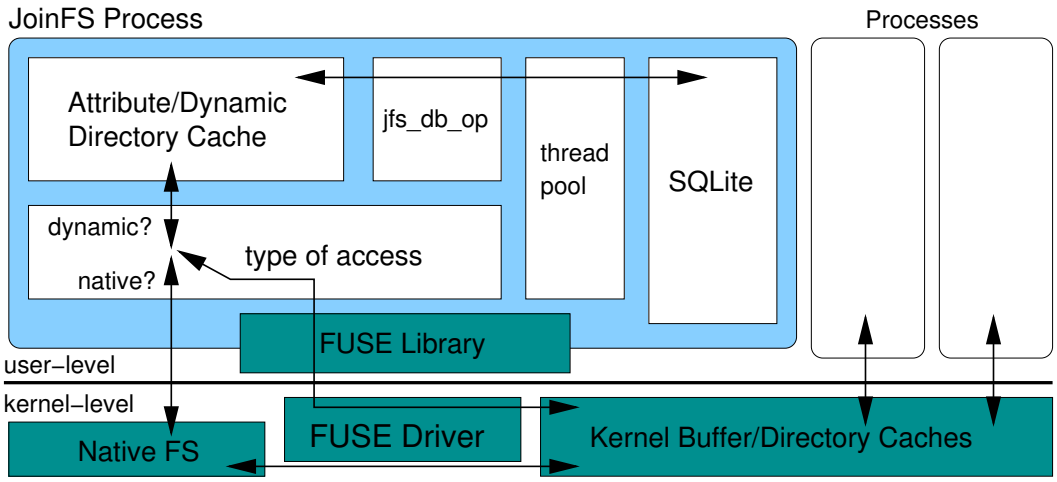

Figure 2. The high-level JOINFS design. The kernel buffers file data, and JOINFS passes file requests on to the native file system. Dynamic directories are handled by the logic and `SQLite` backend.

of one of the artists (i.e. the directories names are the unspecified values associated with given key). Each of these directories includes that artist's `mp3` files. Adding `Music/.jfs_sub_query/.jfs_sub_query` and the attribute `(Music/.jfs_sub_query/.jfs_sub_query,"album=")`, will now create an organization where within `Music`, there is a directory for each artist, in each of those directories, there is a subdirectory for each album for that artist, and in that directory are the actual `mp3` files by that artist *and* in that album.

This example demonstrates how JOINFS *enables the categorization using hierarchical structures of not the data itself, but of the semantics of interest*. Dynamic directories are created for semantic concepts, not data, and JOINFS automatically populates the directory with the appropriate data.

2.1 Metadata Generation

Though the previous example involved `mp3` files, JOINFS is generic in that any file can be annotated with metadata specific to its semantics. This annotation is beyond the scope of JOINFS. We currently use shell scripts to parse `id3` tags for music files, but similar programs could insert semantically relevant metadata for other forms of data such as images, video, or source code.

3 JOINFS Design and Implementation

JOINFS uses a number of existing technologies to avoid recreating the wheel. We implement JOINFS on Linux, and use `FUSE` [1] to integrate with the file system namespace. `FUSE` is "File systems in USEr-space", and enables the implementation of file systems as separate user-level processes. Though `FUSE` has some performance disadvantages [5] compared to native, kernel file systems, it is appropriate for prototyping advanced file system functionality. With `FUSE`, file system operations and requests are translated into inter-process communication (IPC) with the file system process.

JOINFS makes use of a data-base backend to track file system files and meta-data. For this, we use the `SQLite` [8] database. Though `SQLite` does not typically provide competitive write performance compared to more complex databases, it is widely accepted on embedded platforms and is deployed, for instance, in the Android mobile platform. We find in our evaluation (Section 4) that its performance is not prohibitive, and its memory consumption is appropriate for a somewhat capable embedded platform.

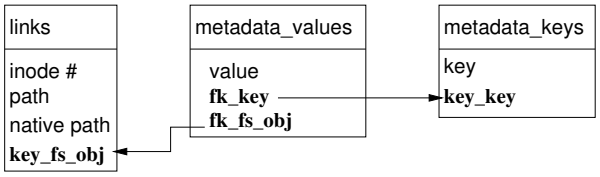

Figure 3. The database schema for JOINFS. Bold items are primary keys; arrows are foreign keys.

The database has a simple schema, and includes only three tables: `links`, `metadata_values`, and `metadata_keys`. The relations between these tables are depicted in Figure 3. The `path` is the complete path through the file system namespace exported by JOINFS to the actual file identified by `filename`.

3.1 Normal File System Operations

A key design decision in JOINFS is to minimize the involvement of the database in normal file operations. Due to this, we do *not* maintain all file data in the database, and

instead use the native file system for holding file data. This means that critical read and write performance are not substantively effected by JOINFS, thus taking JOINFS off of a critical path. In fact, as much functionality as possible is handed off by JOINFS to the native file system. This includes permission checking, real path lookup, hard-link reference counting, and file I/O.

When a request to open, or read a directory is made, JOINFS must determine if the access is made to a dynamic directory and it should be handled via data-base accesses, or to a native file object and it is passed to the native file system. The JOINFS file system is mirrored on the native file system with the exception of the contents of dynamic directories. To determine if a file system request is for dynamic contents, JOINFS simply executes a `stat` on the object in the native file system. This does impose the overhead of an additional system call on each file access that reaches JOINFS (i.e. requests that aren't satisfied by the Linux buffer cache). This is a relic of our implementation on `FUSE` and this overhead would be diminished if JOINFS were implemented in the kernel.

JOINFS does impose overhead on some normal file operations. Namely, creating, deleting, and adding attributes to file system objects. File system object creation and deletion do entail some overhead as the file must be inserted into or removed from the `links` table, and attribute operations have overheads as they are stored in the `metadata` tables. We study this effect in Section 4.

3.2 Dynamic Directories and SQL Generation

When a request is made to read the contents of a directory (*e.g.* via `readdir`), and JOINFS finds that it does exist in the native file system, it must determine if it is a dynamic directory. To do this, JOINFS checks if the attribute `_jfs_query` exists for that directory. If the attribute has a value, the directory has dynamically generated contents. Attributes are cached, so this check can often be performed while avoiding data-base interaction. To return the contents of the dynamic directory based on the query, JOINFS implements a lightweight cache of pathnames used to save the resulting files generated from a data-base query. This technique enables the results of expensive SQL query synthesis, data-base queries, and parsing of the return values, to be saved for future access.

Our experiments demonstrate that the memory overhead for these caches is minimal. These caches serve the important functions of avoiding database accesses when possible.

If the results of listing a dynamic directory are not cached, the SQL query is formed from the key value pairs of all nested hierarchical dynamic directories. They intuitively equate to the intersection (or conjunction) of each of the terms. For each level of dynamic directories, an SQL sub-statement is constructed that matches the current metadata value of the parent directory (e.g. if in the "Beethoven" directory, it matches `key=artist` and `value=Beethoven`), and returns the set of file identifiers (database keys). The intersection of these keys is used to compose the queries of multiple levels of dynamic directories. When the final level of dynamic directories is found, and it is time to generate files, thus a SQL statement takes the file identifiers generated through this process, and returns file information that JOINFS uses as the return values for the `readdir` call.

3.3 Database Interaction

`SQLite` provides a fairly standard database interface. However, for performance, we added a thread pool, and an extra caching layer. All interactions with the database are done using a `jfs_db_op` structure. This API provides functions for creating queries, synchronizing processes with the multithreaded `SQLite` query engine that actually executes the database operations, and cleans up database operations once they finish. The API essentially acts as a layer between JOINFS and the database, enabling the migration away from `SQLite` in the future if desired. JOINFS uses `SQLite` as it has a low footprint, has acceptable performance, and is widely available for many embedded and mobile platforms.

Thread pool for increased concurrency. In order to enable fast concurrent database reads, JOINFS makes use of a thread pool. This thread pool initially starts with the same number of threads as `FUSE`, but can expand and shrink as necessary. The SQL interface adds all database read operations to a job queue. The read pool grabs jobs from this queue, executes them, and returns the results. The thread pool then wakes up the thread that added the job so that they can continue processing. JOINFS prevents write operations from taking place by restricting all database connections to read-only for all reader threads.

The thread pool has the additional benefit of enabling database connection caching with `SQLite`. Without the thread pool, a database connection would have to be made for each database operation which significantly degrades performance.

JOINFS handles writes separately because `SQLite` does not support concurrent write operations. JOINFS instead uses a single writer thread with its own job queue to perform all database writes. JOINFS also supports executing multiple writes at once as a transaction. This enables more complex inserts without having to repeatedly attain an exclusive lock on the database file.

Caching to avoid database access. Even though databases often cache their results in memory, it is desirable in certain situations to have an additional caching layer

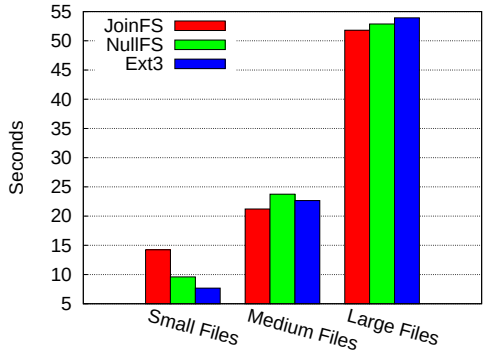

Figure 4. Overheads of transferring a directory hierarchy into JOINFS.

in JOINFS to avoid data-base interaction all-together. Thus, JOINFS caches pathnames and metadata for the contents of dynamic directories to avoid having to query the data-base again.

3.4 Hybrid Directories

As JOINFS marriages file systems with database concepts, there are interesting interplays between the two abstractions. For example, a natural question is what happens when a file is created *inside* a dynamic directory? In such cases, JOINFS stores the file or directories created in the dynamic directory's native backing store [1]. Any files created in these directories will appear in the contents of the dynamic directory. In the future, we will support automatically generating metadata for files placed into dynamic directories. This metadata will be determined by the metadata search terms of the dynamic directory.

4 Experimental Results

To evaluate the performance impact of JOINFS, we use a machine with a 3.3GHz Core i7-920, Intel processor with 6GB of RAM, and a 7200 rpm Western Digital Caviar Black 1 TB SATA drive. We use Ubuntu 10.04 LTS configured to use the ext3 file system and the Ubuntu version of Linux, `kernel-2.6.32-30-generic`.

4.1 Data Transfer

In this section, we wish to investigate the overheads of JOINFS for normal file operations (not to dynamic directories) such as creation, and population of file data. JOINFS is not designed to make these operations faster, and is instead meant to minimize the overheads to these operations.

[1] Each dynamic directory and `.jfs_sub_query` are stored on the native file system. The query results that populate those directories dynamically are not.

Figure 4 plots the amount of time it takes to transfer files from an `ext3` file system mount point into a mount point for 1) JOINFS using a normal directory, 2) NULLFS the `FUSE` file system that simply passes all file system operations on to the Linux file system (in this case `ext3`), and 3) `ext3`, the default Linux file system. We include the results for `ext3` only as a lower-bound on performance. `FUSE` imposes some overhead, especially on transfers for small files [5], so we compare more directly to NULLFS. We transfer three different classes of files: 1) *small files* – the Linux source code directory which includes 37,998 comparably small files totaling 400.3 MB (10.5 KB each on average), 2) *medium files* – a set of 209 music files totaling 1.4 GB, and 3) *large file* – 1 movie file totaling 1.4 GB. In Figure 4, we plot the average of 5 runs for each of the file systems.

For small files, we see that there is some overhead both for NULLFS, and JOINFS, and that JOINFS has an overhead over NULLFS of 48% (14.24 seconds vs 9.59 seconds). Though this overhead is not insignificant, as the file operations become dominated by reads and writes instead of file creation (for medium and large files), we see that all techniques are roughly equivalent (this echos results from [5]). For devices such as mobile phones and embedded systems where the focus is on data consumption rather than creation, we believe these overheads are acceptable.

Filesystem	Memory consumption
NULLFS	6.9 MB
JOINFS	9.8 MB
JOINFS Overhead	75 B per file

Table 1. Memory consumption of JOINFS after creation of 38,208 files.

Table 1 plots the memory usage of JOINFS and NULLFS. Memory consumption of the system is an important factor in analyzing its acceptability in many consumer electronics and embedded applications. We find that JOINFS imposes some overhead due to the data held in the `links` table of 75 bytes per file. Even in memory-constrained devices, we deem this to be acceptable.

4.2 Dynamic Directory Performance

In this section, we investigate the performance of the dynamic directory implementation in JOINFS. We wish to study the basic costs of doing a `readdir` in dynamic directories that queries the meta-data for all files, and produces the resulting directory contents. We study the effects of changing the number of files returned by the query, and the depth in the hierarchy of dynamic directories of the results.

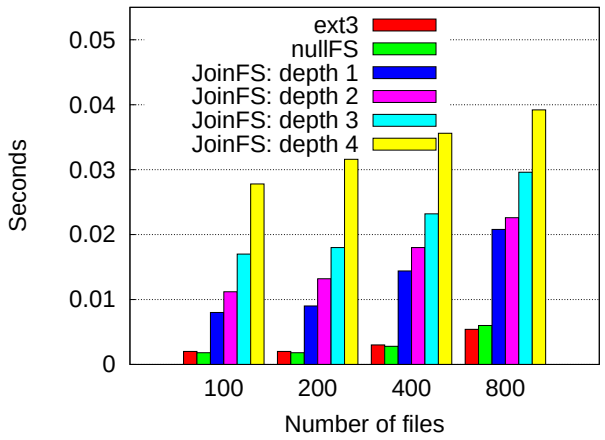

Figure 5. Latency for listing a variable number of files from a directory hierarchy. For JOINFS, we vary the depth of the hierarchy of dynamic directories, and measure query latency.

Figure 5 plots the latency for completing an `ls` command in a directory for 1) JOINFS, 2) NULLFS, and 3) `ext3`. For JOINFS, we plot the latency in a sub-directory of a variable depth into the hierarchy of dynamic directories. The other file systems do not use nested directories. For all file systems, we plot the latency for a varying number of files in the directory. Again, we plot the average of 5 runs.

JOINFS for a dynamic directory at a depth of 1 is within a factor of 5 of the latency for all numbers of query results to NULLFS. As we have nested dynamic directories, the overheads grow as complicated queries are synthesized along the path of directories, and the SQL queries generated increase in complexity. We don't believe these overheads are significant enough to discourage the use of nested dynamic directories for a categorical organization of semantic results.

Filesystem	Memory consumption
NULLFS	0.42 MB
JOINFS	2.8 MB
JOINFS Overhead	396 B per file

Table 2. Memory consumption of JOINFS for 6000 files in dynamic directories.

Table 2 plots the memory overhead after the queries (i.e. `readdirs`) have completed for all number of files (1500 in total), for all dynamic directory depths (thus, 6000 files listed in total). The overhead *above* NULLFS per file queried is 396 bytes per file. This includes index overheads in `SQLite`, and the overhead of JOINFS data-structures. We believe that an overhead of 2.8 MB is not insignificant, but is acceptable for queries that return 6K files.

5 Related Work

Multiple operating systems have investigated incorporating relational concepts into the file system [2, 10]. We are unaware of any previous system that has aggressively mixed a relational data-base with a file system while enhancing both categorical hierarchical structures with search *and* search with categorical, hierarchical structure. *Dynamic directories* provide not only a file-system representation of search, but nested dynamic directories enable the categorical, hierarchical organization of not data, but semantic information.

Semantic search and database file systems. [6] proclaims that hierarchical file systems are dead. JOINFS takes a more nuanced approach in which we still support hierarchical file systems, but integrate in a novel way dynamic directories into a hierarchical structure. [3] provides an implementation of semantic search in a file system, but does not provide nested semantic searches. As JOINFS is implemented on a modern OS, we provide valuable insights into its applicability in embedded and mobile systems. [4] implements a file system interface on top of a data-base. Unlike this approach, JOINFS focuses on backwards compatibility where possible by relying on a traditional file system for data-storage and hierarchical organization while providing dynamic directories as an extension.

Data indexing on top of the file system. Many modern systems use search applications on top of an existing file system to index data and enable search [9, 7]. Even the lowly `locate` command is implemented in this fashion.

6 Conclusions

This paper presents the design and implementation of JOINFS. JOINFS integrates semantic search on top of a hierarchical file system structure in a manner that enables the continued use of optimized, existing file system, while adding *dynamic directories* that are populated by a search term. In integrating the hierarchical namespace with semantic search, JOINFS provides the novel concept of nested dynamic directories. Each sub-directory enables the refinement of the parent directory's search. In such a way, the results of a search can include not only files, but dynamic directories themselves. This enables the normal hierarchical techniques for categorization to be applied not to the data itself, but to the semantic searches.

In integrating search in a natural way with hierarchical file-systems, the power of semantic search is made available to any application without any additional code. A mobile application for displaying `pdf` files can easily access all `pdfs` in the system by listing a dynamic directory. Even shell scripting is enhanced.

We have shown that the processing overheads imposed by JOINFS are acceptable for a great many applications, and that the memory overheads are generally acceptable for a variety of embedded and consumer electronics applications.

The JOINFS source is located at `http://github.com/mharlan/joinFS`.

References

[1] Filesystem in user space: http://http://fuse.sourceforge.net/, retrieved 4/1/11.

[2] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999.

[3] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 16–25, 1991.

[4] M. Olson. The design and implementation of the inversion file system. In *Proceedings of the USENIX Winter 1993 Technical Conference*, 1993.

[5] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 2010.

[6] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in operating systems*, 2009.

[7] Apple spotlight: http://www.apple.com/macosx/what-is-macosx/spotlight.html, retrieved 4/1/11.

[8] SQLite: http://www.sqlite.org/, retrieved 4/1/11.

[9] Windows search: http://www.microsoft.com/windows/products/winfamily/desktopsearch/default.mspx.

[10] WinFS: http://blogs.msdn.com/b/winfs/, retrieved 4/1/11.