Performance of Size-Changing Algorithms in Stackable File Systems

Erez Zadok, Johan M. Andersen, Ion Badulescu, and Jason Nieh
Computer Science Department, Columbia University
{ezk,johan,ion,nieh}@cs.columbia.edu

Abstract

Stackable file systems can provide extensible file system functionality with minimal performance overhead and development cost. However, previous approaches are limited in the functionality they provide. In particular, they do not support size-changing algorithms, which are important and useful for many applications, such as compression and security. We propose fast index files, a technique for efficient support of size-changing algorithms in stackable file systems. Fast index files provide a page mapping between file system layers in a way that can be used with any size-changing algorithm. Index files are designed to be recoverable if lost and add less than 0.1% disk space overhead. We have implemented fast indexing using portable stackable templates, and we have used this system to build several example file systems with size-changing algorithms. We demonstrate that fast index files have very low overhead for typical workloads, only 2.3% over other stacked file systems. Our system can deliver much better performance on size-changing algorithms than user-level applications, as much as five times faster.

1 Introduction

Since the early days of UNIX, file systems have proven to be a useful abstraction for extending system functionality. Stackable file systems are an effective infrastructure for creating new file system functionality with minimal performance overhead and development cost[10, 12, 25, 27, 31, 32]. With stacking, file systems can be developed independently and then stacked on top of each other to provide new functionality. For example, an encryption file system can be stacked on top of a native file system to provide secure data storage[30].

While many stackable file systems have been developed, all of them share a common limitation in functionality. None of them are able to support size-changing algorithms (SCAs), which are important and useful for many applications. Examples of such applications include compression which can save disk space, data format encodings for internationalization which can automatically translate data between unicode and ASCII, and size-changing encryption algorithms which can provide added security. Some ideas have been previously suggested for supporting SCAs in stackable file systems by using cache coherency mechanisms[11, 15], but no complete solution has ever been developed or implemented, much less demonstrated.

The challenge with supporting SCAs in stackable file systems is that the file data layout and page offsets can change from layer to layer. Consider the case of an upper-level compression file system stacked on top of a lower-level native file system. When an application writes a file through the compression file system, the file system will compress the file data, then pass the compressed data to the native file system, which will then store it on disk. The result is an encoded file stored in the lower-level file system. Suppose the application now wants to read a block of data at a given file offset back from the file. The corresponding data in the encoded file then needs to be retrieved. However because of compression, the file offset of the data in the encoded file is generally not the same as the one provided by the application to the file system. The key problem that needs to be addressed is how to map file offsets between layers in stackable file systems. The problem is complicated by the fact that the mapping depends on the SCA used.

We propose fast index files as a solution for supporting SCAs in stackable file systems. Fast index files provide a way of mapping file offsets between upper and lower layers in stackable file systems. Since the fast index file is just a mapping, a lower-layer file system does not need to know anything about the details of the SCA used by an upper-level file system. Each encoded file has a corresponding fast index file, which is simply stored in a separate file in the lower-layer file system. The index file is more than 1000 times smaller than the original data file, resulting in negligible additional storage requirements. The index file is designed to be recoverable if it somehow is lost so that it does not compromise the reliability of the file system. Finally, the index file is designed to deliver good file system performance with low stacking overhead, especially for common file operations. In particular, we introduce an optimization called fast tails to provide performance improvements for writes to the end of a file, a common file operation.

We have implemented fast indexing using stackable templates[31]. This allows us to provide support for SCAs in stackable file systems in a portable way. To demonstrate the effectiveness of our approach, we have used our implemented system to build and measure several example SCA file systems, including a compression file system. Our performance results show (1) that fast index files have very low overhead for typical file system workloads, only 2.3%, and (2) that such file systems can deliver much better performance on size-changing algorithms than user-level applications, as much as five times better.

This paper describes fast index files and is organized as follows. Section 2 details the design of the index files in relation to file operations, and discusses several optimizations. Section 3 overviews important implementation issues. We evaluate our system using several example file systems in Section 4. We survey related work in Section 5. Finally we conclude and discuss future work in Section 6.

2 Design

Traditional file system development is often done using low level file systems that interact directly with device drivers. Developing file systems in this manner is difficult and time consuming, and result in code that is difficult to port to other systems. Stackable file systems build on a generalization of in-kernel files called *vnodes*[16], by allowing for modular, incremental development of file systems using a *stackable vnode interface*[13, 24, 27]. Stacking is a technique for modularizing file system functions by allowing one vnode interface implementation to call another, building upon existing implementations and changing only that which is needed.

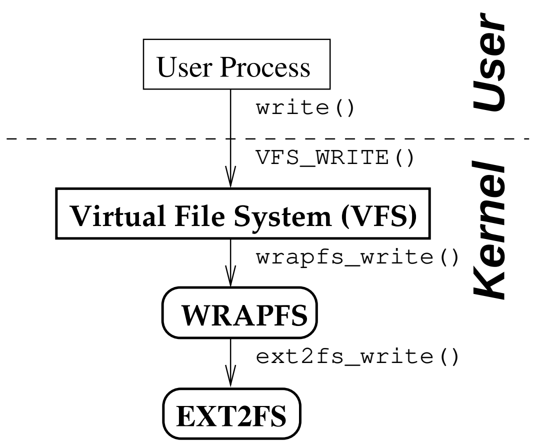

Figure 1: Basic Stacking. A system call is translated into a generic VFS function, which translates into a file-system specific function in our stackable Wrapper file system. Wrapfs then modifies the data passed to it and calls the file system stacked below it with the modified data.

Figure 1 shows the structure for a simple, single-level stackable wrapper file system called Wrapfs. System calls are translated into VFS calls, which in turn invoke their Wrapfs equivalents. Wrapfs then invokes the respective *lower level* file system operations. Wrapfs calls the lower level file system without knowing who or what type it is. In this example, Wrapfs receives user data to write to the lower level file system. Wrapfs may, for example, implement a transparent encryption file system and choose to encrypt the data before passing it to the lower level file system.

Wrapfs is a stackable template system[31] with ports to Linux, Solaris, and FreeBSD. It provides basic stacking functionality without changing other file systems or the kernel. Wrapfs allows developers to define data encoding functions that apply to whole pages of file data, making it easy to produce, for example, a limited subset of encryption file systems. Like other stacking systems, Wrapfs, however, did not support encoding data pages such that the result is of a different size.

Size-changing algorithms (SCAs) may change data offsets arbitrarily: shrinking data, enlarging, or both. A file encoded with an SCA will have offsets that do not correspond to the same offsets in the decoded file. In a stacking environment, the lower level file system contains the encoded files, while the decoded files are accessed via the upper layer. To find where specific data resides in the lower layer, an efficient mapping is needed that can tell where is the starting offset of the encoded data for a given offset in the original file.

We propose an efficient mapping for SCA stackable file systems based on an *index file*. The index file is a separate file that serves as a fast index into an encoded file. It is used to store meta-data information identifying offsets in an associated encoded file. As shown in Figure 2, the basic idea is that an upper level file system can efficiently access the decoded version of an encoded file in a lower level file system by using the meta-data information in an associated index file that resides on the lower level file system.

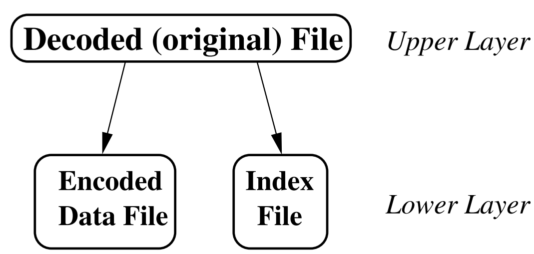

Figure 2: Each original data file is encoded into a lower data file. Additional meta-data index information is stored in an index file. Both the index file and the encoded data files reside on the lower level file system.

Throughout this paper, we will use the following three terms. An "original file" is the complete un-encoded file that the user accessing our stackable file system sees; the "data file" is the SCA-encoded representation of the original file, which encodes whole pages of the original file; the "index file" maps offsets of encoded pages between their locations in the original file and the data file.

Our system encodes and decodes whole pages, which maps well to file system operations. The index table assumes this and stores offsets of encoded pages as they appear in the encoded file.

To illustrate how this works, consider an example of a file in a compression file system as shown in Figure 3. The figure shows the mapping of offsets between the upper (original) file, and the lower (encoded) data file. To find out the bytes in page 2 of the original file, we read the data bytes 3000–7200 in the encoded data file, decode them, and return to the VFS that data in page 2.

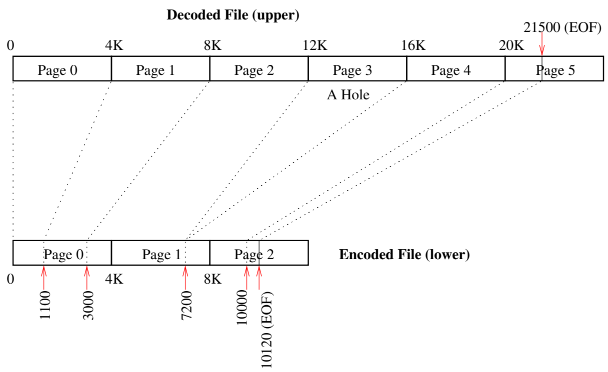

Figure 3: An example of a file system that shrinks data size (compression). Each upper page is represented by an encoded lower "chunk." Holes are supported too. The mapping of offsets is shown in Table 1.

To find out which encoded bytes we need to read from the lower file, we consult the index file, shown in Table 1. The index file tells us that the original file has 6 pages, that its original size is 21500 bytes, and then it lists the ending offsets of the encoded data for an upper page. Finding the lower offsets for the upper page 2 is a simple linear dereferencing of the data in the index file; we do not have to search the index file linearly.

2.1 The Index File

Word	**Representing**	**Regular IDX File**	**With Fast Tail** (ft)
1 (20 bits)	# pages	6	5
1 (12 bits)	flags	ft=0, ...	ft=1, ...
2	orig. file size	21500	21500
3	page 0	1100	1100
4	page 1	3000	3000
5	page 2	7200	7200
6	page 3	7200	7200
7	page 4	10000	10000
8	page 5	10120	

Table 1: Format of the index file for Figures 3 and 4. Fast Tails are described in Section 2.2.1. The first word encodes both the number of pages and flags.

The index information is stored in a separate small file. We measured the impact that the consumption of an additional inode would have on typical file systems in our environment. We found that disk data block usage is often 6–8 times greater than inode utilization on disk-based file systems, leaving plenty of free inodes to use.

For a given data file F, we create an index file called F`.idx`. We decided to store the index table in a separate file for three reasons:

1. The index file is small. We store one word (4 bytes) for each data page (usually 4096 bytes). On average, the index table size is 1024 times smaller than the original data file.
2. The index contains meta-data—original file size and page offsets—which are more logically stored outside the data itself, as is the case with many file systems. That allows us to control the data file and the index file separately.
3. Since the index file is relatively small, we can read it completely into kernel memory and manipulate it in there. To improve performance, we write the final modified index table only when the original file was closed and all of its data flushed to stable media. (Section 2.3.2 discusses how to recover from a lost index file.)

We read the index file into memory as soon as the main file is open. That way we have fast access to the index data in memory. The index information for each page is stored linearly, and each index entry takes 4 bytes. That way we can compute the needed index information very simply, and find it from the index table using a single dereference into an array of 4-byte words (integers). We write any modified index information out when the main file is closed and its data flushed to stable media.

The index file starts with a word that encodes the number of pages in the corresponding original data file and flags. We use the lower 20 bits for the number of pages because 2^{20} 4KB pages (typical on i386 and SPARCv8 systems) would give us the maximum file size we can encode in 4 bytes, explained next. We use the remaining 12 bits for special flags such as whether fast tails were encoded in this file, whether holes were encoded in this file, etc.

The index file also contains the original file's size (second word). We store this information in the index file so that commands like "ls –l" and others using stat(2) would work correctly. That is, if a process looks at the size of the file through the upper level file system, it would get the original number of bytes and blocks. The original file's size can be computed from the starting offset of the last data chunk in the encoded file, but it would require decoding the last (possibly incomplete) chunk (bytes 10000–10120 in the encoded file in Figure 3) which can be an expensive operation depending on the size-changing algorithm. Storing the original file size in the index file is a speed optimization that only consumes 4 more bytes—in a physical data block that most likely was already allocated. By using 4 bytes for the original file size, we are currently limiting the maximum supported decoded file size in our system to 4GB. That is not a major concern at this time, since files greater than 4GB are still fairly rare.

2.2 File Operations

The cost of size-changing algorithms can be high. Therefore it is important to ensure that we minimize the number of times we invoke these algorithms and the number of bytes they have to process each time. The way we store and access encoded data chunks can impact this performance, as well as the types and frequencies of file operations. Files accesses follow several patterns:

- The most popular file system operation is stat(), which results in a file lookup. Lookups account for 40–50% of all operations[19, 23].
- Most files are read, not written. The ratio of reads to writes is often 4–6[19, 23]. For example, compilers and editors read in many header and configuration files, but only write out a handful of files.
- Files that are written are often written from beginning to end. Compilers, user tools like "cp", and editors such as emacs write whole files in this way. Furthermore, the unit of writing is usually set to match the system page size. We have verified this by running a set of common tools and recorded the write start offsets, size of write buffers, and the current size of the file.
- Files that are not written from beginning to end are often appended to. The number of appended bytes is often small. This is true for various log files that reside in /var/log, as well as Web server access logs.
- Very few files are written in the middle. This happens most often when the GNU linker (gnu-ld) creates large binaries: it creates a sparse file of the target size, and then seeks and writes the rest of the file in a non-sequential manner. To estimate the frequency of writes in the middle, we instrumented a null-layer file system with a few counters. We then measured the number and type of writes for our large compile benchmark (Section 4.1). We counted 9193 writes, of which 58 (0.6%) were writes before the end of a file.
- All other operations account for a small fraction of file operations[19, 23].

Given the above access patterns, we designed our system to optimize performance for the more common cases, while not harming performance unduly when the seldom-executed cases occur.

To handle file lookups fast, we store the original file's size in the index table. The index file is usually 1024 times smaller than the original file. Due to locality in the creation of the index file, we assume that its name will be found in the same directory block as the original file name, and that the inode for the index file will be found in the same inode block as the encoded data file. Therefore reading the index file requires reading one additional inode and often only one data block. After the index file is read into memory, returning the file size is done by copying the information from the index table into the "size" field in the current inode structure. All other attributes of the original file come from the inode of the actual encoded file. Once we read the index table into memory, we allow the system to cache its data for as long as possible. That way, subsequent lookups will find files' attributes in the attribute cache.

Since most file systems are structured and implemented internally for access and caching of whole pages (usually 4KB or 8KB), we decided to encode the original data file in whole pages. In this way we improve performance because our encoding unit is the same as that used by the paging system, and especially the page cache. This also helped simplify our code because the interfacing with the VFS and the page cache was more natural. For file reads, the cost of reading in a data page is fixed: a fixed offset lookup into the index table gives us the offsets of encoded data on the lower level data file; we read this encoded sequence of bytes, decoded it into exactly one page, and return that decoded page to the user.

Since our stackable system is page based, it made it easier for us to write whole files, especially if the write unit was one page size. In the case of whole file writes, we simply encode each page size unit, add it to the lower level encoded file, and add one more entry to index table. We discuss the cases of file appends and writes in the middle in Sections 2.2.1 and 2.2.2, respectively.

We did not have to design anything special for handling all other file operations. We simply treat the index file at the same time we manipulate the corresponding encoded data file. An index file is created only for regular files; we do not have to worry about symbolic links, because the VFS will only call our file system to open a regular file. When a file is hard-linked, we also hard-link the index file, using the name of the new link with a the ".idx" extension added. When a file is removed from a directory or renamed, we apply the same operation to the corresponding index file. We can do so in the same context of the file system operation, because the directory in which the operation occurs is already locked.

2.2.1 Fast Tails

One common usage pattern of files is to append to them. Often, a small number of bytes is appended to an existing file. Encoding algorithms such as compression and encryption are more efficient when they encode larger chunks of data. Therefore it is better to encode a larger number of bytes together. Our design calls for encoding whole pages whenever possible. Table 1 and Figure 3 show that only the last page in the original file may be incomplete, and that incomplete page gets encoded too. If we append, say, 10 more bytes to the original (upper) file of Figure 3, we have to keep it and the index file consistent: we must read the 1020 bytes from 20480 (20K) until 21500, decode them, add the 10 new bytes, encode the new 1030 sequence of bytes, and write it out in place of the older 1020 bytes in the lower file. We also have to update the index table for two things: the total size of the original file is now 21510, and word number 8 in the index file may be in a different location than 10120 (depending on the encoding algorithm, it may be greater, smaller, or even the same).

The need to read, decode, append, and re-encode a chunk of bytes for each append grows worse as the number of bytes to append is small while the number of encoded bytes is closer to one full page. In the worst case, this method yields a complexity of $O(n^2)$ in the number of bytes that have to be decoded and encoded, multiplied by the cost of the encoding and decoding of the SCA. To solve this problem, we added a *fast tails* run-time mount option that allows for up to a page size worth of unencoded data to be added to an otherwise encoded data file. This is shown in the example in Figure 4.

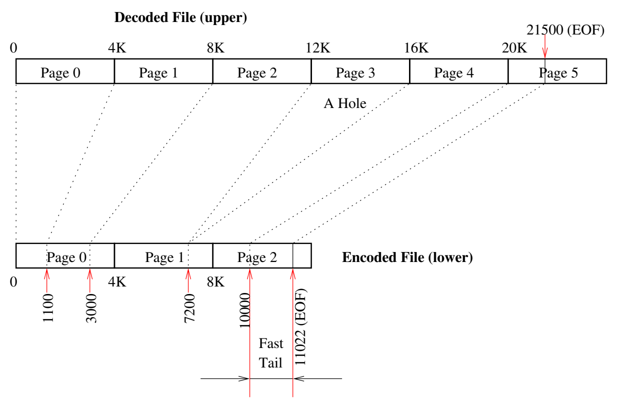

Figure 4: Fast-tails. A file system similar to Figure 3, only here we store up to one page full of un-encoded raw data. When enough raw data is collected to fill a whole fast-tail page, that page is encoded.

In this example, the last full page that was encoded is page 4. Its data bytes end on the encoded data file at offset 10000 (page 2). The last page of the original upper file contains 1020 bytes (21500 less 20K). So we store these 1020 bytes directly at the end of the encoded file, after offset 10000. To aid in computing the size of the fast tail, we add two more bytes to the end of the fast tail, listing the length of the fast tail. (Two bytes is enough to list this length, since typical page sizes are less than 2^{16} bytes long.) The final size of the encoded file name is now 11022 bytes long.

With fast tails, the index file does not record the offset of the last tail, as can be seen from the right-most column of Table 1. The index file, however, does record in its flags field (12 upper bits of the first word) that a fast tail is in use. We put that flag in the index table to speed up the computations that depend on the presence of fast tails. We put the length of the fast tail in the encoded data file to aid in reconstruction of a potentially lost index file, as described in Section 2.3.2.

When fast tails are in use, appending a small number of bytes to an existing file does not require data encoding or decoding, which can speed up the append operation considerably. When the size of the fast tail exceeds one page, we then encode the first page worth of bytes, and start a new fast tail.

Fast tails, however, may not be desirable all the time exactly because they store unencoded bytes in the encoded file. If the SCA used is an encryption one, it is insecure to expose plaintext bytes at the end of the ciphertext file. For this reason, fast tails is a run-time global mount option that affects the whole file system mounted with it.

2.2.2 Write in the Middle

User processes can write any number of bytes in the middle of an existing file. With our system, whole pages are encoded and stored in a lower level file as individual encoded chunks. A new set of bytes written in the middle of the file may encode to a different number of bytes in the lower level file. If the number of new encoded bytes is greater than the old number, we have to shift the remaining encoded file outward to make room for the new bytes. If the number of bytes is smaller, we have to shift the remaining encoded file inward to cover unused space. In addition, we have to adjust the index table for each encoded data chunk which was shifted.

To improve performance, we shift data pages in memory and keep them in the cache as long as possible. That way, subsequent write-in-the-middle operations that may result in additional inward or outward shifts will only have to manipulate data pages already cached and in memory. Of course, any data page shifted is marked as dirty, and we let the paging system flush it to disk when it sees fit.

Note that data that is shifted in the lower level file does not have to be re-encoded. This is because that data still represents the actual encoded chunks that decode into their respective pages in the upper file. The only thing remaining is to change the end offsets for each shifted encoded chunk in the index file.

We examined several alternatives that would have encoded the information about inward or outward shifts in the index table, and even possibly some of the shifted data, but we rejected them for several reasons: (1) it would have complicated the code considerably, (2) it would have made recovery of an index file very difficult, and (3) it would have resulted in fragmented data files that would have required a defragmentation procedure. Since the number of writes in the middle we measured was so small (0.6% of all writes), we do not consider our simplified design to cost too much in performance. Section 4 details our benchmarks and includes testing of files written in the middle.

2.2.3 Truncate

One interesting design issue we faced was with the truncate(2) system call. Although this call occurs less than 0.02% of the time[19, 23], we still had to ensure that it behaved the same. Truncate can be used to shrink a file as well as enlarge it, potentially making it sparse with new “holes.” We had four cases to deal with:

1. Truncating on a page boundary. In this case, we truncate the encoded file exactly after the end of the chunk that now represents the last page of the upper file. We update the index table accordingly: it has fewer pages in it.
2. Truncating in the middle of an existing page. In that case, we result in a partial page: we read and decode the

whole page, and re-encode the bytes within representing the part before the truncation point. We update the index table accordingly: it now has fewer pages in it.

3. Truncating in the middle of a fast tail. In that case we just truncate the lower file where the fast tail is actually located. We then update the size of the fast tail at its end, and update the index file to indicate the (now) smaller size of the original file.
4. Truncating past the end of the file is akin to extending the size of the file and possibly creating zero-filled holes. We read and re-encode any partially filled page or fast tail that used to be at the end of the file before the truncation; we have to do that because that page now contains a mix of non-zero data and zeroed data. All other pages afterwards are made up of zeros, and thus need not be encoded: we just mark them in the index file as zero-filled pages, as shown in Table 1 and Figure 3.

2.3 Additional Benefits

The discussion so far in this section concentrated on performance concerns in our system, since it is an important part of our design. We now consider three additional concerns:

1. **Low Resource Usage**: without harming performance, our system uses as little disk space for storing the index file. The index file is a small fraction of the size of the original file. A special option to support holes and sparse files is available.
2. **Consistency**: since the index file represents important meta-data that is stored separately, it can be recovered on files without holes.
3. **Portability**: our system does not fundamentally change on disk data structures, and it uses stackable file system interfaces. That way it can be ported with relative ease to other stacking systems and other operating systems[31].

2.3.1 Low Resource Usage

We designed our system to use little additional resources over what would be consumed normally. When considering the resource consumption, however, we gave a higher priority to performance concerns.

The index file was designed to be small, as seen in Table 1. It usually includes four bytes for the size of the full original file, four bytes indicating the number of page entries (including flags), and then that many index entries, four bytes each. For each page of 4096 bytes we store 4 bytes in the index file. This results in a reduction size factor of over 1000 between the size of the original file and the index file. Specifically, an index file that is exactly 4096 bytes long (one disk block on an EXT2 file system formatted with 4KB blocks) can describe an original file size of 1022 pages, or 4,186,112 bytes (almost 4MB).

By keeping the index file small, we ensure that the contents of most index files can be read and stored in memory in under one page, and can then be manipulated in fast memory. Since we create index files along with the encoded data files, we benefit from locality: the directory data block and inode blocks for the two files are already likely to be in memory, and the physical data blocks for the two files are likely to reside in close proximity to each other on the physical media.

The size of the index file is less important for SCAs which increase the data size, such as unicoding, uuencoding, and most forms of encryption. The more the SCA increased the data size, the less significant the size of the index file becomes. Even if the case of SCAs that decreased data size, such as compression, the size of the index file may not be as important given the savings already gained from compression.

To save even further of resource usage, we efficiently support zero-length files, as well as sparse files. A zero-length original data file is represented by a zero-length index file. (When the encoded file exists but the index file does not, it indicates that the index file was lost, and can be recovered as described in Section 2.3.2.)

Sparse files are files with "holes"—regions that are filled with zeros. Many file systems optimize disk usage by not writing zero-filled pages to stable media, instead skipping holes when chaining non-zero data blocks. Often, the MMU hardware is used by the operating system to create and zero-fill pages representing holes in files.

We support holes in files on a page-by-page basis. They are indicated in the index table as two or more consecutive entries that end on the same offset. In other words, the difference between them is zero bytes. Such an entry is treated as a zero-filled page in the upper, decoded file. Table 1 and Figure 3 show that page 3 is a hole represented by the lower level encoded data file as zero bytes.

This design to support holes has three implications. First, holes take no space on the encoded file, only four bytes per hole in the index file. Second, holes need not be encoded or decoded, which improves performance. Third, holes are not possible to recover if the index file is lost; we discuss this issue in the next section.

2.3.2 Index File Consistency

By the introduction of a separate index file to store the index table, we now have to maintain two files consistently.

Normally, when a write or create operation occurs on a file, the directory of that file is locked. We keep the directory locked also when the we update the index file, so that both the encoded data file and the index file are guaranteed to be written correctly.

We assume that encoded data files and index files would not get corrupt internally due to media failures. This situation is no worse than normal file systems where a random data corruption may not be possible to fix. However, we do concern ourselves with two potential problems: (1) a partially written or (2) a lost index file.

An index file could be partially written if the file system is full or the user ran out of quota. In the case where we were unable to write the complete index file, we simply remove it and print a warning message on the console. The absence of the index file on subsequent file accesses will trigger an in-kernel mechanism to recover the index file.

An index file could be lost if it was removed intentionally (say after a partial write) or unintentionally by a user directly from the lower file system. If the index file is lost or does not exist, we can no longer tell easily where encoded bytes were stored. In the worst case, without an index file, we have to decode the complete file to locate any arbitrary byte within. However, since the cost of decoding a complete file and regenerating and index table are nearly identical (see Section 4.3), we chose to regenerate immediately the index table if it does not exist, and then proceed as usual as if the index file exists.

We designed our system so that the index file can be recovered reliably in most cases. The three pieces of information needed to recover an index file given an encoded data file are (1) the SCA used, (2) the page size of the system on which the encoded data file was created, and (3) whether fast tails were used. These three pieces of information are available in the kernel to the running file system.

To recover an index file we read an input encoded data file and decode the bytes until we fill out one whole page of output data. We rely on the fact that the original data file was encoded in units of page size. The offset of the input data where we finished decoding onto one full page becomes the first entry in the index table. We continue reading input bytes and produce more full pages and more index table entries. If fast tails were used, then we read the size of the fast tail from the last two bytes of the encoded file, and do not try to decode it (since it was written un-encoded).

If fast tails were not used and we reached the end of the input file, that last chunk of bytes may not decode to a whole output page. In that case, we know that was the end of the original file, and we mark the last page in the index table as a partial page. While we are decoding pages, we sum up the number of decoded bytes and fast tails, if any. The total is the original size of the data file, which we record in the index table. We now have all the information necessary to write the correct index file and we do so.

The one case where we might not be able to recover an index file correctly is when the original data file contained holes. In that case, when reading the input encoded data, we can only find the non-hole pages because holes are not represented at all in the encoded data file: they are only represented in the index file as two consecutive entries with the same offset. For this reason, we also allow support for holes to be turned on or off at run time using a mount time flag. Therefore, support for holes represents a compromise between space (and some performance) savings and recoverability of the index file.

2.3.3 Portability

Our system is based on our Wrapfs templates. The templates provide stacking functionality for a given operating system. While the implementation of each template is different, they all provide a unified API for developers[29, 31]. If developers want to, say, modify file data in a consistent manner, they only need to write two routines: `encode_data` and `decode_data`. One routine is used to encode data pages, and another is used to decode data pages. We changed the templates to support SCAs without changing the encoding and decoding routines' prototypes: developers now may return arbitrary length buffers rather than being required to fill in exactly one output page.

We recently introduced a language called FiST, used to describe stackable file systems at a high level[32]. The language uses file system features common across different operating systems. The language code generator, fistgen, reads in a FiST input file that describes a new stackable file system, reads in the stackable templates for the given operating system, and together it produces a new file system with the desired functionality. With this language, we are able to achieve cross-platform portability for stackable file systems.

Our SCA work is done completely in the templates. The only change we introduced in the FiST language is to add another high level directive that tells the code generator to include SCA support. We decided to make this code optional because it adds overhead and is not needed for file systems that do not change data size. Using FiST, it is possible to write stackable file systems that do not pay the overhead of SCA support if they do not require changing the size of data. Furthermore, FiST can generate fan-out stackable file systems, ones that mount on more than one directory concurrently; it is not necessary that SCA support be active for each branch of the fan-out.

Currently, SCA support is available for Linux 2.3 only. When we port our SCA support to the other templates, we would be able to describe an SCA file system once in the FiST language. From this single description we could produce a number of working file system modules.

3 Implementation

Our original templates, called Wrapfs, served as a good basis for doing this SCA work, since they already had all of the operating system specific stacking infrastructure, and exported an API to developers that asks them to implement one encoding function and one decoding function. We therefore updated our Wrapfs templates based on the design depicted in Section 2. In this section we describe a few interesting implementation issues for support of size-changing algorithms. We also describe three sample file systems we implemented using our new templates.

We implemented our SCA support in Linux 2.3, using our Wrapfs templates. Our Wrapfs templates are ported to Solaris and FreeBSD as well, but we concentrated our efforts on

Linux. The intent of this paper is to show the design and implementation of SCA support and evaluate its performance. Based on our past experiences with other operating systems, we do not expect the performance savings to be significantly different on other platforms.

As mentioned in Section 2.1, we write any modified index information out when the main file is closed and its data flushed to stable media. In Linux, neither data nor meta-data are automatically flushed to disk. Instead, a kernel thread (kflushd) runs every 5 seconds and asks the page cache to flush any file system data that has not been use recently, but only if the system needs more memory. In addition, file data is forced to disk when either the file system is unmounted or the process called an explicit flush() or fsync(). We take advantage of this delayed write to improve performance, since we write the index table when the rest of the file's data is written.

To support writes in the middle correctly, we have to make an extra copy of data pages into a temporary location. The problem is that when we write a data page given to us by the VFS, we do not know what this data page will encode into, and how much space it would require. If it requires more space, then we have to shift data outward in the encoded data file before writing the new data. For this first implementation, we chose the simplified approach of always making the temporary copy. Our code has not been optimized much yet; we discuss avenues of future work in Section 6.1.

3.1 Examples

We implemented three file systems based on our updated templates. For each file system, all we had to implement were two routines: `encode_data` and `decode_data`. These routines take one input buffer, and can produce an output buffer of any size.

1. **gzipfs**: this is a compression file system using the Deflate algorithm[5] from the zlib-1.1.3 package[7, 9], the same algorithm used by GNU zip (gzip))[6, 8]. This file system is intended to demonstrate an algorithm that (usually) reduces data size.
2. **uuencodefs**: this file system is intended to illustrate an algorithm that increased the data size. This simple algorithm converts every 3-byte sequence into a 4-byte sequence. Uuencode produces 4 bytes that can have at most 64 values each, starting at the ASCII character for space (20_h). We chose this algorithm over encryption algorithms that run in Electronic Codebook mode (ECB) or Cipher Block Chaining mode because they do not increase the data size by much[26]. With uuencodefs we were able to increase the data size of the output by one-third.
3. **copyfs**: this file system simply copies its input bytes to its output, without changing data sizes. We wrote this simple file system to serve as a base file system to compare to gzipfs and uuencodefs. Copyfs exercises all of the index management algorithms and other size-changing algorithm support without the costs of encoding or decoding pages.

4 Evaluation

Our overall goals in evaluating this work are to show that we perform well compared to user-level tools, that reads and writes are fast, that lookups are fast, that fast-tails help for repeated small appends, and that writes in the middle do not impact overall performance significantly. For compression, we also aim to show the space savings for different data types.

We conducted a set of benchmarks intended to test the performance of our system and the examples we built. We ran the tests on four identical 433Mhz Intel Celeron machines with 128MB of RAM, a Quantum Fireball lct10 9.8GB IDE disk drive. We installed a Linux 2.3.99-pre3 kernel on this machine.

Each benchmark was run 10 times on a quiet system, using one of several test file systems: ext2fs, wrapfs, copyfs, uuencodefs, and gzipfs. We include figures for ext2fs because it is the basis for comparing to the remaining four stackable file systems. We mount the latter four over ext2fs. Wrapfs simply copies data pages without any SCA support; this helps to evaluate the cost of data page copying in the kernel. Copyfs copies data pages but includes SCA support; this lets us measure the impact of manipulating the index file in kernel and on disk, and the rest of the SCA code. Uuencodefs uses a simple algorithm that increases data size, while gzipfs usually reduces data size; this way we measure both types of size-changing file systems.

Since compression is sensitive to the type of data being compressed, we tested gzipfs on several types of data, ranging from easy to compress to difficult to compress:

1. A file containing the character "a" repeatedly, should compress really well.
2. A file containing English text, actually written by users, collected from our Usenet News server. We expected this file to compress well
3. A file containing a concatenation of many different binaries we located on the same host system, such as those found in /usr/bin and /usr/X11/bin. This file should be more difficult to compress because it contains fewer patterns useful for compression algorithms.
4. A file containing previously compressed data. We took this data from Microsoft NT's Service Pack 6 (sp6i386.exe), which is a self-unarchiving large executable. We expect this file to be very difficult to compress.

For each benchmark, we only read, written, or compiled the test files in the file system being tested. All other user utilities, compilers, headers, and libraries reside outside the tested file system.

To ensure that we used a cold cache for each test, we unmounted all file systems which participated in the given test after the test was done, and mounted the file systems again before running the next iteration of the test. We verified that unmounting a file system indeed flushes and discards all possible cached information about that file system. In one benchmark we also measured the warm cache performance, to show the effectiveness of our code's interaction with the page cache.

We ran all tests without the fast tails option, described in Section 2.2.1. We also repeated all tests with the fast tails option turned on. We report these figures whenever fast tails made a difference.

We measured the standard deviations in our experiments and found them to be small, less than 1% for most micro-benchmarks. We report deviations which exceeded 1% with their relevant benchmarks.

4.1 General Benchmarks

For testing overall performance, we investigated three tests: The Modified Andrew Benchmark (MAB)[21], am-utils (The Berkeley Automounter)[1], and Bonnie[4].

Modified Andrew Benchmark: the MAB benchmark consists of five phase: making directories, copying files, recursive listing, recursive scanning of files, and compilation. MAB was designed at a time when hardware was much slower and resources scarce. On our hardware, MAB completed in under 10 seconds of elapsed time, with little variance among different tests. We therefore opted to use a more intensive compile benchmark.

Am-utils: we configured and compiled a large package inside each file system. We used am-utils-6.0.4: it contains over 50,000 lines of C code in several dozen files. The build process begins by running several hundred small configuration tests intended to detect system features. It then builds a shared library, about ten binaries, four scripts, and documentation. Overall this benchmark contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations such as unlink, mkdir, and symlink. During the linking phase, several large binaries are linked by GNU ld, which exercises our support for holes (although the final linking phase fills all holes with data). Therefore this test is a more realistic general benchmark.

This test is the only test that we also ran with a warm cache. Our system caches decoded and encoded pages whenever possible, so as to improve performance. While normal file system benchmarks are done using a cold cache, we felt that there is value in also showing what the performance impact of our caching is. Also, we ran the test with and without our fast-tails option. We expect that in this general benchmark, fast tails would not have a large impact since the benchmark does not have an unusually large number of small appends.

Figure 5 summarizes the results of the am-utils benchmark. Using a warm cache improves performance by 5–10%.

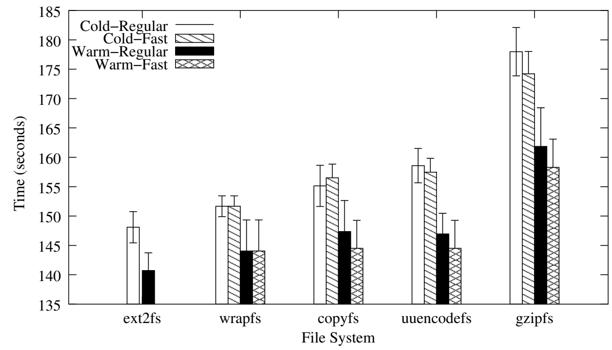

Figure 5: The Am-utils large compile benchmark. The standard deviations for this benchmark was less than 3% of the mean.

Using fast-tails improves performance by at most 2%. Interestingly, the fast-tail code has a small overhead of its own, since it runs different code in our system. For file systems that do not need fast tails, such as copyfs, fast tails actually reduces performance by 1%. We determined that fast tails is an option best used for expensive SCAs where many small appends are occurring, a conclusion demonstrated more visibly in Section 4.2. On average, the cost of data copying without size-changing (wrapfs compared to ext2fs) is an additional 2.4%. SCA support (copyfs over wrapfs) adds another 2.3% overhead. The uuencode algorithm is simple, and adds only 2.2% additional overhead over copyfs. Gzipfs, however, uses a more expensive algorithm (Deflate)[5], and it adds 14.7% overhead over copyfs.

Bonnie: this file system benchmark intensely exercises file data reading and writing, both sequential and random. Bonnie is a less general benchmark than am-utils. Bonnie has three phases. First, it creates a file of a given size by writing it one character at a time, then one block at a time, and then it rewrites the same file 1024 bytes at a time. Second, Bonnie writes the file one character at a time, then a block at a time; this can be used to exercise the file system cache, as cached pages have to be invalidated as they get overwritten. Third, Bonnie forks 3 processes that each perform 4000 random lseek()s in the file, and read one block; in 10% of those seeks, Bonnie also writes the block with random data. This last phase exercises the file system quite intensively.

For our case, we ran Bonnie using files of increasing sizes, from 1KB and doubling in size up to 128MB. The last size is important, because it matches the available memory on the system. Running Bonnie on a file that large, especially in a stackable setting where pages are cached in both layers, is important as the page cache should not be able to hold the complete file in memory.

Since this benchmark exercises data reading and writing heavily, we expect it to be affected by the SCA in use. Figure 6 confirms this. Wrapfs has an average overhead of 20% over ext2fs. Copyfs only adds an additional 8% overhead over wrapfs. Uuencodefs adds an overhead over copyfs that ranges from 5% to 73% for large files. Gzipfs, with its expensive SCA, adds an overhead over copyfs that ranges from 22% to 418% on the large 128MB test file.

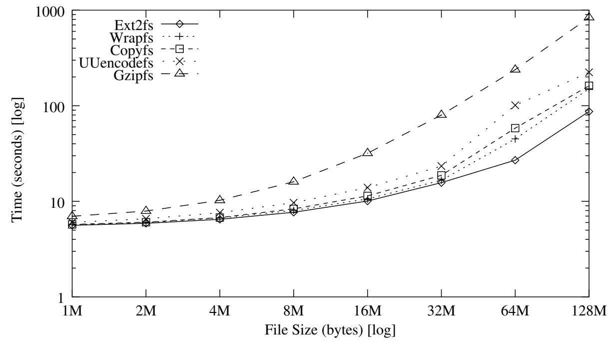

Figure 6: The Bonnie benchmark performs many repeated reads and writes on one file, as well as numerous random seeks and writes in three concurrent processes.

4.2 Micro-Benchmarks

The am-utils benchmark gives us a feel for the overall performance our system has under typical workloads. To a great extent, the Bonnie benchmark exercised our write-in-the-middle code, the code that has to handle possible data shifts. In this section we go a step further and analyze very specific file system operations that we expect to be affected by our system. In particular, we are interested in the following

- Writing files of various sizes. Writing is often more expensive that reading, and when writing files we have to create or modify index files.
- Appending to files. We are interested in the cost of appending a different number of bytes to various files, and how effective our fast-tails code is.
- Getting the attributes of files. In particular the size of the original file is saved in the index file, not in an inode. To get the size of the original file we have to read in the index table, a potentially expensive data reading operation.

Writing Files. We copied files of different sizes into a tested file system, and we tried it with and without our fast tails support. Figure 7 shows this. Wrapfs adds an average overhead of 16.4% over ext2fs; this is the overhead of data page copying. Copyfs adds an overhead of 23.7% over wrapfs; this is the overhead of updating and writing the index file as well as having to make temporary data copies (explained in Section 3) to support writes in the middle of files. The uuencode algorithm adds an additional average overhead of 43.2% over copyfs. For all copies over 4KB, fast-tails makes no difference at all. Below 4KB, it only improves performance by 1.6% for uuencodefs. The reason for this is that this benchmark copies files only once, where fast-tails is intended to work better in situations with multiple small appends.

Compression algorithms behave differently based on the input they are given. To evaluate the write performance of our compression file system, we ran it using four different data types: a file containing the same character repeatedly,

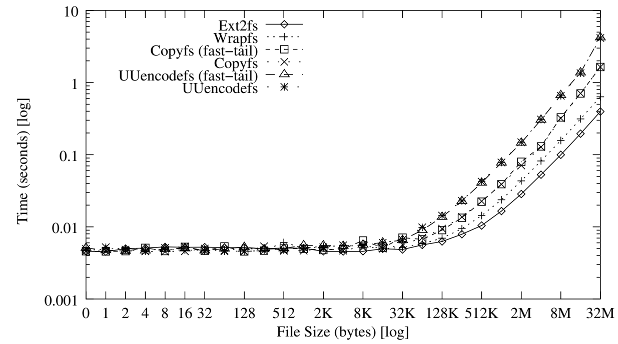

Figure 7: Copying files into a tested file system. As expected, uuencodefs is costlier that copyfs, wrapfs, and ext2fs. Fast-tails do not make a difference in this test, since we are not appending multiple times.

a text file, a binary file, and an already-compressed file. We ran the same tests using GNU zip (gzip)[6, 8], which uses the same algorithm and compression level (9). Figure 8 shows these results.

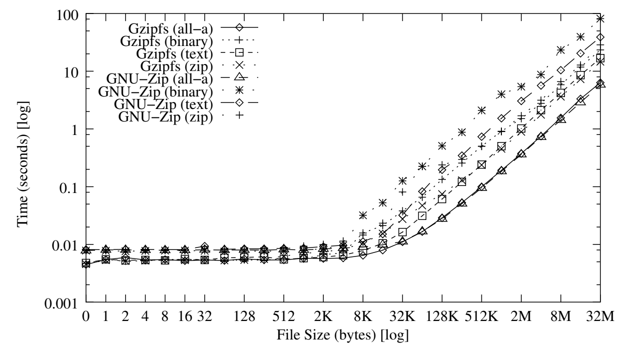

Figure 8: Comparing file copying into gzipfs (kernel) and using GNU zip (user-level) for various file types and sizes.

On average, the gzipfs compresses the “a” files 29.6% faster than GNU zip. For more typical text files and binaries, gzipfs compresses them on average 2.0–2.4 times faster. The larger the file is, the greater the savings: on files only as large as a megabyte, gzipfs runs 3.1–4.8 times faster. This is useful because large text files and binaries are the most likely candidates for compression, especially if they can be compressed quickly.

An interesting result we found is that the performance difference was not too great for compressing the already-compressed file; gzipfs was only 88% faster. The reason for this is that the Deflate compression algorithm will stop looking for potential data patterns to replace if it cannot find them quickly enough. So for a previously compressed file, the Deflate algorithm realizes quickly that it cannot compress it well, and moves on to another area of the file, speeding up overall compression performance for compressed files. Another interesting result to note is that compression is not uniform across a given file, even if the file purports to be of a given type. This is why some of the graphs in Figure 8 are not very smooth: the compression algorithm had to spend a different amount of time in those parts of the files being tested.

Appending to Files. As we explained in Section 2.2.1, we encode whole pages, but the last page of a file is often incomplete. So when such a file is appended to, the incomplete but encoded chunk of data must be decoded first, then the new bytes appended to this decoded chunk, and then the new and longer sequence has to be re-encoded. This expensive process must continue as long as the decoded number of bytes are less than 4KB. The more expensive the algorithm, the worse this process gets; the smaller the number of bytes appended to, and the more appends there are, the worse performance gets. Fast-tails is our technique for deferring the need to decode and re-encode until we have collected enough data. Then we encode a whole 4KB chunk at once, and start a new fast tail with the remaining bytes.

To evaluate the effectiveness of our fast tails code, we read in large files of different types, and used their bytes to append to a newly created file. We created new files by appending to them a fixed, but growing number of bytes. We appended bytes in three different sizes: 10 bytes representing a relatively small append, 100 bytes representing a typical size for a log entry on a Web server or syslog daemon, and 1000 bytes, representing a relatively large append unit. We did not try to append more than 4KB because that is the boundary where fast appended bytes get encoded.

Figure 9 shows the two emerging trends in effectiveness of the fast tails code. First, the more expensive the algorithm, the more helpful fast tails become. This can be seen in the right column of plots. Second, the smaller the number of bytes appended to the file is, the more savings fast tails provide, because the SCA is called fewer times. This can be seen as the trend from the bottom plots (1000 byte appends) to the top plots (10 byte appends). The upper rightmost plot clearly clusters together the benchmarks performed with fast tails support on and those benchmarks conducted without fast tails support.

Not surprisingly, there is very little savings from fast tail support for copyfs, no matter what the append size is. Uuencodefs is a simple algorithm that does not consume too much CPU cycles. That is why savings for using fast tails in uuencodefs range from 22% for 1000-byte appends to a factor of 2.2 performance improvement for 10-byte appends. Gzipfs, using an expensive SCA, shows significant savings: from a minimum performance improvement factor of 3 for 1000-byte appends to as much as a factor of 77 speedup (both for moderately sized files).

Getting the attributes of files. The size of the original file is now stored in the index file, not in the inode of the encoded data file. Finding this size requires reading an additional inode of the index file, and then reading its data. We ran a recursive listing (ls –lRF) on a freshly unpacked am-utils benchmark file set. We report these results in figure 10. Wrapfs add an overhead of 36% to the getattr operation, because it has to copy the attributes from one inode data structure into another. Copyfs adds the most significant overhead, a factor of 2.7 over wrapfs; that is because copyfs includes stackable SCA support, managing the index file in memory and on disk. However, uuencodefs and gzipfs add an overhead of only 12–26% over copyfs, for the getattr operation.

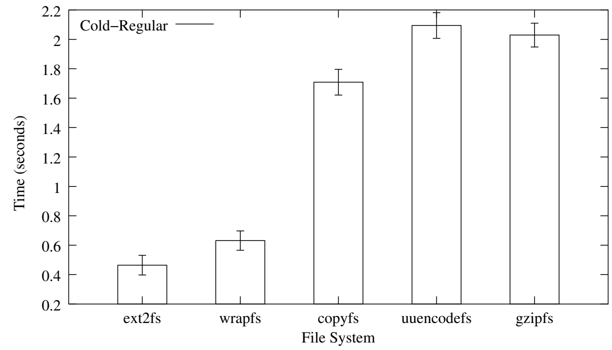

Figure 10: Retrieving file attributes (stat)

While the getattr file operation is a popular one, it is still very fast: the additional inode is likely to be in the locality of the data file, and the index file is over 1000 times smaller than the original data file. In addition, most operating systems cache attributes once they are retrieved. Finally in a typical workload, bulk data reads and writes are likely to dominate any other file system operation such as getattr.

4.3 Additional Tests

We measured the time it takes to recover an index file, and found it to be statistically indifferent from the cost of reading the whole file. This is expected, because to recover the index file we have to decode the complete data file.

One additional benchmark of note is the space savings for our compression file system, gzipfs, compared to the user level GNU zip tools. The Deflate algorithm used in both works best when it is given as much input data to work with at once. GNU zip looks ahead at 64KB of data, while gzipfs currently limits itself to 4KB (one page). For this reason, GNU zip achieves on average better compression ratios: as little as 4% better for compressing previously compressed data, to a factor of 5.6 for compressing the all-“a” file.

We also compared the performance of uuencodefs to user level uuencode utilities. We found the performance savings to be comparable to those with gzipfs compared to GNU zip.

5 Related Work

Most work in the area of stackable or extensible file systems appeared in the early 1990s[10, 12, 25, 27, 28]. Newer operating systems, such as the HURD[3] and Plan 9[22], have an extensible file system interface. Some of these works suggested the idea of stackable compression file systems.

Two additional works in Spring[15, 18] and Ficus[11] discussed a similar idea for implementing a stackable compression file system. Both suggested a unified cache manager that can automatically map compressed and uncompressed pages to each other. Heidemann’s Ficus work provided additional

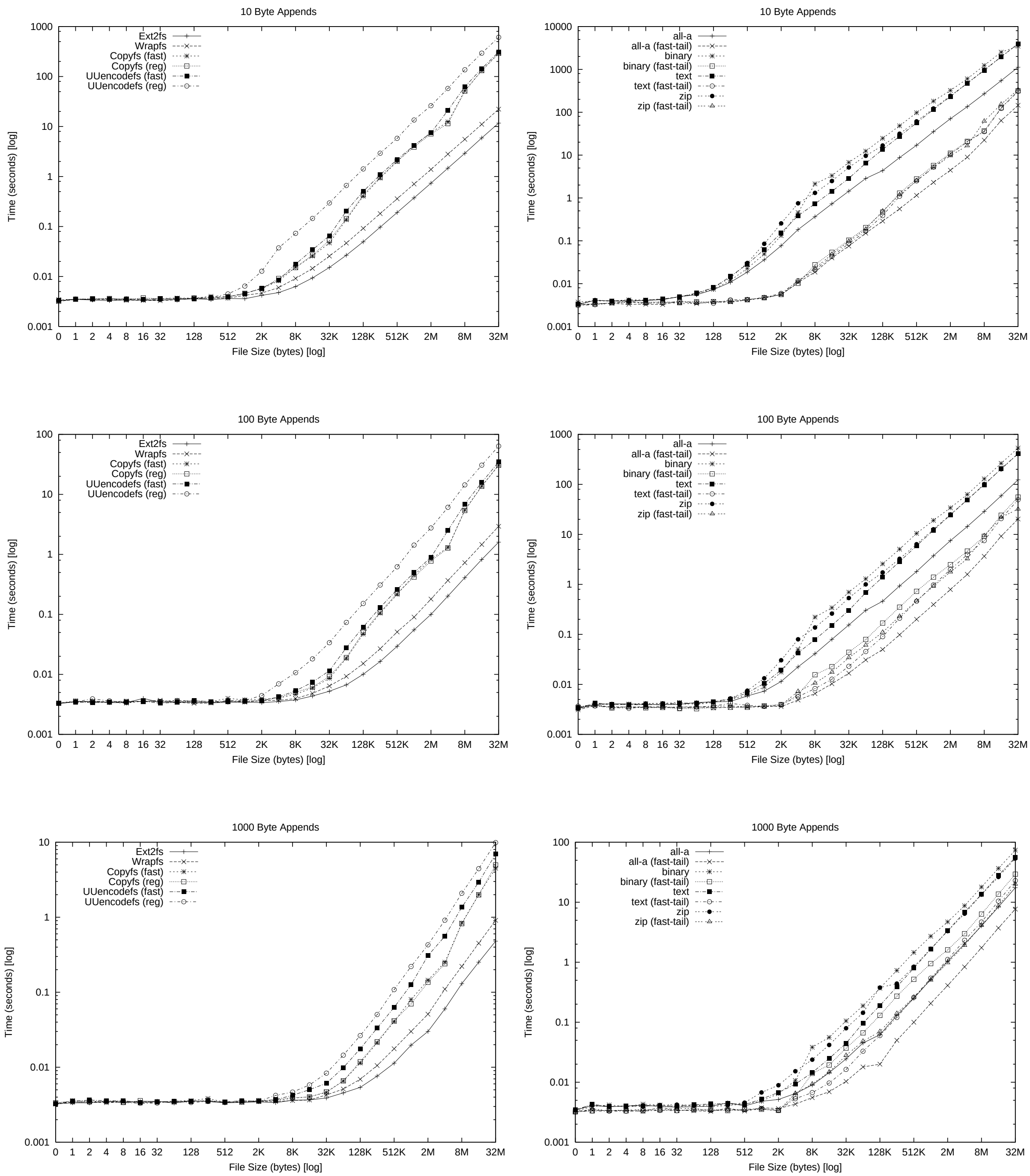

Figure 9: Appending to files. The more expensive the SCA is, and the smaller the number of bytes appended is, the more effective fast tails become, as can be seen on the upper rightmost plot. The standard deviation for this figure did not exceed 9% of the mean.

details on mapping cached pages of different sizes.[1] Unfortunately, no demonstration of these ideas for compression file systems was available from either of these works. In addition, no consideration was given to arbitrary size-changing algorithms, and how to handle difficult file operations efficiently, such as appends, writing in the middle, etc.

In addition, past implementations of stackable file systems required modifications to either existing file systems or the rest of the kernel, limiting their portability significantly, and affecting the performance of native file systems. Although this paper discusses an implementation on one operating system (Linux), we have demonstrated portable stacking twice before[31, 32], and expect to port our SCA support to other platforms (Solaris and FreeBSD) in a short time.

Compression file systems are not a new idea. Windows NT supports compression in NTFS[20]. E2compr is a set of patches to Linux's EXT2 file system that add block-level compression[2]. The benefit of block-level compression is primarily speed. Their main disadvantage is that they are specific to an operating system and one file system, making them very difficult to port to other systems, and resulting in code that is hard to maintain. Our approach is more portable because we use existing stacking infrastructure, we do not change file systems or operating systems, and we run our code in the kernel to achieve good performance.

The Exokernel[14] is an extensible operating system that comes with XN, a low-level in-kernel stable storage system. XN allows users to describe the on-disk data structures and the methods to access and manipulate them in a structure called a libFS. libFSes can implement a size-changing algorithm using a language of their own to determine which blocks in the lower-level file system map to a given file and the total size of a given file. The Exokernel, however, requires significant porting work to each new platform, but then it can run many unmodified applications.

Another transparent compression method possible is in user level. Zlibc is a preloadable shared library that allows executables to uncompress the data files that they need on the fly[17]. It is slow because it runs in user level, it only works on whole files, and it can only decompress files. Furthermore, it has to decompress the whole file before it can be used. Our system is much more flexible, performs well, can work with parts of files or whole files, and supports all file system operations transparently.

GNU zip (gzip))[6, 8] itself maintains some information on the structure of its compressed data. This information includes the un-encoded length of the file, the original file name, and a checksum of the encoded data. The information is useful, but is insufficient for the needs of a file system. Gzip, for example, does not provide support for random-access reading, a requirement for a compressed file system. With gzip, compressed data must be decompressed sequentially from beginning to end.

[1]Heidemann's earlier work[13] mentioned a "prototype compression layer" built during a class project. In personal communications with the author, we were told that this prototype was implemented as a block-level compression file system, not a stackable one.

6 Conclusions

The main contribution of our work is demonstrating that size-changing algorithms can be used effectively and transparently with stackable file systems. Our performance overhead is small and running these algorithms in the kernel improves performance considerably. File systems with support for size-changing algorithms can offer new services automatically and transparently to applications without having to change these applications or run them differently. Our templates provide support for generic size-changing algorithms, allowing developers to write new file systems easily.

Stackable file systems also offer portability across different file systems. File systems built with our SCA support can work on top of any other file system. In addition, we have done this work in the context of our FiST language, allowing rapid development of SCA-based file systems on multiple platforms[32].

6.1 Future Work

Our immediate future work is to add SCA support to our Solaris and FreeBSD templates. Next we would like to add support for large files and 64-bit file systems; this would require storing longer information in the index table, possibly doubling its size. While the index table is currently small compared to the original file, it can be larger on a huge file. For example, on a 64-bit file system, with 64 bit file offsets, an index file may be as large as 8MB for a 4GB original file. When we add support for 64 bits, we would also like to page portions of the index file in and out as needed, instead of reading it in completely.

An additional important optimization we plan to implement shortly is to avoid extra copying of data into temporary buffers. This is only needed when an encoded buffer is written in the middle of a file and its encoded length is greater than its decoded length; in that case, we must shift outward some data in the encoded data file to make room for the new encoded data. We can optimize this code and avoid making the temporary copies when files are appended to or being newly created and written sequentially.

We plan to improve support for sparse files, and in particular to allow holes to be recovered accurately. For that, we plan to add to the data file a bitmap that encodes which pages were holes and which were not. This would pose a small additional space consumption for the data file. For example, in 128 bytes of data we can encode 1024 pages, indicating which ones are holes; 1024 pages of data represent a 4MB file on a system with a 4KB page size. Once we encode information about holes in the data file, we could recover that information in the index file as well. In addition, the bitmap would allow us to report the number of real data blocks in a file that contains holes.

Our design allows the unit of encoding at the upper layer to be any multiple of page size, but our current implementation makes it one page size. We intend to add that support so that SCA-based file systems can work on larger units of data. This can be useful for compression file systems: the more data they compress at once, the better compression ratios they can achieve.

7 Acknowledgments

We would like to thank Jerry B. Altzman for his initial input into the design of the index table. We like to thank John Heidemann for offering clarification regarding his previous work in the area of stackable filing. This work was partially made possible by NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374.

References

[1] Am-utils (4.4BSD Automounter Utilities). Am-utils version 6.0.4 User Manual. February 2000. Available http://www.cs.columbia.edu/~ezk/am-utils/.

[2] L. Ayers. E2compr: Transparent File Compression for Linux. *Linux Gazette*, Issue 18, June 1997. http://www.linuxgazette.com/issue18/e2compr.html.

[3] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin*. Free Software Foundation, 1994. http://www.gnu.org/software/hurd/hurd.html.

[4] R. Coker. The Bonnie++ Home Page. http://www.coker.com.au/bonnie++.

[5] P. Deutsch. Deflate 1.3 Specification. RFC 1051. Network Working Group, May 1996.

[6] P. Deutsch and J. L. Gailly. Gzip 4.3 Specification. RFC 1052. Network Working Group, May 1996.

[7] P. Deutsch and J. L. Gailly. Zlib 3.3 Specification. RFC 1050. Network Working Group, May 1996.

[8] J. L. Gailly. GNU zip. http://www.gnu.org/software/gzip/gzip.html.

[9] J. L. Gailly and M. Adler. The zlib Home Page. http://www.cdrom.com/pub/infozip/zlib/.

[10] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conf. Proc.*, pages 63–71, Summer 1990.

[11] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM SOSP*. ACM SIGOPS, 1995.

[12] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Tech-report CSD-910007. UCLA, 1991.

[13] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *ACM ToCS*, **12**(1):58–89, Feb., 1994.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. *Sixteenth ACM SOSP*, pages 52–65, 1997.

[15] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. *Proceedings of Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14, 1993.

[16] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *USENIX Conf. Proc.*, pages 238–47, Summer 1986.

[17] A. Knaff. Zlibc: Uncompressing C Library. ftp://ftp.gnu.org/pub/gnu/zlibc/zlibc-0.9e.tar.gz.

[18] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conf. Proc.*, 1994.

[19] L. Mummert and M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. Report CMU-CS-94-213. Carnegie Mellon University, Pittsburgh, U.S., 1994.

[20] R. Nagar. Filter Drivers. In *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, 1997.

[21] John Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *USENIX Conference Proceedings* (Anaheim, CA), pages 247–56. USENIX, Summer 1990.

[22] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Proceedings of Summer UKUUG Conference*, pages 1–9, July 1990.

[23] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. To appear in *USENIX Conf. Proc.*, June 2000.

[24] D. S. H. Rosenthal. Requirements for a "Stacking" Vnode/VFS Interface. UI document SD-01-02-N014. UNIX International, 1992.

[25] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conf. Proc.*, pages 107–18. USENIX, Summer 1990.

[26] B. Schneier. Algorithm Types and Modes. In *Applied Cryptography, 2nd ed.*, pages 189–97. John Wiley & Sons, 1996.

[27] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *USENIX Conf. Proc.*, pages 161–74, Summer 1993.

[28] N. Webber. Operating System Support for Portable Filesystem Extensions. *USENIX Conf. Proc.*, pages 219–25, Winter 1993.

[29] E. Zadok and I. Badulescu. A Stackable File System Interface for Linux. *LinuxExpo Conf. Proc.*, 1999.

[30] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 1998.

[31] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. *USENIX Conf. Proc.*, 1999.

[32] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. To appear in *USENIX Conf. Proc.*, June 2000.

Software, documentation, and additional papers are available from http://www.cs.columbia.edu/~ezk/research/fist/.