Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems *

Erez Zadok
SUNY at Stony Brook
ezk@cs.sunysb.edu

Johan M. Andersen, Ion Bădulescu, and Jason Nieh
Columbia University
{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 provide only limited functionality. In particular, they do not support size-changing algorithms (SCAs), which are important and useful for many applications such as compression and encryption. We propose fast indexing, a technique for efficient support of SCAs in stackable file systems. Fast indexing provides a page mapping between file system layers in a way that can be used with any SCA. We use index files to store this mapping. 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 SCAs. We demonstrate that fast index files have low overhead for typical user workloads such as large compilations, only 2.3% over other stacked file systems and 4.7% over non-stackable file systems. Our system can deliver better performance with SCAs than user-level applications, as much as five times faster.

1 Introduction

Size-changing algorithms (SCAs) are those that take as input a stream of data bits and produce output of a different number of bits. These SCAs share one quality in common: they are generally intended to work on whole streams of input data, from the beginning to the end of the stream. Some of the applications of such algorithms fall into several possible categories:

Compression: Algorithms that reduce the overall data size to save on storage space or transmission bandwidths.

Encoding: Algorithms that encode the data such that it has a better chance of being transferred, often via email, to its intended recipients. For example, Uuencode is an algorithm that uses only the simplest printable ASCII characters and no more than 72 characters per line. In this category we also consider transformations to support internationalization of text as well as unicoding.

Encryption: These are algorithms that transform the data so it is more difficult to decode it without an authorization—a decryption key. Encryption algorithms can work in various modes, some of which change the data size while some modes do not [23]. Typically, encryption modes that increase data size are also more secure.

There are many useful user-level tools that use SCAs, such as `uuencode`, `compress`, and `pgp`. These tools work on whole files and are often used manually by users. This poses additional inconvenience to users. When you encrypt or decompress a data file, even if you wish to access just a small part of that file, you still have to encode or decode all of it until you reach the portion of interest—an action that consumes many resources. SCAs do not provide information that can help to decode or encode only the portion of interest. Furthermore, running user-level SCA tools repeatedly costs in additional overhead as data must be copied between the user process and the kernel several times. User-level SCA tools are therefore neither transparent to users nor do they perform well.

Instead, it would be useful for a file system to support SCAs. File systems are (1) transparent to users since they do not have to run special tools to use files, and (2) perform well since they often run in the kernel. File systems have proven to be a useful abstraction for extending system functionality. Several SCAs (often compression) have been implemented as extensions to existing disk-based file systems [2, 3, 18]. Their disadvantages are that they only work with specific operating systems and file systems, and they only support those specific SCAs. Supporting general-purpose SCAs on a wide range of platforms was not possible.

* Appeared in proceedings of the 2001 Annual USENIX Technical Conference.

Stackable file systems are an effective infrastructure for creating new file system functionality with minimal performance overhead and development cost [10, 12, 22, 24, 28, 29, 25]. Stackable file systems can be developed independently and then stacked on top of each other to provide new functionality. Also, they are more portable and are easier to develop [29]. For example, an encryption file system can be mounted on top of a native file system to provide secure and transparent data storage [27]. Unfortunately, general-purpose SCAs have never been implemented in stackable file systems. The problem we set out to solve was how to support general-purpose SCAs in a way that is easy to use, performs well, and is available for many file systems.

We propose *fast indexing* as a solution for supporting SCAs in stackable file systems. Fast indexing provide a way to map file offsets between upper and lower layers in stackable file systems. Since the fast indexing is just a mapping, a lower-layer file system does not have to know anything about the details of the SCA used by an upper-level file system. We store this fast indexing information in *index files*. Each encoded file has a corresponding index file which is simply stored in a separate file in the lower-layer file system. The index file is much smaller than the original data file, resulting in negligible storage requirements. The index file is designed to be recoverable if it is somehow lost so that it does not compromise the reliability of the file system. Finally, fast indexing is designed to deliver good file system performance with low stacking overhead, especially for common file operations.

We have implemented fast indexing using stackable templates [28, 29, 25]. This allows us to provide transparent support for SCAs in a portable way. To demonstrate the effectiveness of our approach, we built and tested several size-changing file systems, including a compression file system. Our performance results show (1) that fast index files have low overhead for typical file system workloads, only 2.3% over other null-layer stackable file systems, and (2) that such file systems can deliver as much as five times better performance than user-level SCA applications.

This paper describes fast index files and is organized as follows. Section 2 reviews the stacking file-system infrastructure used for this work and discusses related work in SCA support in file systems. Section 3 details the design of the index file. Section 4 describes the usage of the index file in relation to common file operations and discusses several optimizations. Section 5 details our design for a consistent and recoverable index file. Section 6 summarizes important implementation issues. Section 7 describes the file systems we built using this work and evaluates our system. Finally, we present conclusions and discuss directions for future work.

2 Background

In this section we discuss extensibility mechanisms for file systems, what would be required for such file systems to support SCAs, and other systems that provide some support for compression SCAs.

2.1 Stacking Support

Stackable file systems allow for modular, incremental development of file systems by layering additional functionality on another file system [13, 15, 21, 24]. Stacking provides an infrastructure for the composition of multiple file systems into one.

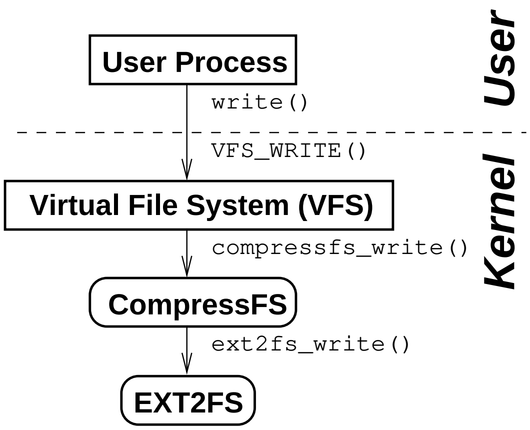


Figure 1: An example stackable compression file system. A system call is translated into a generic VFS function, which is translated into a file-system specific function in our stackable compression file system. CompressFS then modifies (compresses) the data passed to it and calls the file system stacked below it with the modified data.

Figure 1 shows the structure for a simple single-level stackable compression file system called CompressFS. System calls are translated into VFS calls, which in turn invoke their CompressFS equivalents. CompressFS receives user data to be written. It compresses the data and passes it to the next lower layer, without any regard to what type of file system implements that layer.

Stackable file systems were designed to be modular and transparent: each layer is independent from the layers above and below it. In that way, stackable file system modules could be composed together in different configurations to provide new functionality. Unfortunately, this poses problems for SCAs because the decoded data at the upper layer has different file offsets from the encoded data at the lower layer. CompressFS, for example, must know how much compressed data it wrote, where it wrote it, and what original offsets in the decoded file did that data represent. Those pieces of information are necessary so that subsequent reading operations can locate the data quickly.

If CompressFS cannot find the data quickly, it may have to resort to decompression of the complete file before it can locate the data to read.

Therefore, to support SCAs in stackable file systems, a stackable layer must have some information about the encoded data—offset information. But a stackable file system that gets that information about other layers violates its transparency and independence. This is the main reason why past stacking works do not support SCAs. The challenge we faced was to add general-purpose SCA support to a stacking infrastructure without losing the benefits of stacking: a stackable file system with SCA support should not have to know anything about the file system it stacks on. That way it can add SCA functionality automatically to any other file system.

2.2 Compression Support

Compression file systems are not a new idea. Windows NT supports compression in NTFS [18]. E2compr is a set of patches to Linux's Ext2 file system that add block-level compression [2]. Compression extensions to log-structured file systems resulted in halving of the storage needed while degrading performance by no more than 60% [3]. The benefit of block-level compression file systems is primarily speed. Their main disadvantage is that they are specific to one operating system and one file system, making them difficult to port to other systems and resulting in code that is hard to maintain.

The ATTIC system demonstrated the usefulness of automatic compression of least-recently-used files [5]. It was implemented as a modified user-level NFS server. Whereas it provided portable code, in-kernel file systems typically perform better. In addition, the ATTIC system decompresses whole files which slows performance.

HURD [4] and Plan 9 [19] have an extensible file system interface and have suggested the idea of stackable compression file systems. Their primary focus was on the basic minimal extensibility infrastructure; they did not produce any working examples of size-changing file systems.

Spring [14, 16] 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 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 SCAs and how to efficiently handle common file operations such as appends, looking up file attributes, etc.

[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.

3 The Index File

In a stacking environment that supports SCAs, data offsets may change arbitrarily. An efficient mapping is needed that can tell where the starting offset of the encoded data is for a given offset in the original file. We call this mapping the *index table*.

The index table is stored in a separate file called the *index file*, as shown in Figure 2. This file serves as the fast meta-data index into an encoded file. For a given data file F, we create an index file called F`.idx`. Many file systems separate data and meta data; this is done for efficiency and reliability. Meta-data is considered more important and so it gets cached, stored, and updated differently than regular data. The index file is separate from the encoded file data for the same reasons and to allow us to manage each part separately and simply.

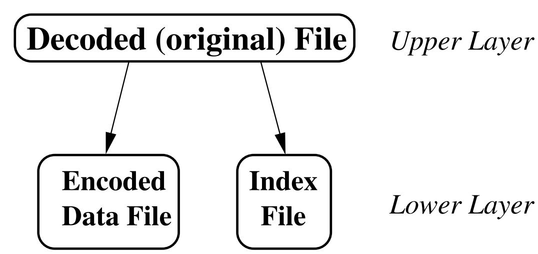


Figure 2: Overall structure of size-changing stackable file systems. Each original data file is encoded into a lower data file. Additional meta-data index information is stored in an index file. Both the index file and the encoded data files reside in the lower level file system.

Our system encodes and decodes whole pages or their multiples—which maps well to file system operations. The index table assumes page-based operations and stores offsets of encoded pages as they appear in the encoded file.

Consider an example of a file in a compression file system. Figure 3 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.

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. Note that our design of the

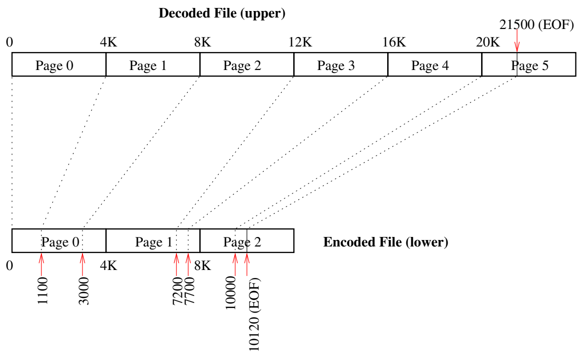


Figure 3: An example of a 32-bit file system that shrinks data size (compression). Each upper page is represented by an encoded lower "chunk." The mapping of offsets is shown in Table 1.

index file supports both 32-bit and 64-bit file systems, but the examples we provide here are for 32-bit file systems.

Word (32/64 bits)	**Representing**	**Regular IDX File**	**With Fast Tail** (ft)
1 (12 bits)	flags	ls=0,ft=0,...	ls=0,ft=1,...
1 (20/52 bits)	# pages	6	5
2	orig. file size	21500	21500
3	page 0	1100	1100
4	page 1	3000	3000
5	page 2	7200	7200
6	page 3	7700	7700
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 4.2. The first word encodes both flags and the number of pages in the index file. The "ls" (large size) flag is the first bit in the index file and indicates if the index file encodes a 32-bit (0) or 64-bit (1) file system.

The index file starts with a word that encodes flags and the number of pages in the corresponding original data file. We reserve the lower 12 bits for special flags such as whether the index file encodes a file in a 32-bit or a 64-bit file system, whether fast tails were encoded in this file (see Section 4.2), etc. The very first bit of these flags, and therefore the first bit in the index file, determines if the file encoded is part of a 32-bit or a 64-bit file system. This way, just by reading the first bit we can determine how to interpret the rest of the index file: 4 bytes to encode page offsets on 32-bit file systems or 8 bytes to encode page offsets on 64-bit file systems.

We use the remaining 20 bits (on a 32-bit file system) for the number of pages because 2^{20} 4KB pages (the typical page size on i386 and SPARCv8 systems) would give us the exact maximum file size we can encode in 4 bytes on a 32-bit file system, as explained next; similarly 2^{52} 4KB pages is the exact maximum file size on a 64-bit file system.

The index file also contains the original file's size in the second word. We store this information in the index file so that commands like `ls -l` and others using `stat(2)` would work correctly; a process looking at the size of the file through the upper level file system 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 SCA. Storing the original file size in the index file is a speed optimization that only consumes one more word—in a physical data block that most likely was already allocated.

The index file is small. We store one word (4 bytes on a 32-bit file system) for each data page (usually 4096 bytes). On average, the index table size is 1024 times smaller than the original data file. For example, 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).

Since the index file is relatively small, we read it into kernel memory as soon as the main file is open and manipulate it there. 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 typically takes 4 bytes. That lets us compute the needed index information simply and find it from the index table using a single dereference into an array of 4-byte words (integers). To improve performance further, we write the final modified index table only after the original file is closed and all of its data flushed to stable media.

The size of the index file is less important for SCAs which increase the data size, such as unicoding, uuencoding, and some forms of encryption. The more the SCA increases the data size, the less significant the size of the index file becomes. Even in the case of SCAs that decrease data size (e.g., compression) the size of the index file may not be as important given the savings already gained from compression.

Since the index information is stored in a separate file, it uses up one more inode. We measured the effect 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. To save resources even further, we efficiently support zero-length files: a zero-length original data file is represented by a zero-length index file.

For reliability reasons, we designed the index file so it could be recovered from the data file in case the index file is lost or damaged (Section 5.) This offers certain improvements over typical Unix file systems: if their meta-data (inodes, inode and indirect blocks, directory data blocks, etc.)

is lost, it rarely can be recovered. Note that the index file is not needed for our system to function: it represents a performance enhancing tool. Without the index file, size-changing file systems would perform poorly. Therefore, if it does not exist (or is lost), our system automatically regenerates the index file.

4 File Operations

We now discuss the handling of file system operations in fast indexing as well as specific optimizations for common operations. Note that most of this design relates to performance optimizations while a small part (Section 4.4) addresses correctness.

Because the cost of SCAs can be high, 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 affect this performance as well as the types and frequencies of file operations. As a result, fast indexing takes into account the fact that file accesses follow several patterns:

- The most popular file system operation is `stat(2)`, which results in a file lookup. Lookups account for 40–50% of all file system operations [17, 20].
- Most files are read, not written. The ratio of reads to writes is often 4–6 [17, 20]. 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 such as `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 edit and build tools on Linux 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 usually small. This is true for various log files that reside in `/var/log` as well as Web server logs.
- Very few files are written in the middle. This happens in two cases. First, when the GNU `ld` creates large binaries, it writes 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 7.3.1). We counted 9193 writes, of which 58 (0.6%) were writes before the end of a file.

  Second, data-base files are also written in the middle. We surveyed our own site's file servers and workstations (several hundred hosts totaling over 1TB of storage) and found that these files represented less than 0.015% of all storage. Of those, only 2.4% were modified in the past 30 days, and only 3% were larger than 100MB.
- All other operations (together) account for a small fraction of file operations [17, 20].

We designed our system to optimize performance for the more common and important cases while not harming performance unduly when the seldom-executed cases occur. We first describe how the index file is designed to support fast lookups, file reads, and whole file writes, which together are the most common basic file operations. We then discuss support for appending to files efficiently, handling the less common operation of writes in the middle of files, and ensuring correctness for the infrequent use of truncate.

4.1 Basic Operations

To handle file lookups fast, we store the original file's size in the index table. 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. Remaining 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, we also encode the original data file in whole pages. This improved our performance and helped simplify our code because 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, decode it into exactly one page, and return that decoded page to the user.

Using entire pages 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 the index table. We discuss the cases of file appends and writes in the middle in Sections 4.2 and 4.3, 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.

4.2 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 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 in two places: 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* runtime 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.

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 file past the fast tail itself, 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 is now 11022 bytes long.

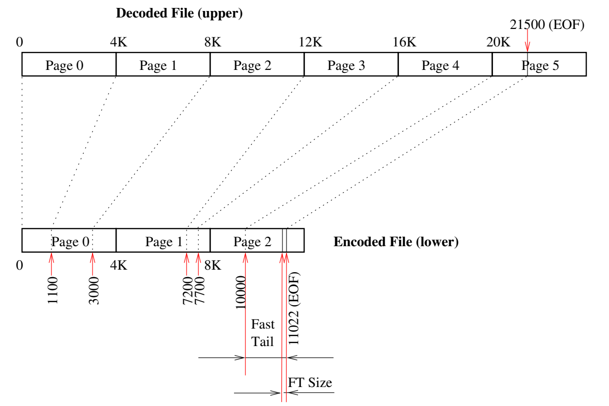


Figure 4: Size-changed file structure with fast-tail optimization. A file system similar to Figure 3, only here we store up to one page full of unencoded raw data. When enough raw data is collected to fill a whole fast-tail page, that page is encoded.

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 (first 12 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 append the length of the fast tail to the encoded data file to aid in reconstruction of a potentially lost index file, as described in Section 5.

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 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 runtime global mount option that affects the whole file system mounted with it. The option is global because typically users wish to change the overall behavior of the file system with respect to this feature, not on a per-file basis.

4.3 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 shift the remaining encoded file outward to make room for the new bytes. If the number of bytes is smaller, we shift the remaining encoded file inward to cover unused space. In addition, we adjust the index table for each encoded data chunk which was shifted. We perform shift operations as soon as our file system's write operation is invoked, to ensure sequential data consistency of the file.

To improve performance, we shift data pages in memory and keep them in the cache as long as possible: we avoid flushing those data pages to disk and let the system decide when it wants to do so. 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. 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 performance optimization alternatives that would have encoded the information about inward or outward shifts in the index table or possibly in some of the shifted data. We rejected them for several reasons: (1) it would have complicated the code considerably, (2) it would have made recovery of an index file 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 consider our simplified design as a good cost vs. performance balance. Even with our simplified solution, our file systems work perfectly correctly. Section 7.3.2 shows the benchmarks we ran to test writes in the middle and demonstrates that our solution produces good overall performance.

4.4 Truncate

One design issue we faced was with the `truncate(2)` system call. Although this call occurs less than 0.02% of the time [17, 20], we still had to ensure that it behaved correctly. Truncate can be used to shrink a file as well as enlarge it, potentially making it sparse with new "holes." We dealt with four cases:

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. This case results in a partial page: we read and decode the whole page and re-encode the bytes within the page 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. We encode all subsequent zero-filled pages. This is important for some applications such as encryption, where every bit of data—zeros or otherwise—should be encrypted.

5 Index File Consistency

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

Normally, when a file is created, the directory of that file is locked. We keep both the directory and the encoded data file locked when we update the index file. This way both the encoded data file and the index file are guaranteed to be written correctly.

We assume that encoded data files and index files will not become 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 three potential problems with the index file: partially written file, a lost file, and trivial corruptions.

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 log a warning message through `syslog(3)`—where the message could be passed on to a centralized logging facility that monitors and generates appropriate alerts. The absence of the index file on subsequent file accesses will trigger an in-kernel mechanism to recover the index file. That way the index file is not necessary for our system to function; it only aids in improving performance.

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 easily tell 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 an index table are nearly identical (see Section 7.6), we chose to regenerate the index table immediately if it does not exist, and then proceed as usual as the index file now exists.

We verify the validity of the index file when we use the index table. We check that all index entries are monotonically increasing, that it has the correct number of entries, file size matches the last entry, flags used are known, etc. The index file is regenerated if an inconsistency is detected. This helps our system to survive certain meta-data corruptions that could occur as a result of software bugs or direct editing of the index file.

We designed our system so that the index file can be recovered reliably in all cases. Four important pieces of information are needed to recover an index file given an encoded data file. These four are available in the kernel to the running file system:

1. The SCA used.
2. The page size of the system on which the encoded data file was created.
3. Whether the file system used is 32-bit or 64-bit.
4. Whether fast tails were used.

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 we do not try to decode it since it was written unencoded.

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.

6 SCA Implementation

Our SCA support was integrated into FiST [29, 25]. The FiST system includes portable stackable file system templates for several operating systems as well as a high-level language that can describe new stackable file systems [26, 28]. Most of the work was put into the stackable templates, where we added substantially more code to support SCAs: 2119 non-comment lines of C code, representing a 60% increase in the size of the templates. Because this additional code is substantial and carries an overhead with it that is not needed for non-size-changing file systems (Section 7), we made it optional. To support that, we added one additional declaration to the FiST language, to allow developers to decide whether or not to include this additional support.

To use FiST to produce a size-changing file system, developers need to include a single FiST declaration in their input file and then write only two routines: `encode_data` and `decode_data`. The main advantage of using FiST for this work has been the ease of use for developers that want to write size-changing file systems. All the complexity is placed in the templates and is mostly hidden from developers' view. Developers need only concentrate on the core implementation issues of the particular algorithm they wish to use in their new file system.

The FiST system has been ported to Linux, Solaris, and FreeBSD. Current SCA support is available for Linux 2.3 only. Our primary goal in this work was to prove that size-changing stackable file systems can be designed to perform well. When we feel that the design is stable and addresses all of the algorithmic issues related to the index file, we will port it to the other templates. We would then be able to describe an SCA file system once in the FiST language; from this single portable description, we could then produce a number of working file systems.

There are two implementation-specific issues of interest: one concerning Linux and the other regarding writes in the middle of files. As mentioned in Section 3, 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 used 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 `fflush(3)` or `fsync(2)`. 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 that new encoding 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, which affects performance as seen in Section 7. While our code shows good performance, it has not been optimized much yet; we discuss avenues of future work in Section 9.

7 Evaluation

To evaluate fast indexing in a real world operating system environment, we built several SCA stackable file systems based on fast indexing. We then conducted extensive measurements in Linux comparing them against non-SCA file systems on a variety of file system workloads. In this section we discuss the experiments we performed on these systems to (1) show overall performance on general-purpose file system workloads, (2) determine the performance of individual common file operations and related optimizations, and (3) compare the efficiency of SCAs in stackable file systems to equivalent user-level tools. Section 7.1 describes the SCA file systems we built and our experimental design. Section 7.2 describes the file system workloads we used for our measurements. Sections 7.3 to 7.6 present our experimental results.

7.1 Experimental Design

We ran our experiments on five file systems. We built three SCA file systems and compared their performance to two non-SCA file systems. The three SCA file systems we built were:

1. **Copyfs**: this file system simply copies its input bytes to its output without changing data sizes. Copyfs exercises all of the index-management algorithms and other SCA support without the cost of encoding or decoding pages.

2. **Uuencodefs**: this is a file system that stores files in uuencoded format and uudecodes files when they are read. It is intended to illustrate an algorithm that increases 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 because it is simple and yet increases data size significantly (by one third).

3. **Gzipfs**: this is a compression file system using the Deflate algorithm [7] from the zlib-1.1.3 package [9]. This algorithm is used by GNU zip (`gzip`) [8]. This file system is intended to demonstrate an algorithm that (usually) reduces data size.

The two non-SCA file systems we used were Ext2fs, the native disk-based file system most commonly used in Linux, and Wrapfs, a stackable null-layer file system we trivially generated using FiST [25, 29]. Ext2fs provides a measure of base file system performance without any stacking or SCA overhead. Wrapfs simply copies the data of files between layers but does not include SCA support. By comparing Wrapfs to Ext2fs, we can measure the overhead of stacking and copying data without fast indexing and without changing its content or size. Copyfs copies data like Wrapfs but uses all of the SCA support. By comparing Copyfs to Wrapfs, we can measure the overhead of basic SCA support. By comparing Uuencodefs to Copyfs, we can measure the overhead of an SCA algorithm incorporated into the file system that increases data size. Similarly, by comparing Gzipfs to Copyfs, we can measure the overhead of a compression file system that reduces data size.

One of the primary optimizations in this work is fast tails as described in Section 4.2. For all of the SCA file systems, we ran all of our tests first without fail-tails support enabled and then with it. We reported results for both whenever fast tails made a difference.

All experiments were conducted on four equivalent 433Mhz Intel Celeron machines with 128MB of RAM and a Quantum Fireball lct10 9.8GB IDE disk drive. We installed a Linux 2.3.99-pre3 kernel on each machine. Each of the four stackable file systems we tested was mounted on top of an Ext2 file system. For each benchmark, we only read, wrote, or compiled the test files in the file system being tested. All other user utilities, compilers, headers, and libraries resided outside the tested file system.

Unless otherwise noted, all tests were run with a cold cache. 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 completed 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 report the warm cache performance, to show the effectiveness of our code's interaction with the page and attribute caches.

We ran all of our experiments 10 times on an otherwise quiet system. We measured the standard deviations in our experiments and found them to be small, less than 1% for most micro-benchmarks described in Section 7.2. We report deviations which exceeded 1% with their relevant benchmarks.

7.2 File System Benchmarks

We measured the performance of the five file systems on a variety of file system workloads. For our workloads, we used five file system benchmarks: two general-purpose benchmarks for measuring overall file system performance, and three micro-benchmarks for measuring the performance of common file operations that may be impacted by fast indexing. We also used the micro-benchmarks to compare the efficiency of SCAs in stackable file systems to equivalent user-level tools.

7.2.1 General-Purpose Benchmarks

Am-utils: The first benchmark we used to measure overall file system performance was am-utils (The Berkeley Automounter) [1]. This benchmark configures and compiles the large am-utils software package inside a given file system. We used am-utils-6.0.4: it contains over 50,000 lines of C code in 960 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: a total of 265 additional files. 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`.

The am-utils benchmark is the only test that we also ran with a warm cache. Our stackable file systems cache decoded and encoded pages whenever possible, to improve performance. While normal file system benchmarks are done using a cold cache, we also felt that there is value in showing what effect our caching has on performance. This is because user level SCA tools rarely benefit from page caching, while file systems are designed to perform better with warm caches; this is what users will experience in practice.

Bonnie: The second benchmark we used to measure overall file system performance was Bonnie [6], a file system test that 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, since cached pages have to be invalidated as they get overwritten. Third, Bonnie forks 3 processes that each perform 4000 random `lseeks` 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, and especially the code that performs writes in the middle of files.

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

7.2.2 Micro-Benchmarks

File-copy: The first micro-benchmark we used was designed to measure file system performance on typical bulk file writes. This benchmark copies files of different sizes into the file system being tested. Each file is copied just once. Because file system performance can be affected by the size of the file, we exponentially varied the sizes of the files we ran these tests on—from 0 bytes all the way to 32MB files.

File-append: The second micro-benchmark we used was designed to measure file system performance on file appends. It was useful for evaluating the effectiveness of our fast tails code. This benchmark read in large files of different types and used their bytes to append to a newly created file. New files are created by appending to them a fixed but growing number of bytes. The benchmark 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 not try to append more than 4KB because that is the boundary where fast appended bytes get encoded. Because file system performance can be affected by the size of the file, we exponentially varied the sizes of the files we ran these tests on—from 0 bytes all the way to 32MB files.

Compression algorithms such as used in Gzipfs behave differently based on the input they are given. To account for this in evaluating the append performance of Gzipfs, we ran the file-append benchmark on four types of data files, 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/X11R6/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 compressed executable. We expect this file to be difficult to compress.

File-attributes: The third micro-benchmark we used was designed to measure file system performance in getting file attributes. This benchmark performs a recursive listing (`ls -lRF`) on a freshly unpacked and built am-utils benchmark file set, consisting of 1225 files. With our SCA support, 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. This micro-benchmark measures the additional overhead that results from also having to read the index file.

7.2.3 File System vs. User-Level Tool Benchmarks

To compare the SCAs in our stackable file systems versus user-level tools, we used the file-copy micro-benchmark to compare the performance of the two stackable file systems with real SCAs, Gzipfs and Uuencodefs, against their equivalent user-level tools, `gzip` [8] and `uuencode`, respectively. In particular, the same Deflate algorithm and compression level (9) was used for both Gzipfs and `gzip`. In comparing Gzipfs and `gzip`, we measured both the compression time and the resulting space savings. Because the performance of compression algorithms depends on the type of input, we compared Gzipfs to `gzip` using the file-copy micro-benchmark on all four of the different file types discussed in Section 7.2.2.

7.3 General-Purpose Benchmark Results

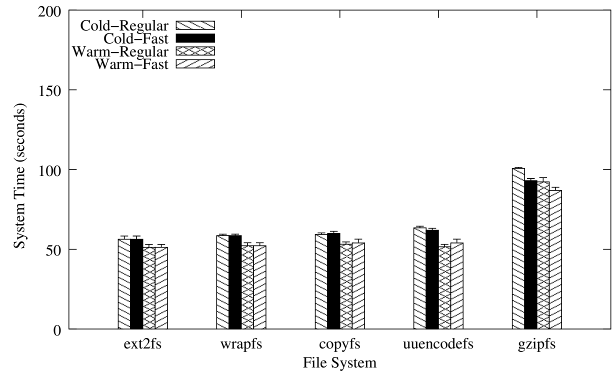
7.3.1 Am-Utils

Figure 5 summarizes the results of the am-utils benchmark. We report both system and elapsed times. The top part of Figure 5 shows system times spent by this benchmark. This is useful to isolate the total effect on the CPU alone, since SCA-based file systems change data size and thus change the amount of disk I/O performed. Wrapfs adds 14.4% overhead over Ext2, because of the need to copy data pages between layers. Copyfs adds only 1.3% overhead over Wrapfs; this shows that our index file handling is fast. Compared to Copyfs, Uuencodefs adds 7% overhead and Gzipfs adds 69.9%. These are the costs of the respective SCAs in use and are unavoidable—whether running in the kernel or user-level.

The total size of an unencoded build of am-utils is 22.9MB; a Uuencoded build is one-third larger; Gzipfs reduces this size by a factor of 2.66 to 8.6MB. So while Uuencodefs increases disk I/O, it does not translate to a lot of additional system time because the Uuencode algorithm is trivial. Gzipfs, while decreasing disk I/O, however, is a costlier algorithm than Uuencode. That's why Gzipfs's system time overhead is greater overall than Uuencodefs's. The additional disk I/O performed by Copyfs is small and relative to the size of the index file.

The bottom part of Figure 5 shows elapsed times for this benchmark. These figures are the closest to what users will see in practice. Elapsed times factor in increased CPU times the more expensive the SCA is, as well as changes in I/O that a given file system performs: I/O for index file, increased I/O for Uuencodefs, and decreased I/O for Gzipfs.

On average, the cost of data copying without size-changing (Wrapfs compared to Ext2fs) is an additional



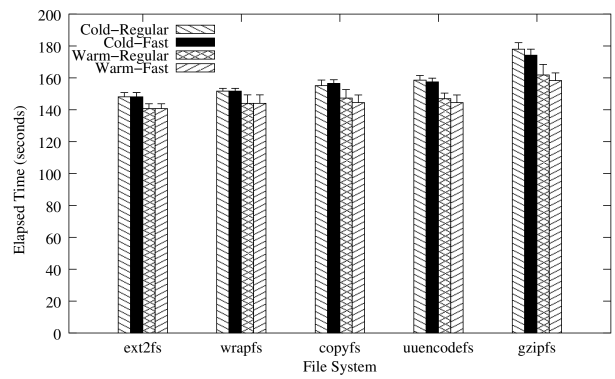


Figure 5: The Am-utils large-compile benchmark. Elapsed times shown on top and system times shown on bottom. The standard deviations for this benchmark were less than 3% of the mean.

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) [7], and it adds 14.7% overhead over Copyfs. Note that the elapsed-time overhead for Gzipfs is smaller than its CPU overhead (almost 70%) because whereas the Deflate algorithm is expensive, Gzipfs is able to win back some of that overhead by its I/O savings.

Using a warm cache improves performance by 5–10%. Using fast tails improves performance by at most 2%. The code that is enabled by fast tails must check, for each read or write operation, if we are at the end of the file, if a fast tail already exists, and if a fast tail is large enough that it should be encoded and a new fast tail started. This code has a small overhead of its own. For file systems that do not need fast tails (e.g., Copyfs), fast tails add an overhead of 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 7.4.2.

7.3.2 Bonnie

Figure 6 shows the results of running Bonnie on the five file systems. Since Bonnie exercises data reading and writing heavily, we expect it to be affected by the SCA in use. This is confirmed in Figure 6. Over all runs in this benchmark, Wrapfs has an average overhead of 20% above Ext2fs, ranging from 2–73% for the given files. Copyfs only adds an additional 8% average 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.

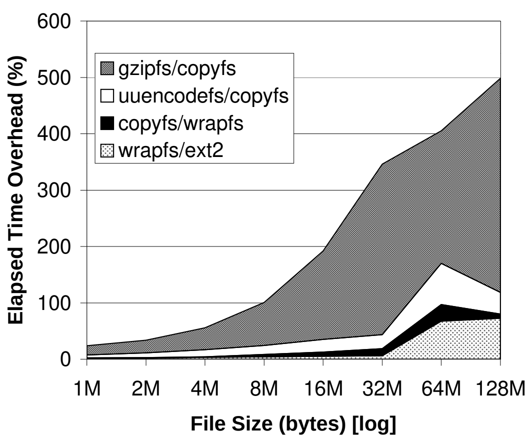


Figure 6: The Bonnie benchmark performs many repeated reads and writes on one file as well as numerous random seeks and writes in three concurrent processes. We show the total cumulative overhead of each file system. Note that the overhead bands for Gzipfs and Uuencodefs are each relative to Copyfs. We report the results for files 1MB and larger, where the overheads are more visible.

Figure 6 exhibits overhead spikes for 64MB files. Our test machines had 128MB of memory. Our stackable system caches two pages for each page of a file: one encoded page and one decoded page, effectively doubling the memory requirements. The 64MB files are the smallest test files that are large enough for the system to run out of memory. Linux keeps data pages cached for as long as possible. When it runs out of memory, Linux executes an expensive scan of the entire page cache and other in-kernel caches, purging as many memory objects as it can, possibly to disk. The overhead spikes in this figure occur at that time.

Bonnie shows that an expensive algorithm such as compression, coupled with many writes in the middle of large files, can degrade performance by as much as a factor of 5–6. In Section 9 we describe certain optimizations that we are exploring for this particular problem.

7.4 Micro-Benchmark Results

7.4.1 File-Copy

Figure 7 shows the results of running the file-copy benchmark on the different file systems. Wrapfs adds an average overhead of 16.4% over Ext2fs, which goes to 60% for a file size of 32MB; this is the overhead of data page copying. Copyfs adds an average 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 6) to support writes in the middle of files. The Uuencode algorithm adds an additional average overhead of 43.2% over Copyfs, and as much as 153% overhead for the large 32MB file. The linear overheads of Copyfs increase with the file's size due to the extra page copies that Copyfs must make, as explained in Section 6. 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, whereas fast-tails is intended to work better in situations with multiple small appends.

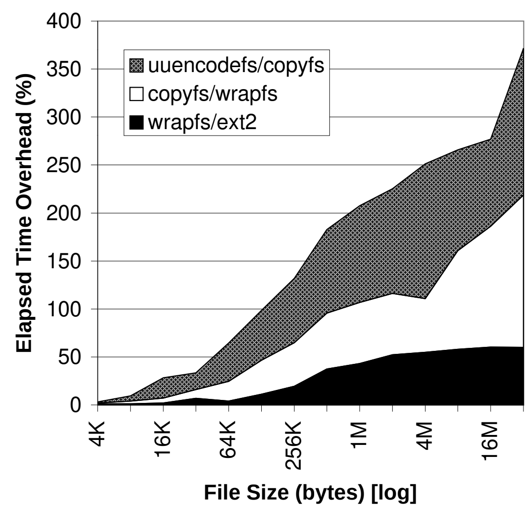


Figure 7: Copying files into a tested file system. As expected, Uuencodefs is costlier that Copyfs, Wrapfs, and Ext2fs. Fast-tails do not make a difference in this test, since we are not appending multiple times.

7.4.2 File-Append

Figure 8 shows the results of running the file-append benchmark on the different file systems. The figure 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

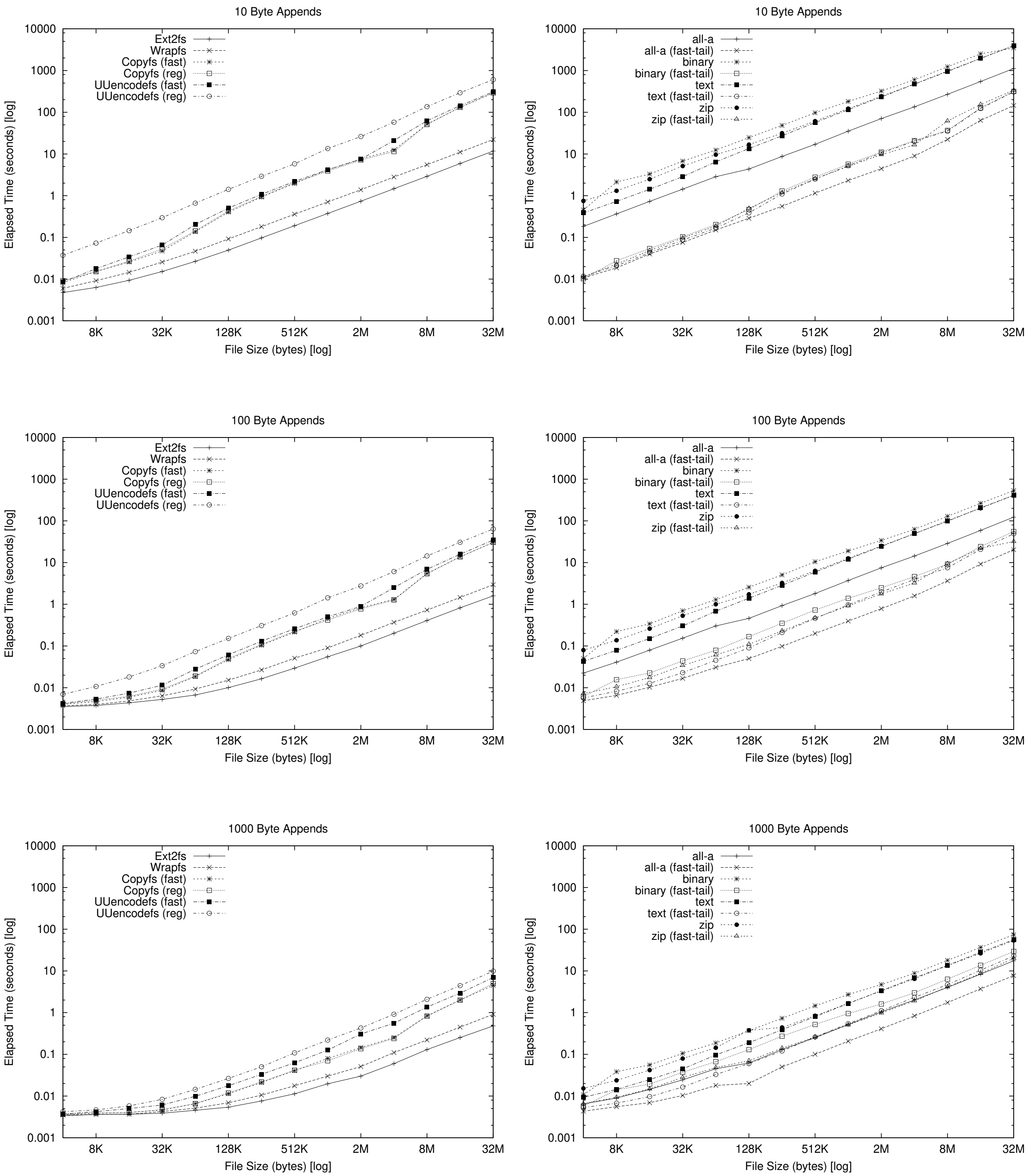


Figure 8: Appending to files. The left column of plots shows appends for Uuencodefs and Copyfs. The right column shows them for Gzipfs, which uses a more expensive algorithm; we ran Gzipfs on four different file types. The three rows of two plots each show, from top to bottom, appends of increasing sizes: 10, 100, and 1000 bytes, respectively. The more expensive the SCA is, and the smaller the number of bytes appended is, the more effective fast tails become; this can be seen as the trend from lower leftmost plot to the upper rightmost plot. The standard deviation for these plots did not exceed 9% of the mean.

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 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).

7.4.3 File-Attributes

Figure 9 shows the results of running the file-attributes benchmark on the different file systems. Wrapfs add an overhead of 35% to the GETATTR file system operation because it has to copy the attributes from one inode data structure into another. SCA-based file systems add the most significant overhead, a factor of 2.6–2.9 over Wrapfs; that is because Copyfs, Uuencodefs, and Gzipfs include stackable SCA support, managing the index file in memory and on disk. The differences between the three SCA file systems in Figure 9 are small and within the error margin.

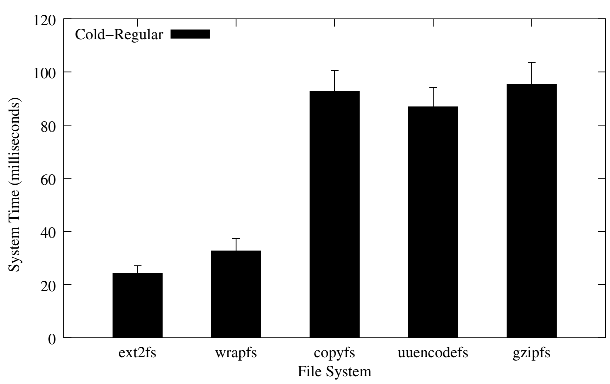


Figure 9: System times for retrieving file attributes using `lstat(2)` (cold cache)

While the GETATTR file operation is a popular one, it is still fast because the additional inode for the small index file is likely to be in the locality of the data file. Note that Figure 9 shows cold cache results, whereas most operating systems cache attributes once they are retrieved. Our measured speedup of cached vs. uncached attributes shows an improvement factor of 12–21. Finally, in a typical workload, bulk data reads and writes are likely to dominate any other file system operation such as GETATTR.

7.5 File System vs. User-Level Tool Results

Figure 10 shows the results of comparing Gzipfs against `gzip` using the file-copy benchmark. The reason Gzipfs is faster than `gzip` is primarily due to running in the kernel and reducing the number of context switches and kernel/user data copies.

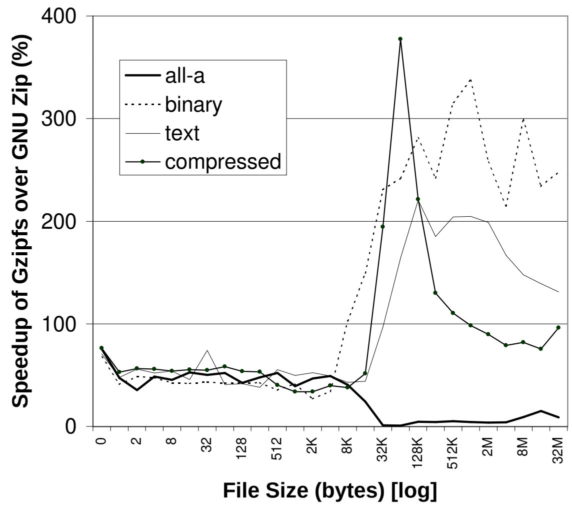


Figure 10: Comparing file copying into Gzipfs (kernel) and using `gzip` (user-level) for various file types and sizes. Here, a 100% speedup implies twice as fast.

As expected, the speedup for all files up to one page size is about the same, 43.3–53.3% on average; that is because the savings in context switches are almost constant. More interesting is what happens for files greater than 4KB. This depends on two factors: the number of pages that are copied and the type of data being compressed.

The Deflate compression algorithm is dynamic; it will scan ahead and back in the input data to try to compress more of it. Deflate will stop compressing if it thinks that it cannot do better. We see that for binary and text files, Gzipfs is 3–4 times faster than `gzip` for large files; this speedup is significant because these types of data compress well and thus more pages are manipulated at any given time by Deflate. For previously compressed data, we see that the savings is reduced to about double; that is because Deflate realizes that these bits do not compress easily and it stops trying to compress sooner (fewer pages are scanned forward). Interestingly, for the all-`a` file, the savings average only 12%. That is because the Deflate algorithm is quite efficient with that type of data: it does not need to scan the input backward and it continues to scan forward for longer. However, these forward-scanned pages are looked at few times, minimizing the number of data pages that `gzip` must copy between the user and the kernel. Finally, the plots in Figure 10 are not smooth because most of the input data is not uniform and thus it takes Deflate a different amount of effort to compress different bytes sequences.

One additional benchmark of note is the space savings for Gzipfs as compared to the user level `gzip` tool. 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, `gzip` achieves on average better compression ratios: as little as 4% better for compressing previously compressed data, to 56% for compressing the all-`a` file.

We also compared the performance of Uuencodefs to the user level `uuencode` utility. Detailed results are not presented here due to space limitations, but we found the performance savings to be comparable to those with Gzipfs compared to `gzip`.

7.6 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.

Finally, we checked the in-kernel memory consumption. As expected, the total number of pages cached in the page cache is the sum of the encoded and decoded files' sizes (in pages). This is because in the worst case, when all pages are warm and in the cache, the operating system may cache all encoded and decoded pages. For Copyfs, this means doubling the number of pages cached; for Gzipfs, fewer pages than double are cached because the encoded file size is smaller than the original file; for Uuencodefs, 2.33 times the number of original data pages are cached because the algorithm increased the data size by one-third. In practice, we did not find the memory consumption in stacking file systems on modern systems to be onerous [29].

8 Conclusions

The main contribution of our work is demonstrating that SCAs 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 SCAs 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 SCAs, 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 [25, 29].

9 Future Work

We are investigating methods of improving the performance of writes in the middle of files by decoupling the order of the bytes in the encoded file from their order in the original file. By decoupling their order, we could move writes in the middle of files elsewhere—say the end of the file (similar to a journal) or an auxiliary file. Another alternative is to structure the file differently internally: instead of a sequential set of blocks, it could be organized as a B-tree or hash table where the complexity order of insertions in the middle is sub-linear. These methods would allow us to avoid having to shift bytes outward to make space for larger encoded units. However, if we begin storing many encoded chunks out of order, large files could get fragmented. We would need a method for compaction or coalescing all these chunks into a single sequential order.

An important optimization we plan to implement 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.

10 Acknowledgments

We would like to thank Jerry B. Altzman for his initial input into the design of the index table. We thank John Heidemann for offering clarification regarding his previous work in the area of stackable filing. Thanks go to the Usenix reviewers and especially our shepherd, Margo Seltzer. This work was supported in part by NSF CISE Research Infrastructure grant EIA-9625374, an NSF CAREER award, and Sun Microsystems.

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. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 2–9.

[4] 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.

[5] V. Cate and T. Gross. Combining the concepts of compression and caching for a two-level filesystem. *Fourth ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA), pages 200-211.

[6] R. Coker. The Bonnie Home Page. http://www.textuality.com/bonnie.

[7] P. Deutsch. Deflate 1.3 Specification. RFC 1051. 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. *Proceedings of the Summer USENIX Technical Conference*, pages 63–71, Summer 1990.

[11] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Proceedings of Fifteenth ACM Symposium on Operating Systems Principles*. 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, February 1994.

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

[15] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the Summer USENIX Technical Conference*, pages 238–47, Summer 1986.

[16] 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 Conference Proceedings*, 1994.

[17] 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.

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

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

[20] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. *Proceedings of the Annual USENIX Technical Conference*, June 2000.

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

[22] D. S. H. Rosenthal. Evolving the Vnode Interface. *Proceedings of the Summer USENIX Technical Conference*, pages 107–18, Summer 1990.

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

[24] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *Proceedings of the Summer USENIX Technical Conference*, pages 161–74, June 1993.

[25] E. Zadok. *FiST: A System for Stackable File System Code Generation*. PhD thesis. Columbia University, May 2001.

[26] E. Zadok and I. Bădulescu. A Stackable File System Interface for Linux. *LinuxExpo Conference Proceedings*, 1999.

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

[28] E. Zadok, I. Bădulescu, and A. Shender. Extending File Systems Using Stackable Templates. *Proceedings of the Annual USENIX Technical Conference*, June 1999.

[29] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. *Proceedings of the Annual USENIX Technical Conference*, June 2000.

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