The Desktop File System

Morgan Clark *Stephen Rago*

Programmed Logic Corporation
200 Cottontail Lane
Somerset, NJ 08873

Abstract

This paper describes the structure and performance characteristics of a commercial file system designed for use on desktop, laptop, and notebook computers running the UNIX operating system. Such systems are characterized by their small disk drives dictated by system size and power requirements. In addition, these systems are often used by people who have little or no experience administering Unix systems. The Desktop File System attempts to improve overall system usability by transparently compressing files, increasing file system reliability, and simplifying administrative interfaces. The Desktop File System has been in production use for over a year, and will be included in future versions of the SCO Open Desktop Unix system. Although originally intended for a desktop environment, the file system is also being used on many larger, server-style machines.

1. Overview

This paper describes a commercial file system designed for use on desktop, laptop, and notebook computers running the UNIX operating system. We describe design choices made and discuss some of the interesting ramifications of those choices. The most notable characteristic of the file system is its ability to compress and decompress files ''on-the-fly.'' We provide performance information that proves such a file system is a viable option in the Unix marketplace.

When we use the term ''commercial file system,'' we mean to imply two things. First, the file system is used in real life. It is not a prototype, nor is it a research project. Second, our design choices were limited to the scope of the file system. We were not free to rewrite portions of the base operating system to meet our needs, with one exception (we provided our own routines to access the system buffer cache).

1.1 Goals

Our goals in designing the Desktop File System (DTFS) were influenced by our impressions of what the environment was like for small computer systems, such as desktop and laptop computers. The physical size of these systems limits the size of the power supplies and hard disk drives that they can use at a reasonable cost. Systems that are powered by batteries attempt to use small disks to minimize the power drained by the disk, thus increasing the amount of time that the system can be used before requiring that the batteries be recharged.

It is common to find disk sizes in the range of 80 to 300 Megabytes in current 80x86-based laptop and notebook systems. Documentation for current versions of UnixWare recommend a minimum of 80 MB of disk space for the personal edition, and 120 MB for the application server. Similarly, Solaris documentation stipulates a minimum of 200 MB of disk space. These recommendations do not include space for additional software packages.

We also had the impression that desktop and notebook computers were less likely to be administered properly than larger systems in a general computing facility, because the primary user of the systems will probably be performing the administrative procedures, often without the experience of professional system administrators. These impressions led us to the following goals:

- Reduce the amount of disk space needed by conventional file systems.
- Increase file system robustness in the presence of abnormal system failures.
- Minimize any performance degradation that might arise because of data compression.
- Simplify administrative interfaces when possible.

The most obvious way to decrease the amount of disk space used was to compress user data. (We

use the term ‘‘user data’’ to refer to the data read from and written to a file, and we use the term ‘‘meta data’’ to refer to accounting information used internally by the file system to represent files.) Our efforts did not stop there, however. We designed the file system to allocate disk inodes as they are needed, so that no space is wasted by unused inodes. In addition, we use a variable block size for user data. This minimizes the amount of space wasted by partially-filled disk blocks.

Our intent in increasing the robustness of the file system stemmed from our belief that users of other desktop systems (such as MS-DOS) would routinely shut their systems down merely by powering off the computer, instead of using some more gradual method. As it turned out, our eventual choice of file structure required us to build robustness in anyway.

From the outset, we realized that any file system that added another level of data processing would probably be slower than other file systems. Nevertheless, we believed that this would not be noticeable on most systems because of the disparity between CPU and disk I/O speeds. In fact, current trends indicate that this disparity is widening as CPU speeds increase at a faster pace than disk I/O speeds [KAR94]. Most systems today are bottlenecked in the I/O subsystem [OUS90], so the spare CPU cycles would be better spent compressing and decompressing data.

Our assumptions about typical users led us to believe that users would not know the number of inodes that they needed at the time they made a file system, and some might not even know the size of a given disk partition. We therefore endeavored to make the interfaces to the file system administrative commands as simple as possible.

1.2 Related Work

Several attempts to integrate file compression and decompression into the file system have been made in the past. [TAU91] describes compression as applied to executables on a RISC-based computer with a large page size and small disks. While compression is performed by a user-level program, the kernel is responsible for decompression when it faults a page in from disk. Each memory page is compressed independently, with each compressed page stored on disk starting with a new disk block. This simplifies the process of creating an uncompressed in-core image of a page from its compressed disk image. DTFS borrows this technique.

The disadvantage with this work is that it doesn’t fully integrate compression into the file system. Only binary executable files are compressed, and applications that read the executables see compressed data. These files must be uncompressed before becoming intelligible to programs like debuggers, for example. The compression and decompression steps are not transparent.

In [CAT91], compression and decompression are integrated into a file system. Files are compressed and decompressed in their entirety, with the disk split into two sections: one area contains compressed file images, and the other area is used as a cache for the uncompressed file images. The authors consider it prohibitively expensive to decompress all files on every access, hence the cache. This reduces the disk space that would have been available had the entire disk been used to hold only compressed images.

The file system was prototyped as an NFS server. Files are decompressed when they are first accessed, thus migrate from the compressed portion of the disk to the uncompressed portion. A daemon runs at off-peak hours to compress the files least-recently used, and move them back to the compressed portion of the disk.

Transparent data compression is much more common with the MS-DOS operating system. Products like Stacker have existed for many years. They are implemented as pseudo-disk drivers, intercepting I/O requests, and applying them to a compressed image of the disk [HAL94].

We chose not to implement our solution as a pseudo-driver because we felt that a file system implementation would integrate better into the Unix operating system. For example, file systems typically maintain counters that track the number of available disk blocks in a given partition. A pseudo-driver would either have to guess how many compressed blocks would fit in its disk partition or continually try to fool the file system as to the number of available disk blocks. No means of feedback is available, short of the pseudo-driver modifying the file system’s data structures. If the pseudo-driver were to make a guess at the time a file system is created, then things would get sticky if files didn’t compress as well as expected.

In addition, a file system implementation gave us the opportunity to employ transaction processing techniques to increase the robustness of the file system in the presence of abnormal system failures. A pseudo-driver would have no knowledge of a given file system’s data structures, and thus would have no way of associating a disk block address with any other related disk blocks. A file system, on the other hand, could employ its knowledge about the interrelationships of disk blocks to ensure that files are always kept in a consistent state.

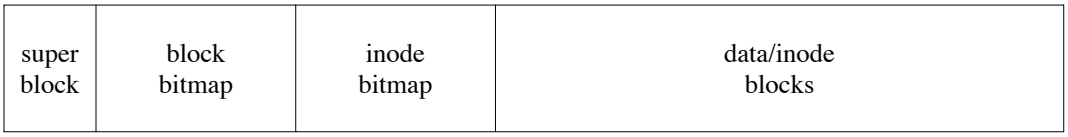

Figure 1. File System Layout

Much more work has gone into increasing file system robustness than has gone into integrating compression into file systems. Episode [CHU92], WAFL [HIT94], and the Log-structured File System [SEL92] all use copy-on-write techniques, also known as shadow paging, to keep files consistent. Copies of modified disk blocks are not associated with files until the associated transaction is committed. (Shadow paging is discussed in Section 5.1.)

2. File System Layout

Figure 1 shows the file system layout of DTFS. The super block contains global information about the file system, such as the size, number of free blocks, file system state, etc. The block bitmap records the status (allocated or free) of each 512-byte disk block. Similarly, the inode bitmap, which is the same size as the block bitmap, identifies the disk blocks that are used for inodes. Finally, the rest of the disk partition is available for use as user data and meta data.

The only critical information kept in the super block is the size of the file system. The rest of the information can be reconstructed by `fsck`. Similarly, the block bitmap can be reconstructed by `fsck`, should it be corrupted by a bad disk block. The inode bitmap is more critical, however, as `fsck` uses it as a guide to identify where the inodes are stored in the file system (inode placement is discussed in the next section). If the inode bitmap is corrupted, files will be lost, so users ultimately have to rely on backups to restore their files.

2.1 Inode Placement

As stated previously, one way DTFS saves space is by not preallocating inodes. Any disk block is fair game to be allocated as an inode. This has several interesting repercussions.

First, there is no need for an administrator to guess how many inodes are needed when making a file system. The number of inodes is only limited by the physical size of the disk partition and the size of the identifiers used to represent inode numbers. This simplifies the interface to `mkfs`.

Second, the inode allocation policy renders the NFS generation count mechanism [SAN85] ineffective. In short, when a file is removed and its link count goes to 0, an integer, called the *generation count*, in the disk inode is incremented. This generation count is part of the file handle used to represent the file on client machines. Thus, when a file is removed, active handles for the file are made invalid, because newly-generated file handles for the inode will no longer match those held by clients.

With DTFS, the inode block can be reallocated as user data, freed, and then reallocated as an inode again. Thus, generation counts cannot be retained on disk. We solve this problem by replacing the generation count with the file creation time concatenated with a per-file-system rotor that is incremented every time an inode is allocated.

2.2 File Structure

A DTFS file is stored as a B^+tree [COM79], with the inode acting as the root of the tree. Two factors led us to this design. The first of these is the need to convert between logical file offsets and physical data offsets in a file. For example, if an application were to open a 100000-byte and seek to byte position 73921, we would need an efficient way to translate this to the disk block containing that data. Since the data are compressed, we cannot simply divide by the logical block size to determine the logical block number of the disk block containing the data.

A simple solution would be to start at the beginning of the file and decompress the data until the requested offset is reached. This, however, would waste too much time. To make the search more efficient, we use logical file offsets as the keys in the B^+tree nodes. The search is bounded by the height of the tree, so it is doesn't cost any more to find byte *n+100000* than it costs to find byte *n*.

The second factor is a phenomenon we refer to as *spillover*. Consider what happens when a file is overwritten. The new data might not compress as well as the existing data. In this case, we might have to allocate new disk blocks and insert them into the tree. With the B^+tree structure, we can provide an efficient implementation for performing insertions and deletions.

When a file is first created, it consists of only an inode. Data written to the file are compressed, stored in disk blocks, and the disk block addresses

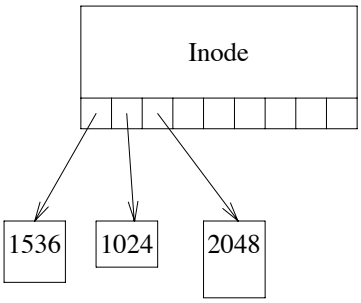

Figure 2. A One-level File

and lengths are stored in the inode (see Figure 2). DTFS uses physical addresses instead of logical ones to refer to disk blocks, because a ''block'' in DTFS can be any size from 512 bytes to 4096 bytes, in multiples of 512 bytes. In other words, a disk block is identified by its starting sector number relative to the beginning of the disk partition.

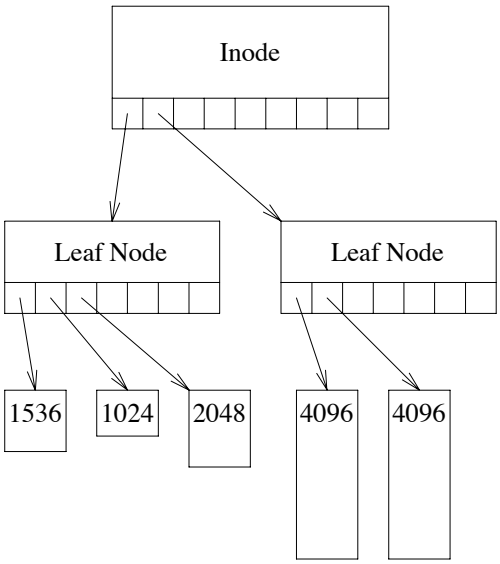

Figure 3. A Two-level File

When the inode's disk address slots are all used, the next block to be added to the file will cause the B^+tree to grow one level. (See Figure 3; note that most links between nodes are omitted for clarity). Two disk blocks are allocated to be used as leaf nodes for the tree. The data blocks are divided between the two leaf nodes, and the inode is modified to refer to the two leaf nodes instead of the data blocks. In this way, as the tree grows, all data blocks are kept at the leaves of the B^+tree.

Figure 4 shows what would happen if the B^+tree were to grow an additional level. The inode now refers to interior nodes of the B^+tree. Interior nodes either refer to leaf nodes or to other interior nodes. Each entry in an interior node refers to a subtree of the file. The key in each entry is the maximum file offset represented by the subtree. The keys are used to navigate the B^+tree when searching for a page at a particular offset. The search time is bounded by the height of the B^+tree.

3. Compression

DTFS is implemented within the Vnodes architecture [KLE86]. DTFS compresses the data stored in regular files to reduce disk space requirements (directories remain uncompressed). Compression is performed a page at a time, and is triggered during the vnode `putpage` operation. Similarly, decompression occurs during the `getpage` operation. Thus, compression and decompression occur ''on-the-fly.''

The choice of compressing individual pages limits the overall effectiveness of adaptive compression algorithms that require a lot of data before their tables are built. Nonetheless, some algorithms do quite well with only a page of data. This was the natural design choice for DTFS, since it was originally designed for UNIX System V Release 4 (SVR4), an operating system whose fundamental virtual memory abstraction is the page. (On an Intel 80x86 processor, SVR4 uses a 4KB page size.)

Each page of compressed data is represented on disk by a *disk block descriptor* (DBD). The DBD contains information such as the logical file offset represented by the data, the amount of compressed data stored in the disk block, and the amount of uncompressed data represented. The DBDs exist only in the leaf nodes of the B^+tree.

Because each page is compressed individually, DTFS is best suited to decompression algorithms that build their translation tables from the compressed data, thus requiring no additional on-disk storage. The class of algorithms originated by Lempel and Ziv [ZIV77], [ZIV78] are typical of such algorithms.

The original version of DTFS only supported two compression algorithms (an LZW derivative [WEL84] and ''no compression''). The latest version of DTFS allows for multiple compression algorithms to be used at the same time. We designed an application programming interface so that customers can add their own algorithms if they should so desire.

4. Block Allocation

With the exception of the super block and bitmaps (recall Figure 1), every disk block in a DTFS file system can be allocated for use as an inode, other meta data, or user data. The basic allocation mechanism is

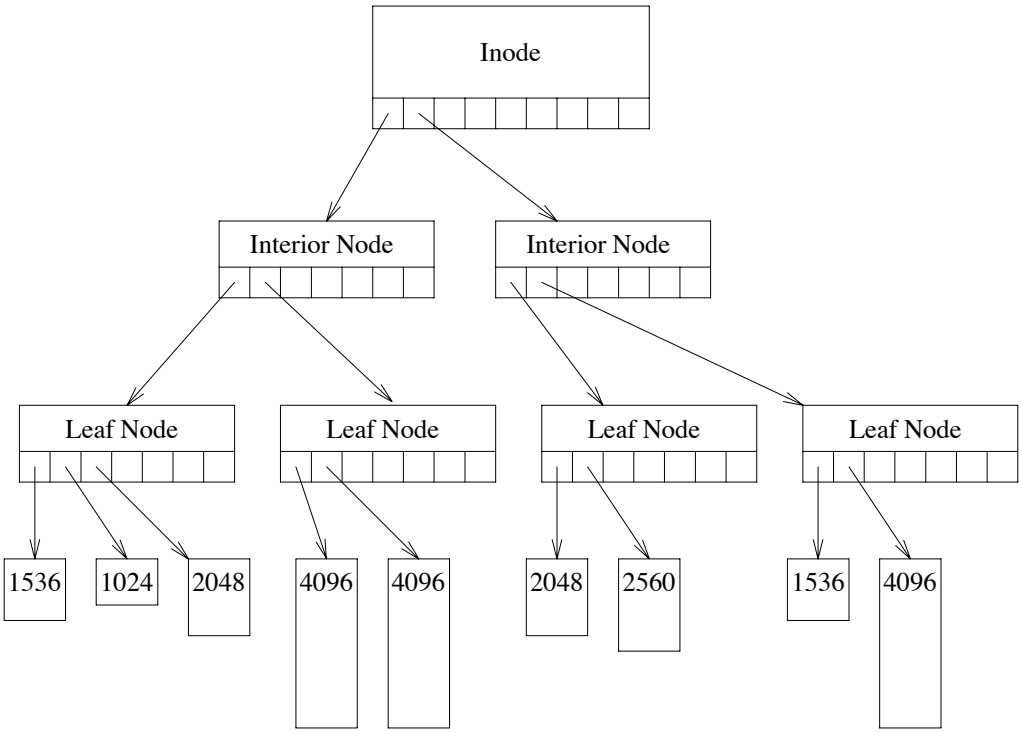

Figure 4. A Three-level File

simple — the file system keeps track of the first available block in the partition and allocates blocks starting from there. Requests to allocate multiple blocks search the file system for a free contiguous region at least as large as the number of blocks requested. If no region is found to be large enough, allocation switches to a first-fit algorithm that returns as many blocks as are contiguously available, starting with the first free block.

Unmodified, this allocation mechanism would cause severe file system fragmentation for several reasons. First, disk blocks backing a page must be allocated at the time a file is written, so that errors like `ENOSPC` can be returned to the application. Usually pages are written to disk sometime later, as the result of page reuse or file system hardening. If DTFS were to defer block allocation until it wrote a page to disk, then DTFS would be unable to inform the application of a failure to allocate a disk block.

With dynamic data compression, after each page is compressed, the leftover blocks are freed back to the file system. So for every eight-block page there might be a run of, say, four blocks allocated followed by four blocks freed. As each page requires eight disk blocks before compression, no other page would use any of the four-free-block regions until the entire file system is fragmented and allocation switches to first-fit.

The second contributor to fragmentation is the shadow paging algorithm, which requires that every overwritten page have new blocks allocated before the old blocks are freed. The new blocks would be allocated from some free region, most likely in a different place in the file system.

The final reason for fragmentation is that any block can be used for any purpose, so inode blocks and user data blocks could be intermingled. This is partly mitigated by the fact that inode and meta data blocks can use the few-block regions between compressed user data blocks.

We use several techniques to avoid fragmentation. First, user data blocks are allocated in clusters, one cluster per inode. When the DTFS `write` function attempts to allocate disk space for a user data page, a much larger chunk of space (typically 32 blocks) is actually allocated from the general disk free space pool, and the blocks for the page in question are allocated from this cluster. The remainder of the cluster is saved in the in-core inode for use by later allocations. When the inode is no longer in use, the unused blocks in the cluster are returned to the file system. This has the effect of localizing disk space belonging to a particular file.

Second, after compression, user data blocks are reassigned from within the cluster to eliminate any local fragmentation. For example, if the first page of

a file allocated blocks 20–27 and the second page allocated blocks 28–35, and after compression each page needed the first four blocks of their respective eight-block region, the second page would be reassigned to use blocks 24–27, eliminating the four-block fragment of free space.

Finally, we keep two first-free block indicators, and allocate inode blocks starting from one point and user data blocks from another.

5. Reliability

Our goal of increasing file system reliability was originally based on our belief that naive uses might not shut a system down properly. File systems like S5 (the traditional System V file system [BAC86]) and UFS (The System V equivalent of the Berkeley Fast File System [MCK84]) can leave a file with corrupted data in it if the system halts abnormally. Unless an application is using synchronous I/O, when an application writes to a file such that a new disk block needs to be allocated, file systems usually write both the user data and the meta data in a delayed manner, caching them in mainstore and writing them out to disk later, to improve performance.

The problem with this approach is that an abnormal system halt can leave a file containing garbage. Consider what happens when the inode's meta data are written to disk, but the system halts abnormally before the user data are written. (This case is more likely to occur than one might think — the system tries to flush ''dirty'' pages and buffers to disk over a tunable time period, usually around 60 seconds. Thus, in the worst case, it is possible for 60 seconds to elapse before user data are flushed to disk.) In this case, the inode's meta data will reflect that a newly allocated disk block contains user data, but the actual contents of the disk block have not been written yet, so the file will end up containing random user data. The contents of the disk block can be as innocuous as a block of zeros, or as harmful as a block that came from another user's deleted file, thus presenting a security hole. (When a block is freed or reused for user data, its contents are usually not cleared.)

This reliability problem can be solved by carefully ordering the write operations. If the user data are forced out to disk before the inode's meta data, then the file will never refer to uninitialized disk blocks. The ordering must be implemented carefully, because forcing the user data out to disk before the meta data can hurt performance. On the other hand, delaying the update of the meta data until the user data have found their way to disk can cause changes to the inode to be lost entirely if the system crashes.

Our choice of compressing user data, as it turned out, forced us to increase reliability to keep the user data and meta data in sync. Because a disk block can contain a variable amount of data, and because that data, once compressed, can represent an amount of data greater than the size of the disk block, we require that the meta data and the user data be in sync at all times. The meta data describe the compression characteristics, and if the information were to be faulty, the decompression algorithm might react in unpredictable ways, possibly leading to data corruption or a system panic.

5.1 Shadow Paging

To solve the problem of keeping a file's meta data and user data in sync, we decided to use *shadow paging* [EPP90], [GRA81], with the intent of providing users with a consistent view of their files at all times.

Shadow paging is a technique that can be used to provide atomicity in transaction processing systems. A *transaction* is defined to be a unit of work with the following ACID characteristics [GRA93]: Atomicity, Consistency, Isolation, and Durability.

The atomicity property guarantees that transactions either successfully complete or have no effect. When a transaction is interrupted because of a failure, any partial work done up to that point is undone, causing the state to appear as if the transaction had never occurred. The consistency property ensures that the transaction results in a valid state transition.

The isolation property (also called serializability) guarantees that concurrent transactions do not see inconsistencies resulting from the possibly multiple steps that make up a single transaction. (A single transaction is usually made up of multiple steps.) DTFS uses inode locks to provide isolation.

The durability property (also known as permanence) guarantees that changes made by committed transactions are not lost because of hardware errors. This is usually implemented through techniques, such as disk mirroring, that employ redundant hardware. DTFS does not provide permanence.

Shadow paging provides only the atomicity property. It works in the following way: when a page is about to be modified, extra blocks are allocated to shadow it. When the page is written out, it is actually written to the shadow blocks, leaving the original blocks unmodified. File meta data are modified in a similar manner, except that the node and its shadow share the same disk block.

When an inode is updated, the user data are flushed to disk. Then the modified meta data are written, followed by a synchronous write of the inode (the root of the B^+tree). Finally, all the original data

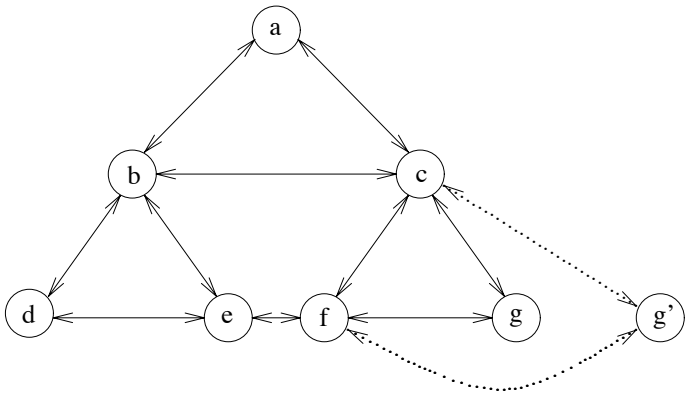

Figure 5. Partial B$^+$tree

blocks are freed. If an abnormal system halt occurs before the inode is written to disk, then the file appears as it was at the time of the previous update.

The shadow of a node is stored in the same disk block as the original node is to avoid a ripple effect when modifying part of a file. In Figure 5, node *g* is shadowed by node *g'*. When node *g'* is added to the tree, we would have to allocate new nodes, *c'* and *f'*, when we updated the nodes with the new pointer to *g'*. Then we would have to update all the nodes that have pointers to *c* and *f*, and so on, until the entire tree has been updated.

We avoid all this extra work by design: *g* and *g'* reside in the same disk block (see Figure 6). This means that when we decide to use *g'* instead of *g*, the nodes that point to *g* don't have to be updated. The way we determine whether to use *g* or *g'* is by storing a timestamp in each node to describe which is the most recently modified of the pair.

Every time we write a node to disk, the inode's timestamp is incremented and copied to the node's timestamp field. Before the inode is written to disk, its timestamp is incremented. Any nodes with a timestamp greater than that of the inode were written to disk without the inode being updated, so they are ignored. Otherwise, we choose the node with the largest timestamp of the two. In the event of a system failure, `fsck` will rebuild the block bitmap, and the user data duplicated by shadowing will be freed.

5.2 Quiescence, Sync-on-Close, and the Update Daemon

To increase reliability, we implemented a kernel process, called the *update daemon*, that checkpoints every writable DTFS file system once per second. The daemon performs a sync operation on each file system, with the enhancement that if the sync completes successfully without any other process writing to the file system during that time, the file system state is set to ''clean'' and the super block is written out to disk.

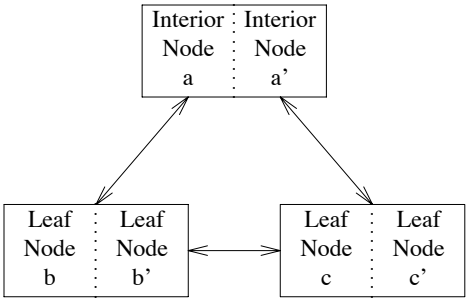

Figure 6. Meta Data Shadows

From this point until the next time any process writes to the file system, a system crash will not require that `fsck` be run on that file system. Studies of our main development machine, with DTFS on all user data partitions, showed that the file systems were in the clean state more than 98% of the time. (DTFS supports an `ioctl` that reports the file system state.)

Another reliability enhancement was to write every file to disk synchronously, once the last reference to the file was released. This mimics the behavior under MS-DOS that once a user's application has exited, the machine can be turned off without data loss. This turned out to cause significant performance degradation, so instead we assigned this task to the update daemon. As the update daemon runs once every second, there can be at most a one-second delay after a user exits his or her applications until the machine is safe to power off. The original sync-on-close semantics are still available as a mount option for even greater reliability.

At first glance, checkpointing each writable file

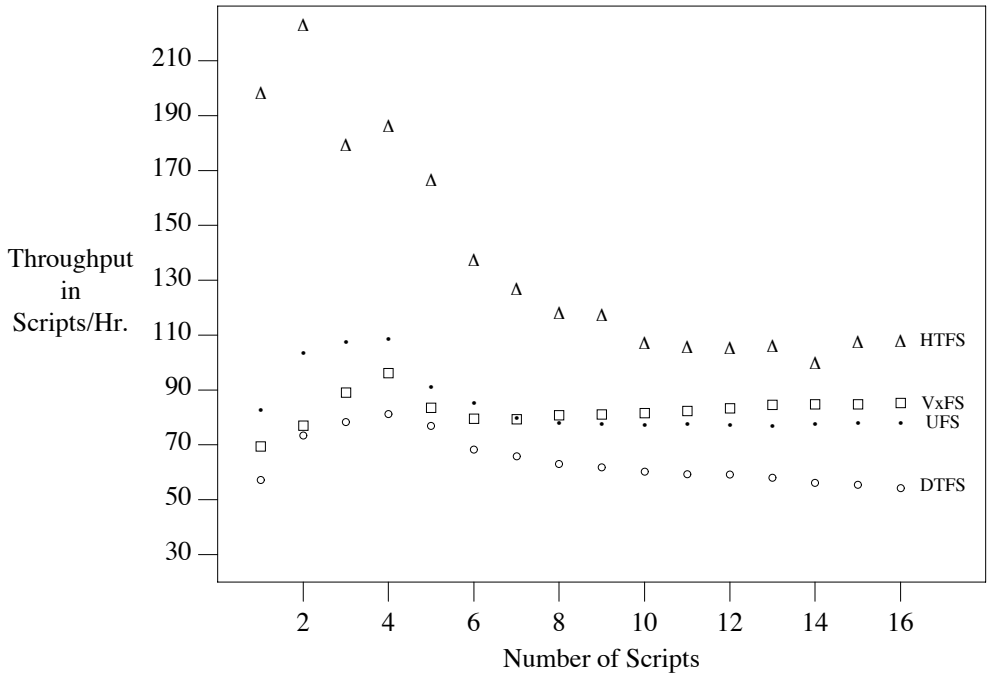

Figure 7. SDET Results

system every second appears to be an expensive proposition, but in practice it turns out to be relatively minor. Our studies have shown that the performance degradation in throughput benchmarks such as the SPEC consortium's SDET benchmark is on the order of ten percent when compared to running with the update daemon disabled. In addition, the update daemon only runs when there are recent file system modifications to be checkpointed, so if there is no DTFS activity, the update daemon overhead is zero.

6. Performance

We compared DTFS to other commercial file systems using the SPEC consortium's SDET benchmark. It simulates a software development environment, measuring overall system throughput in terms of the total amount of work that can be performed in a given unit of time. We think SDET is a good benchmark to measure file system performance because it models real-world system usage, which just happens to exercise the file system quite rigorously.

Our tests were run on an EISA-based 66MHz 80486 machine with an Adaptec 1742 SCSI disk controller in enhanced mode, a 5400 RPM disk with an average access time of 9.5 ms, and 32 MB of memory. We used UnixWare version 1.1 as the host operating system. Figure 7 summarizes the results of the benchmark using the UFS file system (with the file system parameters optimized to match the disk geometry and speed), the HTFS file system (one of our commercial file systems, a UFS-compatible high-throughput file system using intent logging), the VxFS file system (the default UnixWare file system), and the DTFS file system.

DTFS did not perform as well as the other file systems, but its performance was still respectable. At 2 scripts, it performs almost as well as VxFS, and at 5 scripts it reaches 85% of the UFS throughput. At the peak, it attained 74% of the throughput of UFS, 83% of the throughput of VxFS, but only 43% of the throughput of HTFS. Originally we thought the performance loss was from the costs associated with compressing and decompressing user data. When we ran the benchmark with compression disabled, we were surprised to find that performance was slightly worse than with compression enabled. We attribute this to the additional I/O required when files are not compressed, and conclude that the CPU has enough bandwidth to compress and decompress files on-the-fly. The bottleneck in DTFS is the disk layout. We can overcome this by reorganizing our disk layout to perform fewer, but larger, I/O requests (a version number in the DTFS super block allows us to modify the layout of the file system and still support existing formats).

These conclusions are supported by the system timing results summarized in Table 1. The times are expressed in elapsed seconds to facilitate comparison.

Note that with all of the file systems, the benchmark spent about the same time at user level (ignoring rounding errors), which is what we would expect for identical workloads. DTFS spent the most time in the kernel, probably because of the compression and decompression. DTFS also had the most wait-I/O time.

File System	User Time	System Time	Wait-I/O Time	Idle Time
DTFS	24	61	88	0
HTFS	24	36	17	0
UFS	25	41	66	0
VxFS	24	51	67	0

Table 1. SDET Times in Seconds (4 Scripts)

We originally intended to illustrate the effect of the disparity between CPU and I/O speeds by repeating the benchmark with a slower disk drive, expecting the performance of DTFS to be closer to that of the other file systems. With the current disk layout it is not possible to meet this expectation, because the wait-I/O time for DTFS dominates the system time. Nonetheless, we are confident that once we reorganize the disk layout, we will be more able to support our belief that the overhead of performing data compression in the file system will become negligible as CPU speeds increase at a faster pace than I/O speeds.

Since the SDET benchmark measures overall system throughput, the effects of any one component, such as the file system, are diminished compared to benchmarks that isolate that component. For the purposes of comparison, we ran isolated file system throughput tests on the original hardware configuration. Our read test opens a file, optionally invalidates any pages from that file cached in-core, and reads the file. The read size and the total amount of data to be read are configurable. Similarly, our write test creates a file, and writes a data pattern to it in the increment specified, until the file reaches the requested size. Then the test optionally syncs the data to disk and invalidates the pages in-core.

We used a file size of 1 MB and two different I/O sizes to measure the raw throughput through each file system. The results are shown in Table 2. The first DTFS entry was derived using a pattern that only compresses by 25%. For comparison, the second DTFS entry shows the I/O throughput using a pattern that compresses by 85%.

For small writes, DTFS is between 2.0 and 4.0 times faster than VxFS, but is between 2.6 and 4.6 times slower than HTFS and UFS. For large writes,

File System	write 4096	read 4096	write 65536	read 65536
DTFS(25%)	235	366	269	373
DTFS(85%)	416	716	428	731
HTFS	1071	1870	1078	1687
UFS	1085	1860	1085	1678
VxFS	130	2490	471	1830

Table 2. Raw Throughput in KB/s

DTFS is between a factor of 1.1 and 1.8 times slower than VxFS, and is between 2.5 and 4.0 times slower than HTFS and UFS. For reads, DTFS is anywhere from 2.3 to 7.0 times slower than the other file systems, but this is hidden in a system-level benchmark where the page cache is active. Table 3 illustrates that when the page cache is used, DTFS is as fast as the other file systems (or, more correctly, the DTFS `read` vnode operation is as efficient as HTFS and UFS, and better than VxFS, since the reads are satisfied entirely by the page cache). This is one reason why system-level benchmarks like SDET show reasonable performance for DTFS (in other words, the page cache works).

File System	read 4096	read 65536
DTFS	6400	7800
HTFS	6400	7800
UFS	6400	7800
VxFS	6000	7300

Table 3. Cached Raw Throughput (KB/s)

Table 4 summarizes the effectiveness of the DTFS compression on different classes of files. The comparison is against a UFS file system with a block size of 4KB and a fragment size of 1KB. DTFS doesn't compress symbolic links, but the average size of a symbolic link is less than the average size of a symbolic link in a UFS file system. The 50% reduction in size is because a symbolic link requires one fragment (2 disk blocks) in UFS, and in DTFS it only requires 1 to 3 disk blocks, depending on the length of the pathname (short pathnames are stored directly in the inode, and most symbolic links on our system happen to be short).

Table 4. Sample Compression Statistics

When DTFS reports the number of disk blocks needed to represent a file, it adds one for the disk block containing the file's inode. Other file systems store multiple inodes per disk block, so cannot

File Type	Average Compression vs. 4K UFS
`a.outs`	36 %
C Source Files	37 %
Log Files	64 %
Man pages	44 %
Symbolic Links	50 %
X font binaries	53 %

attribute the space to each file as easily. This tends to make comparisons of small files appear as if DTFS cannot compress them, and some even appear as if they require more disk space with DTFS. This is misleading when you consider that DTFS doesn't have to waste disk space preallocating unused inodes.

For example, on a freshly-made 100000-block file system, UFS uses 6.0% of the disk for preallocated meta data. In comparison, VxFS uses about 7.2% of the disk, but DTFS only uses 0.052% of the disk for preallocated meta data.

The bar chart in Figure 8 illustrates the effectiveness of using DTFS as the default file system in a freshly-installed UnixWare system. For the other file systems, the default block sizes were used.

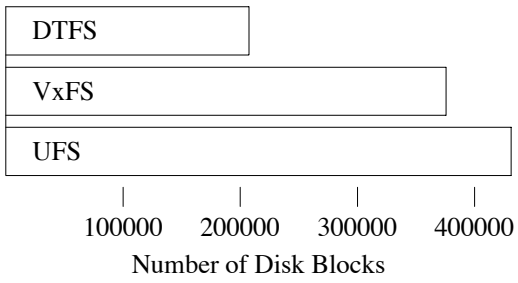

Figure 8. Sample Disk Usage

7. Administrative Interfaces

During the design of DTFS, we simplified several administrative interfaces, mainly because we believed that typical desktop and laptop users are not interested in becoming full-time system administrators, and even if they were, they probably didn't have the necessary skills. Thus, we modified several administrative commands to make them easier to use.

Our `fsck` doesn't prompt the user for directions on how to proceed. The DTFS `fsck` simply tries to fix anything it finds wrong. In addition, no `lost+found` directory is needed with a DTFS file system. Each DTFS inode contains a one-element cache of the last file name associated with the inode, along with the inode number for that file's parent directory. When `fsck` finds that an inode has no associated directory entry, `fsck` creates the entry where it last existed. This frees users from having to use the inode number and file contents to figure out which files have been reconnected by `fsck`.

The DTFS `mkfs` command will automatically determine the size of a disk partition, freeing users from having to provide this information. An option exists to allow users to force a file system to be smaller than the disk partition containing it.

Our `mkfs` has no facility to allow users to configure the number of inodes for a particular file system. DTFS allocates inodes as they are needed, and any disk block can be allocated as an inode, so the only bound on the number of inodes for a given file system is the number of available disk blocks.

The DTFS `fsdb` command is much easier to use than the `fsdb` found with the S5 file system. (We don't expect most users to need `fsdb`, much less have the skills to use it, but some customers have requested that we provide it anyway.) The DTFS `fsdb` presents the user with a prompt, which by itself is a major improvement. The commands are less cryptic than their S5 counterparts. For example, the super block can be displayed in a human-readable form by typing `p sb`. To perform the equivalent operation with the S5 `fsdb`, one would have to type `512B.p0o`, and then interpret the octal numbers. The DTFS `fsdb` command also has facilities to log its input and output to a file and to print a command summary.

8. Lessons Learned

Although designed for use on small computer systems, it turned out that many customers (including our own development staff) use DTFS to store large source file archives. Although DTFS doesn't perform as well as other file systems, performance is good enough for most users so that this difference goes unnoticed.

8.1 Compatibility

One interesting problem resulted from our original inode allocation policy. For simplicity, a file's inode number is the block number of the disk block containing that file's inode. (An exception is made for the file system's root inode, which is hard-coded as 2 for historical reasons.) Since inodes can be allocated anywhere on disk, we were reaching the point where inode numbers greater than 65535 were being allocated. This can break older (SVR3) binary applications that use system interfaces that represent inode numbers as short integers, so we changed the inode allocation policy to bias itself in favor of lower-numbered disk blocks first. This didn't solve the

problem entirely, but it made it less likely to occur.

Except for ‘‘.’’ and ‘‘..’’, which are implicitly created when a directory is made, DTFS does not allow the creation of hard links to directories. We consider this to be an outdated feature since the advent of `mkdir(2)`, `rmdir(2)`, and symbolic links. In addition, it can wreak havoc with the file system hierarchy, and complicate file system implementations. We have found no compatibility problems with this policy.

8.2 System V Kernel Limitations

As stated in the overview, we found it necessary to provide our own routines to access the system's buffer cache. The System V buffer cache interfaces that are used by file systems accept logical block numbers to identify disk blocks. The buffer cache routines then apply the formula

physical block number =
logical block number × *block size*

to convert the logical block number into a physical block number, which is then associated with the disk buffer. This makes the routines useless to file systems that have a variable block size and already represent disk blocks by their physical block numbers. Thus, we found it necessary to provide routines that did not apply this formula.

Another deficiency we found with the System V buffer cache was the lack of a way to invalidate an individual buffer in the cache. A routine existed to invalidate an entire device, but no such routine existed that would invalidate a single disk buffer. DTFS needs to invalidate a disk buffer, if it exists, whenever a disk block is freed. This is because the disk blocks represented by that buffer might later be reallocated to a different sized extent. This could result in aliasing conflicts if the old disk buffer were to remain in the cache. Curiously enough, merely setting the `B_INVAL` or `B_STALE` flag in the buffer header does not prevent the `fsflush` kernel daemon from writing the buffer to disk, if the buffer had been previously delayed-written. Thus, besides an aliasing problem, valid data could be overwritten by stale data, so we had to write a routine to allow us to invalidate a single buffer.

9. Future Work

Our future plans for DTFS include adding new compression algorithms for different classes of data, improving performance, incorporating our HTFS technology to improve system throughput and speed up `fsck` time, and adding features that our customers need. An example of the latter is the ability to recover files removed or truncated accidentally (commonly known as ‘‘undelete’’).

To date, DTFS has been ported to several versions of the Unix operating system, including SVR4, UnixWare, Solaris, and SCO Open Desktop. We intend to port it to other operating systems and hardware architectures.

10. Conclusions

With the disparity between CPU and I/O speeds, performing on-the-fly data compression and decompression transparently in the file system is a viable option. As the performance gap widens, DTFS performance will approach that of other file systems.

DTFS almost cuts the amount of disk space needed by conventional Unix file systems in half. In the introduction, we stated that DTFS was designed to be a commercial file system. The biggest sign of its commercial success is its planned inclusion into future versions of SCO's Open Desktop product.

The architecture of DTFS lends itself well to the inclusion of intermediate data processing between the file system independent layer and the device driver layer of the Unix operating system. In fact, it only took an engineer a few days to convert DTFS from compressing and decompressing data to encrypting and decrypting data.

Acknowledgements

Many people have contributed to the ideas behind the design of DTFS, including George Bittner, Bob Butera, Ron Gomes, Gordon Harris, Mike Scheer, and Tim Williams. We would like to thank the many reviewers of this paper for their helpful comments, especially Margo Seltzer, who also provided several useful references.

References

[BAC86] Bach, Maurice. J. *The Design of the UNIX Operating System*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[CAT91] Cate, Vincent and Thomas Gross. ‘‘Combining the Concepts of Compression and Caching for a Two-Level Filesystem,’’ *4th Interbational Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, CA. April, 1991, pp. 200–211.

[CHU92] Chutani, Sailesh, *et al*. ‘‘The Episode File System,’’ *Proceedings of the Winter 1992 USENIX Conference*. pp. 43–60.

[COM79] Comer, Douglas. ‘‘The Ubiquitous B-Tree,’’ *Computing Surveys*. **11**(2), June 1979, pp. 121–137.

[EPP90] Eppinger, Jeffrey L. *Virtual Memory*

Management for Transaction Processing Systems. PhD Thesis #CMU-CS-89-115, Carnegie Mellon University, Pittsburgh, PA, 1989.

[GRA81] Gray, Jim, et al. ''The Recovery Manager of the System R Database Manager,'' *Computing Surveys*. **13**(2), June 1981, pp. 223–242.

[GRA93] Gray, Jim and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[HAL94] Halfhill, Tom R. ''How Safe is Data Compression?,'' *Byte*. **19**(2), February 1994, pp. 56–74.

[HIT94] Hitz, Dave, *et al*. ''File System Design for an NFS File Server Appliance,'' *Proceedings of the Winter 1994 USENIX Conference*. pp. 235–246.

[KAR94] Karedla, Ramakrishna, *et al*. ''Caching Strategies to Improve Disk System Performance,'' *Computer*. **27**(3), March 1994, pp. 38–46.

[KLE86] Kleiman, Steven R. ''Vnodes: An Architecture for Multiple File System Types in Sun UNIX,'' *USENIX Association Summer Conference Proceedings, Atlanta 1986*. pp. 238–247.

[MCK84] McKusick, Marshall K., *et al*. ''A Fast File System for UNIX,'' *ACM Transactions on Computer Systems*. **2**(3), August 1984, pp. 181–197.

[OUS90] Ousterhout, John K. ''Why Aren't Operating Systems Getting Faster as Fast as Hardware?,'' *Proceedings of the Summer 1990 USENIX Conference*. pp. 247–256.

[SAN85] Sandberg, Russel, *et al*. ''Design and Implementation of the Sun Network Filesystem,'' *USENIX Association Summer Conference Proceedings, Portland 1985*. pp. 119–130.

[SEL92] Seltzer, Margo I. *File System Performance and Transaction Support*. PhD Thesis, University of California at Berkeley, 1992.

[TAU91] Taunton, Mark. ''Compressed Executables: an Exercise in Thinking Small,'' *Proceedings of the Summer 1991 USENIX Conference*. pp. 385–403.

[WEL84] Welch, Terry A. ''A Technique for High-Performance Data Compression,'' *Computer*. **17**(6), June 1984, pp. 8–19.

[ZIV77] Ziv, J. and A. Lempel. ''A Universal Algorithm for Sequential Data Compression,'' *IEEE Transaction on Information Theory*. **IT-23**(3), May 1977, pp. 337–343.

[ZIV78] Ziv, J. and A. Lempel. ''Compression of Individual Sequences via Variable-Rate Coding,'' *IEEE Transaction on Information Theory*. **IT-24**(5), September 1978, pp. 530–536.

Trademarks

Biographies

Morgan Clark is a Member of Technical Staff at Programmed Logic Corporation. Before joining PLC he worked at AT&T Bell Laboratories in Summit, NJ, and as a contractor at IBM. He has a B.A in Computer Science and Physics from Cornell University and an M.S. in Computer Science from the University of Wisconsin-Madison. His interests include operating systems, distributed systems, and performance analysis. He can be reached at morgan@prologic.com.

Stephen Rago is a Principal Member of Technical Staff at Programmed Logic Corporation. His interests include operating systems, networking, and performance analysis. He has a B.E. in Electrical Engineering and an M.S. in Computer Science, both from Stevens Institute of Technology. Stephen is the author of *UNIX System V Network Programming* (Reading, MA: Addison-Wesley, 1993). He can be reached at sar@prologic.com.