SRC Technical Note

2001-001

February 27, 2001

Cloudburst: A Compressing, Log-Structured Virtual Disk for Flash Memory

Gretta Bartels and Timothy Mann

COMPAQ
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
http://research.compaq.com/SRC/

Abstract

Cloudburst is a compressing, log-structured virtual disk implemented in flash memory under the Linux operating system. Existing Linux filesystems (preferably with certain modifications) can run on top of Cloudburst. Unfortunately, more extensive filesystem modifications would be needed to make the results fully satisfactory than we had realized at first, and we did not complete them. Nevertheless, the Cloudburst design has some interesting aspects that may be of use to future efforts in flash file systems.

Introduction

This note reports on Cloudburst, a device driver that enables persistent file storage on handheld computers with flash memory, such as the Itsy pocket computer [1]. Cloudburst is a compressing, log-structured virtual disk, implemented as a Linux block device driver. It was built as a summer intern project in the summer of 2000. Implementing Cloudburst as a virtual disk allowed us to avoid the complexities of building a complete new filesystem with integrated compression and flash support; instead, we can run a conventional Linux filesystem on top of the compressing virtual disk.

Unfortunately, however, a compressing virtual disk unavoidably differs from a real disk in one important way: the amount of available storage space varies greatly depending on the compressibility of the data that is currently stored. A real disk is a fixed-size array of blocks, with the property that any block can be overwritten at any time with any data, barring physical failures in the disk. Let us call the latter property *arbitrary writability.* A compressing virtual disk must also appear to be a fixed-size array of blocks, since that is the interface that conventional filesystems require. The array's virtual size must be larger than the physical size of the underlying media (say, at least twice as large if we hope for an average compression ratio of about 2:1), since otherwise the filesystem would have no way to make use of the space saved by compression. But doing this makes it impossible to achieve arbitrary writability; if the filesystem writes data that does not compress as well as was hoped for, then the physical medium will fill up before all the virtual blocks have been used.

Conventional Linux (and most other) filesystems rely on arbitrary writability at a deep level. First, conventional filesystems allocate blocks within the disk's address range to files and other structures, modify cached copies of the blocks in memory, and write them back to disk later. Failure of such a delayed writeback is a serious problem that may require manual intervention for recovery. With a compressing virtual disk, however, overwriting a block can fail if the new data does not compress as well as the old data. This condition is particularly likely to occur in our application; on a handheld device with limited flash space, the flash is likely to be nearly full much of the time. Second, conventional filesystems do not inform the disk of which blocks are currently in use and which are free space. A real disk does not need this information, but our log-structured, compressing virtual disk needs it for two reasons: the log cleaner is more efficient and runs less often if it can avoid copying data that is no longer in use, and discarding such data leaves more room to store valid data, effectively giving us much better compression.

We found it easy to modify Linux's ext2 filesystem to deal with the second problem. We simply added a *delete* primitive alongside the existing Linux block driver *read* and *write* primitives, and added a call to *delete* from ext2's block freeing routine. We had in fact already made this change to ext2 and the driver interface in order to solve a performance problem with the FTL (Flash Translation Layer) implementation on the Itsy [1].

However, we did not find a simple way to deal with the first problem. Here is a sketch of how it could be done. We add a *reserve* primitive to the driver interface. Calling this asks the driver to reserve enough space to overwrite a given block with new data, assuming worst-case compression. The call returns an error if there is not enough space left; otherwise, the space is reserved until the next *write* or *delete* of the same block is received. The filesystem would then call *reserve* before dirtying any block in the buffer cache (or otherwise committing to being able to write some block in the future). An additional refinement is needed to keep the system from getting stuck when it runs out of space. The only way to free up space is to delete a file, but a file cannot be deleted without writing to its directory. This can be solved by having the driver set aside a few blocks of emergency space to be used only by deletions. Say the emergency space is *e* blocks. An extra argument to *reserve* identifies whether or not the write is in service of a file deletion that will free up space. If it is not a deletion, then *reserve* returns an error if there are *e* blocks or fewer left; if it is, then *reserve* returns an error only there are no blocks left. The latter case should not occur if the filesystem is prompt about actually deleting blocks after reserving space for deletion.

We have not attempted to implement the scheme just described, but it is clearly not a trivial change to ext2 (or other Linux filesystems), and it would have to be implemented separately for each filesystem that is to run on top of Cloudburst. This makes the Cloudburst approach look considerably less attractive than it might appear at first. We suspect that this problem would be somewhat easier to solve (though it still might require nontrivial changes) when adding compression to a fully integrated log-structured flash filesystem; for example, in the current effort to add compression to JFFS [2].

Thus, overall, we do not consider the Cloudburst approach to have been a success. However, we are issuing this technical note because we believe that the detailed design has some interesting aspects that could be of use to future efforts in flash file systems. In the remainder of this note, we first briefly discuss related work, then proceed to the details of Cloudburst's implementation.

Related Work

Many different combinations of compression, log-structuring, flash-friendliness, and virtual disks have been tried before. To the best of our knowledge, the combination of all four features is unique to Cloudburst.

Log-structuring was introduced in Rosenblum's classic paper on the Sprite log-structured filesystem [3]. Burrows *et al.* observed that compression meshes well with log-structuring, and demonstrated a compressing version of Sprite LFS [4]. De Jonge *et al.* applied log-structuring to the implementation of a virtual disk [5]. The industry-standard Flash Translation Layer implements a virtual disk in flash, but without log-structuring or compression [6]. The Itsy implementation of FTL added the *delete* primitive to its interface [1]. Kawaguchi *et al.* reported on a log-structured virtual disk in flash, but without compression or the *delete* primitive [7]. Axis Communications has implemented JFFS, a log-structured filesystem in flash without compression [2]. At this writing, David Woodhouse is reportedly in the process of adding compression to JFFS.

Flash Memory

Flash memory has characteristics that make it different from most other forms of persistent storage. (Specific numbers in this note apply to the flash parts used in the Itsy, which come from the AMD Am29LV or Am29DL series, but they are typical of current CMOS NOR flash technology.) To its advantage, reading flash is nearly as fast as reading RAM. However, writing is relatively slow, and once written, flash cannot simply be overwritten with other data as RAM or disk can. Each bit in a flash memory has an initial *erase* state; say, 1. Each individual bit may be flipped to 0 at any time. But to flip bits back from 0 to 1, an entire *sector* of the flash must be erased simultaneously, changing all of its bits to 1. Sectors are large, often 256 KB or 512 KB, and it takes a long time to erase a sector, typically 0.7 seconds.

For this reason, rewriting small blocks of data in-place on flash memory is very inefficient. For example, suppose we are trying to store an array of 1024 512-byte blocks in a sector of flash. Suppose we need to update one of the blocks, and that some of the bits in the block will flip from 0 to 1 when we do the update. To do the update, we need to:

1. Read the entire 512 KB sector into memory from flash
2. Update the block in memory
3. Erase the sector
4. Rewrite the entire sector from memory to flash

Besides being time- and memory-consuming, this approach also leaves a sizable window of opportunity for data loss, should the system lose power. Furthermore, each sector of flash may only be erased about 100,000 times. After that, it no longer guarantees data integrity.

Cloudburst Rationale

We designed Cloudburst to be:

- log structured, because writing blocks in-place on flash is very inefficient

- compressing, because flash is generally quite small, and file systems expand to fill all available space
- a virtual disk, because implementing complete file systems is tricky and time-consuming. By working at a lower level, we hoped to support many different file systems.

The virtual disk abstraction is a layer between the file system and the storage medium. To the file system, the virtual (or logical) disk acts like a normal, magnetic disk: it reads and writes blocks of a fixed size. However, under the covers, the virtual disk can store the data on the physical storage medium in whatever way it chooses.

In the case of Cloudburst, we store the data in a log, writing blocks sequentially on the flash. Because there is no minimum write granularity on the flash, we can compress the blocks as we write them to the log without losing space due to fragmentation.

In the following sections we give details of the Cloudburst data structures and algorithms.

Recoverability

It is essential that power loss or a system crash while Cloudburst is in the middle of updating the flash should not lose previously written data or otherwise corrupt Cloudburst's data structures. Because typical flash chips (including those that the Itsy uses) can be organized in either 8-bit or 16-bit wide units, we do not assume that anything larger than a single byte can be written atomically. In general, our strategy for writing is to write the data first, then write a separate bit indicating that the write is done. At boot-time recovery, we also check for incomplete writes and resolve them by writing a bit that marks the data as invalid. We have assumed that an incomplete write cannot leave a bit in an indeterminate state where it sometimes reads as 0 and sometimes as 1.

Segments

As in any log-structured system, Cloudburst treats the entire physical storage medium as a large log, built by breaking the medium into *segments* of a fixed size and arranging the segments into a list. Changes to blocks do not overwrite the old, superseded block contents; instead, the new data is appended to the tail of the log. To avoid having the log grow until it fills the disk, the system uses a *cleaner*. Conceptually, the cleaner periodically chooses a segment that contains some superseded blocks, copies all the non-superseded blocks from that segment to the tail of the log, and then removes the cleaned segment from its place in the log and reinserts it at the end, as new free space. As a refinement, the log may have two tails, one for new data (much of which is likely to be superseded soon) and one for copied data (which may be more likely to be retained permanently), but that point is not important for the present discussion.

In Cloudburst, each segment is an integral number of erase units long (typically 1), so that a segment can be erased after it is cleaned. The data structure in a segment looks like this:

```
+---------+------------------------------------+----------+
| id info | compressed disk blocks             |->  free   |
+---------+------------------------------------+----------+
```

The `id` field at the front of the segment is reserved for future use. It could be used to hold information that will help the cleaner choose the best segment to clean, such as when the data in the segment was first written, when the data was last cleaned and recompressed, and the number of times the sector(s) in the segment have been erased (for wear-leveling purposes).

The compressed disk blocks are discussed further below. Since they each contain a length field, compressed disk blocks can be packed one right after the other in the segment. As more blocks are packed into the segment, they are added to the end of the data area, moving the boundary of the `free` area to the right.

The free portion of a segment is entirely filled with 1's. (For simplicity, we assume the type of flash where erased bits read as 1; the data structure could easily be generalized to work with flash that erases to 0's.)

Compressed Disk Blocks

Each compressed disk block is subdivided as shown below. Here "1b" means one bit, while "1B" means one 8-bit byte. There is a 5-byte header, followed by at most 512 bytes of data.

```
  +-----+-----+-----+------+------+-----+-------------------------------+
  | val | ver | run | comp | clen | VBN | cdata                         |
  +-----+-----+-----+------+------+-----+-------------------------------+
     2b    2b   1b     3b     1B     3B    max 512B
```

The `val` field contains two bits indicating the state of this block. If they are 11, then the block has not yet been completely written (in fact, it may not even have been started), and there are no blocks beyond this one in the segment. If they are 01, then the block has been completely written and is currently valid. If they are 00, then either the block was completely written and has been superseded, or the block was incompletely written at the time of a system crash and has been marked invalid by a subsequent recovery. (10 could have been used for the latter case, but there is no real need to distinguish such blocks from superseded blocks.)

The next field, `ver`, contains two bits indicating a virtual block version number. The version number is incremented on each write of a given virtual block, wrapping back to 0 when it overflows. The version number allows the boot time recovery code to disambiguate the case where two blocks have the same virtual block number (VBN). This case can occur if a new version of a block is written and the system then crashes before the old version is superseded. Two bits are sufficient because the two block versions never differ by more than 1. (Note: it might have worked as well to omit the `ver` field and have the boot time recovery code choose one of the two compressed blocks arbitrarily. If the block was being moved by the cleaner when the system crashed, the old and new versions must contain the same uncompressed data. If the block was being newly written when the system crashed, choosing the older version is acceptable, since it leaves the system in the same state it would have been in had the crash occurred a few microseconds sooner, before the new version was written to flash.)

The `run` field contains a bit indicating the beginning of a compression run. If the bit is set, this block is the first block in a compression run.

Next come three bits to indicate the compression scheme. This allows us to use eight different compression schemes (plus no compression, encoded with the `clen` field) in the running system. Thus, blocks can be compressed better as they live longer, or according to the type of data in them. The compression schemes are discussed further below.

The `clen` field gives the length of the block's `cdata` field. The meaning of the `clen` field is shown below.

`clen`	Interpretation
0	This block is uncompressed. We tried to compress it using the scheme listed in the `comp scheme` field and the block did not shrink. The total length of this block (including the 5-byte header) is 517.
1	This block is uncompressed because we have not yet tried to compress it. The segment cleaner should attempt to compress this block later. The total length of this block (including the 5-byte header) is 517.
2-255	This block is compressed and the total length of the compressed block is twice the value in the `clen` field, plus 5 bytes for the header. If the actual length of the compressed data was odd, we pad it with one byte of 0xff to make it even. (Note: this trick wastes one byte of space about 50% of the time, so we should probably have allocated one more bit to `clen` instead.) If length of the compressed data was 1 or 2, we pad it with two or three bytes of 0xff to make the length 4.

The `VBN` field of the headers provides 24 bits for the virtual block number. 24 bits is enough for $2^{24} * 2^9 = 8$ GB of virtual storage. Since compression ratios are not likely to be better than 4:1 on average, we could support up to 2 GB of physical flash with this system.

Finally, the remaining 2 to 512 bytes are the data. The data is compressed when possible, but if the compressor yields a block that is the same size or larger than the uncompressed data, we store uncompressed data.

Virtual to Physical Block Map

The map taking virtual to physical blocks resides in RAM and is regenerated at boot time. A physical to virtual map is not necessary, as each physical block contains its own VBN.

Because the block map must be able to hold many more mappings than will usually be needed, we organize it in a two-level hierarchy. In the current implementation, the top level contains 2^{10} pointers to pages in the next level. Each second-level page contains 2^{14} entries, for a total coverage of all 2^{24} virtual block numbers.

It would be fairly straightforward to modify these parameters, or even to add another mapping level. This might be needed to use RAM more efficiently or to accommodate different amounts and types of flash. The values given here are an example, chosen for the flash used in the Itsy model we were working with (32 MB total, 512 KB in each of 64 sectors).

The map entry layout is shown below. Each entry is two bytes.

```
  +----+--------+--------------+
  | vb | sector | run number   |
  +----+--------+--------------+
    1b     6b       9b
```

The first bit, `vb`, is a valid bit. If the bit is not set, the block is not in use. Blocks that are not in use read as all zeros. The sector field indicates which sector holds the block. The run number indicates which compression run holds the block. (Compression runs are explained further in the next section.) The size of the compression run field seems ample for our purposes and should even scale to larger flash memories.

Each second-level page in the block map takes 32 KB of memory. We expect that under normal usage conditions, only a few of these pages will actually be allocated. File systems with very good compression will have more block map pages allocated.

Block Compression

Choosing Among Compression Schemes

Several factors are important in evaluating compression schemes, including compression ratio, compression speed, temporary space needed during compression, decompression speed, and temporary space needed during decompression. For a given scheme, each factor will vary according to workload. The factors tend to trade off; getting a better compression ratio requires a slower compression algorithm, and perhaps a slower decompression algorithm as well.

In our application, speed is important for two reasons. First, slow compression and decompression algorithms would tend to make the system appear sluggish to the user. Second and more importantly, running the CPU on a battery-powered device consumes scarce energy from the battery. We can ameliorate these problems by using multiple compression algorithms: a fast one when initially writing data, and a slower one that provides a better compression ratio when cleaning. Also, whenever possible, we can postpone cleaning until the device is recharging in its cradle and ample energy is available.

Compression ratio is important for two reasons as well. First, our device's flash is limited in size, and we would like to store as much data in it as possible. Second, if we compress the data better, we will not have to erase and clean the flash as often; this can make the system faster, save energy, and make the flash last longer.

We also place one special requirement on the compression algorithm: it must be incremental, in the sense that it supports building the "compression runs" described in the next subsection. An incremental compressor accepts a sequence of 512-byte blocks and compresses each in turn, returning the result before needing to read the next block. The compressor is expected to learn from the early blocks in the sequence and do a better job of compressing the later ones if they have similar statistical properties. The decompressor can decompress any prefix of the sequence of compressed blocks, but of course cannot start decompressing in the middle of the sequence. The compressor can be reset after any block, starting a new sequence that can be decompressed from its beginning. The compressor implementation must support multiple independent sequences in progress at the same time, each with its own state.

Thus, the amount of temporary space required during compression is also important to us. A compression algorithm that is incremental, but must retain a large amount of state from one input block to the next, would consume a great deal of scarce RAM to hold the states of all the runs that are in progress at any given time (typically two or three).

For our initial implementation, we chose to support two compression schemes: null (no compression) and LZSS [8], [9]. By default, we use null compression when writing new data and LZSS when cleaning. As described above, our data structures provide a 3-bit field to identify the compression scheme, allowing for more schemes to be added in the future.

Compression Runs

512-byte blocks are much too small to get good compression of typical file system data. Therefore, we compress most blocks in runs with other neighboring blocks to improve the compression ratio. One consequence of this choice is that decompression is no longer a per-block operation; we have to decompress from the beginning of a run of blocks to read the desired one. However, compression remains a per-block operation; because our compressor is incremental (as described above), we can add new blocks to an existing compression run as they arrive.

There are several direct consequences for keeping blocks in compression runs:

1. If the blocks are at all related, later blocks will be better compressed.
2. To read a block, the entire run must be decompressed from the beginning.
3. All blocks in a run must be compressed with the same algorithm.
4. The segment cleaner should do its best to sort blocks into appropriate runs, perhaps even placing more frequently-read blocks toward the beginning of a run, to optimize performance.

How should we pack compression runs into segments? One idea would be to make each segment one large compression run. But this seems likely to make the runs too big; it could take a long time to extract blocks that fall late in the segment. Another idea would be to statically subdivide the segments into smaller pieces and put one run in each. However, this would lead to an undesirable amount of internal fragmentation; on average, about half a block (256 bytes) would tend to be wasted between the end of each run and the beginning of the next. Instead, we allow a run to begin at any byte offset within a segment, and we keep a separate table mapping run number to byte offset for each segment.

Next, how many blocks should go into each run? Runs should not be allowed to go on too long, because read time is related to run length. However, runs can't be too small, or compression ratios will be poor. The best length to use for a run might depend on the data being compressed. In our current implementation, we simply start a new run every *N* blocks, where *N* is a tunable parameter, typically set to 8, 16, or 32. Further study would likely yield better heuristics, especially for use in the cleaner.

The Run Offset Table

There may be at most 2^9 runs per segment, since nine bits are allocated to run number in the block map. This seems more than adequate, provided that segments are 512 KB or 1 MB long. Each run has a byte offset of (for now) 19 bits. If we store each offset in three bytes, the run offset table takes $2^6 * 2^9 * 3 = 96$ KB to store. If this seems too large, we could pack the entries more tightly and save 5 bits per entry, or we could dynamically allocate the parts of the table that we don't expect to use.

User-Specified Compression Settings

User processes with good knowledge about the type of data they write might be able to provide useful hints to the virtual disk. The user might want to specify whether data should be compressed now, later, or never, and which compression algorithm to use. We did not implement mechanisms to allow for this, but we consider in this section how it might be done.

Ideally, these hints would be supplied within the call to write, but since we do not have the freedom to change the file system interface, the best that seems possible is to communicate hints through another interface. The user supplies hints via an ioctl, specifying a particular flash minor device number. Once the ioctl is called, all writes to that minor device follow the user's suggestion if possible. If the user process wants to ensure a good mapping between writes and write policies, it should call sync before calling the ioctl. A drawback of this approach is that it is impossible to associate different hints with different files, so if multiple files are being written at once (perhaps by different applications) and should be compressed using different settings, they cannot all be accommodated.

The System in Operation

Boot time

At boot time, Cloudburst must rebuild all its in-memory data structures by scanning the flash.

The system scans each segment, building the block map and the run offset table. Within each segment, it reads the headers of the first block, makes an entry in the table, and uses `clen` to skip down to the next block. When it reaches a block with a valid field of 11, it has reached the end of the successfully written portion of this segment. There may be one partially written block; the next 517 bytes are scanned for values other than 0xff, and if any are found the length of the garbage block is written to the beginning of the block it is marked as superseded. Superseded blocks (with a valid field of 00) are ignored. Blocks with the same VBN are mediated by the version field, and the block with the lower version is marked as superseded.

The current write pointer is set to the end of any partially written segment, or to the beginning of an empty segment. If there are no empty segments, the cleaner must be run right away. If the pointer points to the middle of a compression run chunk, the blocks written to the chunk so far must be decompressed and recompressed so that the current compression state can be recovered.

Reads

On a read, the system looks in the block map, using the VBN as an index. The system then begins reading at the beginning of the compression run chunk containing the desired block, decompressing as it goes along. When it reaches the desired block, it decompresses the block and returns it.

When the compression scheme allows it, we should cache decompression state. Odds are good that the next desired block is also the next block in this chunk, so we can reuse the decompression state.

If the system issues a read to a VBN with an invalid block map entry, a block of all 0's is returned.

Writes

On a write, the system compresses the block using the accumulated compression state. If the compressed block will fit in the current chunk, write it. If not, start the next chunk by clearing the compression state and recompressing the block. Then write the block to the beginning of the next chunk.

To write a block, all of the data except the first header byte containing the valid bits must be written first, and then the byte with the valid bits must be written in a separate call after the other write completes. That way, even if the system shuts down in the middle of a write, no garbage data will live on after boot time.

After a write, we update the write pointer to point to the end of the written block. If the block previously existed in the system, the version should be the previous version plus 1. Otherwise, the version is zero. Next, the mapping in the block map is updated to the new location and version. Finally, the previous version of the block (if any) is invalidated by clearing the valid bits.

As an optimization, if a block of all 0's is written, no data need be written to flash. Instead, we can simply invalidate the existing version of the block (if any).

Deletes

In addition to the standard read and write, our virtual disk also has a delete function. In the absence of a delete function, the only way for the disk to discover that a block is now useless is for the file system to recycle the virtual block number. Therefore, without a delete function, the number of allocated physical blocks must monotonically increase over time, and space that ought to be free may not be reclaimed for a long time. To delete a block, we invalidate the block on the flash and in the block map.

Segment cleaning

The segment cleaner runs periodically. Ideally, the segment cleaner runs when the system is not otherwise busy, but if space becomes low then the cleaner runs immediately. There is some target amount of free space always kept clean.

To minimize the amount of erasing done, the segment cleaner normally chooses the segment with the most invalidated data. However, wear leveling is also a consideration; since each sector can be erased only 100,000 times, we should occasionally move long-lived data to another sector in order to make equal use of the available erase cycles on all sectors. We think that it will work well to simply choose, with some low probability, sometimes to clean the segment that has been erased the fewest times instead of the segment with the most invalidated data. However, we have not done the necessary mathematical analysis to justify this conclusion or to determine the best probability to use.

We have implemented a simple version of the segment cleaner that runs through the valid blocks in order and rewrites them to their new location. The usual read and write procedures are used. When all of the data in a segment has been invalidated, the segment's sector(s) are erased.

A more sophisticated cleaner (which we have not implemented) would be choosier about which blocks it moves. Rather than running down a segment copying each block in turn, it could choose blocks on the basis of their content, their virtual block numbers, or their observed access patterns. Such a cleaner would not produce clean segments as quickly, but it should tend to do a better job of compacting data. This type of cleaner should only be run during periods when there is plenty of energy available and the processor is otherwise idle; for example, when the device is recharging in a docking cradle. This cleaner could also spend more time compressing blocks, perhaps even selecting among several schemes to find the one that provides the best compression for a particular block. The simple cleaner is more appropriate during periods of activity when space suddenly runs short.

Both cleaners need to have separate write pointers, so that unexpected writes from the file system do not interleave with cleaner writes. If the simple cleaner is running, the blocks that have survived long enough to be moved are probably longer-lived than most new blocks and deserve to be packed together. If the sophisticated cleaner is running, then it will invest a significant number of cycles in determining the best block order.

Acknowledgements

We are grateful to Mike Burrows for explaining several compression algorithms to us and for providing many good suggestions on Cloudburst's cleaner algorithms. Mike also gave us helpful comments on the presentation of this note, as did Chandu Thekkath.

References

[1] Joel F. Bartlett, Lawrence S. Brakmo, Keith I. Farkas, William R. Hamburgen, Timothy Mann, Marc A. Viredaz, Carl A. Waldspurger, and Deborah A. Wallach. "The Itsy Pocket Computer". Compaq Western Research Laboratory, Research Report 2000/6, October 2000.

[2] JFFS (Journaling Flash File System) Home Page. Axis Communications. http://developer.axis.com/software/jffs/

[3] Mendel Rosenblum and John K. Ousterhout. "The design and implementation of a log-structured file system". ACM Transactions on Computer Systems, February 1992, volume 10, number 1, pages 26-52.

[4] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. "On-line Data Compression in a Log-structured File System". Proc. 5th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pages 2-9. Also available as Research Report 85, Systems Research Center, Digital Equipment Corporation, April 15, 1992.

[5] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. "The logical disk: A new approach to improving file systems". Proc. 14th Symposium on Operating Systems Principles, December 1989, pages 15-28.

[6] See the M-Systems Corporation web site, http://www.m-sys.com/.

[7] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. "A flash-memory based file system". USENIX 1995 Technical Conference.

[8] Mark Nelson and Jean-Loup Gailly. The Data Compression Book. M & T Books, 2nd Edition (1996), pages 215-253.

[9] J. A. Storer and T. G. Szymanski. "Data Compression Via Textual Substitution", Journal of the ACM, 1982, volume 29, number 4, pages 928-951.