The Linear Tape File System

David Pease*, Arnon Amir[†], Lucas Villa Real[‡], Brian Biskeborn[§], Michael Richmond[¶] and Atsushi Abe[‖]
* IBM Almaden Research
pease@almaden.ibm.com
[†] arnon@almaden.ibm.com
[‡] lucasvr@us.ibm.com
[§] bbiskebo@us.ibm.com
[¶] mar@almaden.ibm.com
[‖] IBM Yamato Lab
piste@jp.ibm.com

***Abstract*—While there are many financial and practical reasons to prefer tape storage over disk for various applications, the difficultly of using tape in a general way is a major inhibitor to its wider usage. We present a file system that takes advantage of a new generation of tape hardware to provide efficient access to tape using standard, familiar system tools and interfaces. The Linear Tape File System (LTFS) makes using tape as easy, flexible, portable, and intuitive as using other removable and sharable media, such as a USB drive.**

I. Motivation

In today's digital world, more and more companies keep all of their data, including their most valuable assets, in digital form. The broadcast and movie industries, for example, are changing to all-digital, file-based workflows as they go through a major transition. In what is often referred to as the Digital Media Transformation, traditional film and video tapes are being replaced with file-based workflows.

This transformation helps drive a burgeoning demand for storage capacity. More importantly, it creates new challenges for storage technologies. A detailed report by the Academy of Motion Pictures Art and Sciences [1] explains the industry's mission to keep and preserve digital movies for the next hundred years. It states that today, no media, hardware or software exists that can reasonably assure long-term accessibility to digital assets. When it comes to data preservation, the information technology community can learn from the more than a century of experience of the movie industry. Hundred year old film can still be projected and scanned today. In contrast, data put on floppy disks 20 years ago is already very hard or impossible to recover.

Within today's storage technology, data tape is still the preferred media for archive. Global tape archive capacity in 2008 was 5,210 PB, accounting for 51% of total archive storage, and is projected to grow 50% annually to 24,400 PB in 2012 [2]. The capacity of a single, industry-standard Linear Tape Open Generation 4 (LTO-4) cartridge is 800 GB (without compression), and will nearly double with the introduction of LTO Generation 5 this year. A single LTO-4 tape drive can read and write at a sustained rate of 120 megabyte per second (140 MB/sec with LTO-5), faster than a single hard drive. A tape library can host dozens or even hundreds of drives operating in parallel. A case-study comparison [3] found the cost ratio for a terabyte stored long-term on SATA disk versus LTO-4 tape to be about 23:1, and the energy cost ratio is as high as 290:1. Furthermore, the bit error rate of a SATA hard drive is at least an order of magnitude higher than of LTO-4 tape [4]. Tape longevity is typically rated by customers at thirty years [1] (though one would still need to keep an operational drive to read the tapes). Tape's economy, scalability, robustness, high density and low power consumption are unmatched.

Despite these many advantages, tape is rarely mentioned in the same breath with hard drives. Inherent functionality and usability differences often create the perception that tape is inferior to hard drives. Hard drives provide random access to files and blocks within milliseconds, while tapes might require tens of seconds seek time. More importantly, hard drives typically contain a file index, managed by a file system. Applications can access files on a hard drive using standard sets of APIs common to nearly all operating systems and programming languages. These file systems can be exported over local and wide area networks. Multiple files can be opened and modified simultaneously by multiple users. Hard drives can be made portable across platforms and operating systems.

In contrast, tapes can only be written in a linear, sequential fashion. Because of the way in which tape data is recorded, update-in-place is not possible. Hence tape is used as an append-only device, in which new content is added but old content cannot be modified or purged, and blocks cannot be reclaimed.

Typically, a tape contains no general-purpose file index (which would need to be updated often). Rather, an index is often kept in an external database, stored on a hard drive and managed by a storage system running on a host computer. Applications can only interface with this storage system through its special APIs, or access files which are staged by a hierarchical storage management (HSM) system. Therefore, data stored on an individual tape cannot be recovered without the presence of external databases and proprietary storage systems. Not only does this model diminish tape portability, it is often a major concern when it comes to recovery after a major catastrophe, such as when a storage system, its server, or its database go down. All of these components need to be rebuilt or recovered in order to regain access to the data on tapes.

Alternatively, files can be packaged in groups prior to writing them to tape, and a file index can be added and encapsulated within the data. This is commonly done using the ancient tape archive (tar) format. Access to individual files requires retrieval of the entire "tarball" (assuming one has a way of knowing which tar contains the files), and then accessing its internal index and unpacking the files. In discussions with customers who use tar on tapes, it is often considered cumbersome, inefficient, unreliable ("sometimes the tar header gets corrupted and we lose the entire set of files"), and incompatible (there is no universal tar standard).

Hence data tapes are much less portable than one would imagine, and are typically kept inside large automated tape libraries, managed by storage management systems and controlled by experienced system administrators. Even more striking is the comparison to digital video tapes, which have no external dependencies, require only a video tape recorder (VTR) to play, are portable, ubiquitous, and can be operated by anyone. Video tapes can be archived in a vault for many years and the content can be retrieved at any time.

Is there a way to make data tapes independent, portable, and as simple to use as removable hard drives or video tapes? This question led us to create the Linear Tape File System (LTFS). Simply put, our goal is to make data tape an equal member in the family of portable storage devices. Towards that goal we identified and addressed several key requirements:

- A self-describing tape cartridge, containing an index of its contents and sufficient information to allow complete recovery of the files from the tape without any additional external information.
- An index that includes a hierarchical directory structure, and files with attributes such as file name, date, and size. An extent list allowing for multiple file extents, and file modification through block relocation.
- An index that is easily accessible and updatable, independent of the data files stored on the tape.
- Access to tape data using standard utilities and programs through file system interfaces.
- Support for user- or application-specified extended attributes on files.
- Support for small, special-purpose data files close to the beginning of the tape. These files are expected to be domain-specific, and defined by the user's application. They are quickly available after loading and mounting the tape.
- Support for more than just individual tape drives, but also for library automation.

II. Related work

Early work on tape file system was mainly around the log-structured file system (LFS)[5], [6], utilizing linear block-based writing for optimal write performance. It was later extended to tertiary storage systems by work such as Kohl's HighLight [7]. These however did not aim at making tapes self-describing but only to provide an efficient file system access to tapes, first directly and later through tiers of storage. However in this work we particularly focus on a self-describing, stand-alone portable tape cartridge.

In [8] the authors describe a file system for a hybrid of tapes and RAID arrays. This work is more related to HSM systems, as it implements the file system in a global fashion, and does not allow for removal tapes from the system in order to use them in other environments.

Our work is most closely related to the work of Zhang et al. on the High Performance Tape File System (HPTFS) [9]. The authors describe a single-partition self-describing tape format with an index at the end of the data segment. They developed the file system using FUSE, and measured performance in several usage scenarios. In contrast, LTFS uses a dual-partition tape and leverages it in many ways, including rapid loading at mount time, the ability to keep multiple index versions and roll back to an earlier version, saving metadata files on the index partition, and more.

While both HPTFS and LTFS are file system implementations, only LTFS supports a hierarchical directory structure on tape, allowing preservation of the original directory structure and the file organization of users' data. LTFS also supports extended attributes for files and for directories. HPTFS works only at the single tape level; it does not support automated libraries. A major difference impacts the performance of seek(), a function commonly used for partial file retrieval or update. Because HPTFS uses block headers to associate blocks with files, its index does not contain a list of extents for each file. This approach requires the system to first find the beginning of a file, then read linearly while identifying and counting the relevant blocks and skipping the others, until the desired seek point is located. With LTFS, the block number of a seek command can be calculated from the index alone, before the tape is moved. The tape can then be moved directly to the beginning of that block, saving considerable time. An advantage of HPTFS is in the handling of streaming by interleaving files.

III. Tape Background

In order to understand some of the details of the LTFS implementation, it is helpful to understand something about modern linear tape. Because LTFS was written to work on linear tape such as LTO (Linear Tape Open))[10], our description will use the LTO format as its model. LTO is an open tape standard, and several companies produce and market compatible LTO drives and media. All members of the LTO Consortium must approve any changes to the LTO specification.

In general a new generation of LTO is introduced every 18-24 months, and typically brings increased recording density and an accompanying increase in tape capacity (and often in speed).

The current generation of LTO is Generation 4 (LTO-4). However, by the time this paper appears, Generation 5 (LTO-5) will be available. LTO-5 will have an uncompressed capacity of 1.5TB. As in the current generation, it will support both compression and encryption of data in the drive. Its maximum streaming data rate will be 140MB/second.

Unlike tapes of old, LTO tape is not written from the beginning to the end in a single pass. Rather, the tape is recorded in what is referred to as a serpentine fashion: a set of tracks is written from the beginning of the tape to the end, then the read/write head is moved some distance across the tape, and the tape is written back from the end to the beginning. Each end-to-end pass is referred to as a "wrap". So, writing 2 wraps returns the tape to its starting point. The current generation of linear tape records 80 wraps across the tape.

Due to requirements for backwards compatibility with earlier LTO generations, current (and future) generation of LTO write in a mode called shingling. Shingling means that the tape head erases a wider path than it records. This shingling effect (along with some other peculiarities of tape behavior) guarantees that tape data cannot be overwritten in place without affecting the data around it. Even if the tape drive could guarantee to rewrite a block in the same place and for the same length as an existing block (which it cannot), shingling would cause adjacent data to be erased.

As a result of the inability of tape to update blocks in place, tape is an append-only medium. Data cannot be written to other than the logical end of the tape (end of data) without destroying some data that follows. This means that any file system (or other scheme) that writes data to a tape must be designed so that updating data involves writing to the end of the tape, then changing any references to the old data block(s) to point to the updated one(s).

A major enhancement to the LTO specification in LTO Generation 5 is the addition of partitions. LTO-5 supports up to two partitions on tape; each partition can be written completely independently of the other. The smallest partition size allowed is 2 wraps, for a nominal capacity of 37.5GB. On a dual-partition tape, the largest partition can be a maximum of the tape size minus 4 wraps (2 wraps for the second, small partition, and two more that are automatically set aside as a guard area between the two partitions).

LTFS has been designed specifically to take advantage of the dual-partition tapes and drives available with LTO Generation 5. (Much of the original prototype work was done on LTO-4 drives with microcode modified to allow two partitions.)

IV. Linear Tape File System

A. What is LTFS?

In a nutshell, LTFS is a file system that makes a tape as easy to use as a removable hard drive. After an LTFS tape cartridge with data on it is loaded into a drive and mounted by the file system, a user can see the directories and files stored on the tape through the standard tools available on their system (for example, a graphical file explorer, application open menus, command-line tools, etc.). They can traverse the directories (or folders), read and modify files from within their applications, and open applications by double-clicking file icons. Files can be updated, and in general can be used as if they were on disk.

The performance of tape files in random-access applications can be significantly different from that of disk-resident files, and users should realize that tape and disk are fundamentally different. Nonetheless, LTFS makes it possible to use tape-resident files from standard applications and utilities without worrying about the fact that they are on tape, and without having to copy files from tape to disk. Perhaps equally importantly, it also creates self-describing tapes, and makes it possible to view the contents of tapes using the tools that users are familiar with.

The LTFS technology consists of two major components: the on-tape index format and the file system implementation. LTFS requires the multi-partition capability found in tape systems like LTO generation 5; however, it is not LTO-specific. (In fact, it has already been shown running on an non-LTO drive modified to provide dual-partition capability.)

In a basic LTFS implementation, one partition, typically a comparatively small one referred to as the index partition, is used to record the tape index; another partition, made up of the remainder of the tape and referred to as the the data partition, is used to store file data blocks.

B. On-tape XML schema

An important aspect of the design of LTFS was to make the format of the data on tape open, accessible, and easy to use. Our goal in this respect was to make it easy for others to write applications that read LTFS-formatted tape or to write tapes in a format that LTFS implementations can load and use. (Our group has already modified one existing IBM tape utility to write LTFS-compatible tapes.)

To that end, the format of the LTFS index has been designed with considerable attention to its ease of interpretation and processing. We have chosen to represent the index in a simple XML schema in which all values are represented in a human-readable format. As an example, we represent file timestamps in a `yyyy-mm-dd hh:mm:ss.mmmmmmmmm` format rather than in the more common seconds since the beginning of an epoch that file systems often use internally; while this takes a few more characters for each timestamp, it makes it easy for anyone to interpret the values in the XML schema. It also helps support another goal of LTFS, platform independence and interchangeability.

In order to be truly useful, an LTFS tape written on any operating system platform must be able to be read on any other OS platform. (This is the same type of interchangeability that we take for granted when using USB flash storage, due to the universal use of the FAT [11] format on such devices.) Our file system supports constructs that are common to all modern

```xml
<?xml version="1.0" encoding="UTF-8"?>
<index version="0.9">
    <creator>IBM LTFS 0.20 - Linux - ltfs</creator>
    <volumeuuid>9710d610-5598-442a-8129-48d87824584b</volumeuuid>
    <generationnumber>3</generationnumber>
    <directory>
        <name>LTFS Volume Name</name>
        <creationtime>2010-01-28 19:39:50.715656751 UTC</creationtime>
        <modifytime>2010-01-28 19:39:55.231540960 UTC</modifytime>
        <accesstime>2010-01-28 19:39:50.715656751 UTC</accesstime>
        <contents>
            <directory>
                <name>directory1</name>
                <contents>
                    <file>
                        <name>binary_file.bin</name>
                        <length>10485760</length>
                        <extentinfo>
                            <extent>
                                <partition>b</partition>
                                <startblock>8</startblock>
                                <byteoffset>0</byteoffset>
                                <bytecount>720000</bytecount>
                            </extent>
                            <extent>
                                <partition>b</partition>
                                <startblock>18</startblock>
                                <byteoffset>0</byteoffset>
                                <bytecount>9765760</bytecount>
                            </extent>
                        </extentinfo>
                        <extendedattributes>
                            <xattr>
                                <key>uservalue</key>
                                <value>fred</value>
                            </xattr>
                        </extendedattributes>
                    </file>
                    <file>
                        <name>read_only_file</name>
                        <length>0</length>
                        <readonly/>
                    </file>
                </contents>
            </directory>
        </contents>
    </directory>
</index>
```

Figure 1. Sample LTFS XML Schema

file system implementations, such as hierarchical directories, long file names, file timestamps, and extended attributes. It does not support attributes that tend to be platform-specific, such as UNIX-style user and group permissions or Windows ACLs.

Figure 1 shows an abbreviated example of the XML schema used in the LTFS directory. It tries to adhere to the idea that simpler is better. For example, a directory entry can consist of as little as a directory name, under which can be files and other directories. File entries can contain as little as a file name, timestamps, a file length, and the extent information that identifies the location of the blocks of the file on tape.

We expect that most LTFS files will initially be written sequentially and in a single pass. Thus, most files will initially have a single extent entry with a byte offset in the first block of zero, and a byte count equal to the file length. Files are initially written in fixed-size blocks of one megabyte in length, with the last block typically being a short block [1]. However, our extent list format supports short blocks anywhere in the file, as well as blocks in which not all bytes represent current file data.

[1]The one megabyte block size is the default, and can be changed at tape format time.

As an example of the use of this capability, imagine a single-extent file where one byte in the middle of a one-megabyte block is updated. One approach to this would scenario be to read the entire modified block, update the byte in question, append the updated block to the physical end of the tape, and split the single-extent file into three extents. However, this could involve considerable seeking across the tape to find and read the block being modified and then repositioning to the end of tape. Our approach would simply write the modified byte as a short block at the end of the tape, then split the file into three extents. If many updates are made to a file, our scheme allows new data to be written to the end of the tape without any intervening tape movement, and is thus far more efficient.

Another thing to note in the extent list format is the presence of a partition field. This field exists for two reasons. First, while LTO generation 5 supports only two partitions tapes in the future could support more, and since each partition starts with relative block number of zero, it is necessary to identify the partition to which an extent refers. Second, our implementation supports storing small data files in the index partition for caching at tape mount time (more information on this capability is presented below). Also note that this system allows different extents of a file to be in different partitions if that becomes necessary or useful.

A final comment on the XML schema concerns the presence of an index generation number in the schema. When using LTFS in a mode where tape index information can be cached on a hard drive (such as in Library Mode, described below), it is possible for the cached and on-tape versions of the index to get out of synch. A simple timestamp is not suitable for determining the latest version of the index, since the tape might have been updated on a system whose clock is wildly different from the system on which the tape was previously written. Instead, we use a monotonically increasing index version to resolve questions of index currency.

C. Extended attributes

An important feature of our index schema is support of user metadata for files, in the form of extended attributes. These extended attributes are key/value pairs that can be set or queried using standard POSIX xattr [12] system calls (as well as Windows system calls).

As an example, extended attributes could be used in cartridges that contain multimedia streams to indicate the offset in a file where certain events can be found, or to provide indices so that the user can seek back and forth in time on a video stream. They could also be used to hold frequently accessed information related to a file, which could then be accessed without seeking to the file data.

In order to provide efficient access to metadata, LTFS stores extended attributes in the XML schema as part of the file description. Once a tape cartridge is loaded and parsed by LTFS, such information becomes instantly available for reading with no need for additional tape I/O operations.

D. Architecture and Implementation

Our first LTFS file systems have been implemented on Linux and Mac OS X. On both of these platforms, the file systems use the FUSE [13] framework, and implement the actual file system logic in user space. The use of the FUSE infrastructure allowed us to build and debug the file system logic much more quickly than would have been possible with kernel-level development. Figure 2 illustrates the relationships between user applications, the VFS layer, the FUSE module, LTFS, and the tape device driver.

Our two initial implementations share most of their code. The major difference between the Linux and Mac OS X implementations is at the device driver level. For Linux, an open source tape driver, LinTape [14], that supports LTO drives exists. Unfortunately, no such driver yet exists for OS X, and we were forced to use raw SCSI CDBs [15] to interface with the drives on that platform.

There is no reason we could not port our implementations to native kernel-level file systems. However, unless there are significant performance gains to be realized from such a port, there is not much incentive to do so. Unfortunately, there are many commercial UNIX variants for which FUSE implementations do not exist; for those systems a kernel-level file system would appear to be the only choice (short of porting FUSE). There is also currently no complete, working implementation of FUSE for Windows (though one appears to be under serious development).

Until we finish development of a Windows implementation, our current approach for Windows (as well as for other UNIX variantss) is to export the LTFS file system from Linux or Mac OS through either CIFS [16] (using Samba [17]) or NFS [18]. However, this approach is unlikely to be acceptable in the long term, especially in the Windows world where Linux and Mac systems and skills may be rare. We are have started work on a native Windows LTFS implementation, but in our experience Windows is the most difficult of file system development environments.

E. Single-drive mode

The current implementations of LTFS have two operating modes, which we call “single-drive mode” and “library mode”. Single-drive mode is the simplest of LTFS modes. It is designed for an environment without tape automation (libraries), where one (or perhaps two) tape drives are attached to a workstation or server. In this mode, using LTFS is similar to using a removable hard drive or flash drive. When a tape is mounted the contents of the tape are accessible through the file system, and when the tape is unmounted no memory of the tape contents is kept on the host machine.

This mode is intended for environments where tape automation is unnecessary or too costly. It is especially well-suited to scenarios where tape is used as an exchange medium, such as small audio/video processing companies.

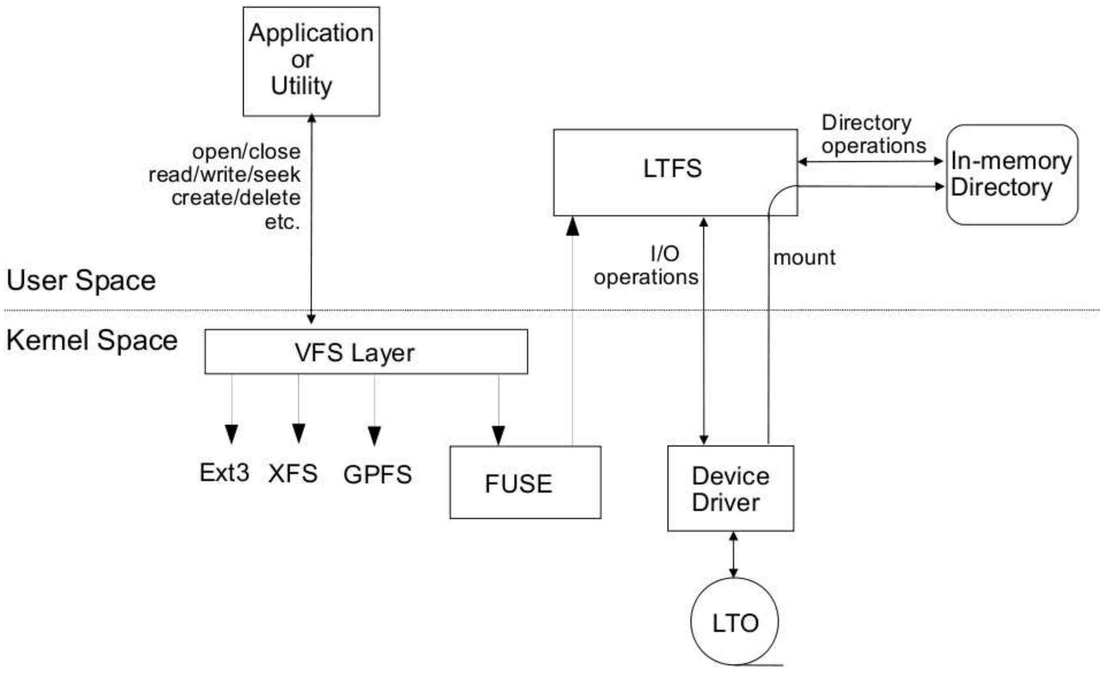

Figure 2. Relationship between LTFS, the operating system and tape hardware

F. Small Library mode

LTFS has also been designed to support tape libraries. A tape library is a storage system that contains one or more tape drives, a number of slots for holding tape cartridges, and a robot (automation) that moves tapes to and from the drives under software control. The automation identifies cartridges by scanning a bar code that is visible on the outside of the cartridge.

When LTFS is running in Library Mode, it caches tape index information to hard disk as it accesses tapes [2]. This allows the LTFS system to expose the contents of all the tapes in the library without having to mount any tapes. When a library is first mounted, LTFS creates a directory for each known tape volume under the library mount point, and under each of those directories the file and directory information for the associated tape is visible. A user can search or browse the tape contents without the tape being accessed. When a user or application opens a file on one of the tapes, LTFS instructs the library automation to mount the tape in a free drive.

The index generation number mentioned earlier allows resolution of conflicts should the disk copy of the index get out of synch with the version on tape.

G. Data files in Index Partition

As mentioned earlier, LTFS supports the ability to store data files in the index partition. This capability is intended to support environments where small data files describe much larger files in the data partition, and those small files may be referenced frequently. (An example of such a scenario is the use of an MXF [19] “moe” file to provide an index into a very large MPEG video file.) When LTFS mounts a tape it reads the index partition, and caches any data files it finds there either in memory or on local disk for fast access.

LTFS provides the ability to specify the criteria for inclusion of data files in the index partition at tape format time. The criteria that can be used are the maximum file size and file name pattern(s). The maximum size can also be set to zero, which will disable any writing of data files to the index partition.

Files written to the index partition are expected to be small and relatively few. In LTO-5 the size of the index partition is approximately 37.5 gigabytes out of a tape total of 1.5 terabytes (both numbers are uncompressed). If the index partition appears to be in danger of filling up, the file system can choose to write even data files that meet the index partition criteria to the data partition.

While our LTFS implementation supports data in the index partition, that does not mean that other implementations of LTFS-compatible formats would have to. When writing a tape, a simpler implementation could always write all data to the data partition. When reading, an implementation simply uses the partition number in the extent list, without specifically worrying about whether that extent is in the index or data partition.

H. Multiple index generations

Since tape is an append-only medium, even data that has been logically deleted or updated is still physically on the tape. This fact makes it very easy to implement a form of versioning

[2]Synchronization of the XML index to disk can happen at various times, including at unmount time (if the tape contents have changed), and when requested by the user or application through a file system sync request.

file system. Blocks written to the tape are never overwritten by LTFS, with the sole exception of XML index. Older copies of the index are written to the data partition, interleaved with data, and a chain of back pointers allows LTFS to walk back through the array of older indexes.

When a tape is mounted, the file system reads the most recent index information from the index partition. (A redundant copy is also kept at the end of the data partition, in case of an error.)

If the user specifies at mount time that they want to revert to an earlier version of the tape contents, LTFS traverses back through the chain of index files to find the one that most closely matches the user's request, and makes that version of the tape data available for reading. (A separate tape reclamation utility can permanently revert a tape to an earlier version, and allow it to subsequently be rewritten from that point.)

V. Results

A. NAB Prototype Demonstration

We initially implemented LTFS on IBM's LTO-4 drives using modified firmware designed to support dual partition operation. (As mentioned earlier, this dual-partition capability will be available natively as part of LTO Generation 5.) The prototype system used FUSE to provide file system access to a mounted tape; it supported both single drive and library modes of operation. We demonstrated the LTFS prototype at the National Association of Broadcasters (NAB) conference in April of 2009. The demonstration showed the basic functionality of the prototype, including reading and writing files, browsing the directory structure on the tape, exporting the file system to Windows machines via Samba, and performing partial file recall on an MXF video file.

We have subsequently reimplemented the LTFS file systems in a much more robust fashion using LTO-5 hardware. (We have chosen to retain the FUSE-based architecture.) We will make this code publicly available during the 2010 NAB show in April of this year.

B. Performance

Tape drives perform best for streaming workloads: an LTO-5 drive can require 90-100 seconds to seek in the worst case, but it can perform sequential reads and writes at 140 MB/s. Because of the serpentine nature of the LTO format, throughput is constant over the entire medium except for the few seconds required to turn around at the end of each wrap. Once at least one wrap is filled, average seek time is nearly constant as more data are added to the tape; because each wrap holds only 18.75 GB, this limit is reached quickly.

The performance characteristics of a tape are predictable, but they do not necessarily reflect those of a file system running on the tape. LTFS always appends to the tape when writing, so in the absence of file system overhead, it should achieve the full write throughput of the tape drive for any series of file writes using a large enough block size. Similarly, LTFS should achieve the full read throughput of the drive provided that the requested blocks are sequential on the tape. Non-sequential reads will be completely dominated by seek time.

Our test system contains two quad-core Intel Xeon processors (Core 2 architecture, running at 2.66 GHz) and 24 GiB of RAM; it is connected to a full-height LTO-5 drive via a 4 gigabit-per-second Fibre Channel network.

File system write performance was measured by filling 10 wraps (175 GiB) of an LTO-5 tape with zeros. This amount of data provides even coverage over the length of the medium, so the results obtained are representative of the entire medium. The tape was formatted with a 1 MiB block size. To determine large file performance, 1 GiB files were used, for a total of 175 files. These files were written and read using 1 MiB write requests. To determine performance on smaller files, 1 MiB files were used for the second set of tests, for a total of 179200 files spread across 175 directories. These files were written and read using 128 KiB write requests.

For each file size, sequential write performance was tested by measuring the total time required to write all files to the tape. Sequential read performance was tested by measuring the total time required to read all files from the tape. Seek time (which dominates random read performance) was measured by choosing 100 files at random, then reading the first byte of each file in the chosen order. The reported seek times are averages of all 100 seek requests. The file system was remounted between write and read tests to eliminate any possible caching effects.

Table I summarizes the results of these benchmarks. We have measured raw throughput on the test system at 133.0 MiB/s. LTFS consistently achieves the full throughput of the tape drive except when writing small files. The lower small file write throughput likely occurs because LTFS flushes data to the tape drive when closing a file, but we have not confirmed this explanation.

Average seek time is roughly 37 seconds and is independent of file size. Since LTFS can only append blocks to the tape, it turns random writes into sequential ones. Therefore, only read workloads are directly subject to long seek times. Random I/O may be acceptable on small files, especially if the entire file can fit in the file system cache (either in LTFS or in the kernel). But for general random workloads, LTFS is limited by the inherent access time of the underlying medium.

VI. Discussion and Future Work

This work defines a new class of portable storage media which is based on open standards, has the economy, robustness, high density and low power of tape with many of the functionality and usability of hard drive. A new open format for self-describing tape is proposed, utilizing dual-partition tape which is part of the LTO-5 industry standard. A large data partition is used to store the files' content and a much smaller index partition is used to keep the file index and metadata. The index can be updated and metadata files can be added to the index partition independent of the main content that is stored on the data partition. The use of a dual partition tape increases

File size	Write (MiB/s)	Read (MiB/s)	Seek time (s)
1 GiB	132.9	132.2	37.2
1 MiB	92.8	133.0	37.6

Table I
WRITE, READ AND SEEK PERFORMANCE OF LTFS.

efficiency and improves resilience against unexpected power outages and other system crashes. A new LTFS Linear Tape File System was implemented on both Linux and OS X using FUSE.

Our LTFS implementation on both Linux and OS X uses FUSE for code development in user space. It supports both a single-cartridge mode and a library mode - in which many tapes show up as folders under a single mount point. Hence tape ingest into libraries is made easy by reading just the content of the index partition into LTFS and populating and content management system with the metadata found on the index partition. The tape is made portable and compatible across platforms.

A first system prototype was implemented and successfully demonstrated at NAB 2009. We have since built a version suitable for general use, and will make it available as open source in early 2010. We have completed a first version of the LTFS Format Specification, and will make that openly available at the same time.

More work is required to support multiple concurrent requests, scale up, and closer integration with other file systems.

This work is part of the DuraBytes project, whose goal is to make tape better and "smarter". In other parts of the project we have developed a prototype hybrid tape cartridge embedding 4 GB of flash memory. The flash is accessible while the cartridge is either loaded into the drive or is connected through USB to any computer; it allows browsing the directory tree and file attributes for the tape's content. We also address domain-specific tape requirements such as tools for MXF wrapping and metadata extraction and support efficient handling of file-based media workflows.

REFERENCES

[1] "The Digital Dilemma, Strategical issues in archiving and accessing digital motion picture material," 2007, the Science and Technology council of the academy of motion picture arts and sciences.

[2] J. McKnight, M. Turner, B. Babineau, and J. Gahm, "2007 File Archiving Survey: End-User Requirements & Priorities," Dec 2007, enterprise Strategy Group.

[3] D. Reine and M. Kahn, "Disk and Tape Square Off Again – Tape Remains King of the Hill with LTO-4," Feb 2008, clipper Notes.

[4] H. Newman, "The Tape Advantage: Benefits of Tape over Disk in Storage Applications," Apr 2008, white paper, Instrumental.

[5] H. Robinson, "A mass storage subsystem using ANSI X3B6 ID-1 recorders," K. D. Friedman and B. T. O'Lear, Eds. Monterey, CA: IEEE 10th Symposium on Mass Storage Systems, May 1990, pp. 43–45, iEEE catalog number 90CH2844-9, 7–10.

[6] C. Staelin and J. Kohl, "The Coconut file system: Utilizing Tape-based Robotic Storage," in *USENIX File Systems Workshop Proceedings*. USENIX, May 1992, pp. 141–142.

[7] J. T. Kohl, "Highlight: Using a Log-structured File System for Tertiary Storage Management," Master's thesis, UC Berkeley, Berkeley, CA 94720, 1993.

[8] D. Feng, L. Zeng, F. Wang, and P. Xia, "TLFS: High Performance Tape Library File System for Data Backup and Archive," in *Proceedings of 7th International Meeting on High Performance Computing for Computational Science*. Rio de Janeiro, Brazil: Springer, June 2006.

[9] X. Zhang, D. Du, J. Hughes, and R. Kavuri, "HPTFS: A High Performance Tape File System," in *Proceedings of 14th NASA Goddard/23rd IEEE conference on Mass Storage System and Technologies*, College Park, MD, May 2006.

[10] Aberdeen Group, "LTO Tape Has It All," Dec 2001. [Online]. Available: http://www.lto-technology.com/pd f/Aberdeeni_LTO_has_it_all.pdf

[11] Microsoft, "Microsoft Extensible Firmware Initiative FAT32 File System Specification," Dec 2000. [Online]. Available: http://www.microsoft.com/whdc/sy stem/platform/firmware/fatgen.ms px

[12] I. IEEE, *IEEE Standard for Information Technology: Portable Operating Sytem Interface (POSIX). Part 1: System Interface*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Standards Association, 2001.

[13] M. Szeredi, "File system in user space (FUSE)," 2009. [Online]. Available: http://fuse.sourgeforge.net

[14] IBM, "LinTape, Linux Tape Driver," Sep 2009. [Online]. Available: http://www-01.ibm.com/support/do cview.wss?rs=577&uid=ssg1S400078 7

[15] G. Field and P. M. Ridge, *The Book of SCSI*, 2nd ed. 555 De Haro St., Suite 250. San Francisco, CA 94107: No Starch Press, Jun 2000.

[16] C. R. Hertel, *Implementing CIFS: The Common Internet File System*. Upper Saddle River, NJ, USA: Prentice Hall, Aug 2003.

[17] T. S. Team, "The samba project," Dec 2009. [Online]. Available: http://samba.org/samba

[18] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3 - Design and Implementation," in *In Proceedings of the Summer USENIX Conference*, 1994, pp. 137–152.

[19] B. Devlin, J. Wilkinson, M. Beard, and P. Tudor, *The MXF Book: An Introduction to the Material eXchange Format*. 11830 Westline Industrial Drive, St. Louis, MO 63146, USA: Focal Press, Apr 2006.