An I/O Scheduler for Dual-Partitioned Tapes

Lucas C. Villa Real
IBM Research
lucasvr@br.ibm.com

Michael Richmond
Infoblox
mrichmon@acm.org

Brian Biskeborn
Google Inc.
bbiskebo@gmail.com

David Pease
IBM Research
pease@us.ibm.com

***Abstract*—For a long time, tapes have had a single logical partition that was efficiently operated by dedicated software in batch mode. Nowadays, after the introduction of the LTO 5 standard, tapes support logical partitioning and, with the arrival of the Linear Tape File System (LTFS), their contents are exposed through the file system interface as regular files and directories. As a consequence, tape medium can now be accessed by concurrent processes that may create files in parallel. This creates two potential problems: interleaved data blocks and expensive partition switches. This paper presents the design and implementation of Unified, an I/O scheduler for LTFS that addresses these problems. We observe the effectiveness of delayed writes on decreased file fragmentation and the reduction of partition switches with the use of buffering and of redundant file copies. We also find out that software-based read prefetching, commonly used to manage disk devices, does not improve read times on tapes, but rather introduces potential overhead. With the techniques described in this paper, the Unified scheduler allows tape operations to be performed close to the raw hardware speed.**

1. Introduction

Magnetic tapes have always played an important role in long term archiving and backup solutions because they are cost-effective when compared to other storage media [11]. Tapes are also used in combination with high speed devices in hierarchical storage management systems: data files that are frequently accessed are kept in memory, or on fast hard disks, and automatically migrated to and from data tape according to data access policies [4]. Cloud-based storage providers [13] and the Large Hadron Collider [1] are two examples of users of such technology.

Common tape management and backup software [15] are designed so that sets of files can be transferred to tapes in a single run. The advantage of that class of software is that each file can be transferred sequentially to tape, resulting in optimal block allocation for streaming I/O. Often a single process manages access to the tape hardware. That fact, combined with the opportunities to arrange files sequentially on medium, has caused past research on tape I/O to focus on scheduling read operations alone so that reads could be served with a minimum number of tape seeks.

With the introduction of the POSIX [7] compliant Linear Tape File System (LTFS) [12], the contents of the tape are no longer controlled by dedicated applications. Rather, they are exposed to the user through the operating system's file system interface. By doing so, users can use tapes just like hard disk or external mass storage devices, performing drag and drop operations and using existing desktop applications to concurrently read from / write to tape in many unpredictable ways. Consequently, LTFS needs to employ scheduling strategies for situations that previous generations of tape software did not have to deal with.

Moreover, modern enterprise tape systems, beginning with the 5^{th} generation of the Linear Tape-Open (LTO) specification [6], have the ability to *segment* tapes into two (or more) separate partitions where applications can read from and write to. A consequence of supporting writes to different partitions is that the performance of applications can be degraded due to the expensive tape head movements and longitudinal seeks to locate the data blocks on the different partitions. LTFS is the first file system to support logical partitioning on modern tapes, which means that this is one of the first studies on optimizing tape partition switches.

In this paper we present the design of Unified, an I/O scheduler for LTFS, which is, to the best of our knowledge, the first one designed to optimize performance for on-line access to dual-partitioned tapes. Since tape systems are commonly overlooked in today's world of memory-based storage devices, we begin Section 2 by providing background on how modern tapes are designed, with emphasis on how logical partitioning works. Following that, we describe our aims and objectives with Unified as part of Section 3. Next, Section 4 presents a study of prior work that we took into consideration when looking into improving the performance of LTFS. The architecture of the I/O scheduler is given in Section 5, including annotations about this paper's main contributions. Section 6 then presents an analysis of the performance gains obtained by asynchronous writes and the effect of delayed operations, buffering, and redundant file copies on the spatial arrangement of files and on savings to the number of partition switches. Our results are compared against the First-Come, First-Served scheduler available on LTFS. We also investigate a prefetching technique commonly adopted by disk I/O schedulers and discuss about its drawbacks when applied to software-based tape I/O schedulers. Section 7 draws our conclusions and makes final remarks.

2. Tape design

A partition in an LTO tape is defined as a sequence of *wraps* that run through the entire length of the data tape. Each wrap has a direction: it can run from the begin of tape to the end of tape or the other way round. A sequential write operation is performed in a serpentine pattern that

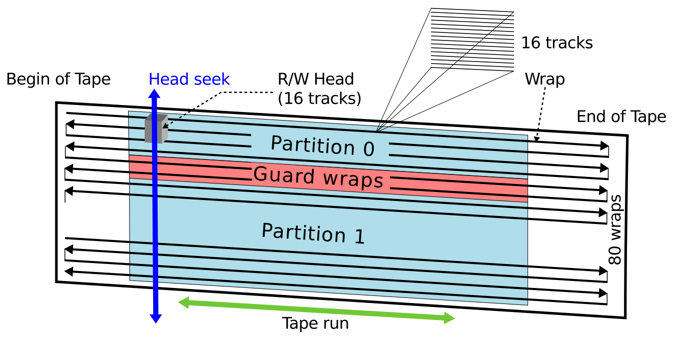


Fig. 1: LTO-5 track recording in a dual-partitioned tape.

follows the directions of the wraps. The first wrap is written while running the tape forward; the second wrap is written while the tape is run backwards; the third wrap is written while the tape runs forward, and so on.

Data writes to LTO tapes are done through a technology called Shingled Magnetic Recording (SMR) [8]. The idea behind SMR is to partially overlap previous wraps on new writes so that the written “bits” occupy the minimum wrap (or track) size possible, thus leading to increased storage density. The way in which tape drives implement SMR is by applying a high power current to the write head that not only affects the currently positioned wrap but also the adjacent wraps. That is, writing to a wrap N will partially overlap the previous wrap $N-1$ and also affect any existing data at the next wrap $N+1$.

A. Tape partitions

Due to the use of Shingled Magnetic Recording, a write operation on a single-partitioned tape causes all data following the written logical block address to be deleted – after all, the contents of the next wrap will be partially overwritten by that operation.

With LTO 5 onwards, it is possible to segment tapes into different logical partitions that support independent write operations without altering the data stored on any other partition. That is possible through the use of buffer or *guard* wraps that serve as a physical boundary to separate partitions. Then, writes to the last wrap of a partition P only interferes with the guard wrap that separates it from partition $P+1$, thus guaranteeing data integrity of $P+1$'s first wrap. Likewise, writes to the first wrap of $P+1$ do not affect P's last wrap.

Logical partitioning is restricted to the section on tape reserved for *user data*. The remaining sections near the beginning and end of the tape are reserved by the drive for housekeeping and calibration and are not subject to the partition split. This organization is shown in Figure 1, which depicts a dual-partitioned tape.

B. Partitions in LTFS

In the LTFS tape format, the physical media is divided into two logical partitions in accordance with the LTO specification.

The first partition is primarily used to store the LTFS Indexes that represent the filesystem metadata for the volume. In that partition, small files can be optionally stored according to data placement rules determined at format or mount time – users can, for example, configure a tape so that metadata files smaller than 2MiB and with names ending on ‘.XML’ are always copied to the Index Partition. The advantage of having a few selected small files on that partition is that, in general, they will be close to the beginning of tape and to the LTFS Index, making it possible to quickly retrieve their contents.

The second partition is significantly larger than the former and is used as the primary location for file storage on the volume.

C. Data extents

A data extent is a set of one or more sequential logical blocks used to store file data. In LTFS, all blocks within a data extent have a fixed size (determined at format time) except for the last block, which may be smaller. The data arrangement of a file on an LTFS tape is described by extent lists containing information such as the partition where each extent resides, its starting tape block number, its size in bytes, and the corresponding logical file offset [19].

The fewer extents a file has, the smaller its representation in the LTFS Index. Additionally, a file with fewer extents will incur fewer tape seeks during access, resulting in a more efficient access pattern to read the file data back from tape.

3. Aims and objectives

Due to their longevity and cost-effectiveness, tapes are often the industry's choice for long-term archiving. A nearly 20-year study on evolutionary archival trends of scientific and historical long-term data indicates that workloads have shifted in the past few years from read-dominated operations towards write-heavy ones [3]. That analysis suggests that current archival storage systems could be more optimized for write activities, which is the main topic of this paper as well as Unified scheduler's major aim.

Concerning the dual-partitioning of tapes, the motivation for designing Unified is to lower the large delays caused by partition switches [20] when writing to tape. Although the LTFS specification does not require implementations to support writing user data to both partitions, all major LTFS software distributions do so. Through data placement rules [19] users indicate which files can be stored on the first partition (named Index Partition, or IP for short) and their maximum allowed size. With a specialized I/O scheduler, we expect to minimize the number of partition switches demanded by that feature while preventing the Index Partition from becoming too full.

Next, being supported on three contemporary multi-user operating systems (Linux, Windows, and Mac OS X), it is important for an LTFS I/O scheduler to treat concurrent access to the tape in a fair manner. We focus our design so that parallel write requests can be efficiently orchestrated, with the objective of reducing data block fragmentation.

Moreover, the file system interface utilized by LTFS on Linux may translate single large write requests into multiple synchronous writes of smaller sizes to the LTFS user space process. By doing so, the Linux kernel reduces the number of dirty memory pages that it needs to hold before the user space file system writes them down [14]. Hence, to achieve maximum throughput, the architecture of the I/O scheduler needs to include mechanisms that handle writes asynchronously. The same is not necessary for reads, as discussed in Section 5.5.

Last, data blocks in LTFS are always written to the end of the selected partition. This append-only constraint is imposed by the characteristics of tape media and simplifies the design of Unified: data blocks from deleted files cannot be recycled by the file system on subsequent write operations. Therefore, all optimizations related to file writes are focused on data appends at the tape media.

4. Related work

This section presents related work on tape I/O scheduling. We note that even though tapes have been around for a very long time, there are few studies in that area. One of the possible reasons is that tape management software frequently operates as an interface for transferring fixed lists of files from disk to the tape and back. By doing so, the software can operate the medium in batch mode and significantly reduce the complexity and cost of its implementation. Nevertheless, some interesting techniques found in the literature can be still applied to dynamic I/O scheduling of requests to tapes.

A. Tape reads

Several studies in the literature aim to improve reads on serpentine tapes. Some authors have proposed mathematical models that estimate the seek times for random accesses to the tape medium [5]. These estimates are then used to create algorithms that consider the tape layout when scheduling reads [17].

The seek time between two positions on a tape is determined by four factors: (a) the longitudinal distance between the source and destination positions, (b) the need for a track change, (c) the need to change winding direction, and (d) whether locating the closest key point[1] on tape influences the seek. Sandstå and Midtstraum [16] propose a generic model that can estimate the cost of seeks based on these factors for a given sequence of block addresses. The inconvenience of that method is that low level information about the tape layout needs to be input by the user, otherwise cost estimates can be wrong. Even if that process is automated through software, the time spent doing the characterization of the tape can be very long, which makes it prohibitive to integrate this model in a file system meant for production environments.

[1]The tape drive maintains a set of evenly-spaced reference points that are sought before the requested data block address can be located.

In a study to optimize the throughput of data restore from tapes, Schaeffer and Casanova [18] implemented TReqS, a scheduling system placed in front of a hierarchical storage mananagement (HSM) that queues read requests from files. Inside that queue, requests are sorted according to their physical position on tape (given by the underlying HSM). If a read request is received while HSM is processing a queue, then that request's block address is compared against the tape's current position. If the tape position is greater than the requested block address, then that request is put on a new queue that is processed later. That condition guarantees that the tape always moves sequentially when serving requests from a given queue and that the drive only rewinds the tape when the new queue is activated. We adopted the same strategy of serving reads based on the associated block addresses in the Unified scheduler.

B. Tape writes

To improve the performance of writing small files to tape, Murray et al. [10] proposed modifications to the CASTOR storage manager system that runs at CERN. That system stores a file on tape by surrounding the actual file data with three special *tape marks*[2]. Differently from a regular data byte written to tape, a tape mark is written using a particular SCSI command that also causes the tape drive to flush all outstanding buffers to the medium [2]. In the system evaluated by Murray's team, the cost of writing a tape mark was of 1.7 seconds, on average — meaning that about 5 seconds would be spent syncing buffers when writing the 3 tape marks next to a new file. Their proposal was to introduce a new Linux system call to the SCSI tape driver that writes a tape mark without forcing a subsequent flush operation. Their legacy software has then been modified in a way that only the last of the 3 tape marks written would trigger a flush.

When writing the LTFS specification, we determined that tape marks would not be used to delineate the location of files, but rather to determine the locations of the LTFS Indexes. Because these Indexes are important to assert a consistent file system view, it is critical to flush outstanding buffers when writing a tape mark. In order to achieve a compromise between data protection and performance, LTFS provides three policies to control how often the Index is flushed to tape. In the first, the flush is programmed to occur when a file opened for writing is closed. In the second, a timer determines how often to save the Index if changes to the tape's contents have been made. In the third, the LTFS Index is flushed only when the filesystem is unmounted.

Other authors have focused on determining data placement strategies for tertiary storage on hierarchical storage management environments (HSM). Li and Prabhakar [9] have developed probability functions that estimate the relationships between file objects on a primary or secondary storage. Once objects with a strong connection

[2]Tape marks are special in the sense that the drive can be programmed to seek at high speed to the next or previous mark. For that reason, they are frequently used to delineate the beginning of new contents on tape.

are identified, they can be migrated to the same tertiary storage (tape) medium. The rationale behind this approach is that by grouping files based on their relationship the cost of future read and seek operations will be reduced. Given the heterogeneity of the environments in which the Unified scheduler is expected to be used, we deliberately chose not to make assumptions about anticipated workloads or inter-data dependencies (which, in fact, may not be easily determined on non-HSM environments). Instead, we focus on reducing file fragmentation, which also leads to improved read times[3]. Section 6 holds details about the gains obtained when reading back scattered files written with and without special treatment for fragmentation.

5. I/O Scheduler Design

In this section we describe the design of the Unified I/O scheduler, which is implemented as a loadable module in the user-space LTFS software stack. Sections 5.1 and 5.2 introduce essential data structures that our design relies on. The core novelties introduced by this article are presented in Section 5.3: (1) the use of redundant file copies combined with delayed writes, employed to reduce the number of partition switches and catch outliers, and (2) an effective caching management, used to reduce internal file fragmentation.

A. Request cache objects

The *request cache object* (RCO) is a fixed-size data structure that holds outstanding data written to LTFS. Its size is defined to match the low-level tape block size so that optimum efficiency is reached when flushing a file's list of RCO entries to tape. The RCO has three major members: the logical *file offset* where the data must be written, the *size* of the data it holds, and its *state*, which can be one of the following:

- `PARTIAL`: the RCO buffer is not full;
- `DATA_PARTITION`: the RCO buffer is fully utilized and is ready to be written to the Data Partition;
- `INDEX_PARTITION`: the RCO buffer is fully utilized, has already been written to the Data Partition and is now waiting to be written to the Index Partition.

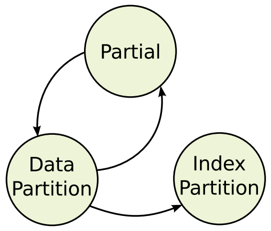


Fig. 2: Transitions and states of a request cache object.

[3]LTFS runs on a variety of HSM stacks that could still group files according to user policies before submitting them to tape through the Unified scheduler.

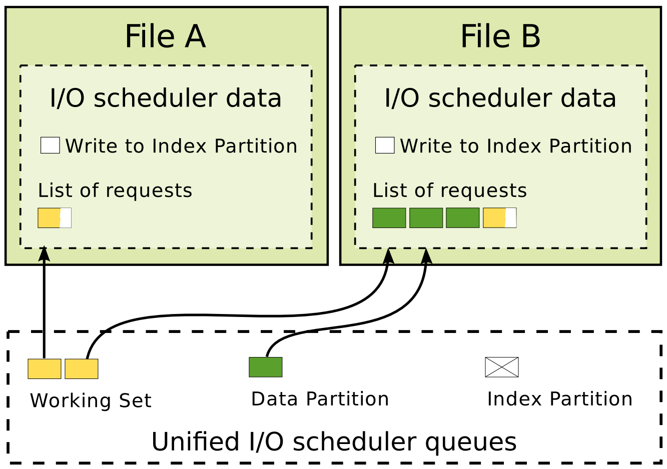


Fig. 3: Management of request cache objects and the Unified queues.

Figure 2 shows the possible states and transitions that a RCO can make. The transition from `DATA_PARTITION` to `PARTIAL` can only happen on a file truncation operation that affects an RCO which was previously fully utilized. The inverse transition is made when more contents are appended to the RCO buffer causing it to become full. On the other hand, once an RCO state is set to `INDEX_PARTITION` it never changes – the RCO is either written to tape or removed from the file's RCO list if it no longer needs to be written to that partition. Finally, at most one RCO of a given file can be in `PARTIAL` state, in which case it will be found at the end of that file's RCO list.

B. Queue Membership

Unified manages outstanding requests with the assistance of three queues that provide efficient access to files that have dirty data[4]. At any time, a file can be a member of zero or more queues, depending on how many RCOs of that file are queued for writing and on whether the file meets the Index Partition data placement rules of that tape or not.

The first queue, called "Working Set", points to files that have partial requests. That is the case of files A and B in Figure 3. The second queue is the "Data Partition" queue, whose members are files that have at least one full request. The "Index Partition" queue holds pointers to files that satisfy the data placement rules for being written to the Index Partition. The example depicted in Figure 3 shows no files fulfilling that rule.

C. Write operations

Optimizing write operations is key to fully exploit the streaming capabilities of tape hardware. Unified handles these operations by queueing writes in the request lists and by regulating the conditions in which the data is flushed to tape.

[4]Data that was written by the application but not yet written to the medium.

The scheduler implementation uses insertion sort to arrange the RCO entries of a file according to their starting *offset*. The traditional insertion sort algorithm has been slightly optimized to benefit sequential operations: if the *offset* passed by a write call is beyond the end (given by the last entry on that list) then the request is simply appended to the list with no need for complete traversal. This optimization provides our insertion sort algorithm an *O(1)* complexity on streaming writes.

The write algorithm also deals with overlapping addresses in the cached objects. If an RCO already holds one or more bytes of data starting at the *offset* passed by a new write call, then that RCO's data blocks at the overlapped addresses are superseded by the new data. Any extra bytes that will not fit in the updated cache object are then either merged with the next object in the linked list or placed in a new `PARTIAL` cache object.

Unified employs a dedicated writer thread to control requests that are ready to be moved from the cache objects to the medium. The writer thread is critical to the performance of the file system. Consequently, there are four circumstances in which the writer thread awakes to process outstanding requests:

1. Explicit file flushes: To satisfy the predictability expected by some applications, the `fsync` system call always ensures that all outstanding buffers of the corresponding file are flushed to tape.

2. Fully utilized RCOs: When the first RCO in a request list changes state from `PARTIAL` to `DATA_PARTITION` the writer thread is woken to prepare for potential data bursts. The list of RCOs of each file belonging to the Data Partition queue is searched by that thread and any requests with the state `DATA_PARTITION` are written to tape. Afterwards, the file is removed from the Data Partition queue.

Next, if the file meets the Index Partition criteria, then the processed requests are changed to state `INDEX_PARTITION` and the file is added to Unified's Index Partition queue. The processing of data written to the Index Partition is detailed below.

This exposes one key aspect of the I/O scheduler's design: even when a file matches the criteria to be saved to the Index Partition, it is still written to the Data Partition. There are three related reasons for this design. First, alternating between partitions on tape is expensive due to seek delays. Second, by indefinitely postponing data writes, events like power outage may lead to data loss if no data blocks from a file ever met the tape; by writing these blocks to the Data Partition, we are protected against such problems. Third, it is possible to catch outliers — files that were subject to the Index Partition placement policy but that have grown beyond the size allowed by that policy. These should not benefit from the privileged storage in the Index Partition.

Moreover, by flushing data to tape one file at a time, we reduce file fragmentation when there are concurrent write requests for different files. Because many cached buffers from a given file are written at once, large file extents can be produced. That translates to more compact LTFS Indexes and to more efficient read operations.

3. Cache pressure: When the number of RCOs in use reaches the maximum value established for the current LTFS instance, Unified enters a *cache pressure* mode. All incoming write requests are put on hold until some of the pending write requests are flushed to the medium.

Under cache pressure, Unified always tries to free at least twice as many cache objects as there are blocked writer threads. It first checks for requests in state `DATA_PARTITION` that are not eligible for placement in the Index Partition. If there are enough of these, the scheduler flushes the Data Partition queue and returns. This avoids inefficient partial-block writes or a costly Index Partition seek.

If the condition above is not satisfied, Unified checks the number of requests in `INDEX_PARTITION` state against a high water mark. If the high water mark has been reached, the scheduler flushes the Index Partition queue. Otherwise, it flushes both the Data Partition and Working Set queues, writing all eligible blocks (including partial sized ones) to the Data Partition.

4. File system unmount: In this case, the writer thread first writes RCOs whose state are either `DATA_PARTITION` or `PARTIAL` to tape (see Figure 4a). Those RCOs that are marked for writing to the Index Partition have their state changed to `INDEX_PARTITION` thereafter and are written to tape together with other RCOs that share that same state (Figure 4b).

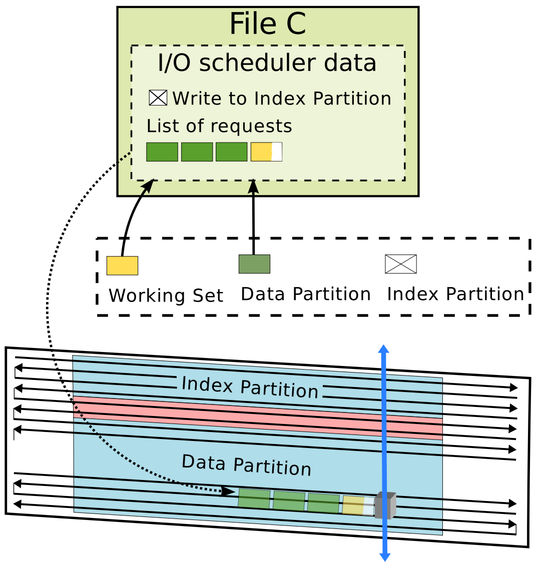
D. File synchronization

Having support for asynchronous writes in the file system has one major implication for applications: they may not have a chance to get a notification from the file system if an error condition has been raised after a write operation. Since the `write` call returns immediately after submitting its data buffers to the I/O scheduler's queue, I/O errors associated with these requests may be only detected long afterwards.

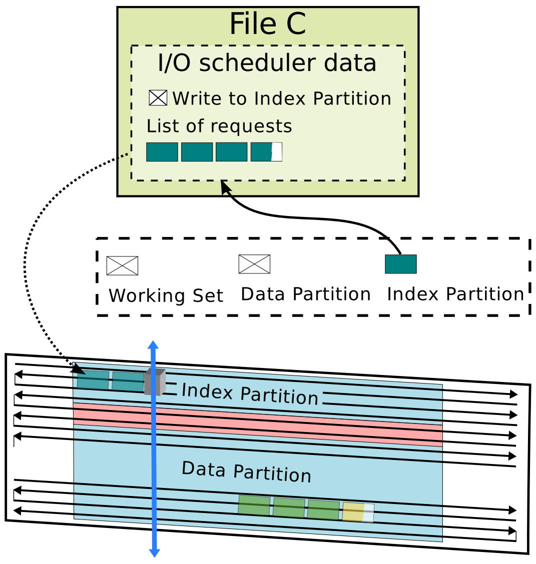
To let applications become aware of file I/O problems, the `close` system call has been made synchronous for files that were either created or updated. Any queued entries in the request list of the file are processed, and errors, if any, are propagated through the return of that system call. The synchronization process on `close` is the same one performed by Unified on an explicit `fsync`.

E. Read operations

Reading files through Unified may benefit from the RCOs of a file. If the data asked for is completely cached in the RCOs, then the requested can be served without involving access to the tape medium. If it is not, then the scheduler queues up reads from tape to get the missing data. Similarly to write requests, queued up reads are sorted by the logical file offset to enable sequential tape reads.



(a) Flush of the Data Partition and the Working Set queues



(b) Flush of requests that migrated to the Index Partition queue

Fig. 4: Unified queue flushes.

Many file systems offer some level of prefetching when reading data from the medium. We have analyzed two approaches to prefetching in LTFS: file-based and block-based.

File-based prefetching is implicitly performed by stream-based operations that utilize buffered I/O. That is the case of the `fread` and `fwrite` family of functions. The drawback with that approach is that there are situations in which the file extents on tape are interleaved with extents from other files. If parallel reads are issued for two such files, then seeking the tape forwards to prefetch data for one file may cause tape backhitch[5] when data for the other file is demanded.

Block-based prefetching, on the other hand, has the potential to improve performance in two situations. First, by asking more data to the drive than requested by the application, it is possible to sustain faster streaming rates. Second, it can save expensive backwards seeks when files with interleaved extents are read together, as long as the prefetching window is large enough.

Since tape drives are optimized for streaming I/O, the hardware normally keeps reading ahead until the application stops requesting to pull more data. Depending on the caching strategies implemented by the hardware, however, it is possible that data blocks at smaller addresses than the current tape position are no longer in that cache. Therefore, we investigate block-level prefetching as a way to optimize that situation alone.

Support for block-level prefetching has been implemented in the LTFS tape driver layer, which is the lowest layer of the LTFS software stack. That layer's main duty is to issue low level *SCSI* commands that drive the tape hardware via the operating system. A number of sequentially prefetched blocks are stored in a cache within that layer. When an application asks to read from tape, the requested file offset is translated to a block address and, if that address happens to be in cache, then a tape I/O is avoided. Our evaluation of block prefetching is presented in the next section of this paper.

6. Performance Analysis

In this section we investigate to which extent the features introduced by Unified improve LTFS and examine why certain mechanisms have failed to show good results. We use synthetic workloads that reproduce the behavior of having a user drag-and-dropping several files to the LTFS mount point and of having applications reading back these files from tape (as is often the case of anti-virus software when several files may be scanned in parallel).

Our evaluation of the performance gains analyzes four aspects that directly impact LTFS efficiency: (1) the placement of extents in tape medium, (2) streaming reads and writes to tape, (3) reading two or more files with interleaved extents, and (4) the effect of deferred Index Partition writes on throughput.

We compare the results obtained with the stock First-Come, First-Served (FCFS) scheduler available on LTFS using metrics like throughput, the number of data extents allocated per file, and the number of seconds spent on I/O.

Environment

The sequential read and write throughput of LTFS is measured using an IBM LTO-5 tape drive, attached via SAS to a modern server class machine running Linux kernel 3.14.1. The tape drive in question is rated for a maximum throughput of 140 MiB/second, excluding the

[5]The repeated motion made by the tape drive when seeking back and forth.

effects of hardware compression (which was disabled for testing). The same class of hardware is driven by Linux 2.6.18 in our data placement experiments. Parallel I/O operations are measured on the same class of tape drive on an Intel Xeon E5530 machine with two Quad Core processors that expose a total of 16 logical processing units.

A. Placement of file extents on tape

In this scenario we evaluate the number of data extents created and their sizes when writing to files in parallel. In our test, the writes issued by the application reach LTFS in blocks with a maximum size of 128 KiB, which is determined by the Linux FUSE driver. Due to the divergence between that size and the tape block size, extents of sub-optimal, fixed sizes are created when writes are orchestrated by the FSFS scheduler, as shown in Figure 5a. In this first experiment, the data blocks of each file created are distributed over an average of 277 extents, with a standard deviation of 3.

The same process is repeated with the Unified scheduler, resulting in a reduction to an average of 4 extents per file with a standard deviation of 0.75. By visually inspecting the placement of file blocks (Figure 5b) we observe two details of the initial implementation of our scheduler:

1) *The first extents written to tape are small.* This is a consequence of waking up the writer thread as soon as a complete cache object is full: when there are too few requests on the request lists, small extents may be produced. We approach this problem by deferring writes until a minimum number of requests are queued.
2) *The majority of extents are very large.* Once continuous data flows start to arrive at the I/O scheduler, a substantial number of cache objects accumulate in the request lists and coalesced writes are possible, resulting in extents of greater size.

The fifth row of Figure 5a, toward the left edge, also shows a sequence of 11 contiguous blocks belonging to File G. That layout differs from the distribution of blocks of the other files and suggests that other writers may have been starved of access to the media during this period of execution. This starvation is an artifact of task scheduling by the kernel. Our introduction of asynchronous I/O operations afforded by Unified masks these kinds of task scheduling artifacts from the tasks performing I/O to LTFS.

B. Scattered reads

We evaluate two scenarios to gauge the impact of spatial distribution of file extents on write throughput. In both scenarios we measure the throughput by reading 10 files that had been previously written in parallel to the tape.

In the first scenario, the files are written with the LTFS FCFS algorithm which produces 4498 extents per file, on average, with a fixed size of 128KiB. In the second scenario, files are written through the I/O scheduler which produces an average of 76 extents per file with a standard deviation of 11. The average size of each extent is of 7.7MiB.

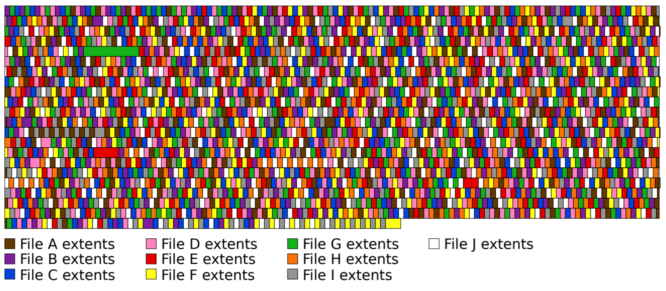
Figure 6 shows 10 files (named from A to J) grouped in pairs. The vertical bars on the left of each pair represent the time to read the files produced with the FCFS policy; the bars on the right hand side represent the elapsed time while reading files written by Unified. Further, each vertical bar is split in two, with the CPU time of the read process being depicted at the bottom and the I/O time on top. All reads are performed with the I/O scheduler disabled to turn off buffering at the file system level.

As expected, the CPU time is mostly the same across different runs. The difference in I/O time, on the other hand, clearly shows the gains obtained by the aggregated writes of our scheduler.

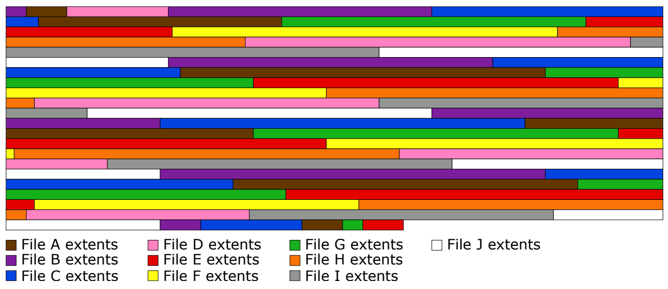
C. Sequential throughput

For each test run, the GNU `dd` utility is used to either read or write 2 GiB of data in two iterations of 1 GiB each. The first iteration is used to force the tape drive to start moving, and throughput is measured on the second iteration. I/O size is varied from 1 KiB to 64 KiB, and the mean measured throughput from 10 runs is taken at each size, ignoring any data points where the tape drive reaches the end of the tape and turns around (a known source of reduced throughput). For read testing, the kernel's buffer cache is cleared before each run.

Results are presented in Figure 7. Write throughput



(a) Data extents with no I/O schedulers loaded



(b) Data extents produced with the Unified I/O scheduler

Fig. 5: Data placement of different files written in parallel on LTFS. File extent sizes are shown in the X axis. Several lines are used due to space restriction.

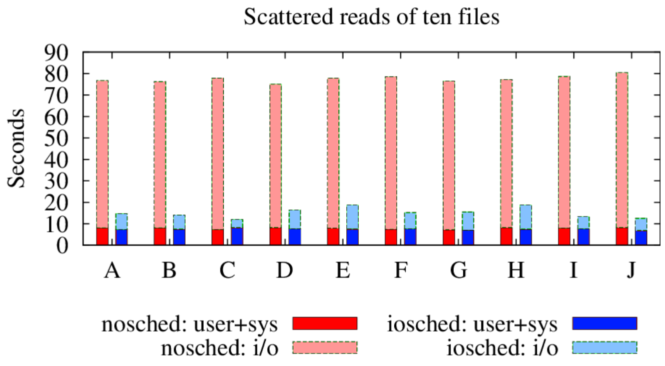


Fig. 6: Time to read back scattered extents from 10 files of 500MB each.

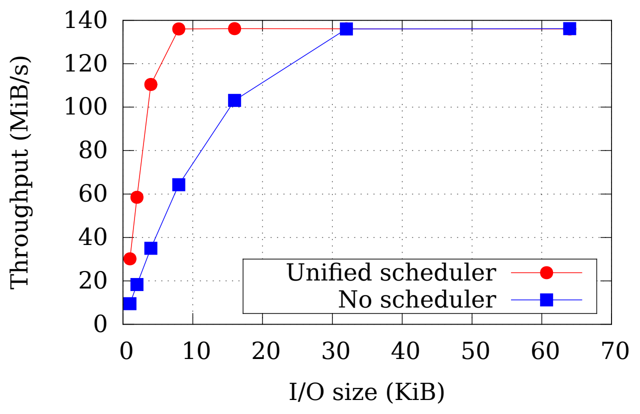


Fig. 7: Write throughput versus I/O size with and without the I/O scheduler.

without an I/O scheduler in LTFS requires 32 KiB writes to reach the limits of the hardware. Turning on the I/O scheduler enables full throughput with 8 KiB writes, with increased throughput compared to the no-scheduler case at smaller I/O sizes. Reduced throughput at lower I/O sizes is likely explained by the need to perform a context switch to the LTFS process for every request. Error bars for the measurements are too small to show on the plot, with a measured standard deviation of 3 MiB/s or less at all I/O sizes.

Read throughput consistently reaches 136 MiB/s regardless of I/O size and is not plotted.

D. Reading files with interleaved extents

In this scenario we measure the total time required to read a sequence of 200 files of 10MiB each. Every file features two discontiguous extents that are interleaved with the the first extent of the next file. Two prefetching strategies are evaluated: (1) look-ahead and (2) a combination of look-ahead with look-behind (in which prefetching actually starts before the block address requested by the user).

The first test compares the performance of no prefetching against different read-aheads setups, whose results can be seen in Figure 8. The configuration with 12 threads

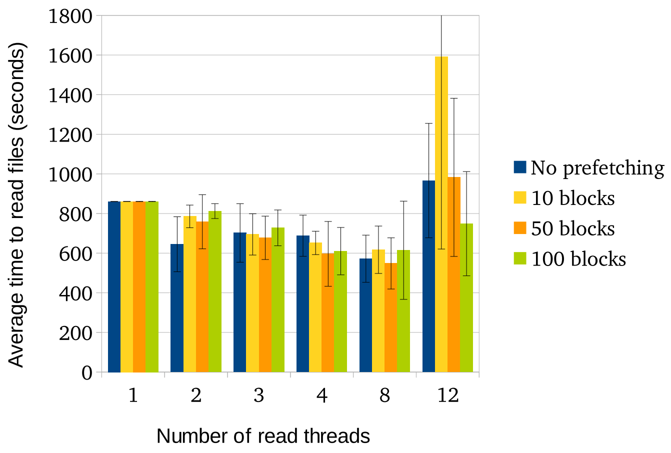


Fig. 8: Block prefetching times with different look-ahead configurations and number of parallel read threads.

shows a considerable performance degradation due to the excessive number of active threads at a time: besides the reader tasks, there are several FUSE threads serving requests from these tasks, plus the LTFS process. The remaining configurations, especially the one with a single thread, do not benefit from prefetching: as the tape hardware employs this same type of optimization, prefetching in software not only shows no gains but may even introduce overhead to the file system. That is clearly reflected by the run with 2 threads.

Motivated by the reasoning that commercial tape drives do not implement look-behind, we evaluate the combination of look-behind and look-ahead prefetching. Due to the performance degradation observed with 12 threads on the previous test, that configuration is not evaluated this time. Two combinations of prefetching are used. In one, 50% of blocks are read behind the requested address followed by 50% of blocks being read ahead of it. In the other, that ratio is changed to 75% and 25%.

The results shown in Figure 9 indicate that prefetching is highly sensitive to the number of blocks chosen, as the configuration with 50 blocks attests. With an increase in the number of blocks prefetched, the chance of resource contention elevates and may increase the length of tape that other threads need to run on seeks. Also, as indicated by the configuration with 8 threads, the arrangement of the files used in the test do not gain real benefits from prefetching.

Based on the numbers obtained in this study, we come to two conclusions. First, due to tape hardware optimizations, software-based look-ahead prefetching does not translate to tape read improvements. Second, while the combination of look-behind and look-ahead may sound appealing, only certain file arrangement combined with particular access patterns may benefit from it, granted that the right choice of number of blocks to prefetch is used.

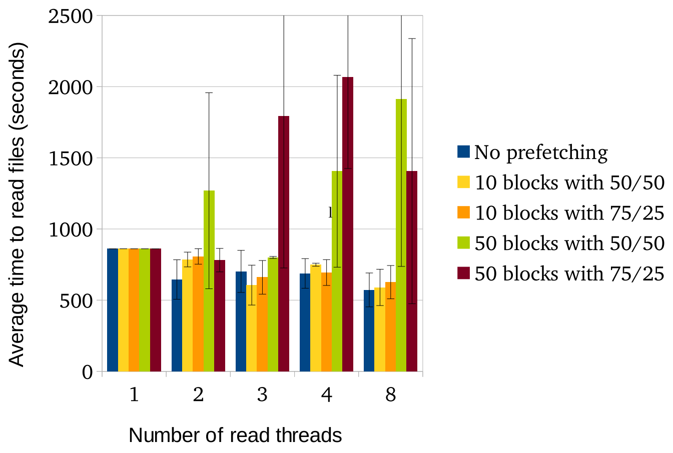


Fig. 9: Block prefetching times with look-behind and look-ahead configurations.

E. *Deferred Index Partition writes*

The effect of deferred Index Partition writes is demonstrated by varying the scheduler's cache size. Cache size control is a standard feature of our LTFS implementation. In each test, 1000 files of 10 MiB each are written to the tape, with 10 of these files matching the Index Partition placement criteria set at mount time. The total size of Index Partition blocks is therefore 100 MiB, larger than the scheduler's default cache size of 50 MiB. The `fsync` system call is invoked on every file before it is closed, so at any given time the scheduler's cache should contain blocks bound for the Index Partition plus at most 10 MiB of Data Partition writes.

With the default cache settings, writing all 1000 files takes 159 seconds, for an average throughput of 63 MiB/s. Increasing the cache size to 1 GiB decreases the write time to 82 seconds, for an average throughput of 122 MiB/s. This number is still lower than the sequential throughput reported in the subsection above because this experiment does not control for tape drive start/stop time.

With the larger cache, all Index Partition writes are deferred until the tape is unmounted. Since the data placed on the Index Partition should be very small to reduce read access time (e.g. 1 GiB or less), increasing the cache size to prevent Index Partition flushes is a reasonable choice.

7. Conclusion

In this paper, we have detailed the design and operation of Unified, an I/O scheduler for dual-partitioned magnetic tape media, and evaluated several optimization techniques to address situations that former tape management software did not have to handle. The outcome of this work is a set of recommendations for optimizing on-line I/O on tapes and on other log-structured file systems.

A novel contribution of this paper relates to the management of dual-partitioned tapes. We realize that the costs incurred by interleaved writes to different partitions is addressed by letting all data blocks of a file land on the Data Partition first. Then, by delaying seeks to the Index Partition our scheduler is able to: (1) catch outliers that no longer meet the data placement criteria, (2) dramatically improve write throughput of small files, and (3) reduce the risk of data loss caused by deferred writes.

We observe that internal file fragmentation caused by parallel writes to tapes can be significantly reduced by the use of simple buffering techniques. In Unified, we developed a sophisticated in-memory data cache managed at the file level that allows the scheduler to coalesce data writes, leading to the creation of larger data extents on the tape media and to more efficient use of SCSI I/O operations. Our performance evaluations show that these techniques allow file-system read/write operations at data rates close to the speed of the raw tape hardware. Further, We believe that this optimization may also benefit other storage media such as disk-based devices.

A related benefit provided by buffering is that I/O operations from user tasks are not exposed to scheduling artifacts that result from the kernel task scheduler. Because outstanding write requests are delayed and grouped on a per-file basis, such artifacts are masked.

We also find that block-based prefetching, which is commonly used by disk I/O schedulers to improve reads of data near the last blocks accessed, does not bring the same benefits to tapes when that functionality is already present on tape hardware. Read-behind prefetching, although not implemented by the tape drives used in our evaluation, turn out to cause the prefetcher to hold the tape for more time than needed, potentially introducing more overhead to parallel operations in the file system.

The main findings presented in this paper and the implementation of the Unified scheduler have been integrated in the open source LTFS software for single tape drives that is used by several companies who offer LTFS solutions.

References

[1] A Cavalli, L dell'Agnello, A Ghiselli, D Gregori, L Magnoni, B Martelli, M Mazzucato, A Prosperini, P P Ricci, E Ronchieri, V Sapunenko, V Vagnoni, D Vitlacil, and R Zappi. StoRM-GPFS-TSM: A new approach to hierarchical storage management for the LHC experiments. *Journal of Physics: Conference Series*, 219(7):072030, 2010.

[2] Gary Field and Peter M Ridge. *The Book of SCSI*. No Starch Press, 555 De Haro St., Suite 250. San Francisco, CA 94107, 2nd edition, Jun 2000.

[3] Joel C Frank, Ethan L Miller, Ian F Adams, and Daniel C Rosenthal. Evolutionary trends in a supercomputing tertiary storage environment. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 411–419. IEEE, 2012.

[4] S. Ghandeharizadeh and C. Shahabi. On multimedia repositories, personal computers, and hierarchical storage systems. In *Proceedings of the Second ACM*

International Conference on Multimedia, MULTIMEDIA '94, pages 407–416, New York, NY, USA, 1994. ACM.

[5] Bruce K Hillyer and Avi Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 170–179. ACM, 1996.

[6] IBM Corporation. IBM TotalStorage LTO Ultrium tape drive: SCSI reference, Second edition, Feb 2013.

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

[8] G.A. Jaquette. LTO: A better format for mid-range tape. *IBM Journal of Research and Development*, 47(4):429–444, July 2003.

[9] Jiangtao Li and Sunil Prabhakar. Data placement for tertiary storage. In *Tenth Goddard Conference on Mass Storage Systems and Technologies: in cooperation with the Nineteenth IEEE Symposium on Mass Storage Systems*, volume 2002, pages 193–208. National Aeronautics and Space Administration, Goddard Space Flight Center, 2002.

[10] S Murray, V Bahyl, G Cancio, E Cano, V Kotlyar, G Lo Presti, G Lo Re, and S Ponce. Tape write-efficiency improvements in CASTOR. *Journal of Physics: Conference Series*, 396(4):042042, 2012.

[11] Henry Newman. The Tape Advantage: Benefits of Tape over Disk in Storage Applications, Apr 2008. White paper, Instrumental.

[12] David Pease, Arnon Amir, Lucas Villa Real, Brian Biskeborn, Michael Richmond, and Atsushi Abe. The Linear Tape File System. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *MSST*, pages 1–8. IEEE Computer Society, 2010.

[13] Varun Prakash, Xi Zhao, Yuanfeng Wen, and Weidong Shi. Back to the future: Using magnetic tapes in cloud based storage infrastructures. In David M. Eyers and Karsten Schwan, editors, *Middleware*, volume 8275 of *Lecture Notes in Computer Science*, pages 328–347. Springer, 2013.

[14] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM.

[15] Rainer Richter. LTFS hits the mark in Media & Entertainment: An in-depth introduction to LTFS for digital media, Jun 2012.

[16] O. Sandstå and R. Midtstraum. Improving the access time performance of serpentine tape drives. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 542–551, Mar 1999.

[17] Olav Sandstå, Thomas Maukon Andersen, Roger Midtstraum, and Rune Sætre. Access time modeling of a MLR1 tape drive. In *Proceedings of Norsk Informatikkonferanse, Voss, Norway*, pages 267–278, Nov 1997.

[18] Jonathan Schaeffer and Andrs Gmez Casanova. TReqS: The Tape REQuest Scheduler. *Journal of Physics: Conference Series*, 331(4):042040, 2011.

[19] Storage Networking Industry Association (SNIA). Linear Tape File System (LTFS) format specification, Dec 2013. LTFS Format Version 2.2.0.

[20] JieBing Yu and David J DeWitt. Processing satellite images on tertiary storage: A study of the impact of tile size on performance. In *Proceedings of the 1996 NASA Conference on Mass Storage Systems*, pages 460–476, Sep 1996.