Tape write-efficiency improvements in CASTOR

S Murray[1], V Bahyl[1], G Cancio[1], E Cano[1], V Kotlyar[2], G Lo Presti[1], G Lo Re[1] and S Ponce[1]

[1] European Organization for Nuclear Research CERN, CH-1211 Genève 23, Switzerland
[2] State Research Center of Russian Federation Institute for High Energy Physics, RU-142281, Protvino, Moscow region, Russia

E-mail: Steven.Murray@cern.ch, Vladimir.Bahyl@cern.ch, German.Cancio.Melia@cern.ch, Eric.Cano@cern.ch, Victor.Kotlyar@ihep.ru, Giuseppe.LoPresti@cern.ch, Giuseppe.Lo.Re@cern.ch and Sebastien.Ponce@cern.ch

Abstract. The CERN Advanced STORage manager (CASTOR) is used to archive to tape the physics data of past and present physics experiments. For reasons of physical storage space, all of the tape resident data in CASTOR are repacked onto higher density tapes approximately every two years. Improving the performance of writing files smaller than 2GB to tape is essential in order to keep the time needed to repack all of the tape resident data within a period of no more than 1 year. This paper reports on the solution to writing efficiently to tape that is currently in its early deployment phases at CERN.

1. Introduction

The CERN Advanced STORage manager (CASTOR) [1] is used to archive to tape the physics data of past and present physics experiments. The current size of the tape archive is approximately 66 PB. Data is migrated (repacked) from older, lower density tapes to newer, high-density tapes approximately every two years to follow the evolution of tape technologies and to keep the volume occupied by the tape cartridges relatively stable. Improving the performance of writing files smaller than 2GB to tape is critical in order to keep the time needed to repack all of the tape-resident data to within a period of no more than 1 year.

The tape format used by CASTOR is based on the ANSI AUL tape format [2]. The specifics of the format are described in [3]. Figure 1 illustrates how a single user file appears on tape. Three files are written to tape for each user file archived; a header file followed by the user file followed by a trailer file. Each tape file is separated from the previous one by a tape mark. A tape mark is recorded to tape in different ways depending on the tape technology used. The header and trailer files contain metadata and both have a length of 240 bytes.

Header file	Tape mark	**User file**	Tape mark	Trailer file	Tape mark

Figure 1. The CASTOR tape-format writes 3 tape files for every user file

Before any of the write efficiency improvements described in this paper were introduced, CASTOR always flushed the write buffers of the underlying tape system after writing a tape mark. The contents of these buffers are stored in volatile RAM and a power cut could lose their contents before they are flushed to tape. The buffers must therefore be flushed before a file can be guaranteed to be safely on tape. It takes approximately 1.7 seconds to flush the buffers to tape. Given the tape format, 3 tape marks are written per user file and therefore before the tape write-efficiency project started, the buffers were flushed 3 times per user file. This equates to approximately 5 seconds (3 × 1.7 seconds) being spent on flushing buffers per user file. The current write speeds of enterprise tape drives are reaching over 240MB per second, therefore an enterprise tape drive can write approximately 1.2 GB in 5 seconds. The average size of a file in CASTOR is 290MB; therefore the write efficiency would be significantly increased if the time spent flushing were instead used to write data. This paper reports on the recently deployed CASTOR installation at CERN where tape write-efficiency has been considerably improved by reducing the number of times the write buffers are flushed.

The tape write-efficiency project investigated two different solutions. The first is presented in [4] and was to change the tape format. The second and chosen solution was to control exactly when the tape write-buffers are flushed. The following section describes these two approaches and explains why the latter was chosen based on the main advantages and disadvantages of both. The section after that describes the WRITE-FILEMARKS SCSI-command and how it can be used to control when the tape write-buffers are flushed. The subsequent section describes the architecture of CASTOR before the current write-efficiency solution was implemented. The installation of the efficiency improvements was divided into two incremental deployments in order to reduce the risk of corrupting user data. The two following sections describe these deployments. The next section after these describes the methodology taken with respect to handling the modification of the legacy code responsible for the mission critical task of reading from and writing to tape. The penultimate section presents some performance plots showing the actual results obtained by the tape write-efficiency project. The final section then presents some conclusions.

2. Changing the CASTOR tape-format versus controlling exactly when to flush

This section explains why the tape write-efficiency project chose to improve efficiency by controlling exactly when the write buffers of the underlying tape system are flushed as opposed to changing the tape format. The explanation is based on describing and comparing the advantages and disadvantages of both solutions.

The solution to increase tape write-efficiency by changing the tape format is presented in [4]. This is a common solution and the UNIX tar command is a good example. By default the tape write-buffers are synchronously flushed when a tape mark is written to indicate the end of a tape file. The solution of putting many user files into a single tape file therefore decreases the number of flushes without the need to use any advanced SCSI features.

The main advantage of changing the tape format is the possibility to add more metadata. If more metadata could be added then they would include the size, checksum and path of each user file. This metadata is currently stored in the CASTOR namespace database. Both the header and trailer files of the tape format include a unique numeric identifier that can be used to lookup these 3 pieces of information in the namespace database. If the size and checksum of each user file were stored on tape then the integrity of a tape could be tested without the need to connect to the namespace database. If the path of each user file were stored on tape then tapes shipped to other sites could then have their contents listed without having the original namespace database available or reachable. Please note that the proposed new format specified in [4] stores the path of a user file only once per user file. The path is not written to tape again if the file is renamed or moved to a different directory. This means listing the contents of tape without connecting to the namespace database would give the original location of each user file.

The main disadvantage of changing the tape format would be the relatively big code changes incurred on the legacy code responsible for the mission-critical task of reading from and writing to tape. The modifications for writing to tape would not be so dramatic, but the code responsible for positioning a tape for recall would require major modifications.

The positioning logic of the legacy code relies on the fact that a user file is written to tape as a single tape file; there are no aggregations of 2 or more user files into a single tape file. A tape file is written to tape in blocks and always starts on a block boundary. A tape drive can position to the start of a tape file using two methods. A drive can position by file sequence number (position to the nth file) and by block id (position to the nth block). CASTOR uses both positioning methods for recall. Positioning by block id is tried first, because historically this has been the faster method. If this positioning method is not possible then positioning by file sequence number is tried. When positioning by file sequence number, the drive can revert to simply counting the files from the beginning of the tape. If many user files were to be contained within a single tape file then positioning by block id would still work for positioning to the beginning of a user file, but positioning by file-sequence number would require an additional block offset. Locating the end of the user file would also be a problem. CASTOR currently relies on a user file ending on a tape file boundary, so a block offset from the beginning of the user file would also have to be taken into account in order to determine the end of the file.

Another approach would be to recall the entire tape file in which a specific user file was located and then extract the user file in a later step. This approach would be inefficient because many non-requested user files would be recalled that might not be of interest.

Positioning for read by file sequence number plus a block offset would require significant modifications to the legacy code responsible for reading from tape. This is also true for the approach of always recalling complete tape files.

The second and chosen solution to improving write efficiency is to keep the existing tape format and to control exactly when the tape write-buffers are flushed using the WRITE FILEMARKS SCSI-command.

The main advantage of controlling exactly when the tape write-buffers are flushed is the fact that the existing tape format and all of its positioning logic can be kept as is. Modifications still need to be made to the legacy code; however these will be small and well contained because they are either a modification to hold back from flushing or the addition of new code to make an explicit flush.

CASTOR uses the SCSI command named WRITE FILEMARKS in order to write a tape mark between each tape file. According to the SCSI standard, the WRITE FILEMARKS command allows a program to write zero or more tape marks to tape, with or without flushing the write buffers. A program specifies whether or not the buffers are to be flushed by specifying whether or not the WRITE FILEMARKS command should be executed in immediate mode. When the WRITE FILEMARKS command is executed in immediate mode, the command does not flush the buffers and returns immediately to the calling program, hence the name given to the mode. When the WRITE FILEMARKS command is executed in non-immediate mode, the command writes the specified number of tape marks, flushes the buffers and does not return until the flush has been completed.

The main disadvantage of controlling the flushing of tape write-buffers is, or more importantly was, the fact that at the beginning of the tape write-efficiency project there was no UNIX distribution with a SCSI tape-driver that provided a way to execute the WRITE FILEMARKS command in immediate mode. CERN overcame this problem by working together with the main developer of the Linux SCSI tape-driver (Kai Makisara).

In order to keep the number of Linux system-calls to a minimum and to make the Linux kernel extensible, the Linux `ioctl` system-call is used to perform device specific operations that Kernel developers did not know about at the time they developed a specific version of the kernel or they deemed not worthy of a dedicated Linux system-call. The Linux SCSI tape-driver supports 3 `ioctl` requests: the `MTIOCTOP` request to perform a tape operation, the `MTIOCGET` request to get the status and the `MTIOCPOS` request to get the tape position. When the caller of the `ioctl` system-call passes an `MTIOCTOP` request, the caller must specify an `mtop` structure that specifies the tape operation to be performed and how many times it should be performed. The `MTWEOF` tape operation writes a specified number of tape marks. Underneath it executes a WRITE FILEMARKS SCSI-command in non-immediate mode. The code snippet of figure 2 illustrates how CASTOR used the `ioctl` system-call before the start of tape write-efficiency project in order to write a single tape mark and synchronously flush the write buffers.

```
struct mtop mtop;

mtop.mt_op    = MTWEOF;
mtop.mt_count = 1;

ioctl(tapefd, MTIOCTOP, &mtop);
```

Figure 2. Using the `ioctl` system-call to write a file-mark and synchronously flush

The collaboration between CERN and the main developer of the Linux SCSI tape-driver resulted in the addition of the `MTWEOFI` tape operation. This operation executes a WRITE FILEMARKS SCSI-command in immediate mode. The `MTWEOFI` tape operation is now available as an official SLC5 Kernel-patch and is in the vanilla Linux-Kernel as of version 2.6.37. Documentation about the new tape operation can be found in the `Documentation/scsi/st.txt` file of version 2.6.37 of the official Linux kernel source [5].

Given the above advantages and disadvantages it became clear that the safest course of action to be taken in order to increase tape write efficiency was to preserve the existing tape format and use the WRITE FILEMARKS SCSI-command to control exactly when the tape write-buffers are flushed. In short the decision was made by giving a higher priority to reducing risk than to adding new metadata functionalities that were not part of the original mandate of the tape write-efficiency project.

3. Using the WRITE FILEMARKS SCSI-command to flush the tape-drive write-buffers

In order to control exactly when the tape write-buffers are flushed, the new code of the tape write-efficiency project uses the WRITE FILEMARKS SCSI-command for effectively two different actions. Firstly to write a tape-mark without flushing the tape write-buffers and secondly to simply flush the buffers.

The new write-efficient code always writes a tape mark by executing the WRITE FILEMARKS command in immediate mode. The code snippet of figure 3 illustrates how this is done using the `ioctl` system-call and the new tape operation `MTWEOFI`.

```
struct mtop mtop;

mtop.mt_op    = MTWEOFI;
mtop.mt_count = 1;

ioctl(tapefd, MTIOCTOP, &mtop);
```

Figure 3. Using the `ioctl` system-call to write a file-mark without flushing

In order to ensure data integrity, CASTOR has to determine at regular intervals which files have been safely written to tape. This means the flushing of the tape write-buffers cannot be delayed indefinitely because their contents are stored in volatile RAM and could be lost due to a power cut. The WRITE FILEMARKS command takes as an argument the number of tape marks that should be written to tape. The tape write-buffers can be flushed synchronously by setting the number of tape marks to 0 and setting the execution mode to non-immediate. The code snippet of figure 4 illustrates how the new write-efficient code does this by using the `ioctl` system-call to make an `MTIOCTOP` request to perform an `MTWEOF` tape operation with a count of 0 file marks.

```
struct mtop mtop;

mtop.mt_op    = MTWEOF;
mtop.mt_count = 0;

ioctl(tapefd, MTIOCTOP, &mtop);
```

Figure 4. Using the `ioctl` system-call to flush

4. The CASTOR architecture before the tape write-efficiency improvements

Figure 5 shows how the components of CASTOR wrote to tape before the tape write-efficiency improvements were introduced.

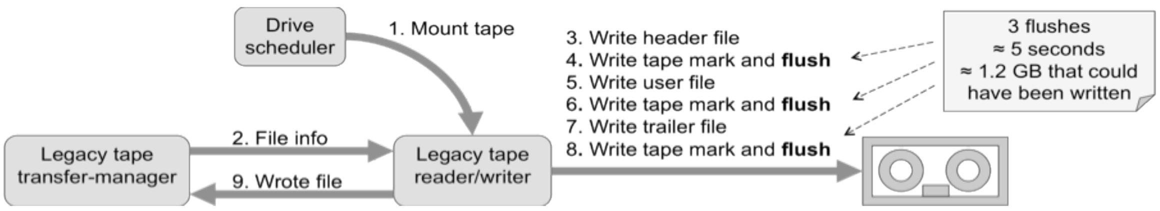
Drive scheduler
1. Mount tape
Legacy tape transfer-manager
2. File info
9. Wrote file
Legacy tape reader/writer
3. Write header file
4. Write tape mark and **flush**
5. Write user file
6. Write tape mark and **flush**
7. Write trailer file
8. Write tape mark and **flush**
3 flushes ≈ 5 seconds ≈ 1.2 GB that could have been written

Figure 5. Writing to tape before the tape write-efficiency improvements

When a file should be written to tape and there is a tape with free space and a compatible tape drive available, the drive scheduler sends a mount-tape message (step 1 of figure 5) to the tape reader/writer on the tape server connected to the drive. The tape reader/writer contains the legacy code for reading from and writing to tape. The tape write-efficiency project was given the stringent constraint of minimizing the number of modifications to be made to this legacy code. The tape reader/writer gets the information about the file to be written to tape from the legacy tape transfer-manager (step 2). This information contains such details as the location of the file on the disk cache. The tape reader/writer ensures the tape is mounted and then writes a header file (step 3), a tape mark with a synchronous flush (step 4), the user file being archived from disk to tape (step 5), a second tape mark with a synchronous flush (step 6), a trailer file (step 7), and a third and final tape mark with a synchronous flush (step 8). The tape reader/writer then reports back to the tape transfer-manager that the file has been safely written to tape (step 9). This architecture meant that the tape write buffers were flushed 3 times per user file. Assuming the write speeds of the latest enterprise tape-drives speeds of around 240 MBs^{-1} and a flush time of 1.7 seconds, the 5 seconds spent flushing (3 × 1.7 s) would have been better spent writing 1.2 GB of data (5 s × 240 MBs^{-1}).

5. The CASTOR architecture after the first tape write-efficiency improvements

Figure 6 shows how a file was written to tape after the first set of write-efficiency improvements were added to CASTOR. The goal of the modifications was to only introduce the use of the WRITE FILEMARKS SCSI-command in immediate mode (using the `ioctl` system-call to make an `MTIOCTOP` request to perform an `MTWEOFI` tape operation).

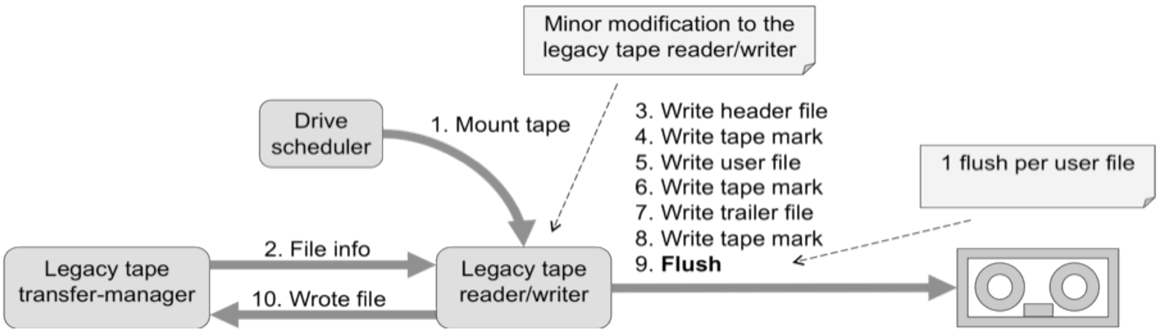

Figure 6. Writing to tape after the first tape write-efficiency improvements

The steps followed to write a file are nearly the same as those before the tape write-efficiency improvements were added. The only difference is that the tape-write buffer is only flushed once per user file instead of 3 times.

Assuming the average size of a file in CASTOR is 290MB, the size of the header and trailer files is negligible, the drive write speed is 240 MBs^{-1} and the time to flush the tape write-buffer is 1.7 seconds, then the time taken to write a user file of average size before the first tape write-efficiency improvements was 6.3 seconds (290 MB ÷ 240 MBs^{-1} + 3 × 1.7 s) or approximately 46 MBs^{-1} (290 MB ÷ 6.3 s). By only flushing the tape-write buffers once, the time taken to write a user file of average size after the first tape write-efficiency improvements was 2.9 seconds (290 MB ÷ 240 MBs^{-1} + 1.7 s) or 100 MBs^{-1} (290 MB ÷ 2.9 s). This means a performance gain of a factor of 2.2 for 290MB files (100 MBs^{-1} ÷ 46 MBs^{-1}).

6. The CASTOR architecture after the complete set of tape write-efficiency improvements

At CERN there is one CASTOR disk cache installation for each large experiment, however due to the relatively high cost of tape drives and robotics there is a single and central tape backend. Upgrades to CASTOR are first deployed to the central tape backend and then to the disk caches of each of the large CERN experiments, one experiment at a time in order to reduce the risk of introducing possible new bugs everywhere at the same time. The second and final set of tape write-efficiency improvements required changes to the protocols used between the many disk caches and the central tape backend. Given these facts the main goal of the final set of tape-write efficiency improvements, besides making further increases in tape write-efficiency, was to allow both legacy and new disk cache installations to communicate with a new central tape backend. Figure 7 illustrates this compatibility requirement.

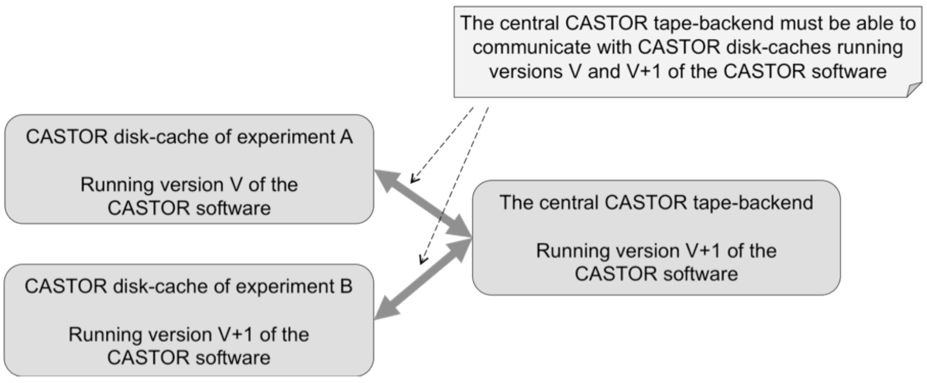
The central CASTOR tape-backend must be able to communicate with CASTOR disk-caches running versions V and V+1 of the CASTOR software
CASTOR disk-cache of experiment A
Running version V of the CASTOR software
The central CASTOR tape-backend
Running version V+1 of the CASTOR software
CASTOR disk-cache of experiment B
Running version V+1 of the CASTOR software

Figure 7. Both legacy and new disk caches need to communicate with a new central tape backend

Figure 8 shows how a file is written to tape after the complete set of tape write-efficiency improvements have been applied.

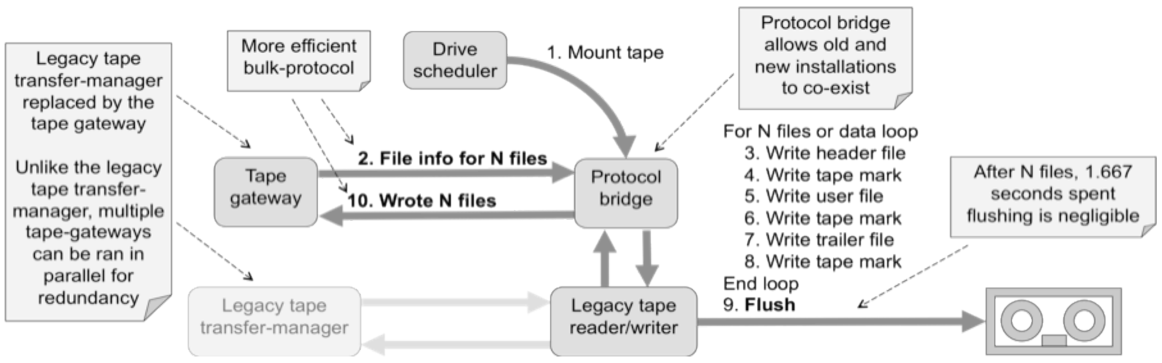
Legacy tape transfer-manager replaced by the tape gateway
Unlike the legacy tape transfer-manager, multiple tape-gateways can be ran in parallel for redundancy
More efficient bulk-protocol
Drive scheduler
1. Mount tape
Protocol bridge allows old and new installations to co-exist
Tape gateway
2. File info for N files
10. Wrote N files
Protocol bridge
For N files or data loop
3. Write header file
4. Write tape mark
5. Write user file
6. Write tape mark
7. Write trailer file
8. Write tape mark
End loop
9. Flush
After N files, 1.667 seconds spent flushing is negligible
Legacy tape transfer-manager
Legacy tape reader/writer

Figure 8. Writing to tape after the final set of write efficiency improvements

The final set of tape write-efficiency improvements includes the addition of two new components, the tape gateway and the protocol bridge. The tape gateway was added to replace the legacy tape transfer-manager. Unlike the legacy tape-transfer manager, multiple tape gateways can be ran in parallel for the same disk cache for redundancy. The tape gateway also communicates files to be transferred using a bulk protocol where many file transfers are described in a single TCP/IP message. This bulk protocol was deemed necessary in order to fully exploit the new tape write-efficiency improvements whilst lowering the overhead of handling the associated metadata. The protocol bridge enables the tape gateway to communicate with the legacy tape reader/writer. Legacy tape-transfer managers are still able to talk directly with the legacy tape reader/writer. This feature allows both old and new disk cache installations to work with the same central tape backend.

Step 1 of figure 8 is slightly different than usual. When a file should be written to tape and there is a tape with free space and a compatible tape drive available, the drive scheduler sends a mount-tape message to the new protocol bridge. Without going into further detail the drive scheduler knows if the tape to be mounted is for an old disk cache installation using the legacy tape transfer-manager or if it is for a new disk cache installation using the new tape gateway. Step 2 is also different. Instead of the tape reader/writer getting the information about a single file-transfer from the legacy tape-transfer manager, the protocol bridge gets the information about many file transfers in a single bulk message from the tape gateway. Steps 3 through to 8 are executed in a loop for each of the file transfers sent within the bulk message. For each file to be written, the usual three tape files, header, user file and trailer are written; however the write buffers are not flushed until the end of the trailer file of the last user file. All of the file transfers listed within a single bulk message are treated as a single transaction.

No communication is made with the tape gateway during the many iterations of the file-writing loop. If there is any failure then the whole transaction of carrying out the many individual file transfers to tape is rolled back. All of the file transfers within the transaction are considered never to have completed. Once rolled back the file transfers are retried at a later point in time, most probably on a different tape server. When all of the files of a given transaction have been written to tape and the write buffers have been successfully flushed, it is guaranteed that all of the files are safely on tape. Only at this point is step 10 carried out. In this step the protocol bridge sends a message back to the tape gateway stating that all of the files of the transaction are safely stored on tape.

7. Methodology used for modifying the legacy code

The legacy code of the tape reader/writer is critical for the safe storage and retrieval of data. Even though modifications to this legacy code were kept to the strict minimum, modifications had to be made nonetheless. The legacy code of the tape reader/writer was written in C. The main strategy taken when modifying the legacy code was to use unit tests whenever and wherever possible. A unit test is a small and independent test carried out on a relatively small piece of a larger program. Such a test in a procedural language like C usual tests a particular use case of a single function or procedure. The legacy code of the tape reader/writer did not come with accompanying unit tests nor was it structured in a way that facilitated the adding of unit tests; therefore as many modifications as possible were implemented as separate functions. These functions were developed along with their own unit tests. Even if the existing legacy code could not be put under a set of unit tests, at least all of the newly added functions would be. This technique is described in [6] where it is given the name "sprout method".

The unit test framework used was Cppunit. A total of 189 unit tests were written during the tape write-efficiency project. Test complexity ranged from single function calls through to testing large portions of the TCP/IP application protocol used by the protocol bridge. Local domain or UNIX domain sockets were found to be of great use when developing unit tests for the TCP/IP application-protocols. They were found to be useful for two reasons. Firstly they did not require TCP/IP ports to be used during the execution of the tests and secondly they reduced the need for multiple threads when testing. A call to connect to a TCP/IP port is blocking by default. A call to connect to a local domain socket is non-blocking by default. This means a test can create a local domain socket, bind it as a listen socket, connect to it, write a message, then accept the connection and read out the reply all in the same thread.

8. Results obtained after deploying the new software of the tape write-efficiency project

The best measure of the success of the tape write-efficiency project is the measurements made of the actual tape write speeds achieved using the new software. Figures 9 and 10 give such measurements and clearly show substantial increases in tape write performance.

Figure 9 shows the write performance of an Oracle STK T10KB drive when used in a controlled environment with the latest write efficient version of CASTOR configured to show tape write speeds before the tape write-efficiency project (3 flushes per user file), after the deployment of the first set of improvements (1 flush per user file) and after the deployment of the second set of improvements (1 flush per several GB). Figure 9 shows an approximate performance improvement of a factor of 3 for the average file size of 290 MB (120 MBs^{-1} ÷ 40 MBs^{-1}).

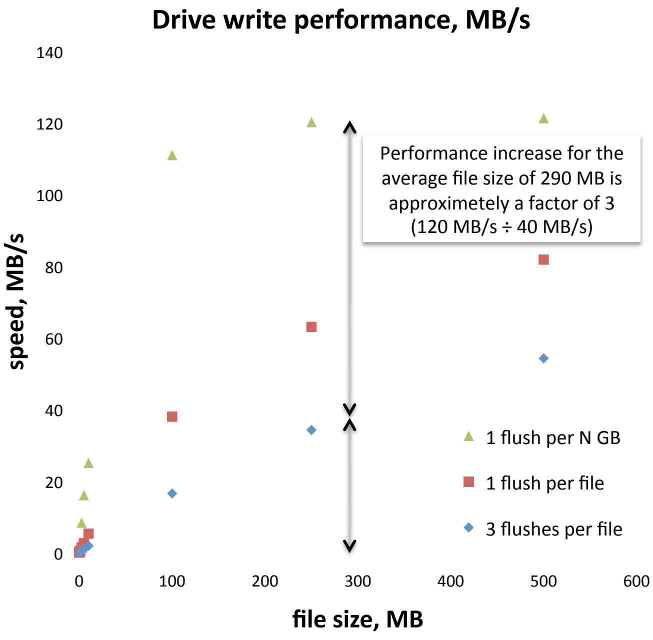

Figure 9. Tape write-performance with an Oracle STK T10KB drive

CERN has a public CASTOR installation that provides general-purpose access to all users at CERN. Figure 10 shows the tape write performance of the public disk cache installation before and after both of the tape efficiency upgrades, that is to say the first upgrade from 3 flushes per user file down to 1 flush and the second upgrade from 1 flush per user file to 1 flush per 32GB worth of user files. Labels 1 and 2 of figure 10 show the first upgrade increased the tape write speed from 13 MBs^{-1} to 39 MBs^{-1}. Labels 3 and 4 show the second upgrade increased the write speed from 51 MBs^{-1} to 144 MBs^{-1}. The increase between the two upgrades from 39 MBs^{-1} to 51MBs^{-1} is explained by the fact that new higher performance tape drives were introduced during this period. The combined increase in performance from 13 MBs^{-1} to 144 MBs^{-1} shows an overall 10-fold increase in tape write speed.

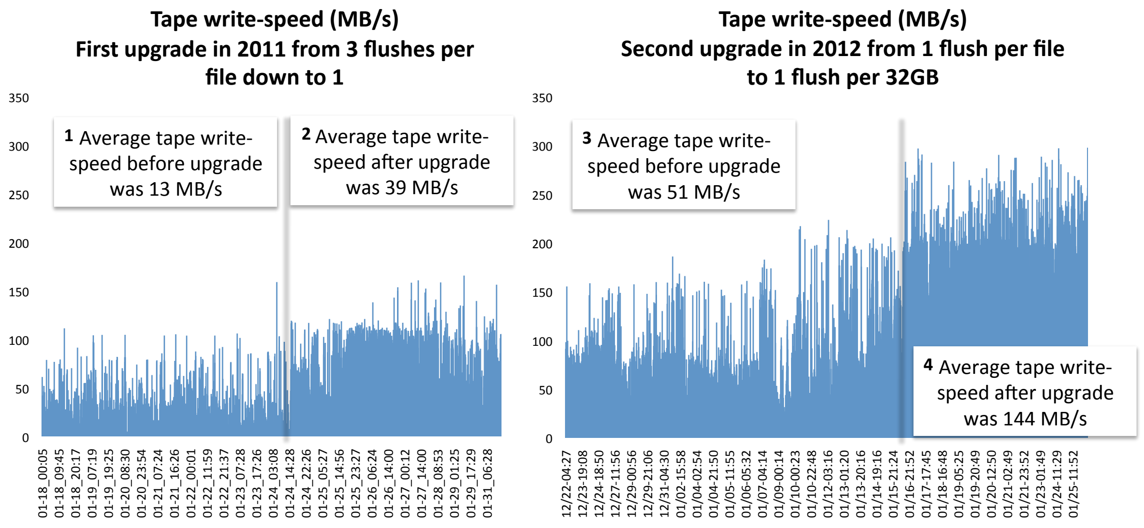

Figure 10. Factor 10 performance gain from upgrading the public CASTOR instance at CERN

9. Conclusion

The tape write-efficiency project identified the main cause of tape write inefficiencies in CASTOR to be the excessive flushing of the write buffers of the underlying tape system. These buffers must be flushed before the files being written can be guaranteed to be safely stored on tape. Before the tape write-efficiency project was started, CASTOR flushed the tape write-buffers three times per user file stored on tape. The strategy of the tape write-efficiency project was therefore to flush the buffers once every N files. After thorough investigations the project chose the solution of preserving the existing tape format and using advanced SCSI commands to control exactly when the buffers were flushed.

Before the tape write-efficiency project started, the standard SCSI tape-drivers of all UNIX distributions always synchronously flushed the tape write-buffers when a tape mark was written to mark the end of a file. CERN successfully collaborated with the main developer of the Linux SCSI tape-driver in order to extend the driver to permit the writing of a tape mark without flushing these buffers.

The new software of the tape-write efficiency project was installed at CERN in two incremental deployments. By taking two relatively small steps as opposed to one big step the risks of interfering with data taking and causing data loss were kept to the bare minimum. The tape write-speeds of CASTOR were approximately increased by a factor of 10 by the combination of the two deployments. The repack operation planned for 2014 should now be comfortably completed as planned. The increase in write efficiency has also led to the possibility of using less tape drives and therefore reducing the overall running costs of the tape backend of CASTOR at CERN.

References

[1] CASTOR homepage *http://castor.web.cern.ch/*

[2] 1991 Using Magnetic Tape Labels File Structure *International Business Machines Corporation Document SC26-4565-01*

[3] Curran C 2008 Tape Labels, ANSI and IBM *http://it-dep-fio-ds.web.cern.ch/it-dep-fio-ds/Documentation/tapedrive/labels.html*

[4] Bessone N, Cancio G, Murray S and Taurelli G 2009 Increasing the efficiency of tape-based storage backends 2010 *J. Phys.: Conf. Ser.* **219** 062038

[5] Makisara K 2010 Documentation/scsi/st.txt *Version 2.6.37 of the Linux kernel source http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.37.tar.bz2*

[6] Feathers M C 2004 *Working Effectively with Legacy Code (Prentice Hall)*