Improving File Tree Traversal Performance by Scheduling I/O Operations in User space

Carl Henrik Lunde, Håvard Espeland, Håkon Kvale Stensland, Pål Halvorsen
Department of Informatics, University of Oslo and Simula Research Laboratory, Norway
email: {chlunde, haavares, haakonks, paalh}@ifi.uio.no

Abstract

Current in-kernel disk schedulers provide efficient means to optimize the order (and minimize disk seeks) of issued, in-queue I/O requests. However, they fail to optimize sequential multi-file operations, like traversing a large file tree, because only requests from one file are available in the scheduling queue at a time. We have therefore investigated a user-level, I/O request sorting approach to reduce inter-file disk arm movements. This is achieved by allowing applications to utilize the placement of inodes and disk blocks to make a one sweep schedule for all file I/Os requested by a process, i.e., data placement information is read first before issuing the low-level I/O requests to the storage system. Our experiments with a modified version of `tar` *show reduced disk arm movements and large performance improvements.*

1 Introduction

There is an ever increasing demand for fast I/O. However, the properties of the mechanical, rotational disks and the way current systems manage these storage devices raise large challenges. In this area, a large number of data placement and disk scheduling mechanisms have been proposed [5], addressing different scenarios, application characteristics and access patterns. Current commodity operating systems implement several policies such that the system administrator can configure the system according to the expected workload, e.g., Linux has the NOOP, Deadline I/O, Anticipatory I/O and Complete Fair Queuing (CFQ) schedulers. What is common for all these schedulers is that they sort the issued, but non-served requests available in the scheduling queue according to the placement on disk to optimize performance.

In modern operating systems, the disk schedulers operate at the kernel level. However, with respect to operations requiring multiple file reads, the current approaches fail to optimize seeks and thus I/O operation performance in an application-wide manner. For Linux, this include applications like (recursive) copying (`cp -r`), deleting (`rm -r`), archiving (`tar` and `zip`), searching (`find`), listing (`ls -R` and file managers), synchronizing (`rsync`) and calculating file tree size (`du`). As exemplified by the `tar` archiving application (see listing 1), the reason for failing is simple and straight forward. These operations read one file at a time. Requests within a single file may be optimized by the operating system, but there is currently no way of ordering requests between different files as the scheduling queue only contains requests for one file. Thus, the inter-file seek distance may be large, reducing the overall performance by orders of magnitude.

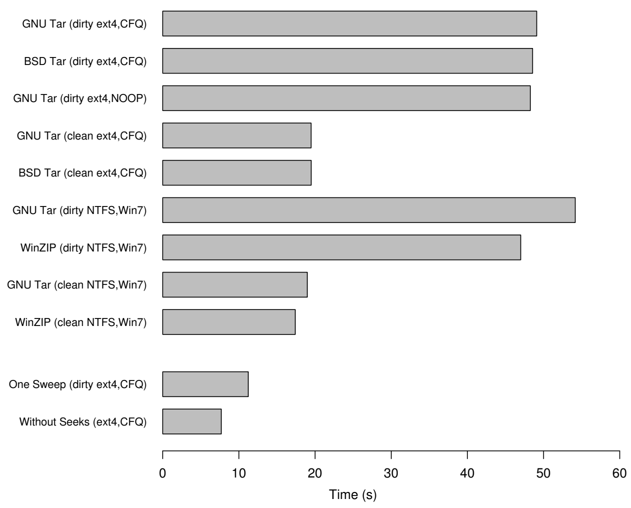


Figure 1. Performance of programs traversing directory trees showing potential improvement

In figure 1, we see the performance of a multi-file read operation performing `tar` on a Linux source tree containing about 22500 files stored on a 7200 RPM Maxtor DiamondMax 10 disk. We observe a high execution time regardless of application implementation (BSD vs. Gnu), scheduling policy (NOOP vs. CFQ), file system (ext4 vs.

NTFS) and operating system (Linux vs. Windows 7[1]). A new, clean file system reduce the execution time, but as the system age, the execution time increase due to a more fragmented file system (dirty vs. clean). Thus, this is a cross-platform and cross-file system problem. The last two bars in the figure also indicate that there is a large potential performance gain, i.e., if the order of the files (or disk blocks) can be sorted according to disk placement on an application-wide basis instead of on a per-file basis. This is shown by the ”one sweep” and ”without seeks” bars where the data is read in the file system's logical block number order or combined in one file without any inter-file seeks, respectively. In both cases, the total execution time is drastically reduced.

In this paper, we prove that this kind of high-level scheduling is possible in practice by first retrieving file location information and then performing the file I/O requests in logical block number order. Furthermore, we have modified the `tar` program under Linux to schedule the requests in this manner, and we show experimentally that the performance gains are substantial with a negligible CPU and memory overhead, e.g., the time to archive the Linux source tree is reduced from 82.5 seconds to 17.9 seconds - a reduction of about 78 % - on an aged ext4 file system running on a Seagate Barracuda 7200 RPM disk.

The rest of this paper is structured as follows: In section 2, we outline the basic idea of the user space scheduling and in section 3, we look at related work. Section 4 describes our implementation and a modified application example, and in section 5, we present our experiments and results. A brief discussion of our approach is given in section 6, and finally, section 7 summarizes and concludes the paper.

2 Why user space scheduling?

Traditionally, disk scheduling is performed in the kernel. The scheduler orders the requests after their logical block numbers[2] in the file system by taking into account current disk arm placement and move direction. Thus, based on the *concurrent, available* I/O requests in the queue, a schedule is made where a minimized disk arm movement and optimized throughput and response time are the goals.

However, to perform scheduling across multiple files that are read one by one, current kernel-level approaches fail, because the only requests available for sorting are from one file only. Consider the example in figure 2 starting to read and process file A, then file B and then finally file C. In this scenario, the individual blocks of each file might be efficiently scheduled, but the inter file seeks means moving the disk head from cylinder 2 (file A) to cylinder 0 (file B) and ending at cylinder 6 (file C). Obviously, it would be more efficient to start at cylinder 0, go to cylinder 2 and stop at cylinder 6. Furthermore, taking into account that high-end disks like Seagate Cheetah X15.6 have an average seek of 3.6 milliseconds, a track-to-track seek of 0.2 milliseconds and a full-stroke seek of 7 milliseconds, it is beyond doubt that any reduction in the number of cylinders traversed give great savings of disk access time. In order to make an efficient schedule based on location and thus reduce the seek overhead we need information about block placement for all files that need to be processed (which is not available to the kernel scheduler due to the one file at a time type of operation).

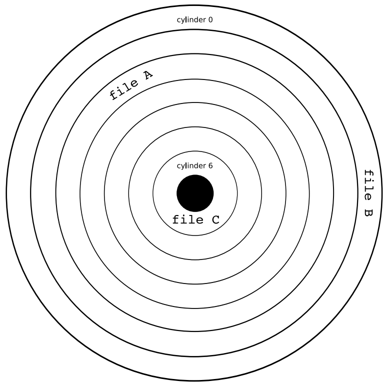


Figure 2. Example file placements on disk within a file tree

To address this scenario, we propose to implement a user space scheduler, recommending the application developers to utilize available information from the file system when making I/O requests. We base our approach on two sets of information, i.e., meta-data block number and block number (if available). Often, the order of the meta-data structures reflects the order of the disk blocks, and in file systems like the Linux ext4 and XFS, the ordering between inode blocks is the same as the order of inode numbers. Furthermore, in file systems like ext2/3 and XFS, the mapping of disk block extents is available to a superuser [4] using the `FIBMAP (FIle Block MAP) ioctl` system call. When using ext4, the new `FIEMAP (FIle Extent MAP) ioctl` [3] call provides such information to all users.

The basic idea is to use the available information about meta-data (e.g., inode) placement and disk block placement to make a cross-file schedule where we aim for a one-sweep disk arm movement for the whole recursive file tree traversal. In the `tar` scenario, the meta-data is read first in inode

[1]We have also tested on Windows Vista, but the performance is slightly lower than Windows 7, so the results are not included.

[2]The logical block numbers used by the operating system are assumed to correspond to the physical placement of the blocks on disk, but the disk often hides the true block layout. The physical placement and the logical block numbers however usually corresponds, but there may be inaccuracies.

order to recursively discover all files and directories, and second, all the file data is ordered by file data position.

3 Related Work

Similar ideas were discussed in an lkml-email exchange in 2003 about an ordered `readdir` operation and sorting in user space to minimize seeks [10]. Nearly four years later, Sun [8] filed a patent application which claims the method of accessing files in order of physical location when doing tree traversal. However, to the best of our knowledge, no real implementation nor measurements exist. We have therefore implemented such a scheduler in user space, and further improved the technique [7]. In this paper, we examine user space scheduling in detail and evaluate the performance by using `tar` as an example.

An alternative approach to user-space scheduling is to include support for optimized traversal in the file system itself. The new Btrfs [6] file system adds a secondary index which can be used to iterate over directory trees. The secondary index in Btrfs is currently very simple, which only sort entries according to creation time. However, this index could be extended to optimize full tree traversal. Our approach does not require change to the underlying file system, and can be used with any file system as long as the block address of the file data is exposed to the user space application.

4 Implementation

To evaluate our technique, we have chosen to adapt the archiving program `tar`. This is an old Unix utility that has been around for decades, so there are several implementations available. GNU Tar [2] is the default implementation on most Linux distributions and is widely used. Since GNU Tar is not developed as a two-part program, i.e., a tar library and a command line front end, we chose instead to adapt BSD Tar [1] as our reference implementation. The performance of GNU Tar and BSD Tar is very similar (as shown in figure 1). Our adaptation of BSD Tar, which does user space reordering, is referred to as qtar.

The tar program does not use any metadata information to read the files, directories, and metadata in any other order than returned by the system calls for doing post-order traversal of a directory tree. Pseudo-code for the GNU and BSD traversal strategy of tar is shown in listing 1. Qtar is our adaptation of tar which sorts the requests based on the physical locations to minimize hard drive seek time. This location is obtained using the `FIEMAP ioctl` on Linux, which supports files, but not directory entries. Qtar first traverses the directory tree (by C-SCAN order of the inodes) and adds all files therein to a queue sorted by block order.

```
def archive(path):
    for file in path:
        stat file
        read file
        add to archive

    for subdirectory in path:
        stat subdirectory
        add to archive
        archive(subdirectory)
```

Listing 1. Tar traversal algorithm

```
def archive(path):
    next = path
    do
        stat next
        if is_directory:
            add to archive(next)
            for file in next:
                inode_cscan_queue.add(next)
        else:
            FIEMAP next
            block_sort_queue.add(next)
    while (next = inode_cscan_queue.next())

    flush()

def flush():
    for file in block_sort_queue:
        add to archive
```

Listing 2. Qtar traversal algorithm

The last operation is to process the files by the order of the sorted queue. Pseudo-code for the qtar algorithm is shown in listing 2.

Finally, a memory overhead of a couple of hundred bytes for each file is required, i.e., storing metadata such as file name, parent directory, inode and block number. However, if an upper bound of consumed memory is required, the `flush()` method (see listing 2) can be called at any time, e.g., for every directory, for every 1000 files or when n MiB RAM have been used.

This method of reading metadata in C-SCAN order and sorting the full directory tree before accessing the files, has not been described in previous work known to us.

5 Experiments and Results

We have conducted experiments to evaluate different aspects of this kind of user space scheduling and the qtar technique in particular. The test setup is a machine running GNU/Linux using ext4 and XFS on a workstation-class Seagate Barracuda 7200.11 hard drive. In our tests, we did not use any limit on memory, so around 6 MiB of memory were used for the largest directory tree (22 500 directory entries).

5.1 Running time and head movement

As shown in figure 3, the running time can be reduced using application level sorting. By running qtar on the Linux kernel source tree consisting of about 22 500 files, we get an average runtime of 17.9 seconds for five runs on ext4 (11.6 seconds on the setup in figure 1). The corresponding time for GNU tar is 82.5 seconds (48.3 seconds on the setup in figure 1) giving a performance increase of a factor of more than 4.5. Furthermore, the elapsed running time of qtar is fairly close to the time required for reading the data using the theoretical *one sweep* benchmarks where we assume that all files and directories immediately can be sorted according to block numbers. However, *one sweep* is somewhat faster than practically possible, because we have several data dependencies (directory data blocks must be read to find new files and subdirectories, and inodes must be read to find file data).

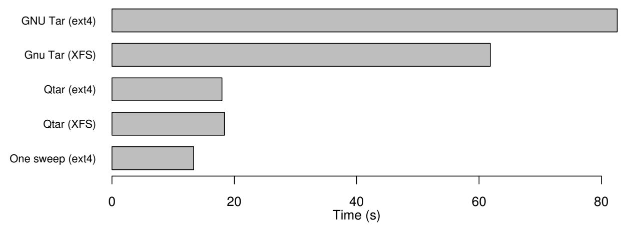
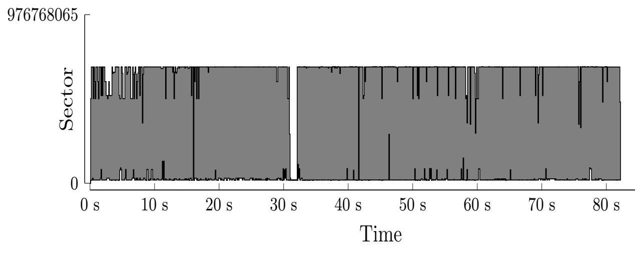
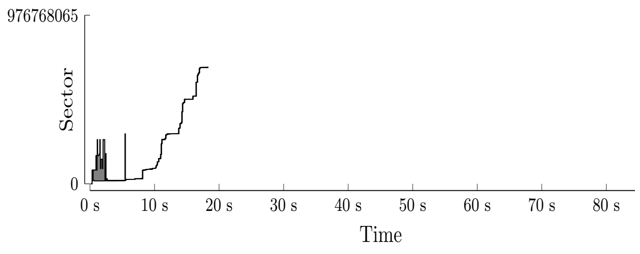


Figure 3. Time to `tar` the Linux source tree (22500 files)

In figure 4, we visualize the runtime and seek footprint of GNU Tar and Qtar for the same experiments, by showing the maximum and minimum disk sector accessed for every 100 millisecond intervals. The data for the figure was recorded using `blktrace`. We could not include the full seek pattern, because there were too many seeks for the GNU Tar implementation. Therefore, the figure shows only the upper and lower bound of the disk head position, even though the finer details are lost. As we can see, the traditional solution requires multiple sweeps over the disk platters as a result of multiple SCAN operations performed by the kernel-level I/O scheduler. However, the footprint of the improved qtar solution is radically different from the original solution. Almost all seeks are going in the same (C-SCAN) direction, except for some directory blocks which need to be read before new inodes are discovered. Our implementation orders directories by inode number, which is one of the reasons for the seeking done during the directory traversal the first few seconds. After the directory tree has been traversed, there do not seem to be any unnecessary seeks at all. This means that the file system allocator did not have to resort to any file fragmentation in order to fit the file on the file system. Ext4 has implemented delayed allocation, so the full size of the file is known before the physical



(a) GNU tar (ext4)



(b) Qtar (ext4)

Figure 4. Disk head movement every 100 ms

blocks of the file are allocated. A video showing the disk head movement on an actual disk using GNU tar and qtar can be downloaded from [3].

5.2 Partial Sorting

The implementation we have tested so far read the full directory tree and ensured that both metadata (inodes) and file data were read according to disk placement. In figure 5, we show the performance of an experiment reading a directory tree on ext4 with 1000 files using various alternative implementations of tar with only partial sorting:

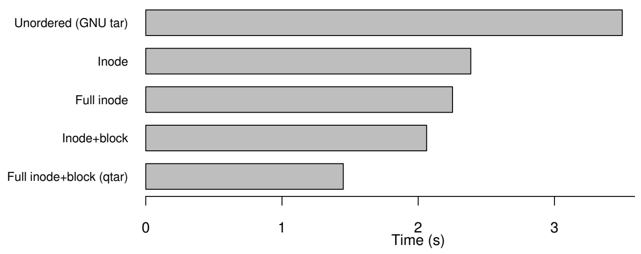


Figure 5. Sorting alternatives compared

- **Unordered** (GNU Tar): This implementation read all the metadata for all files in a directory in the order in which they were retrieved from the file system. Then, it read the file data in the same order, before recursively repeating the procedure for each subdirectory.

[3] `http://www.ping.uio.no/~gus/misc/diskhead-low.avi`

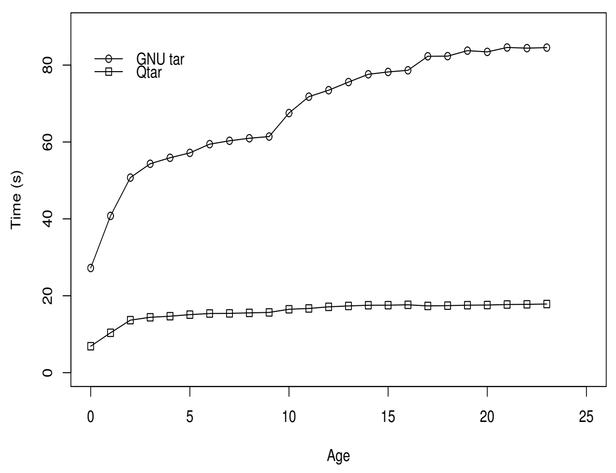


Figure 6. Performance as file system ages

- **Inode sort**: This implementation sorted the directory entries by inode before reading the metadata. It also read the file data in the same order, i.e., sorted by inode.
- **Full inode sort**: This implementation traversed the full directory tree before reading all the metadata ordered by inode, and then it read all the file data ordered by inode.
- **Inode and block sort**: This implementation read all the metadata for a directory ordered by inode. Then, it read all the file data for the file in the directory, ordered by the position of the first block of each file. This procedure was repeated for each subdirectory.
- **Full inode and block sort** (Qtar): This implementation traversed the full directory tree and read all the metadata ordered by inode. Then, it read all the file data ordered by the first block of each file.

The results clearly show that doing the full inode and block sort gives the best performance, but also that even the simplest change, ordering directory entries by inode, helps a lot. This shows that there is a high correlation between the inode number and the position of the file data on disk on the ext4 file system, i.e., information that could be used to implement applications doing directory tree traversal more efficient. We do not know if such an assumption can be made on other file systems than ext.

5.3 Effect of File system Aging

An interesting detail is how the performance changes as a file system ages. To evaluate this, we ran our qtar and GNU Tar five times throughout an aging procedure which repeatedly checked out 24 different versions of the Linux source code. In short, our aging procedure replays development activity on the kernel source tree by incrementally checking out new versions. The archiving processes were again run on the full Linux kernel source, and we used the ext4 file system. In figure 6, we show the average runtime for the five runs. The amount of change for each step in the aging procedure is different, which is why step 1, 2, 10 and 17 are especially steep for GNU Tar. It is interesting to see in this graph that the age has much less impact on the improved qtar implementation. As the file system ages, the improvement factor increases from 3.95 to 4.75.

5.4 Other File Systems

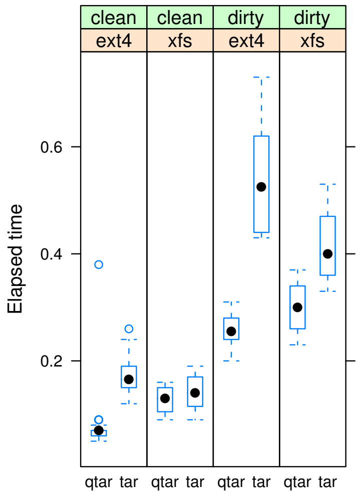
To evaluate how the underlying file system impacts tar and qtar, we compared the performance running on ext4 and XFS. The tests were run 5 times, and the average runtime was used. Each test was measured for 16 different directories which were created in the same way, but at different times. We also ran the tests on 16 aged (dirty) directories.

Detailed boxplots with absolute timings are shown in figure 7, and a summary of the results in terms of relative performance gain (GNU tar / qtar) can be seen in figure 8. We can see that the improvement obtained with qtar is larger on ext4 than on XFS. Still, qtar outperforms tar in all cases, even on clean file systems.

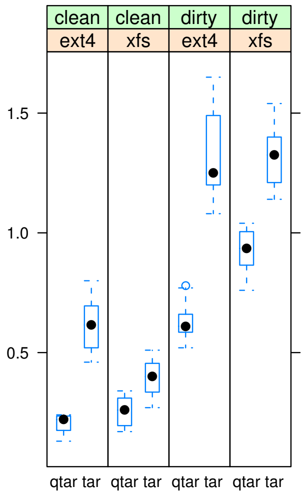
The difference in improvement factor for the file systems relates to the order the file systems returns files in when reading directories. Note that the variance for the boxplots mainly shows variance in the file system allocator, not variance in runtime with the same directory. Thus, in summary, this experiment shows that qtar improves the performance over GNU tar for different file systems regardless of the size of the file tree and the fragmentation of the file system. Using the smallest file tree of 100 files, the improvement is ”only” about 10% on a clean XFS file system. However, for the benchmarks we performed, the improvements increase with the file system age and file tree size, i.e., giving a performance increase of about 200% using an aged XFS on the Linux source tree (22500 files). The improvement factor on ext4 for the clean filesystem approaches that of the dirty filesystem for the run with 22,500 files. Why we see this slight decrease of improvement in figure 8 is unknown to us, but it still runs at about four times the speed of GNU tar for both clean and dirty ext4.

6 Discussion

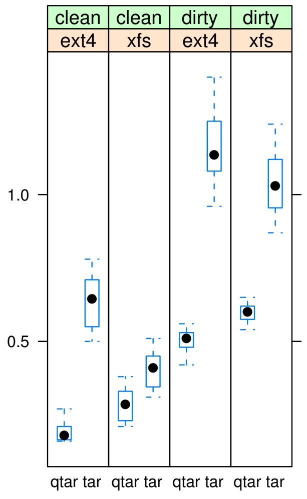
The performance gain of the user space scheduling approach, i.e., first retrieving meta-information from the file system about file placement on disk and then making logical-block-number-sorted I/O requests, is promising.



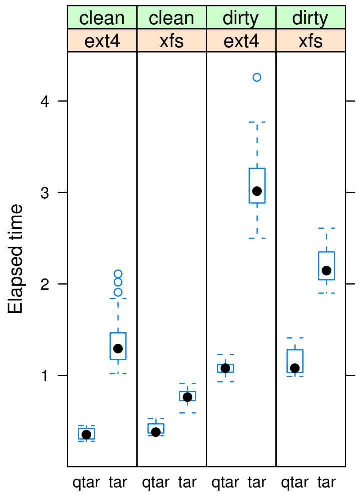
(a) 100 files



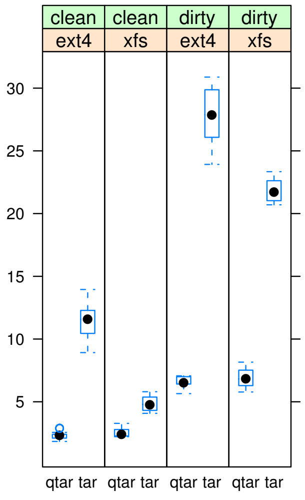
(b) 250 files



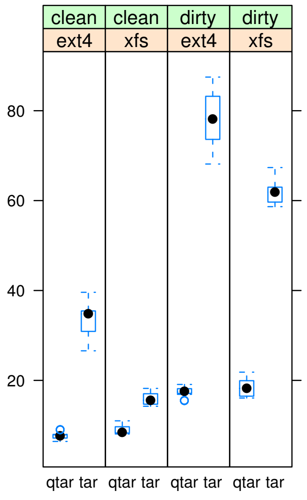
(c) 500 files



(d) 1000 files



(e) 5000 files



(f) 22500 files

Figure 7. Boxplots for qtar and tar performance on the ext4 and XFS file systems

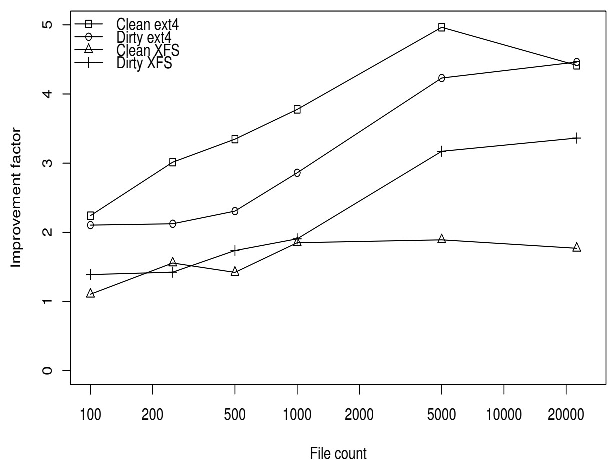


Figure 8. Improvement factor for qtar over GNU tar comparing results on ext4 and XFS

However, there are a number of issues that must be considered.

6.1 Applicability

To increase the application performance by reordering requests, it obviously must have multiple I/O operations to choose from. The potential improvement in execution time will generally depend on the number of I/O operations that can be reordered. In this respect, there are many examples where this kind of user space scheduling is useful, i.e., where file and directory traversal is the core functionality. However, there are also examples where the added complexity does not pay off. This includes applications that depend on external input, such as interactive applications and databases; there is no way to know what data to read until a user requests it. There is also a limitation for applications with data dependencies; when one I/O operation decides which operation to do next, or when one write operation must be written to disk before another for consistency reasons. As an example of the former case, consider a binary search in a database index; the database only knows about one operation, *read the middle block*, while the next operation, *read the left branch* or *read the right branch*, depends on the data found by the first operation in the middle block. Another examples of dependencies when we need to read a directory block to find a file's inode number, and to read the file inode before we can read the file data. Finally, this inter-file-seek optimizing is most efficient with many small files. For large files, there is usually not enough external fragmentation in modern Linux file systems to warrant reordering of requests, neither within a file nor between files. In these cases, we will likely introduce intrusive changes in the program flow, and yield relatively low performance gains. Another consideration is whether it is likely that files are cached, i.e., if they have been recently used. A compiler such as `gcc`, or rather the preprocessor `cpp`, may seem like a good candidate for this technique because it reads many small header files (around 20 for many standard C headers in GNU libc, almost 100 for GTK+). But in most work flows, header files will be cached, and therefore, the average gain over time will be low.

6.2 Implementation alternatives

The traditional approach is to let the kernel I/O scheduler do the scheduling. Our approach is application-specific, and is implemented in user space since only the application knows (and can adapt) its disk access pattern. The main argument against doing scheduling in user space is that it adds complexity. Nevertheless, a solution where the kernel does the sorting could be implemented in several ways. For example, an application can ensure that the I/O scheduler has multiple files to choose from by using the existing asynchronous APIs or by using multiple threads each issuing I/O requests. There are problems with this solution, which leads us to believe that such a solution would be slower and more intrusive:

- **Complexity**: The application must handle multiple buffers and multiple pending asynchronous requests, which is more complex than sorting a list of files.
- **Memory overhead**: The performance depends on the depth of I/O queue. For efficiency, we should allocate buffers for all the file data we want to have pending. If we use smaller buffers, we may end up with more seeks due to inter-file request interleaving. For example, if we want a queue depth of 500 and the average file size is 20 KiB, the kernel approach needs 4 MiB RAM. Our approach uses 200 bytes of memory per file giving the equivalent of a queue depth of around 20.000.
- **Scheduler deadlines**: The I/O scheduler in the kernel assigns deadlines to prevent starvation. However, in our target scenarios, the total finishing time is the important metric, and individual request can therefore be delayed longer than the I/O scheduler deadlines.
- **Metadata API**: Another performance issue is the lack of efficient asynchronous APIs for metadata operations. There are several projects implementing such APIs, such as fibrils in Linux. However, our experiments revealed that the kernel serializes operations on files in the same directory using the `i_mutex` lock, i.e., the I/O requests will still arrive one at a time.

The alternative approach of modifying the file system to return files in `readdir` by sorted order also have limitations.

The first being decrease in performance for software which will not traverse the tree, e.g., programs only interested in the filenames. Looking up the block address of every file in a directory may add a large performance penalty. A solution could be to add a new parameter to the `open` system call to indicate that the program will do tree traversal. The problem with this approach is that the sorting is limited to one directory at time, and as shown in figure 5, this decreases the performance considerably compared to a full inode and block sort. Another approach is to implement a special kernel API for this operation, but we would need to move user space logic into the kernel. For example, a file manager which generates previews of all files in a directory would only be interested in the embedded thumbnails for JPEG images (a small part of the file), and the first page of a multi-page document. Since this behavior is application-specific, the best place to put this logic is in user-space.

Using our technique for directory traversal is somewhat more complex than traditional blocking I/O found in programs like tar. To avoid code duplication and allow system wide disabling of the technique, it should be implemented as a library instead of in the application itself. The technique can then be optionally enabled at runtime and adapted to the underlying file system. For partial sorting (see inode and inode+block in section 5.2), the `readdir` call can simply be wrapped to return files in traversal order. However, using full inode+block sort (used in qtar) requires a new interface that allows traversing files in an order not limited by its containing directory.

6.3 Non-rotational devices

The technique is applicable for all types of hard drives that experience seek times which increase relative to the distance between the previous block read and the next. However, we have only tested in on storage devices using rotational disks. If other types storage devices are used, like solid state disks, there is no disk arm movement to optimize. Since version 2.6.29, Linux exposes if a block device is rotational or not through the `sysfs` interface which could be used to determine if user space scheduling should be turned on or off.

6.4 Portability

The block address is not always available to user space applications without root privileges. For ext4 and XFS, this is exposed on Linux for normal users through `FIEMAP ioctl`, but currently requires root access for ext3 using `FIBMAP ioctl`. On Windows, the block address of files can be found by sending the `FSCTL_GET_RETRIEVAL_POINTERS` command using the `NtFsControlFile` function, which is part of the defragmentation interface [9].

6.5 Physical vs. logical block numbers

Disk scheduling in general requires knowledge of where the data is physically located on disk. Commodity hard drives do not expose this information directly, but instead present logical block numbers. There is no guarantee of a direct mapping between the block numbers used by the operating system and the actual physical placement on the disk, but usually the disk layout resembles the logical disk block numbers. Thus, differences in the assumed correct mapping will influence all disk I/O schedulers.

7 Conclusion

Traversing directory trees and reading all files one by one is a common operation in many applications. Traditional, in-kernel scheduling fail to optimize inter-file seek operations, i.e., this is a problem that cross different schedulers, file systems and operating systems. We have therefore proposed to take all file placements on disk into account using a user space scheduler where I/O request ordering can be adapted to the application's requirements. Finally, our experiments demonstrate that this can give huge performance improvements.

References

[1] libarchive. http://code.google.com/p/libarchive/.

[2] Tar - gnu project. http://www.gnu.org/software/tar/.

[3] A. Cox. Why is fibmap ioctl root only? Linux Kernel Mailing List, 2007. http://lkml.org/lkml/2007/11/22/97.

[4] M. Fasheh. Fiemap, an extent mapping ioctl, 2008. http://lwn.net/Articles/297696/.

[5] P. Halvorsen, C. Griwodz, V. Goebel, K. Lund, T. Plagemann, and J. Walpole. Storage system support for continuous-media applications, part 1: Requirements and single-disk issues. *IEEE Distributed Systems Online*, 5(1), 2004.

[6] J. Kára. Ext4, btrfs, and the others. In *UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference*, pages 99–111, Oct. 2009.

[7] C. H. Lunde, H. Espeland, H. Stensland, A. Petlund, and P. Halvorsen. Improving disk i/o performance on linux. In *UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference*, pages 61–70, Oct. 2009.

[8] O. Manczak and E. Kustarz. Speeding up traversal of a file system tree. *US Patent Application*, (US 20080172387), 2008.

[9] M. Russinovich. Inside windows nt disk defragmenting. *Microsoft TechNet*, 2007. http://technet.microsoft.com/en-us/sysinternals/bb897427.aspx.

[10] T. Ts'o. Re: [bug 417] new: htree much slower than regular ext3. Linux Kernel Mailing List, Mar. 2003. http://lkml.org/lkml/2003/3/7/348.