Investigating ext2 optimizations in Linux

Markus Peloquin
markus@cs.wisc.edu

2009-09-30

Abstract

The Linux implementation of the ext2 filesystem contains a number of optimizations to hide artifacts of the implementation. Read-ahead (buffering data yet to be requested) completely hides any noticeable timing effects of both the block size and the way in which blocks are specified by the filesystem. Caching allows for some set of blocks to be read repeatedly without having to return to the disk. The only parameter initially discoverable is the size of the read-ahead. Only by disabling this, one can further observe the block size, the layout of blocks for a file, or the size of the file cache.

1 Introduction

In more primitive operating systems, block size was something you had to work around. The Unix function `stat` returns a value `st_blksize` as a hint to programmers so sequential reads could be better optimized. The standard C library's I/O routines gave programmers a cleaner interface that buffered I/O automatically, but this was done by the library. With modern Unix operating systems, even the 'unbuffered' `read` system call gets buffered, but now in the kernel by something called *read-ahead* or *prefetching*.

Read-ahead is accomplished in the kernel by guessing which blocks will be read next, and then buffering them in the background. There are many different access patterns, and the programmer is allowed to select one with the `posix_fadvise` system call. Usually, it is expected that reads will be more or less sequential.

Another optimization is file caching. If a file or set of files were used recently, they may be stored in memory for subsequent use. This is useful in such cases as repeatedly listing a directory's contents, or a file that is often *sourced* by various shell scripts. Another case it can be useful is when a file gets rewound.

2 Method

To understand how read-ahead or caching work, we must understand the workings of operations. As such, the block size is important to learn. The filesystem's block size is not the only factor affecting performance, but also the layout of block addresses. Strategies for revealing the parameters of the filesystem and their optimizations are given in the following sections.

2.1 Block size

The filesystem reads data from the disk a block at a time. If only a single byte is needed, the whole block that it is contained in must be buffered. A trivial optimization then is to keep that block in memory, since it costs little to keep it and would cost a lot to regain it. Subsequent reads to the first block should then not require any disk activity. To derive the size of a block, we can just keep reading from what is presumably the same block in small increments. These should all be very fast. Once the time taken to read a small increment takes a long time, then it must be because the next block is being read.

The above strategy would work, but it presumes there are no optimizations in effect. Read-ahead would cause the filesystem to start reading the subsequent blocks, and so we would not find a small segment of data that took a long time at all. The Linux kernel allows for this through a general filesystem `ioctl` ('I/O control') called `BLKRASET`. Setting this to zero deactivates read-ahead.

We still have caching to consider. An acceptable strategy would be to read from some large separate file. This would fill the buffer up with useless data, so reading from the file that we *do* care about would require a disk read. This was not the approach taken, as Linux allows for a simpler way to achieve this result.

All that is needed to clear the cache is a simple `echo 1 > /proc/sys/vm/drop_caches`, which allows the tests to run quicker.

Another approach that might reveal the block size is to attempt to read a large file with different assumed block sizes (e.g. 512, 1024, 2048). Theoretically, this should perform the fastest when the incremental read size is a multiple of the real block size. In practice it reveals little information.

2.2 Read-ahead

When data is read from the disk, the read-ahead algorithm will read more than it needs asynchronously. Even when the reading program reads past the first block, it will not need to wait for subsequent blocks. The strategy is then similar to detection of block sizes: read in increments, and find the points that it takes longer than usual. The incremental read size should be larger, since presumably the read-ahead size is a multiple of the block size, which we now should know. Further, the more data read at a time, the faster we can get to the end of the read-ahead buffer, hopefully leading to a wait at the end of each read-ahead buffer. Again, the caches should be cleared before running the test.

2.3 Caching

A simple model for caching is to keep the last C blocks in memory in a FIFO scheme. If a block is read while still in the cache, then it is moved to the front of the queue if necessary, and no block is removed. The strategy is to read C blocks into the queue so that the next block takes longer. Since we do not know C, the strategy is to read the first block, then the first two blocks, then the first three blocks, etc. As long as we are only reading the first C blocks of a file, the reads to the first block should be relatively constant, since only the memory would need to be consulted. Once the time to read the first block begins to increase, we know that the buffer has been filled.

As usual, the caches need to be cleared, and read-ahead disabled. Read-ahead must be disabled since we want to read the blocks in a very specific order, and read-ahead would violate this.

2.4 Block layout

Like the Unix filesystem (UFS or FFS), ext2/3 stores block pointers in the inode structures (ext4 does this in addition to a more optimal method called *extents*). First, there is some number of direct pointers, followed by an indirect pointer, a double-indirect pointer, and a triple-indirect pointer. What becomes important is that upon reading an inode structure, the filesystem immediately knows where to find the first D blocks of data. To read block $D+1$, it must first read the indirect block from the disk, and so requires an extra disk access. Theoretically, reading block $D+1$ should take twice as long as it did to read block D.

As usual, any caches should be cleared. Read-ahead must be disabled again, otherwise the extra delay in finding block $D+1$ will not be noticeable.

3 Results

It should be noted that these tests were run on a 32-bit machine. While this does not change the results of the filesystem layout, it may affect the values relevant to the optimizations (caching and read-ahead). The x86 `rdtsc` instruction was used to measure time, and CPU frequency scaling disabled.

Each test was run 16 times. For all but the caching test, the read rates at each point in the file were averaged between the tests. The read rate itself was computed either as the centered difference approximation or a one-sided approximation, depending on what sample points exist. Let h be the separation between samples. The approximations are

$$f'(x) \approx \frac{1}{2h}(f(x+h) - f(x-h))$$
$$f'(x) \approx \frac{1}{2h}(-3f(x) + 4f(x+h) - f(x+2h)).$$

This was used to give the rate in terms of ticks per byte. Taking the inverse gives the desired units bytes per tick. These are the rates that are averaged, giving the average read rate at a point in the file.

3.1 Block size

Segments of 16 bytes were read in at a time ($h = 16$). The size does not particularly matter, as it must only be a divisor of any possible block sizes. There is a clear pattern in Figure 1, showing read drops at regular intervals. When reading data in from a cached block, the execution time is mostly due to the system call. At intervals of 4096 bytes, the read operation takes a much longer time, and so it is concluded for the rest of the experiments that the block size is 4 kB.

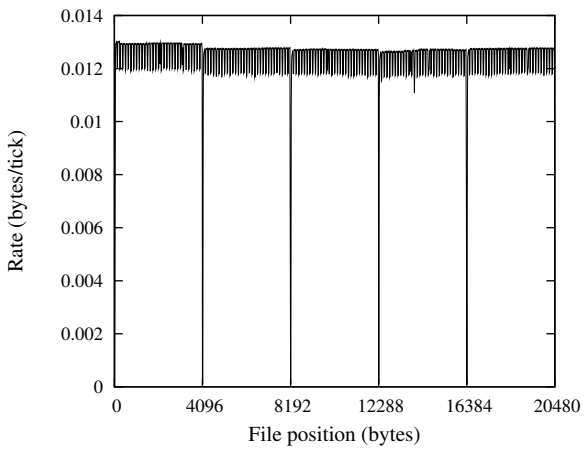

Figure 1: The read rate at each position in the file when read 16 bytes at a time. The rate drops are at 4 kB intervals, taken to mean one block.

3.2 Read-ahead

Segments of 512 bytes were read at a time ($h = 512$). The size should not matter so much, except that the results were most clear with this value. Figure 2 shows the read rate at different points of the file. There are four evenly spaced points at which a large decrease is noticeable, and the separation between consecutive points is exactly 126976 bytes, 248 sectors, 124 kB, or 31 blocks.

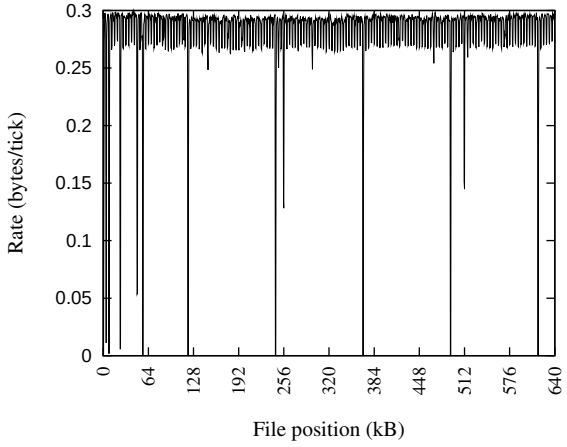

Figure 2: The rates of consecutive reads at 512 byte increments. Large drops at 122880, 249856, 376832, ... bytes are at intervals of 126976 bytes or 248 sectors, the read-ahead size.

That is not the only observable pattern in the figure. At exactly 256 kB and 512 kB are two less-significant rate drops. The reason is unknown, but the more significant drops were used as the basis of the read-ahead size.

Another thing of interest is all the drops near the beginning. There are multiple possible reasons for this. It may be that read-ahead works differently at the beginning. OpenBSD's FFS implementation does a 'clustered' read of the first few blocks at the beginning of the file, and follows a different algorithm afterwards.

3.3 Caching

The file was read in one block at a time in the order 0, 0–1, 0–2, 0–3, ... The time reported in Figures 3 and 4 is the time required to read the first block. This graph is less clear, but the increase in time seems to begin at block 175 (700 kB), where the time shifts upward slightly. After almost a megabyte, the cache effects begin to wear off, so it seems likely that the cache size is less than a megabyte.

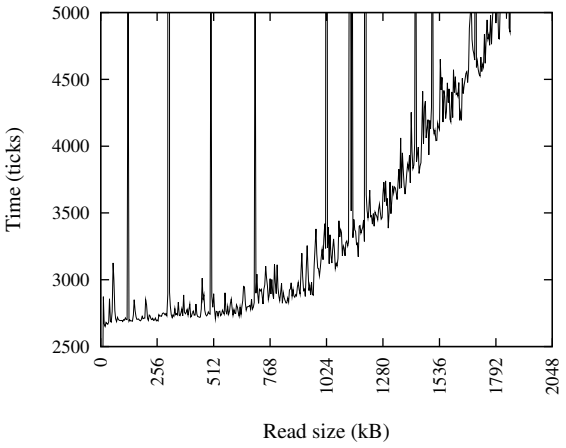

Figure 3: The time to read the first block with respect to how much of the file is already in the cache. Beginning around block 175 (716800 bytes, 700 kB), there is a slight increase in the time taken, and another increase at around 241 blocks (987136 bytes, 964 kB).

One potential reason for the increase in slope is the hard disk's own 8 MB cache. Another potential reason is that the disk must seek to read data no longer in its cache, but this would mean the graph should be linearly increasing.

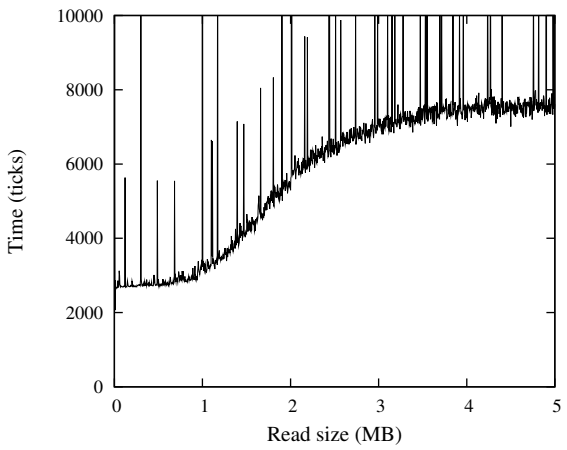

Figure 4: A wider version of Figure 3. The graph continues linearly after 4 MB.

3.4 Block layout

A file was read in at half block increments ($h = 2048$). The tests were not as revealing with any increments as large as the block size (partially a consequence of the rate approximation used). Like the test for read-ahead, Figure 5 has rates computed as a centered difference approximation. The main dip is at 49152 bytes or 12 blocks.

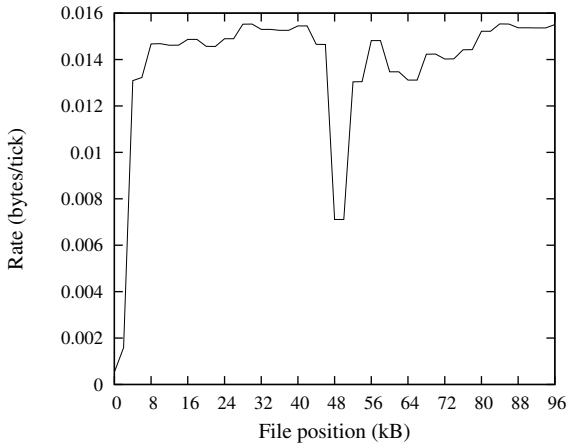

Figure 5: The rate of each consecutive read at half block increments. The first main dip is at 49152 bytes or 12 blocks.

The graph is near zero for the first 2 kB because of limitations of the rate estimation. The second observable dip occurring at around block 15 was consistently present, possibly the result of some filesystem optimization.

4 Conclusions

The block size reported in §3.1 is correct. Upon inspecting the superblock of the filesystem, the block size is reported as, indeed, 4 kB.

In a similar way to deactivating the read-ahead, the default read-ahead can be obtained (before setting it to zero) by calling `ioctl` on the filesystem's device with the request `BLKRAGET`. Before it was reset to zero, the value returned is 248: the exact result obtained in §3.2.

Definitively determining the cache size of the operating system (§3.3) or the filesystem was not as straightforward. Unlike read-ahead, its state can never be modified, except to clear what is already in it.

The number of direct block pointers inside an ext2/3 filesystem is no secret. It has, and probably always had, room for 12 direct block pointers, confirming the result observed in §3.4.