File System Forensics : Measuring Parameters of the ext4 File System

Madhu Ramanathan
Department of Computer Sciences, UW Madison
madhurm@cs.wisc.edu

Venkatesh Karthik Srinivasan
Department of Computer Sciences, UW Madison
venk@cs.wisc.edu

Abstract

Operating systems are rather complex software systems. The File System component of Operating Systems is defined by a set of parameters that impact both the correct functioning as well as the performance of the File System. In order to completely understand and modify the behavior of the File System, correct measurement of those parameters and a thorough analysis of the results is mandatory. In this project, we measure the various key parameters and a few interesting properties of the Fourth Extended File System (ext4). The ext4 has become the de facto File System of Linux kernels 2.6.28 and above and has become the default file system of several Linux Distros. We measure and report:

- The size of the basic allocation and transfer unit, otherwise called as the block size
- The prefetch size used by the file system
- The size of the buffer cache
- The number of in-inode extents

1. Introduction

1.1 A primer on the ext4 File System

Ext4 [2] was released as a functionally complete and stable file system in Linux 2.6.28 and has been included in every Linux distro that uses kernels 2.6.28 and above. It has been used as the default file system in several Linux distros. ext4 is a Journalling File System like its ext3 counterpart but supports volume sizes upto Exa Byte (1 EB) and file sizes upto 16 TB. It is the first file system in the ext family to depart from the traditional mechanism of using indirect blocks to index data blocks. When indirect blocks are used, accessing data blocks need to pass through an extra level of indirection. Since mapping information is maintained for every block, regardless of the blocks' being physically contiguous with allocated adjacent blocks, block maps tend to be huge and the extra indirection becomes a huge overhead. Further, files tend to become extremely fragmented on disk, the gravity of which is determined by the block allocator's intelligence. For these very reasons, 'extents' are used in ext4.

An extent is a group of physically contiguous blocks. Allocating extents instead of indirect blocks reduces the size of the block map, thus, aiding the quick retrieval of logical disk block numbers and also minimizes external fragmentation. An extent is represented in an inode by 96 bits with 48 bits to represent the physical block number and 15 bits to represent length. This allows one extent to have a length of 2^{15} blocks. An inode can have at most 4 extents. If the file is fragmented, every extent typically has less than 2^{15} blocks. If the file needs more than four extents, either due to fragmentation or due to growth, an extent HTree rooted at the inode is created. A HTree is just like a B-Tree with high fanout. The extents occur at the leaf nodes and they point to contiguous disk blocks. Figure 1 shows the structure of an inode with the extent tree.

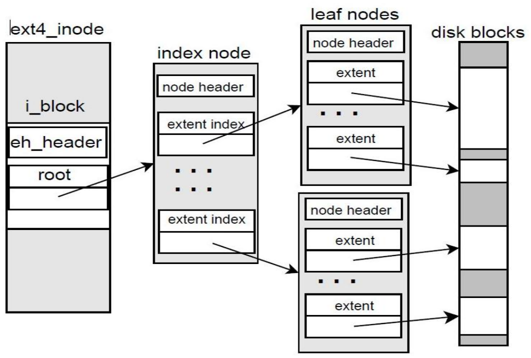

Figure 1. Structure of the ext4 inode with the Extent HTree

1.2 Measured Parameters

A very basic parameter of any file system is the size of the basic allocation and storage unit, also called as the block size. Our experiments first determine the block size of the ext4 file system. This parameter is used by the rest of the experiments. The blocksize is tricky to determine if the file system prefetches data on a buffer cache miss. ext4, being a complex and intelligent file system, is no exception to this fact. We had to design an algorithm that triggers no form of prefetching, but, at the same time, measures the block size accurately.

The next parameter that we measure is the prefetch size of ext4. Whenever there is a buffer cache miss, instead of just servicing the block miss, it would be advantageous to fetch the next few contiguous disk blocks too. This would prove fruitful if the file were to be accessed in a sequential manner. We designed a simple experiment to measure the number of blocks prefetched by ext4.

Table 1. Experimental Infrastructure

Processor	Intel Core i7
Core Clock Freq.	2.66 Ghz
No. of Cores	2
LLC Size	4.0 MB
Memory Bandwidth	17.1 Gb/s
Memory Size	4.0 GB
Hard Disk Capacity	500GB
Hard Disk RPM	7200
Linux Ditro	Ubuntu 10.10
Kernel Version	Linux 2.6.35-22-generic

We then attempt to measure the buffer cache size of the file system. Reading from a disk can be very slow when compared to accessing the main memory. In addition, it is common to read the same part of a disk several times during relatively short periods of time. By reading the information from disk only once and then keeping it in memory until no longer needed, one can speed up all but the first read. For this purpose, disk buffering is done and the pages once read are stored in the buffer cache, which occupies a significant part of the main memory.

Unlike ext2, the ext4 file system does not have the concept of direct pointers and indirect pointers to store the data block addresses. This is because, if the files are large, then it would require a large amount of disk blocks to store the metadata itself. Ext4 overcomes this disadvantage by storing data in contiguous blocks of memory. The metadata about each contiguous block of memory up to a maximum size of 128MB is stored in an extent. Each inode supports the storage of 4 such extents in the inode itself. If the number of extents is greater than 4, then they are stored in a HTree whose root is stored in the inode. We attempt to verify the number of extents stored in the inode directly.

2. Experimental Setup

We carried out our experiments in a Lenovo Thinkpad T510 Laptop running Ubuntu 10.10 distro, built over a 2.6.35-22-generic Linux kernel. The T510 had an Intel Core i7 processor. The processor is a dual core processor and so, we affinitized our programs to run on a single core with high priority. Our system had a 4 GB RAM. We used a 500 GB ATA Hitatchi 7200 RPM disk for our measurements. The detailed hardware infrastructure used for our experimenst is listed in Table 1. Our programs were coded in C.

3. Design and Implementation

Taking measurements for a software system as complex as the ext4 file system requires carefully formulated methodologies and extreme care so as to eliminate the effects of noise. ext4 is a pretty complex file system and noise can be introduced by a variety of factors. We had to take several things into consideration to get accurate measurements.

- **Selection of Timer:** The selection of the timer used for measurements is very critical for the project. Since memory references require delays in the order of microseconds to complete, we wanted to use a high resolution timer that gave an accuracy finer than microseconds. The *rdtsc* is an assembly instruction in the x86 ISA that reads a 64-bit register dedicated to storing the number of clock cycles elapsed since the register was reset. Since hardware counters provide the highest accuracy possible in terms of CPU cycles, we decided to use the *rdtsc* instruction. However, our processor had two cores, with two separate registers and, depending on which core the process is currently running on, this might cause errors while reading the register. To ensure that this does not happen, we affinitized the process to run in a single CPU using the sched_set_affinity() system call wrapper provided by the Linix kernel API. Further, we raised the priority of the process so that no other process is scheduled to that CPU. We multiply the resulting CPU cycles obtained, by the CPU frequency to obtain the time measurements.

- **Flushing and Clearing the Buffer Cache:** Several times, during our experiments, we had to clear the buffer cache or flush the buffer cache to disk or both. For flushing the cache, we made use of the fsync() system call. For clearing the buffer cache, we made use of the "drop_caches" utility, found in Linux kernels 2.6.16 and above.

- **Choosing a representative measurement:** Once we took a series of measurements of a specific parameter, we had to use a value that best represents the set of measurements taken. For this, we used the arithmetic mean of a filtered subset of measurements. While measuring the time for file system operations, we need to take into account that there is a mechanical device involved, the hard disk. The time that we report encompasses the necessary head movement. To ensure that the head movement is stable, we repeated the experiment many times and disregarded the first 3 or 4 measurements. First few measurements might include the time taken by the head to move from its current poistion and this might cause unnecessary noise in the measurements. This was how we arrived at the filtered subset of measurements. Further, we did not print the measurements in a file and we printed them to the console to prevent buffer cache pollution or worse, stray head movements.

- **Measurement validation:** Since ext4 uses extents, which are physically contiguous blocks of arbitary sizes, we had to find some way of veryfying the results we obtained, pertaining to extent boundaries. For this, we used the *debugfs* tool, an interactive file system debugger in Ubuntu that can be used to examine the state of the ext4 file system. We used it to manually uncover the extent map of our files and to cross-verify our measurements.

3.1 Design of individual experiments:

Now, we present the mechanism we used to determine the various parameters of the ext4 file system. In each subsection, we briefly explain the methodology used to measure the specific parameter and a short listing of the algorithm used.

3.1.1 Block Size

The size of the block is the fundamental unit in which transactions with the file system are done. Only on determining the block size can the remaining experiments be performed. We wanted to come up with a foolproof method so that no form of prefetching kicks in, and thus, we had to avoid reading the file sequentially in any direction. A random read would not help much in this scenario since the block size is unknown. Hence, we came up with an algorithm to estimate the block size. The buffer cache is cleared before every run of the experiment. Every run of the experiment assumes a block size. During each run, a rifle shot read is done to the byte located at the position of an arbitary prime number multiplied by the assumed block size. The next read is directed to the previous byte. If both reads take time in order of milliseconds, the assumed block size is the actual block size. The rifle shot reads serve the purpose of random reads and the previous byte is read next to counter initial sequential prefetching. A prime number is chosen to prevent reporting false block boundaries. For example, if 4 were chosen as the number, and if the assumed block size were 1024 but the actual block size were 4096, $4 * 1024^{th}$ byte and the previous byte would go to different blocks and result in a false positive.

```
Pseudocode:
Affinitize process to a CPU;
Make priority of this process the highest;
Clear buffer cache;
Choose a random prime number P;
Assume a block size X;
Repeat {
Read the $(P * X)^{th}$ byte;
Let R1 = Time required by the read;
Read the $(P * X) - 1^{th}$ byte;
Let R2 = Time required by the read;
if ( R1 $\approx$ R2 )
Output X as actual block size;
else
Increment X by a suitable increment;
Clear buffer cache;
}
```

3.1.2 Prefetch Size

ext4 has a prefetching mechanism through which the file system prefetches sequentially adjacent blocks while a block is read from the disk. The prefetching makes sequential reads very fast. To find how many blocks ext4 prefetches, we formulated an experiment. Before every run of the experiment, the buffer cache is cleared. Every run assumes a prefetch size. First, an arbitary block is read from the disk. A *sleep()* of a second is given to provide enough time for blocks to be prefetched. Then the block that is located 'assumed number of blocks' away from the previously read block, is read. The first read takes time in the order of milliseconds for disk access. If the second read takes time in the order of milliseconds, we are readindg a block that is beyond the prefetch size. So the size of the prefetch buffer is the previously assumed prefetch size.

```
Pseudocode:
Affinitize process to a CPU;
Make priority of this process the highest;
Clear buffer cache;
Choose a random number P;
Assume a prefetch size X;
Let B = Block size determined from the previous
experiment;
Repeat {
Read the $P^{th}$ block;
Let R1 = Time required by the read;
Sleep() for a second;
Read the $(P + X)^{th}$ block;
Let R2 = Time required by the read;
if ( R1 $\approx$ R2 )
Output previous X as actual prefetch size;
else
Increment X by suitable step;
Clear buffer cache;
}
```

3.1.3 Buffer Cache Size

As described earlier, buffering helps minimize the time wasted in repeated reads of the same data. In our experiment, we first assume the buffer cache size to be a certain value. During the first pass, we read the assumed number of blocks from a file. Since this is the first access made to these blocks, a compulsory cache miss will occur for each of the blocks and so the average read time will be more (in the range of milliseconds). Now, we read the same sequence of blocks for the second time. If the actual buffer cache size is greater than the assumed buffer cache size, then the average read time during the second pass will be very less (in the range of microseconds). If on the other hand, the buffer cache size is smaller, then cache misses will occur for every block being accessed and the average read time during the second pass will also be similar to the first pass (in the range of milliseconds). We keep repeating this experiment by increasing the assumed buffer cache size until this condition is reached. The buffer cache size can then be determined to be in the range where such a transition occurs and the experiment can be repeated by making smaller increments in this range to more accurately measure the buffer cache size.

At the beginning of the experiment and after each iteration the buffer cache is cleared so that the results of the previous iteration do not affect the measurements. Also, to avoid blocks from being prefetched during both passes we read the blocks in reverse order from the file.

```
Pseudocode:
Affinitize process to a CPU;
Make priority of this process the highest;
Clear buffer cache;
Let X = Assumed buffer cache size;
Repeat {
Read X number of blocks from file in reverse order;
Let R1 = Average access time for a block;
Read the same X blocks again;
Let R2 = Average access time for a block;
Increment X by the step size;
} until ( R1 $\approx$ R2)
Clear buffer cache;
```

However, the buffer cache size available for a given process is not constant and it varies depending upon the memory usage by other active processes. So, we have to conduct the experiment in a stable ambiance, preferably having the current process as the only process under execution. This will give us the maximum value that the buffer cache size can assume.

3.1.4 Number of in-inode extents

We conducted our experiments based on the concept that, when the number of extents exceeds the number that can be stored in memory then some additional time would be required to allocate extra blocks that would store the HTree. We start by writing block by block to a newly created file and find the time taken for each block write.

```
Pseudocode:
Affinitize process to a CPU;
```

```
Clear cache;
Create a new file;
Let X = Number of blocks to be written;
while ( X ≥ 0)
{
    Write a block of data to the file;
    Store the time taken for the write;
    X = X - 1
}
Plot graph of Block number vs time taken
```

The corresponding graph is shown in Figure 7. The graph showed a number of small peaks and one sharp peak at the end. To cross check if these were the blocks that correspond to points where new extents are created, we used the *debugfs* utility. With the help of *debugfs* we could trace the physical address corresponding to a given logical block and, through trial and error, we traced the beginning of each extent. However, after comparing the results of *debugfs* with the graph, we could see that the small peaks did not match with points where the extents started. This could be because ext4 file system uses delayed allocation mechanism, where contiguous physical blocks get allocated only when the buffer overflows or when the file is written to the disk. This improves performance and reduces fragmentation by improving block allocation decisions based on the actual file size. The sharp peak towards the end could thus be attributed to the time taken to finally allocate contiguous blocks in the disk.

Since delayed allocation prevents us from identifying extents we altered our experiment slightly and timed the reads instead of writes hoping to see an increase in the read time as the block numbers increase. The cache was cleared after each read so as to avoid irregularities due to prefetching. However, except for a few peaks due to noise we could not observe any increase in the read time as expected. This behavior could be because large contiguous blocks of memory are allocated and hence the number of extents is so few that the additional time required to trace the HTree is not that expensive.

4. Results and Evaluation

4.1 Block Size

To implement our experiment to determine the block size, a rifle-shot read of 1 byte was done. 47 was chosen as the random prime number. The experiment was done for assumed block sizes ranging from 256 to 4096, stepping in powers of 2. While smaller assumed block sizes reported time in the order of microseconds for the second read, 4096 bytes reported a read time of about 7 milliseconds. Thus, we concluded that 4096 bytes was the block size. The results of the experiment are shown in Figure 2.

4.2 Prefetch size

To determine the prefetch size, larger reads of size 4096 bytes were done at a random block number and then to the block that was at the end of the assumed prefetch size. The assumed prefetch sizes were varied from 32 to 256 blocks in powers of 2. A second disk access occured for an assumed prefetch size of 256 blocks, which meant that the actual prefetch size was between 128 blocks and 256 blocks. With a second run

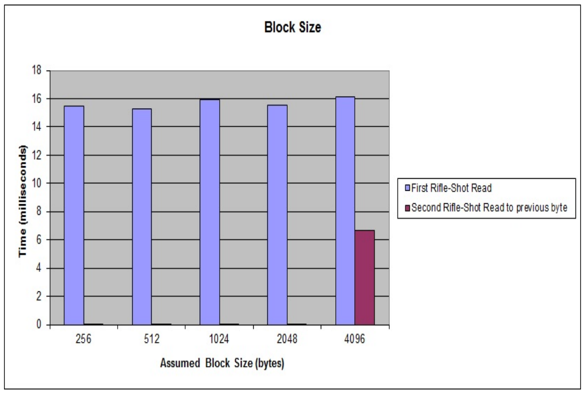

Figure 2. Time for First and second rifle shot reads for different assumed block sizes

and further granularity, the prefetch size was determined as 200 blocks or 800KB. The results are shown in Figure 3.

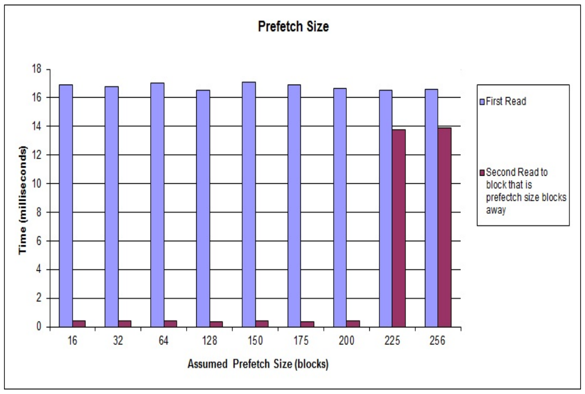

Figure 3. Time for first and second block reads for different prefetch sizes

4.3 Buffer Cache Size

To measure the approximate buffer cache size, three passes of the algorithm in section 3.1.3 were executed, each time with decreasing values of step size. In the first run the increment was done in powers of 2 to speed up the process of identifying the range in which the buffer cache lies. It was found to be between 512MB and 1GB. The second run was executed with 100MB as step size and the third run with 20MB as the buffer size. From the graphs it can be concluded that the buffer cache size is somewhere in the range of 850MB.

4.4 Number of in-inode extents

In our experiment we created a file of size around 100MB. We conducted the experiment as described in section 3.1.4 and the corresponding graph is shown in Figure 7. As can be seen the write time is almost constant except for a few peaks due to noise. It could also be noted that the write time is much less than what could be normally expected in file systems such a ext3. This could be because in ext2 blocks are allocated the moment data is written to them. Ext4 however follows delayed allocation mechanism and hence the average write time is much less throughout and there is just one peak at the end which corresponds to the point where the actual allocation of contiguous

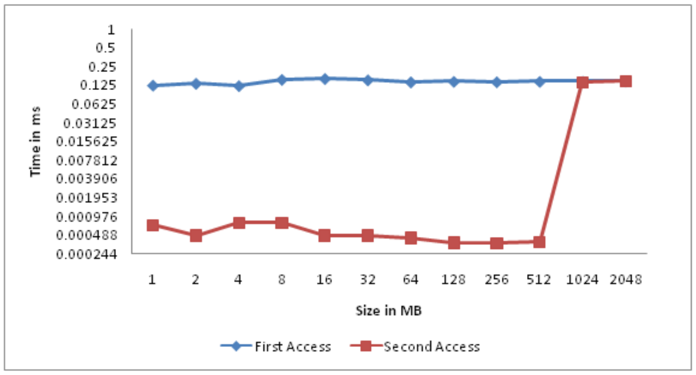

Figure 4. First run where assumed buffer size is incremented in powers of 2 starting from 1MB as the assumed buffer cache size

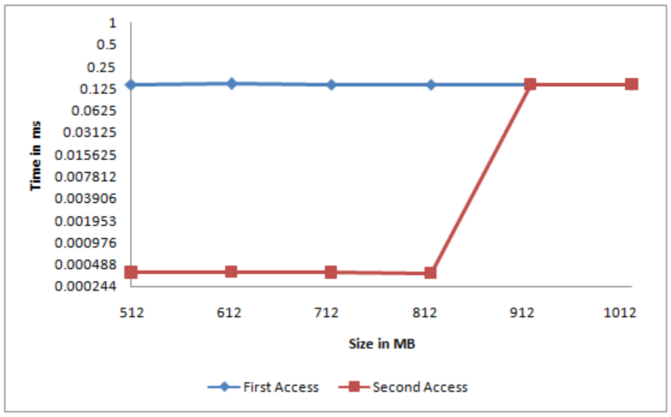

Figure 5. Second run where the step size is fixed as 100MB starting with 512MB as the assumed buffer cache size.

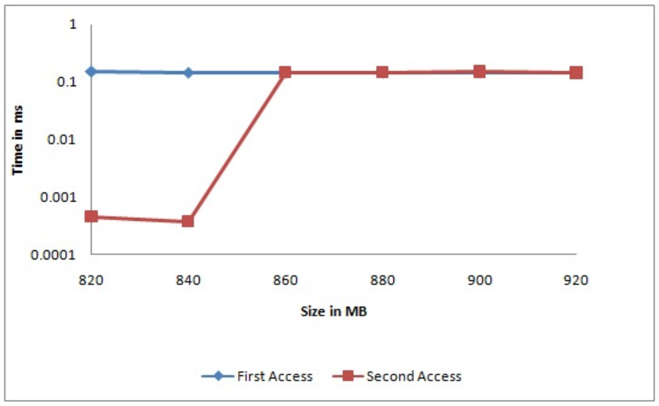

Figure 6. Third run where the step size is reduced to 20MB and starting with 820 MB as the assumed buffer cache size

block of memory in disk is made. This avoids fragmentation and also improves performance to a great extent.

In the second part of our experiment, we measured the read access time for each block for the above file. The graph is shown in Figure 8. Except for a few peaks due to noise, the graph does not exhibit any increase in access time, as the block number

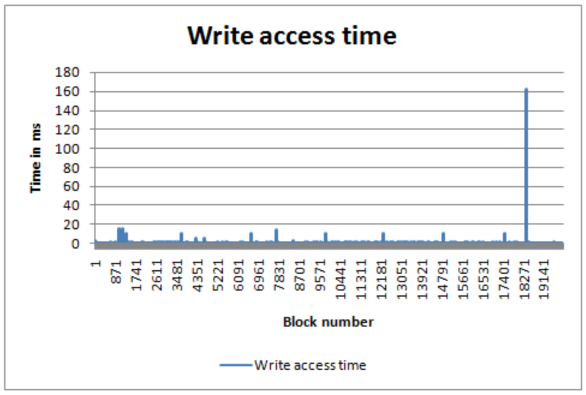

Figure 7. Graph showing block number vs time required to write the block

increases. This may be because, large contiguous blocks are grouped together as extents and hence, even for a file as large as this, the number of extents is small enough such that it doesn't require significant access time overhead.

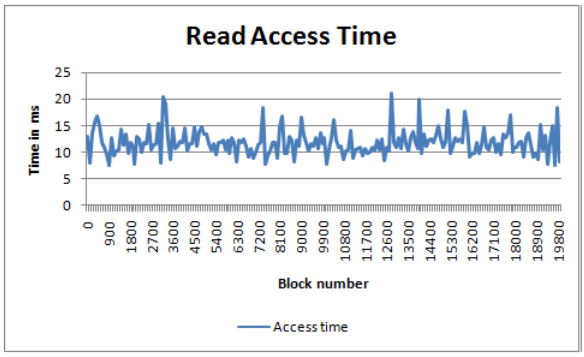

Figure 8. Graph showing Block number vs read access time

This was the most difficult of all experiments. Though we got to understand the various techniques involved to improvise the performance of file access, we could not however measure the number of direct extents present in the inode. The inherent modeling of the ext4 file system makes it very difficult to measure the number of extents or the beginning of the extents. We could however conclude that the ext4 file system could be observed to produce a large amount of performance improvement over the earlier file systems.

5. Conclusion

We have measured the basic block size, prefetch size, buffer cache size and we have analysed the characteristics of the extent records that are used to store the file metadata. We learned how to use system utilities such as drop cache, that helps to clear the cache; nice, that helps to increase the priority of a process; set_affinity which makes a process execute on a single core. We also learned how to use the tool called debugfs that was very helpful in tracing the beginning of extents. We also learned how noise affects measurements in systems and how to counteract noise by taking necessary precautions while taking measurements.

From the experiments coducted we learned a few major characteristics of the ext4 filesystem behaviour

1. ext4 is a very intelligent file system with advance block allocation techniques and smart prefetching mechanisms.
2. The ext4 filesystem produces a huge performance improvement over the earlier version such as ext2 by using delayed allocation.
3. It also decreases to a large extent, the huge amount of metadata stored in the inode by having just one extent refer to a huge contiguous block of memory.

References

[1] http://en.wikipedia.org/wiki/Ext4

[2] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, *The new ext4 filesystem: current status and future plans*, IBM Linux Technology Center