Improving the Performance of *fsck* in FreeBSD

Marshall Kirk McKusick
1614 Oxford Street
Berkeley, CA 94709-1608

1. Introduction

While listening to the presentation of the first paper at FAST 2013, ''ffsck: The Fast File System Checker'' [Ma, Dragga, Arpaci-Dusseau, & Arpaci-Dusseau, 2013], I immediately wondered if I could implement some of the ideas in FreeBSD. The researchers' goal was to reorganize the Linux ext3 filesystem and to rewrite its filesystem checker so that a complete check of the filesystem could be done more quickly. With the addition of a couple of hundred lines of code, I was able to implement both the improvements to *fsck* and the layout policy in the FreeBSD filesystem (FFS).

Although the thrust of the paper was to make changes to the layout of the filesystem to enable *fsck* to run more quickly, some of the changes resulted in a reduction in performance of the filesystem. As I am unwilling to accept a reduction in filesystem performance solely for the purpose of speeding up *fsck*, I chose to consider only on the subset of their changes that improve both.

2. Implementation

The paper describes changes that they made to the on-disk layout of the filesystem. It is very difficult to get folks to change to a different filesystem format that is incompatible with the existing filesystem format. So, in my implementation, I was not willing to change the filesystem format beyond using one of the spare fields in the superblock to tune the layout policy. Even with these limitations, I was able to get an impressive improvement in *fsck*'s running time and some small improvements in filesystem performance.

In FFS the disk space is broken up into groups of contiguous blocks called cylinder groups similar to the ext3 block groups. The first block of each cylinder group contains the cylinder group descriptor that includes a map showing the free and allocated blocks and a map showing the free and allocated inodes in that cylinder group. Following the cylinder group descriptor are blocks that contain the metadata (inodes) for the files in that cylinder group. The organization of an inode is shown in Fig. 1. The remainder of the cylinder group is made up of blocks that contain the indirect blocks and data blocks for the files and directories contained in the filesystem. An inode may reference blocks in one or more cylinder groups in the filesystem though the policy is that small files have their blocks allocated in the same cylinder group in which the inode resides. For details, see Chapter 8 of [McKusick & Neville-Neil, 2005].

The key idea in the paper is to reserve a small area in each cylinder group immediately following the inode blocks for the use of metadata, specifically indirect blocks and directory contents. It requires that metadata be allocated in this area and does not allow data blocks to be allocated in this area. Thus, the paper has a long discussion of how to size this area. If it is improperly sized the filesystem will report as being full when it in fact still has plenty of available space since it reports a filesystem full error when either the metadata area or the non-metadata area fills up.

The FFS separates the allocation of data blocks and inodes into two distinct layers: policy and implementation. The policy layer is responsible for picking what it views as the ideal place to allocate the inode or the data block. For example, when asked to allocate a block for a file, it will usually ask for the block that immediately follows the previously allocated block.

The implementation layer is responsible for managing the allocation bitmaps and ensuring that resources do not get double allocated. Thus, the policy layer does not have to worry about requesting an already allocated block. If the implementation layer finds that a requested block is already allocated, it simply scans through the map to find the closest available free block. The result of this separation is that once the implementation layer is working properly, filesystem designers are free to try out whatever hair-brained policy ideas that they want without fear of corrupting the filesystem. In the case of FFS, the implementation layer was written and debugged in 1982 and has not been changed since. Further refinements to the filesystem have been done at the policy layer.

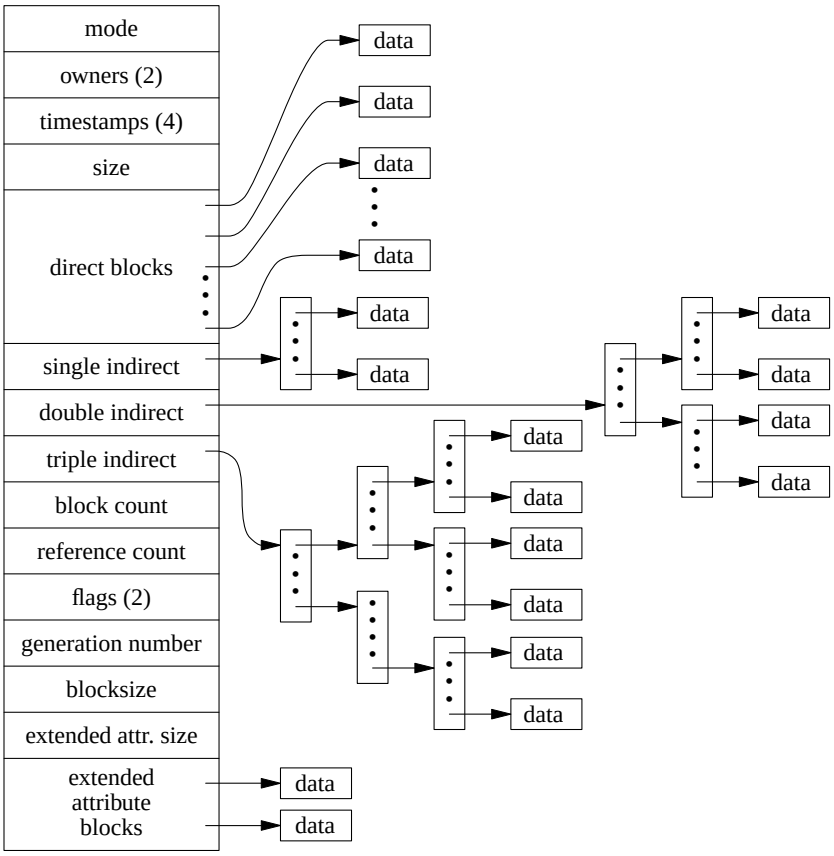

Figure 1. The structure of an inode

Following these design principles, I chose not to change the filesystem layout or the implementation layer. Instead I chose to implement it entirely as a new policy. Specifically, the new policy is to hold about the first 4% of the data blocks in each cylinder group for use of metadata. The policy routines preferentially place metadata in the metadata area and everything else in the blocks that follow the metadata area. In my implementation the size of the metadata area does not matter as it is just used as a hint by the policy routines. If the metadata area fills up, then the metadata just gets put in the regular blocks area and vice versa. And this decision happens on a cylinder group by cylinder group basis (e.g., some cylinder groups can overflow their metadata area while others do not overflow it). For filesystem performance, it is usually better to have the metadata in the same cylinder group as its inode than it is to push it to the metadata area of another cylinder group as is done by the design in the paper.

Another area where I chose to take a different approach than the paper is in the allocation policy for the first indirect block of the file. The BSD fast filesystem tries to place the first (single) indirect block inline with the file data (e.g., it tries to contiguously lay out the first 12 direct blocks followed immediately by the indirect block followed immediately by the data blocks referenced from the indirect block). One of the performance slowdowns in the paper occurs for files that spill into only the first part of their first indirect block. The slowdown comes from moving this first indirect block to the metadata area thus causing two extra seeks when reading it. To avoid this slowdown, I do not change the layout of the first indirect (leaving it inline). Only the second and third level indirects along with the indirects that they reference are moved to the metadata area. The nearly contiguous allocation of this metadata close to the inode that references it noticeably improves the random access time to the file as well as speeding up the running time of *fsck*. Also, as noted in the paper, the disk track cache is often filled with much of a file's metadata when the second-level indirect block is read thus often speeding up even the sequential reading time for the file (though in limited testing I did not see statistically significant differences in sequential reading times).

In addition to putting indirect blocks in the metadata area, Ma, et al suggest that it is also helpful to put the blocks holding the contents of directories there too. They found that putting the contents of directories in the metadata area gives a speedup to directory tree traversal since the data is a short seek away from where the directory inode was read and may already be in the disk's track cache from other directory reads done in its cylinder group. I added this hint to the FFS block preference routines and observed a similar improvement in the speed of pathname lookup and in the shorter running time of *fsck*.

The final observation that I plucked from the paper specifically for speeding up *fsck* is to save an in-memory copy of the cylinder groups during pass1 so as not to need to re-read them in pass5. This nearly doubles the memory footprint of *fsck*, so if memory runs short (e.g., its mallocs begin to fail) this cache is released as needed to make room for other allocations.

3. Results

I have been testing on an Intel Quad-core CPU running at 2.83GHz with 2Gb of memory and a 2Tb Western Digital 7200rpm testing disk running FreeBSD 8.3-STABLE (Subversion revision r246915M). Filesystems are created with their default settings: 16K blocks, 2K fragments, soft updates, and 4% of the data blocks held for metadata. For these tests, the filesystem is 75% full mostly populated with big files (to exaggerate the metadata effects). In each case a new filesystem was created and all the data copied into it so that the new layout could have maximal effect. There are few files and hence little directory information, so the benefit to the running time for directories is minimal in these tests. I am presently running tests on a more conventionally populated filesystem.

Fsck times are also better as the filesystem has not been aged. However aging effects in the FFS filesystem tend to be a lot less noticeable than in others due to its use of dynamic block reallocation. Notably, the Harvard folks found that I/O performance dropped off by only about 10% after ten months of simulated aging [Seltzer & Smith, 1996]. Also, *fsck* times are low because of the small number of files in the filesystem and hence the smaller number of inodes needing to be inspected. Finally a technique similar to the metadata compression discussion in the Ma, et al paper has been in use in *fsck* for the directory metadata since 1988 which cuts down on running time.

Executive summary on running time of *fsck*:

- Baseline before any changes: 284 seconds (4 min 44 sec)
- Storing second and third level metadata (and their referenced indirect blocks) but not first indirect block in the metadata area: 135 seconds (2 min 15 sec)
- Adding directory data blocks to metadata area: 134 seconds (2 min 14 sec)
- Caching cylinder group blocks in pass1 to avoid the need to read them in pass5: 84 seconds (1 min 24 sec)

In Appendix 1 are the summary statistics for each run. I/O listed as ‘‘Double Level Indirect’’ includes all double-indirect blocks referenced from inodes and all the single level indirect blocks below them. Similarly, ‘‘Triple Level Indirect’’ includes all triple-indirect blocks referenced from inodes and all the single- and double-level indirect blocks below them. The key observation is that while the number of I/Os of each type of data remain similar from run to run, the percentage of time for reading the metadata has dropped dramatically.

I ran just a few tests on the speed with which data could be read from or written to files. Random read times improved a bit. The remaining tests were not statistically significantly different. More thorough tests would need to be run to get a reasonable idea of whether it makes any difference. But first results imply no degradation and some hints at improvement.

4. Conclusions

This work has once again shown the power of separating the filesystem layout policy routines from the implementation routines. I was so excited by the possibilities presented by this paper that I skipped lunch after hearing it so I could try implementing it in FFS. By the time the 90 minute lunch break was over I had fully written the 100 lines of changes (half of which were comments) to the block layout policy

routine to implement the reserved metadata area. And I had no fears of bringing it up on my primary server to test it out since I knew that at worst I would get some badly laid out files; certainly I was not running the risk of corrupting my filesystems.

By retaining the same on-disk format, I did not need to make any changes to *fsck*. The stock *fsck* just ran faster because of the new layout of metadata. I did need to make about 100 lines of changes to *fsck* to add the caching of cylinder groups between pass1 and pass5. But that was a trivial change and one that will provide equal improvement whether or not the new filesystem layout is in use. The vast majority of my time has been spent measuring the effects of the changes and writing this paper. Having spent time writing or tuning *fsck* for the past 30 years, I never would have guessed that so much improvement in running time could be gotten out of *fsck* for so little effort.

The lesson to be learned is that separating policy from implementation is an important design principle when architecting software systems, especially when they are mission-critical systems. The policy layer allows new ideas to be implemented and tested quickly. Once validated, those ideas can be deployed without danger of compromising the integrity of the system.

I commend the authors of the paper for their work. Unfortunately the filesystem on which they worked is not separated into policy and implementation layers so they had to make several thousand lines of changes in areas where bugs would compromise the filesystem integrity. The monolithic architecture lead to a great deal more effort on their part than would otherwise have been necessary. Finally, the scope of the change and the possibility of destabilizing a production filesystem will make it far more difficult for them to get their changes accepted back into the mainline code base.

5. References

Ma, Dragga, Arpaci-Dusseau, & Arpaci-Dusseau, 2013.

A. Ma, C. Dragga, A. Arpaci-Dusseau, & R. Arpaci-Dusseau, “ffsck: The Fast File System Checker,” *USENIX FAST ’13 Conference*, available from www.usenix.org/conference/fast13/ffsck-fast-file-system-checker (February 2013).

McKusick & Neville-Neil, 2005.

M. K. McKusick & G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley, Reading, MA (2005).

Seltzer & Smith, 1996.

M. Seltzer & K. Smith, “A Comparison of FFS Disk Allocation Algorithms,” *Winter USENIX Conference*, pp. 15-25, available from www.eecs.harvard.edu/margo/papers/usenix96-ffs (January 1996).

Appendix 1
More details on the performance measurements

#
Baseline with no changes on 16K/2K filesystem
#
305 files, 447783558 used, 191768583 free (71 frags, 23971064 blocks, 0.0% fragmentation)
17.401u 0.830s 4:44.39r 6.4% 152t+978d (1131tot/43104max) 48723+33io 21361r+0pf+0w

Final I/O statistics
Running time: 284.263 sec

Buffer reads by type	Count		I/O Read Time	
Cylinder Group:	14038	28.8%	94.199 sec	35.3%
Single Level Indirect:	218	0.4%	2.152 sec	0.8%
Double Level Indirect:	23553	48.3%	141.592 sec	53.0%
Triple Level Indirect:	3859	7.9%	23.330 sec	8.7%
Inode Block:	7020	14.4%	5.412 sec	2.0%
Directory Contents:	26	0.0%	0.068 sec	0.0%

#
Indirect blocks in reserved area on 16K/2K filesystem
#
329 files, 449870118 used, 186694875 free (91 frags, 23336848 blocks, 0.0% fragmentation)
18.072u 0.690s 2:15.84r 13.8% 156t+1007d (1163tot/42880max) 48780+31io 21267r+3pf+0w

Final I/O statistics
Running time: 135.760 sec

Buffer reads by type	Count		I/O Read Time	
Cylinder Group:	13972	28.6%	104.148 sec	88.5%
Single Level Indirect:	235	0.4%	1.086 sec	0.9%
Double Level Indirect:	23683	48.5%	6.022 sec	5.1%
Triple Level Indirect:	3859	7.9%	0.511 sec	0.4%
Inode Block:	6986	14.3%	5.757 sec	4.8%
Directory Contents:	26	0.0%	0.145 sec	0.1%

#
Include directory data blocks in reserved area
#
311 files, 448187974 used, 188377019 free (83 frags, 23547117 blocks, 0.0% fragmentation)
17.847u 0.802s 2:15.20r 13.7% 154t+995d (1149tot/43022max) 48656+38io 21332r+0pf+0w

Final I/O statistics
Running time: 134.734 sec

Buffer reads by type	Count		I/O Read Time	
Cylinder Group:	13972	28.7%	103.372 sec	88.5%
Single Level Indirect:	223	0.4%	1.006 sec	0.8%
Double Level Indirect:	23581	48.4%	6.014 sec	5.1%
Triple Level Indirect:	3859	7.9%	0.510 sec	0.4%
Inode Block:	6986	14.3%	5.754 sec	4.9%
Directory Contents:	26	0.0%	0.141 sec	0.1%

```
#
# Add cylinder group block caching in fsck
#
311 files, 448187974 used, 188377019 free (83 frags, 23547117 blocks, 0.0% fragmentation)
17.539u 0.911s 1:24.75r 21.7% 152t+982d (1135tot/99182max) 41670+50io 49383r+0pf+0w

Final I/O statistics
Running time: 84.622 sec
```

Buffer reads by type	Count		I/O Read Time	
Cylinder Group:	6986	16.7%	53.453 sec	79.8%
Single Level Indirect:	223	0.5%	1.098 sec	1.6%
Double Level Indirect:	23581	56.5%	5.965 sec	8.9%
Triple Level Indirect:	3859	9.2%	0.503 sec	0.7%
Inode Block:	6986	16.7%	5.754 sec	8.5%
Directory Contents:	26	0.0%	0.141 sec	0.2%