A Critique of Seltzer's LFS Measurements

John Ousterhout / john.ousterhout@scriptics.com

The paper "File System Logging Versus Clustering: A Performance Comparison", by Margo Seltzer et al., appears in the Winter 1995 USENIX Conference. This paper is the second in a series of papers analyzing the performance of Seltzer's implementation of a log-structured file system in the BSD kernel (the first appeared in the Winter 1993 USENIX Conference).

The new paper corrects a number of flaws in the earlier paper having to do with the BSD LFS implementation and the choice of benchmarks, but I believe that the new paper is still misleading in several ways:

1. Section 5 claims to measure the effect of fragmentation on small file performance, but the benchmarking approach is such that it doesn't really measure small files; it measures only large files. Small file performance under FFS will be worse than claimed in the paper.
2. The measurements in Section 4 are counter-intuitive, and the back-of-the-envelope calculations used to support those measurements are incorrect. At present neither Seltzer nor I can explain the measurements.
3. BSD LFS does not include a simple optimization that should substantially improve the LFS results in Section 4.

Thus, of the three main results sections (Sections 3, 4, and 5) only the results in Section 3 are completely reliable; the others should be taken with a grain of salt.

The following sections explain the problems in more detail. Several other Web pages are also available on the topic of FFS vs. LFS:

- Seltzer's response to these comments.
- My response to Seltzer's response, which shows that Seltzer's comments validate my concerns.
- A critique of Seltzer's 1993 USENIX paper.

Section 5: FFS Fragmentation

The most serious problem with the paper is Section 5, which attempts to measure the impact of free space fragmentation on FFS performance. This was done in Section 5.3 by taking snapshots of several production FFS disks. The snapshots were used to recreate the disks' block allocation structures on fresh disks, then benchmarks were run on the recreated disks. These results of the benchmarks were compared with the same benchmarks running on clean (unfragmented) disks to measure the performance degradation due to fragmentation.

Seltzer claims that real workloads will experience the same degradation seen for the benchmarks, but this is not the case. In fact, real workloads are likely to see much greater performance degradation than Seltzer measured, because the benchmarks really measure the performance of very large files; small files tend to suffer more from fragmentation than large files.

The reason for this is explained in Section 5.2 of the paper. In FFS the free space in a cylinder group is unevenly distributed. The first part of a cylinder group tends to be highly fragmented (only a few free blocks distributed far apart), and most of the free space is concentrated at the end of the cylinder group. As explained in Section 5.2, small files tend to be allocated entirely from the beginning of the cylinder group so they are likely to be fragmented. Large files, on the other hand, may consume all of the fragmented free space and have most of their blocks allocated from the more concentrated free space at the end of the cylinder group. Thus large files have better layout than small files. The authors of the paper measured this effect and the measurements appeared in earlier drafts of the paper (these measurements were dropped from the final draft, but I believe that they are available in reference 12 cited by the paper).

Consider the benchmark used in Section 5.3, which creates a large number of files of a given size and measures the throughput. The problem is that the benchmark writes 25 files in a row to a single directory without deleting any files. All 25 of these files will probably fall in the same cylinder group. The first files will consume all of the cylinder group's fragmented space, and the later files will all be allocated in the "good" space at the end of the cylinder group, just as if a single very large file had been written.

Now compare this to real workloads. It is unlikely in practice for 25 new files to be created without any intervening deletes. Real workloads tend to have interspersed creates and deletes, so that new space is constantly being freed throughout the cylinder group and the fragmented free space is never consumed as in the benchmark.

Since the benchmark in Section 5.3 writes 25 files in a row, it is really measuring the fragmentation for files about 25 times as large as those actually used. For example, Section 5.3 used three file sizes: 8KB, 64KB, and 1MB. The 8KB case is really equivalent to writing a single 200KB file, and the 64KB case is equivalent to a single 1.5MB file. Thus the benchmark doesn't really measure small file performance at all (in practice, less than 5% of all files are larger than 200 KB and only about half of all bytes transferred are to files larger than 200 KB). Small-file fragmentation is much worse than large-file fragmentation, so the real-world degradation for 8KB and 64KB files is likely to be worse than Seltzer reports.

There is also a second problem with Section 5, having to do with 8 KB files. Figure 8 shows a performance degradation of a few percent for 8 KB files, but in fact there should be none: 8 KB files occupy only a single block on disk so contiguous block allocation isn't even an issue. It is not obvious to me why there should be any degradation at all for 8KB files; in any case, it would be better to use 16KB files as an example of the smallest files where fragmentation is an issue.

Section 4: Transaction Processing

The other problematic section of the paper is Section 4, which uses a transaction processing workload to compare LFS to FFS. I believe that the results in this section don't make sense without further explanation; until they can be explained I think they should be viewed with skepticism.

The problem with the results in Figure 7 is that LFS shows almost no variation in performance as the disk utilization changes. This benchmark is a pathological case for LFS due to its random access patterns, and I believe that its performance should be very sensitive to disk utilization. See, for example, Figure 4 of the original LFS paper by Mendel Rosenblum (in ACM Transactions on Computer Systems, February 1992).

Seltzer attempts to explain the performance by doing two back-of-the-envelope calculations near the end of Section 4; ultimately she claims that there is only a 10% gap between expected and measured performance. Unfortunately, though, this calculation fails to include two important factors: variation in utilization and the short lifetimes of the indirect blocks. These are explained in the subsections below. In fact, the expected LFS performance is 25-30% higher than what was measured.

Variance in segment utilization

Seltzer's calculation assumes that when an LFS disk is 50% utilized, every segment cleaned will be 50% utilized. However, this is not the case. Under random access patterns there will be variations in the utilizations of segments, and the LFS cleaner can choose the least-utilized segments to clean. Thus, in practice, the utilization of the segments cleaned will be less than the utilization of the disk as a whole. This effect is described in Rosenblum's paper; from Figure 4 in Rosenblum's paper I calculate that if the disk as a whole is 50% utilized, the segments cleaned should be only about 33% utilized, which will yield much better cleaning performance.

Let's redo Seltzer's back-of-the-envelope calculation (the last one in the paper, in the next-to-last paragraph of Section 4) using Seltzer's disk parameters of 14.5 ms for a random seek and rotational delay, 2.3MB/sec read bandwidth, 1.7 MB/sec write bandwidth, and a transaction rate of 43 TPS (with cleaning costs ignored). At a 33% utilization of cleaned segments, LFS will read 3 dirty segments, write 1 cleaned segment, and have room

for 2 segments worth of new transactions. It will take 450ms to read each segment and 603ms to read the cleaned segment. Then there will be room for 2*168 = 336 new transactions at a rate of 43 TPS, or 7.81s. Thus the total time for 336 transactions (including both the transactions and the cleaning associated with them) will be 3*450ms + 603ms + 7810ms = 9763, for a rate of about 34.4 transactions per second. The measured time in Figure 7 is 27 TPS, so there is a discrepancy of about 27% (not 10% as reported in the paper).

Let's also do a calculation for a disk utilization of 80%. From Rosenblum's Figure 4 I calculate the utilization of the cleaned segments to be about 66%, so LFS will read 3 segments for cleaning, write 2 cleaned segments, and have 1 segment for 168 transactions of new data. The total time for 168 transactions will then be 3*450ms + 2*603ms + 3900ms = 6456ms, or 26 TPS. This is closer to the measured rate of about 24 TPS, but it is still off by about 8%. Also note that this number is 30% lower than the estimate for 50% disk utilization.

Short-lived data

The second factor not included in Seltzer's calculation is the effect of indirect blocks. BSD LFS flushes all of the dirty indirect blocks to disk whenever it writes data blocks (I think this is a bug; more on this below). In the benchmark of Section 4 the file has 58 indirect blocks and the cache holds 173 blocks dirty blocks when disk writes occur. Virtually all the indirect blocks will be dirty whenever a write occurs, so 58 out of every 173 new blocks written to disk will be the indirect blocks. Furthermore, these indirect blocks die almost immediately: they will all be overwritten with fresh copies the next time the cache is flushed. The indirect blocks therefore have a much shorter lifetime than the data blocks of the file, and they constitute about 1/3 of all the disk writes.

Rosenblum's paper shows that LFS performs much better when some of the data is short-lived than when all of it is uniformly long-lived (see Figure 7 in the Rosenblum paper), so the measured performance should be better than what was estimated above. I haven't been able to find a way to estimate cleaning costs under this set of conditions that I'm completely comfortable with. My best guess is that at 50% disk utilization the utilization of cleaned segments will be around 25% and at 80% disk utilization the utilization of cleaned segments will be around 57%. This translates into overall performance of 36 TPS and 29 TPS respectively, which are 33% and 20% higher than the measured values. However, I don't completely trust these estimates, so use them with caution.

I don't claim that the measured numbers are wrong, but I do think there is something going on that we don't understand right now. For example, there may be some artifact of the BSD implementation that is affecting the measurements. Until there is a logical explanation for the measurements I think that they should be taken with a grain of salt.

BSD-LFS's writing policy

I believe that there is a bug in the writing policy of BSD-LFS that affects the performance in the measurements of Section 4. The bug is that BSD-LFS writes all dirty indirect blocks whenever it writes any dirty data blocks. In this benchmark, 58 out of every 173 blocks written to disk are indirect blocks. The indirect blocks get modified almost immediately after being written, so they result in a lot of free space on disk that must be reclaimed by cleaning.

I believe that the best policy is to write dirty indirect blocks for a file only if the file has been closed or if the LRU replacement policy determines that the block should be replaced in the cache. In the normal case where a file is opened, written sequentially, and immediately closed, this guarantees that the indirect blocks will be near the file's data on disk. In the case where a file is open a long time and is being accessed randomly, as in the transaction processing benchmark, there is nothing to be gained by writing the indirect blocks. They are not needed for reliability, since there is already enough information in the log to recover the data blocks after a crash without the indirect blocks. If a file is open there will probably be more I/O to the file soon, so the indirect blocks are likely to be modified soon, which invalidates any copies on disk. Writing the indirect blocks just wastes write bandwidth and makes more work for the cleaner.

In the last paragraph of Section 4 Seltzer tries to justify the BSD implementation by arguing that the cache space occupied by dirty indirect blocks is needed for dirty data blocks. However, virtually every indirect block will be touched again before the cache fills, so in fact all the indirect blocks will stay in the cache anyway and no space will be freed for dirty data blocks. If in fact a dirty indirect block ages out of the cache according to the cache's LRU policy, then I agree that it makes sense to write it to disk, but this isn't the case in this particular benchmark.

Thus the results for LFS in Section 4 are a bit pessimistic. For example, this should make a difference of about 1.2ms per transaction in the case where there is no cleaner, which should increase the transaction rate from about 43 TPS to over 45 TPS. I can't estimate how much this will affect LFS performance when the cleaner is turned on, since the absence of the indirect blocks will change the cleaning cost in ways I don't know how to estimate. However, the relative impact of the optimization should be much greater with the cleaner turned on at high disk utilizations than when the cleaner is turned off.