A Response to Ousterhout's Critique of LFS Measurements

John Ousterhout has posted a web page questioning measurements in the 1995 Usenix paper **Logging versus Clustering: A Performance Evaluation** This page responds to the issues raised by Ousterhout.

- Ousterhout references "a number of flaws in the earlier paper" (the 1993 Usenix paper **A Log-Structured File System for UNIX**), but does not detail these flaws. To date, we are aware of no technical inaccuracies in the paper.
- Ousterhout cannot explain the measurements in Section 4, stating that they are inconsistent with the *simulated* results in the Rosenblum/Ousterhout paper that appeared in the February 1992 ACM Transactions on Computer Systems. The simulated results in the TOCS paper are for a different algorithm and different parameters than used by either Sprite-LFS or BSD-LFS. This is explained in more detail below.
- The optimization, suggested by Ousterhout, to improve the transaction processing performance has not yet been implemented. This optimization affects the ability of the file system buffer cache to cache dirty data, and the evaluation of its impact is beyond the scope of the 1995 Usenix paper. We agree that this work will be interesting and encourage Ousterhout to pursue this research avenue.
- Ousterhout claims that the results in Section 5 are invalid. We present even more data below that demonstrates that the results in Section 5 are indicative of real-world performance.

A Clarification of the 1993 Usenix Paper

There is one graph that, while not incorrect, is only partially explained in the paper. Figure 9 on page 320 depicts the write performance of BSD-LFS, BSD-FFS and the raw disk on a benchmark that synchronously writes data to disk. In the paper, this difference is attributed to the fact that FFS begins asynchronously writing blocks to disk as soon as they are filled, but LFS delays writes until approximately 800 KB of data have accumulated. This phenomenon does account for part of the performance gap, but a second phenomenon also contributes. The maximum transfer unit between memory and the disk is 56 KB. The FFS places a rotational delay between 56 KB units; LFS does not. Therefore, when data is being written to disk, LFS loses a rotation between 56 KB writes while FFS loses only a *rotdelay* (approximately 1/4 revolution).

Section 4 Cleaner Measurements

Ousterhout's questions about the measurements in Section 4 stem from the following misinterpretation of the results presented in the Rosenblum/Ousterhout paper in ACM Transactions on Computer Systems, February 1992 (hereafter referred to as *the TOCS paper*).

- The *simulated* results presented in Figure 4 of the TOCS paper are for the greedy cleaning algorithm. This is **not** the cleaning algorithm implemented in either Sprite-LFS or BSD-LFS, and therefore the results of the simulation do not apply.
- The results presented in the TOCS paper are for a single file system size, segment size, and cleaning cluster size (the number of segments cleaned per cleaner invocation). None of these parameters is the same for the system measured in the 1995 Usenix paper or for the Sprite-LFS system measured in the TOCS paper. All of these parameters affect performance.

- The cost-benefit calculation discussed in the text of the TOCS paper is different than that used in the simulation (the one described in the text is the one used in both Sprite-LFS and BSD-LFS).

- The age metric used in the cost-benefit simulation is different than the age metric used in either Sprite-LFS or BSD-LFS.

These four misunderstandings render Ousterhout's comparisons questionable.

The TOCS paper discusses two cleaning algorithms. The first is the *greedy* algorithm, which always cleans the segment(s) with the lowest utilization. This is the algorithm simulated to produce the results in Figure 4 of the TOCS paper. The paper then goes on to explain that the greedy algorithm is sub-optimal for workloads that show skewed distributions. This observation leads to the second algorithm: *cost-benefit*. The cost-benefit algorithm uses a combination of segment utilization and segment age to select segments for cleaning.

What the TOCS paper fails to discuss is the selection of the age metric in the cost-benefit algorithm. The TOCS paper presents *simulated* results of a log-structured file system. An examination of the simulator that we believe was used to derive these results reveals that the simulated file system measures age in terms of the number of blocks that have been written. The calculation of the cost-benefit metric uses the log of the number of writes since a segment was created. Therefore, the age metric grows slowly. For example, on a 1 GB file system with a 1 MB segment and 4 KB blocks, the maximum age possible after filling the file system the first time is 18. A segment that survives another writing of the complete disk (all but one of the 1024 segments) will have an age of 19.

In contrast to this simulated cleaning algorithm, both Sprite-LFS and BSD-LFS use elapsed time as the measure of segment age. The age of a segment is the age of the segment's newest file. In the cost-benefit calculation, Sprite-LFS expresses the age in minutes while BSD-LFS expresses the age in seconds. In both cases, the selection of segments is affected by the write rate of the application and the number of segments in the file system. For example, consider the 1 GB file system described above. On BSD-LFS, the first segment written on a pass over the entire file system will have an age of 613 for the first pass and will grow to 1226 for the second pass. On Sprite-LFS, the age will be 10 for the first pass and 20 for the second pass.

Our measurements show that when the disk is 50% full, the average segment utilization is 48%, not the 25% or 33% expected by Ousterhout. In our measurements for TPC/B, we increase the fullness of the file system by creating a large file that is never accessed. In effect, this reduces the size of the file system and reduces the impact of the age metric, because segments are recycled more quickly. The effects of the age metric were never addressed in the TOCS paper. These effects are subtle and not yet well-understood. We believe that these effects warrant further investigation that is beyond the scope of the 1995 Usenix paper.

Cleaning Optimization

Ousterhout proposes keeping indirect blocks in the cache when a file's data blocks are written. This is not an unreasonable suggestion. However, retaining indirect blocks is more complicated than Ousterhout suggests.

First, Ousterhout ignores the fact that periodically, during checkpoint, indirect blocks must be written to allow file system recovery. Indirect blocks can be reconstructed by rolling forward after a crash. However, there must be a consistent snapshot from which to begin the roll forward process. When segments are cleaned, they cannot be rolled forward. Therefore, the only segments eligible for cleaning are those written before the most recent checkpoint.

Second, normal cache replacement in 4BSD depends on the fact that the file system can evict a dirty buffer from the cache at any time. In LFS, this is not true. The creation and writing of a segment requires the use of additional buffers to store segment meta-data such as segment summaries and inode blocks. This has two main effects on buffer cache management. Dirty LFS blocks cannot be kept on the normal LRU queue and segment writes must be initiated before there is danger of running out of buffers. This is discussed in more detail in Section 4.1.2 of **the 1993 Usenix paper**.

Retaining dirty indirect blocks in the cache will reduce the effective size of the buffer cache, penalizing all files being served by the LFS. Although this modification might improve performance of the transaction processing benchmark, the effects on other workloads is unknown, and it is of questionable value to modify an algorithm to improve the performance of one benchmark, potentially at the expense of others. For example, we have modifications to the FFS that substantially improve the performance of the create benchmarks in Sections 3 and 5. However, since we do not yet understand the long term ramifications of these modifications, we did not include them in this study.

FFS Fragmentation: Section 5 Results

The benchmark used in Section 5 is a slight modification of that used in Section 3. The benchmark, designed by John Ousterhout, creates, reads, writes, and deletes a large number of files and gathers data for a wide range of file sizes. In Section 3, the benchmark places 100 files in each directory while in Section 5, we place only 25 files in each directory and report results only for reading and creating.

The purpose of Section 5 is to examine how effectively the Fast File System can allocate files as a file system ages. In order for FFS to achieve the performance shown in Section 3, it must succeed in allocating blocks contiguously on disk. As files are created and deleted, the free space on a file system could become sufficiently fragmented that FFS would be unable to allocate files contiguously. We evaluate the performance over time by running the same benchmark on recreations of old file systems (in particular, in the Usenix paper, we show the performance for 4 points in time over the past year).

Ousterhout has two complaints with this methodology. First, he believes that since the files are created all at once, they will exhibit different characteristics than files created under normal use. Second, he does not understand why our measurements show degradation in performance for files of 8 KB since these files inhabit only a single block.

Let's begin with the second point, since it is trivial to explain. The create test of an 8 KB file involves three writes: the new inode, the directory in which the file is being added, and the data block. In the "empty disk" case (i.e. running the benchmark on a newly created file system), the directory and the 25 files contained in it are allocated contiguously from the beginning of the cylinder group. During the benchmark, the inodes and the modified directory are written synchronously. The data is written asynchronously but is scheduled before the next create begins. Therefore, the disk head seeks back and forth between the inode area and the first data cylinder in the cylinder group.

When the benchmark is run on the recreated file system there is no guarantee that either the directory or any of the files appear at the beginning of the cylinder group. Therefore the seeks between the inode area and the data blocks of the directory and new files are likely to be longer. These longer seeks lead to lower performance.

Read performance is substantially better than write performance as multiple files are in the disk's track buffer after the first file is accessed. However, read performance can also degrade on an older file system. Since the files are not allocated contiguously, there is less benefit from the track buffers, and performance suffers.

Ousterhout's first criticism of the benchmark makes reference to our ongoing research in file layout characterization. The reference to which Ousterhout refers is File Layout and File System Performance. In this report, we define a *layout score* that indicates what fraction of a file's blocks are allocated optimally. A file that is allocated optimally (contiguously) has a layout score of 1.0 and a file with no two blocks contiguous has a layout score of 0.0. Consider a four-block file. We assume that a seek is always required to access the first block of a file and use only the remaining blocks in the layout score calculation. In the case of our four-block file, let's say that the first two blocks are contiguous and the last two blocks are contiguous, but a seek is required between the second and third blocks. There are a total of three blocks considered in the layout score. Of those, 2 are allocated optimally (the second and fourth) while one (the third) is not. This yields a layout score of 0.67.

Our research shows that layout score is a moderate indicator of performance (its linear regression coefficient is 0.7, indicating that the layout score is responsible for approximately 50% of the variance in performance we see). The other component is the length of the seek/rotation required between blocks that are not allocated contiguously. Layout score does not model this.

Ousterhout states that information about layout scores was present in earlier drafts of the paper but has been omitted from the final paper. This is true. At the time of submission, we did not yet have a tool that permitted reconstruction of a file system from a snapshot. We now have this tool. Rather than provide an abstract score that only partially explains performance, we provide actual measured performance in the final paper. The detailed discussion of layout scores appears in the File Layout and File System Performance technical report.

To address Ousterhout's concern over the validity of the results, we calculated the layout scores for the 64 KB files created by the benchmark and then calculated the layout score for the 64 KB files on the recreated file system before the benchmark. We refer to these two sets of files as the *benchmark files* and the *old files* respectively. In most cases, the layout scores of the benchmark files was better than that of the old files, while in one case it was actually worse.

The table below shows the six file systems analyzed in the Usenix paper. For each file system, we show three snapshot dates and the layout scores for the benchmark files and the old files. The fifth column shows the difference in layout score between the two sets of files. Using the layout score of the old files and the equations derived from linear regression of layout score versus performance, we show a predicted read/write performance. Then we show the actual performance that we measured on the benchmark files.

		LAYOUT SCORES			RELATIVE PERFORMANCE as a fraction of empty fs perf.			
File system	date	bench layout	old layout	diff	Pred read	Actual read	Pred write	Actual write
cnews	26Apr94	0.88	0.23	0.65	0.70	0.98	0.63	0.91
cnews	26Jul94	0.84	0.14	0.69	0.67	0.97	0.58	0.91
cnews	20Oct94	0.78	0.15	0.62	0.67	0.94	0.59	0.86
glan5	26Apr94	0.89	0.94	-0.05	0.99	0.97	0.94	0.88
glan5	26Jul94	0.89	0.95	-0.06	0.99	0.96	0.95	0.86
glan5	29Oct94	0.88	0.96	-0.08	0.99	0.96	0.96	0.89
staff	26Apr94	0.70	0.57	0.13	0.84	0.89	0.71	0.79
staff	26Jul94	0.69	0.55	0.14	0.83	0.88	0.69	0.76
staff	20Oct94	0.69	0.62	0.07	0.86	0.89	0.74	0.85
usr4	26Apr94	0.95	0.82	0.13	0.93	0.98	0.83	0.91
usr4	26Jul94	0.85	0.83	0.02	0.93	0.96	0.84	0.92
usr4	29Oct94	0.85	0.84	0.00	0.94	0.92	0.85	0.80
usr6	26Apr94	0.75	0.55	0.20	0.88	0.92	0.74	0.87
usr6	26Jul94	0.69	0.59	0.10	0.88	0.92	0.77	0.85
usr6	29Oct94	0.74	0.60	0.14	0.88	0.92	0.77	0.82
white	26Apr94	0.83	0.79	0.04	0.91	0.90	0.81	0.80
white	26Jul94	0.87	0.71	0.15	0.86	0.98	0.71	0.93
white	29Oct94	0.88	0.55	0.34	0.77	0.98	0.53	0.91

The data shows that the benchmark files have worse layout scores than old files on one file system and better layout scores on the other five file systems. On those five file systems the difference in layout score produces differences between predicted and actual read performance that are typically less than 5%. For writes, the difference between predicted and actual performance is typically less than 10%. Even using the predicted numbers, in most cases, read performance over the one year period is at least 85% of the best performance and write performance is at least 75% of the best performance.

This analysis is not perfect in that the regression equations were derived from the measured numbers and may not be correct for the files that exist in the file system prior to the benchmark. The only other option would be to select a random set of files to read/write. It is unclear that this provides a better metric. The real question is for a given file system configuration, how fragmented are the files that *are actually accessed*. A trace-drive study is probably the next step in this particular examination.

Margo Seltzer / *margo@das.harvard.edu*
Keith Smith / *keith@das.harvard.edu*