First submission
2/94

A Comparison of Logging and Clustering

John K. Ousterhout, Hari Balakrishnan, Keith Bostic, Jacqueline Chang, Sara McMains, Venkata N. Padmanabhan

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Margo Seltzer, Keith A. Smith

Division of Applied Sciences
Harvard University
Cambridge, MA 02138

E-mail: *ouster@cs.berkeley.edu*

Abstract

This paper compares the performance of the LFS and EFS storage managers in BSD UNIX. It expands on the measurements made in an earlier paper by Seltzer et al. [6], and also reflects improvements made to the BSD implementation. Whereas EFS was generally faster than LFS in the Seltzer measurements, we found in our expanded benchmark set that the performance of LFS was generally equal to or better than that of EFS. We also found that EFS can potentially suffer from fragmentation of free space under actual use, which may limit the ability of the system to allocate large clusters.

1. Introduction

Is a log-structured file system more efficient than a traditional block-based system with extensions for clustering? This question was first addressed in the USENIX Winter 1993 Conference by Seltzer et al. [6]. They described a new log-structured file system (LFS) implementation for BSD UNIX and presented a few preliminary benchmark results comparing it to the Berkeley fast file system (FFS) [3] and an extended version of FFS (EFS) that permits clustering as described by McVoy and Kleiman [4]. In most of the Seltzer benchmarks EFS outperformed LFS, but Seltzer noted that the results were preliminary and recommended additional investigations.

This paper will provide additional investigations. We have re-run all of the Seltzer benchmarks, and we have added several additional benchmarks to cover a larger range of usage patterns. For example, the Seltzer paper focussed on large files, but we have also included small and medium-sized files. In addition, we have measured several variants of file writing such as creates instead of overwrites, and asynchronous writes. The paper will also include more detailed measurements of cleaner overheads in LFS, and it addresses the issue of fragmentation in EFS, which has never before been measured.

In addition to expanding the range of measurements, this paper will also analyze the effects of tuning on both LFS and EFS. For example, the LFS implementation measured by Seltzer was completely untuned: it wrote more metadata than necessary, which increased cleaner overheads, and it did not use fragments, which impacted efficiency for small files. This paper will measure a tuned implementation where these problems have been repaired. We will also report on a modified version of EFS that performs writes more efficiently by performing them asynchronously like LFS.

Because of the additional measurements and tuning, this paper reaches a somewhat different conclusion than the Seltzer paper. In the case of an empty disk, which is ideal for both file systems, LFS matches or exceeds the performance of EFS for nearly all of our tests (the main exception being reads of medium-size files). Performance with a full disk depends on cleaning overheads for LFS and fragmentation for EFS. Unfortunately there are not enough production LFS disks for us to add much to the existing cleaner measurements of Rosenblum [5]. However, we have been able to measure production EFS disks, and our measurements suggest that there is fragmentation of free space under the current EFS implementation. Additional analysis is needed both for LFS cleaner overheads and EFS fragmentation under production use, but overall we are cautiously optimistic about the performance of LFS.

2. Overview of EFS and LFS

A disk storage manager such as EFS or LFS is measured by how efficiently it uses the disk. For maximum performance, the storage manager must avoid seeks and perform large I/Os. To achieve this, the storage manager must do three things. First, it must ensure that there are always large contiguous areas of free space on disk. Second, it must use the free areas to lay out related information (such as consecutive blocks of a file or consecutive files in a directory) adjacent to each other and transfer them together in large I/O operations. Third, it must avoid synchronous writes (those that must complete immediately) so that many related pieces of data can be collected and written together. This section will provide a brief introduction to EFS and LFS and describe their advantages and disadvantages with respect to the goals.

We will describe the EFS approach in terms of its *bit map*, which keeps track of the free space, and its *cylinder groups*, which are used to group some things and separate others. Information is arranged on disk in terms of three units: blocks, partial blocks called *fragments*, and contiguous ranges of blocks called *clusters*. In principle, this allows a high degree of locality on disk. In practice, there are two potential problems. The first is that many write operations in EFS are small and/or synchronous. For example, it takes five distinct small I/O operations to create a new 1-kbyte file, of which two are synchronous. The second potential problem is that EFS does not currently take any special measures to preserve large free clusters. If a file system has been in use for a long time, will it become so fragmented that most free clusters are only a few blocks long?

We will then describe BSD LFS. Its log-based approach guarantees a high degree of locality, and synchronous writes have been almost entirely removed, so in theory it should be able to provide very high performance. However, it too has a potential problem with preserving free extents. LFS uses a copying garbage collector called the *cleaner* to regenerate large free extents. In many situations it may be possible to run the cleaner during idle periods so that it doesn't interfere with normal file accesses. However, during periods of high activity it may be necessary to run the cleaner concurrently with normal file accesses. Furthermore, depending on the file access patterns, cleaning can potentially be very expensive, and this can degrade system performance. (The paper will also discuss the potential risks of performing writes asynchronously.)

3. Performance with an Empty Disk

The paper will present benchmark results in two sections. The first section will present "best case" measurements, taken with an empty disk. In this case EFS will not suffer from fragmentation and LFS will not incur any cleaning overhead. We use the same three benchmarks as in the Seltzer paper, with a few additions. Our benchmarking setup consists of a SPARC-2 with 40MB of memory and several 3.5-inch SCSI drives, running BSD 4.4 (more details will be in the paper). Both the CPU and the disks are slightly faster in our configuration than in the Seltzer configuration, and the machine has more memory.

The first benchmark measures sequential I/O performance as a function of file size. Each run of the benchmark reads or writes several files (all the same size) from beginning to end. The total amount of data transferred in each run is 32 Mbytes; at the end of the run the effective bandwidth is computed. Different file sizes are used in different runs to produce a graph of bandwidth versus file size. The paper will include one graph for reads and four graphs for writes. The write graphs will measure the following scenarios:

(a) Create new files on an empty disk.

(b) Overwrite existing files on an otherwise empty disk.

(c) Overwrite existing files on an otherwise empty disk, but use `fsync` to force each file to be written synchronously.

(d) Create new files on an empty disk, but use a modified version of EFS that eliminates most of its synchronous writes.

Our measurements extend those of Seltzer et al. in three ways. First, Seltzer et al. measured only scenario (c). Second, we will present the data using log-log graphs so that the behavior for both small and large files is clearly visible (the linear scales in the Seltzer paper made it hard to see the performance for file sizes less than about 100 kbytes, and these sizes account for a substantial fraction of file activity in UNIX [1]). Third, our scenario (d) allows

us to separate the logging aspects of LFS from the asynchronous aspects, so that we can see how much benefit accrues from each.

The following table gives our preliminary data for reads and for writes under scenarios (a), (b), and (c). The block size was 8-K for both EFS and LFS; LFS does not yet support fragments.

File Size (Kbytes)	Read (Mbytes/sec)		Write (a) (Mbytes/sec)		Write (b) (Mbytes/sec)		Write (c) (Mbytes/sec)	
	EFS	LFS	EFS	LFS	EFS	LFS	EFS	LFS
4	0.72	0.62	0.10	0.73	0.21	0.33	0.20	0.30
8	1.11	1.21	0.20	1.42	0.33	1.01	0.33	0.92
16	1.68	1.24	0.30	1.43	0.50	1.22	0.49	1.08
32	1.87	1.32	0.38	1.43	0.69	1.32	0.66	1.15
64	2.16	1.35	0.90	1.46	0.82	1.37	0.37	1.20
128	1.03	1.50	0.69	1.40	0.62	1.04	0.60	0.93
256	1.37	1.89	0.85	1.46	0.86	1.22	0.82	1.14
512	1.53	1.90	1.03	1.48	1.10	1.30	1.05	1.23
1024	1.74	1.99	1.26	1.50	1.36	1.34	1.33	1.30
2048	1.91	2.05	1.45	1.48	1.52	1.39	1.44	1.33
4096	2.01	2.11	1.48	1.49	1.55	1.45	1.49	1.42
8192	2.04	2.16	1.51	1.65	1.51	1.48	1.50	1.47

The final paper will measure even smaller file sizes, down to 1KB, and write scenario (d) (LFS measurements for files smaller than 8 KB aren't meaningful until LFS is modified to use fragments). The paper will also contain a detailed explanation of the factors behind the observed performance, such as the use in EFS of 8-KB gaps between 64-KB clusters (which helps some files and hurts others), the loss of substantial write bandwidth in LFS because of limitations on the sizes of write operations, and the impact of indirect blocks, track buffering, and synchronous I/Os on performance.

Overall, this benchmark shows LFS to perform as well as EFS or better under most conditions. At its best, LFS runs at about 7x the speed of EFS when creating small files; at its worst, LFS runs a few percent slower than EFS for creating some large files, and it runs about 50% slower than EFS for reading some medium-sized files.

The second benchmark is the Andrew benchmark developed by M. Satyanarayanan [2]. It copies a small file hierarchy, scans the files in the hierarchy, then compiles and links them. This benchmark attempts to recreate the workload in typical software development environments. Although the Andrew benchmark is not heavily I/O bound, in many respects it is a more accurate reflection of actual UNIX usage than the preceding benchmark.

We ran this benchmark in the same single-user and multi-user configurations used by Seltzer, and our results are very similar to the Seltzer results. For the single-user case, LFS is about 10% faster than EFS because of its asynchronous writes. For the multi-user case, we found LFS to be slightly faster than EFS whereas Seltzer et al. found LFS to be slightly slower. However, we found a high variance between individual measurements in the multi-user case, which is much larger than the differences between EFS and LFS. Thus we conclude that the systems have about equal performance in the multi-user case.

The third benchmark is the transaction processing benchmark used by Seltzer, which is based on the industry standard TPC-B benchmark. The benchmark simulates the sort of

activity that would occur at an automatic teller machine, consisting of deposits and withdrawals for accounts in a large customer database. Most of the disk I/O consists of random read and write accesses to a large file containing the customer accounts.

Our preliminary results for this benchmark on an empty disk show that EFS is 50% slower than LFS. These results disagree with the original Seltzer measurements, where EFS was only about 20% slower than LFS. We are currently trying to understand these differences, and expect to have a full explanation for the final paper.

4. Performance with a Full Disk

Any file system that attempts to allocate large extents on disk will find its job more difficult as the disk fills up and there is less free space. The main question for EFS is whether free space will gradually fragment over time, so that the system cannot allocate full-size clusters. To answer this question, we have examined the free space on several production disks at Harvard that use the implementation of EFS in SunOS as described by McVoy [4]. Our preliminary results show that free space within a cylinder group is unevenly distributed: most of the free space is at the end of the cylinder group, and the free space at the beginning of the cylinder group is often fragmented. The EFS allocation policy tends to allocate files at the beginning of a cylinder group whenever possible, which both explains the free space distribution and suggests that it may be hard for the system to allocate full-size clusters.

For the final paper we will present these measurements in detail. We will also analyze the actual file layouts on disk to see how effective clustering is in practice and to see whether clustering degrades as a disk ages. The paper will also discuss possible improvements to the EFS layout algorithm to reduce fragmentation.

For LFS, there are two questions. First, how much work must the cleaner do to generate large free extents? And second, how much of that work can be hidden by cleaning when the file system is idle? These questions can only be answered by measuring real LFS systems in use over long periods of time, and the BSD implementation is not yet mature enough to make such measurements. Rosenblum measured the Sprite implementation of LFS and found the cleaning costs to be relatively low [5], but these measurements are based on limited usage. This paper will include one additional measurement of cleaning costs. We will run the transaction processing benchmark long enough for the cleaner to start running, and measure the degradation in throughput that occurs when the cleaner is running. Furthermore, we will vary the disk space utilization to produce a graph of performance versus disk utilization. We have not made this measurement yet because the cleaner is not working in the current version of BSD LFS; this will be fixed in time for the final paper.

Our LFS cleaner measurements will differ from Seltzer's in two ways. First, we will measure the cleaner over a range of disk utilizations, whereas Seltzer measured only a single disk utilization (at 80% disk utilization, the cleaner caused the benchmark to run 40% slower). Second, our measurements will use a tuned version of LFS that writes significantly less metadata to disk than the version used by Seltzer. This should reduce the cleaning costs relative to the Seltzer measurements.

5. Conclusions and Contributions

We expect to have three overall conclusions. The first conclusion is that LFS performs better relative to EFS in our measurements than in the Seltzer measurements. This is due in part to the expanded scope of our measurements and in part to performance improvements in

LFS. Our second conclusion is that EFS appears to be vulnerable to fragmentation of free space, which may cause production EFS file systems to perform worse than the clean file systems typically used for benchmarking. Our third conclusion is that the performance of LFS cannot be certified without more measurements of cleaning costs in LFS in production systems. We hope that the BSD implementation of LFS can serve as a vehicle for making these measurements.

Aside from the individual benchmark results, this paper will increase the overall understanding of both the EFS and LFS file systems. For example, this paper provides the most detailed performance analysis of EFS to date, and it is the first to measure fragmentation in production EFS systems.

6. Note to Reviewers

We apologize for the fact that our measurements are only about two-thirds complete at the time of submitting this abstract and that several more bugs still need to be fixed in LFS. However, we promise (!!) that they will all be done for the final paper. We think it's very important that the paper appear in the Summer '94 USENIX. Although Seltzer et al. pointed out that their results were preliminary, many people concluded from them that EFS is a clear winner over LFS. We'd like to publicize our additional measurements as soon as possible.

7. References

[1] Baker, M.G., et al. "Measurements of a Distributed File System." *Proc. 13th Symposium on Operating Systems Principles*, October, 1991, pp. 198-212.

[2] Howard, J.H. et al. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 51-81.

[3] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.

[4] McVoy, L.W. and Kleiman, S.R. "Extent-like Performance from a UNIX File System." *Proc. USENIX Winter Conference*, January 1991, pp. 33-43.

[5] Rosenblum, M. and Ousterhout, J. "The Design and Implementation of a Log-Structured File System." *ACM Transactions on Computer Systems*, Vol. 10, No. 1 (February 1992), pp. 26-52.

[6] Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C. "An Implementation of a Log-Structured File System for UNIX." *Proc. USENIX Winter Conference*, January 1993, pp. 307-326.