EREZ ZADOK, VASILY TARASOV, AND PRIYA SEHGAL

the case for specialized file systems, or, fighting file system obesity

Erez Zadok is an Associate Professor of Computer Science at Stony Brook University. His research interests involve file systems and operating systems, performance benchmarking and tuning, and green computing.

ezk@cs.sunysb.edu

Vasily Tarasov is a third year PhD student in the Computer Science Department of Stony Brook University. His scientific interests include operating systems, computer architectures, software engineering, green technologies, and services science.

vtaras@cs.sunysb.edu

Priya Sehgal is a Master's student at the Computer Science Department of Stony Brook University. Her research interests include operating systems, green computing, and computer architecture.

psehgal@fsl.cs.sunysb.edu

THE COMPLEXITY OF MODERN FILE systems has increased drastically in recent decades, and it keeps increasing. The ext2 file system in the Linux kernel has over 8,000 lines of code (LoC), ext3 doubles this, and ext4 doubles it again. A working development version of btrfs file system already has over 52,000 LoC, XFS is over 77,000 LoC, and other network-based file systems easily exceed 100,000 LoC. We believe that the amount of functionality provided in modern file systems is overkill for many of the usage scenarios, and this often hurts performance, energy efficiency, and even reliability [1]. Instead of creating gigantic general-purpose file systems that are hard to develop, debug, maintain, and tune for specific workloads, we propose to develop minimalistic file systems, each tuned for a particular case.

The growth of complexity is mainly caused by the expanding functionality integrated in a file system. In fact, the list of the features supported by modern file systems is impressive: journaling, B-tree-based search for objects, flexible data extents, access control lists (ACLs), extended attributes, encryption, checksumming, etc. ReiserFS allows programmers to write plugins for it; large file systems such as zfs and btrfs integrate complex storage pool management and deduplication. Features such as access-permission checks, hardlinks and symlinks, unlimited file name length and file size, as well as arbitrary directory depths, are no longer considered extra features: any self-respecting file system *must* support them. But should this "must" really be so strict?

Too Many Features

The large variety of features supported by modern file systems is, in part, the desire of file system developers to satisfy as many end users as possible. Depending on the specific situation, different characteristics are required from a file system. In emergency cases, reliability is the most important factor; for storing military data, security is crucial; enterprise servers require high performance; and in mobile platforms, energy efficiency plays an important role. When all corresponding features go into one file system, the final user obtains not only the functionality *they* require, but also all the functionality that *other* users may need. The number of tunable parameters of a file system grows proportionally to the functionality of a file system. Ext2 alone allows users to specify over 10 format options and over five mount options, resulting in at least a 10×5 = 50 parameter space; often, many of these options are not mutually exclusive, making the parameter space exponential (e.g., as large as 2^{50} in ext's case). It is extremely difficult for the end user to find an optional point in this space where performance is best. Our experiments show that the default format and mount parameters (often considered by the users as universally best) are up to 50% suboptimal and in some cases nearly an order of magnitude worse than a carefully tuned system [2].

From the developers' point of view it is hard to support, maintain, and develop large file systems. Integration of new features takes a lot of time: one needs to ensure that new functionality interoperates correctly with all other features that are already implemented in the file system. Consequently, the amount of effort spent on adding each new feature grows exponentially. The number of different code paths in a large file system is huge, leading to an exponential number of states to explore, which considerably complicates debugging and performance analysis. New developers spend a lot of time understanding the details of a complex file system before they can fix bugs or change file system behavior in some way.

Most of the users do not need all of the functionality incorporated in a modern file system at once. Actually, in certain cases only minimal file system functionality is enough. We held discussions with scientists who sought our help in designing efficient HDF-based file formats for complex images [3]. These scientists have diverse backgrounds—in neutron and X-ray imaging, molecular and structural biology, optical microscopy, macro-molecular imaging, 3D cryo-electron microscopy, and astrophysics—and use various clusters, with a range of file systems installed, analyzing terabyte-sized data sets on a daily basis. It was surprising to find out that they do not care about even basic features available in modern file systems. They do not use hardlinks, softlinks, or ACLs. The sequence of open-unlink-close (which is painful to implement in a file system) as well as directory renaming are very rare in their environments. They do not use deep directories: most files often reside in one flat directory or a shallow hierarchy. Files typically have known names of fixed length. The input and output file sizes in an experiment are often known in advance. Reliability features (e.g., journaling) are usually not crucial, because lost data can be regenerated easily by rerunning an experiment; for long-running experiments, periodic checkpointing is performed at the application level. With all this in mind, many scientists do not have a preferred file system, because most present file systems provide all the bare features the scientists require.

Simpler File Systems

We looked at all the difficulties related to developing and using the functionality in "obese" file systems, as well as the lack of necessity for the full set of features they offer. We propose creating minimalistic file systems with the functionality incorporated only on an as-needed basis. In this case the code size of a file system can be much smaller, which allows programmers to develop the file system quickly and then support it with less effort. Additionally, such file systems can be tuned more tightly for specific workloads, and without creating a myriad of parameters to confuse the end user. Note that inmany cases (e.g., the aforementioned scientists), users already know the target usage of the file system and the characteristics of the workloads they are running. In our recent work we showed that careful tuning of existing file systems can increase their performance and power efficiency by as much as a factor of nine [2]. Developing a specialized file system would increase these numbers even more.

Creating a new file system is not as hard as one might think. To demonstrate this, we conducted an experiment within the graduate Operating System class at Stony Brook University. Four teams of 2–3 first-year MS students developed a *very simple real file system* (VSRFS). The functionality was limited, but varied from group to group: fixed/variable number of files and file sizes, no directories vs. simple directories, support of extended attributes, time-stamp storing, etc. It took only 3–4 calendar weeks for the students to create a working file system, with code sizes of 1000–2000 LoC. We therefore hypothesize that file system development time is not linear with respect to the code size and that it is easier to develop many small file systems instead of a few larger, feature-rich file systems. To facilitate filesystem development more, one can take advantage of templates technology similar to the one used in FiST for automatic generation of stackable file systems [4]. Another alternative is to design file systems to be modular: minimal sets of features could be loaded on demand based on workload characteristics.

In conclusion, file systems have become kitchen sinks in recent years; they integrate many hard-to-implement features that many do not use. This fact complicates the development of file systems and makes them less efficient for specific usage. We think that the adoption of small custom file systems is a feasible alternative that facilitates development and increases the efficiency of future file systems.

REFERENCES

[1] V. Prabhakaran, N. Agrawal, L.N. Bairavasundaram, H.S. Gunawi, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. “IRON File Systems,” *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP ’05 (ACM Press, 2005), pp. 206–220.

[2] P. Sehgal, V. Tarasov, and E. Zadok, “Evaluating Performance and Energy in File System Server Workloads Extensions,” *Proceedings of FAST ’10: 8th USENIX Conference on File and Storage Technologies* (USENIX Association, 2010), forthcoming.

[3] The HDF Group, Hierarchical Data Format, 2009: www.hdfgroup.org.

[4] E. Zadok and J. Nieh, “FiST: A Language for Stackable File Systems,” *Proceedings of the 2000 USENIX Annual Technical Conference* (USENIX Association, 2000), pp. 55–70.