

CHRISTOPH HELLWIG

XFS: the big storage file system for Linux



Christoph Hellwig is a freelancer providing consulting, training, and, last but not least, contract programming for Linux storage and file systems. He has been working on Linux file systems since 2001 and is one of the most widely known developers in this area.

hch@lst.de

XFS IS A FILE SYSTEM THAT WAS DESIGNED from day one for computer systems with large numbers of CPUs and large disk arrays. It focuses on supporting large files and good streaming I/O performance. It also has some interesting administrative features not supported by other Linux file systems. This article gives some background information on why XFS was created and how it differs from the familiar Linux file systems. You may discover that XFS is just what your project needs instead of making do with the default Linux file system.

BACKGROUND AND HISTORY

For years the standard Linux file system was ext2, a straightforward Berkeley FFS derivative. At the end of the 1990s, several competitors suddenly appeared to fill the gap for a file system providing fast crash recovery and transactional integrity for metadata. The clear winner in mainstream Linux is ext3, which added journaling on top of ext2 without many additional changes [7].

XFS has been less well known to many average Linux users but has always been the state of the art at the very high end. XFS itself did not originate on Linux but was first released on IRIX, a UNIX variant for SGI workstations and servers, in December 1994, almost 15 years ago. Starting in 1999, XFS was ported to Linux as part of SGI's push to use Linux and Intel's Itanium processors as the way forward for its high-end supercomputing systems. Designed from the start for large multiprocessor systems and disk arrays [1] rather than for small, single-disk consumer workstations, it was for a long time positioned above the mainstream Linux market. Today even low-end workstations with a small number of CPU cores and disks come close to the limits of ext3 (see Table 1). While there is another adaption of the FFS concept called ext4 under development to mitigate these limits to a certain extent, it seems as though basic FFS design is close to maxed out.

To address these limits, ext3 is evolving into ext4 by incorporating features pioneered by XFS such as delayed allocations and extents. Even with these improvements taking the basic FFS design as far as it can go, it is difficult to match the scalability limits of XFS, which has been designed for large storage systems from day one. In a few years, btrfs, a new file system initiated by Oracle, will mature from development status to hopefully become the

new standard file system. As a new design that includes advanced management and self-healing features, btrfs will compete heavily with XFS on the lower end of the XFS market, but we will have to see how well it does on the extreme high end.

Today XFS is used by many well-known institutions, with CERN and Fermilab managing petabytes of storage for scientific experiments using XFS, and kernel.org serving the source code to the Linux kernel and many other projects from XFS file systems.

Limit	ext3	ext4	XFS
max file system size	16 TiB	16 TiB	16 EiB
max file size	2 TiB	16 TiB	8 EiB
max extent size	4 kiB	128 MiB	8 GiB
max extended attribute size	4 kiB	4 kiB	64 kiB
max inode number	2^{32}	2^{32}	2^{64}

(All numbers assume the maximum 4 kiB block size on x86 Linux systems.)

TABLE 1: FILE SYSTEM LIMITS FOR XFS, EXT3 AND EXT4

Space Allocation and Management

Each XFS file system is partitioned into regions called allocation groups (AGs). Allocation groups are somewhat similar to the block groups in ext3, but AGs are typically much larger than block groups and are used for scalability and parallelism rather than disk locality. Allocation groups are typically sized between 0.5 and 4 gigabytes and keep the size of the XFS data structures in a range where they can operate efficiently and in parallel [2].

Historical UNIX file systems, including ext3, use linear bitmaps to track free space, which is inefficient especially for larger contiguous allocations. XFS instead uses a pair of B+ trees in each allocation group. Each entry in the B+ tree nodes consists of a start block and length pair describing a free-space region. The first B+ tree is indexed by the starting block of the free region, and the other is indexed by the length of the free region. This double indexing allows the allocator to consider two goals for new data placement: locality to existing file data, and best fit into free space.

A similar extent concept is used for tracking the disk blocks assigned to each file. In addition to the start block on disk and the length of the contiguous range, the extent descriptor also contains two additional fields. The first one is the logical offset into the file, which allows for efficient sparse file support by skipping ranges that do not have blocks allocated to them. The second one is a simple one-bit flag to mark an extent as unwritten, a concept that will be explained later in this article.

For most files, a simple linear array of extent descriptors is embedded into the inode, avoiding additional metadata blocks and management overhead. For very large files or files containing many holes, the number of extents can be too large to fit directly into the inode.

In this case, extents are tracked by another B+ tree with its root in the inode. This tree is indexed by the offset into the file, which allows an extent descriptor for a given file offset to be found quickly, with no linear search overhead. Figure 1, showing the time needed to remove a large file such as an HD video or virtual machine image, demonstrates how management overhead can be reduced by using extents.

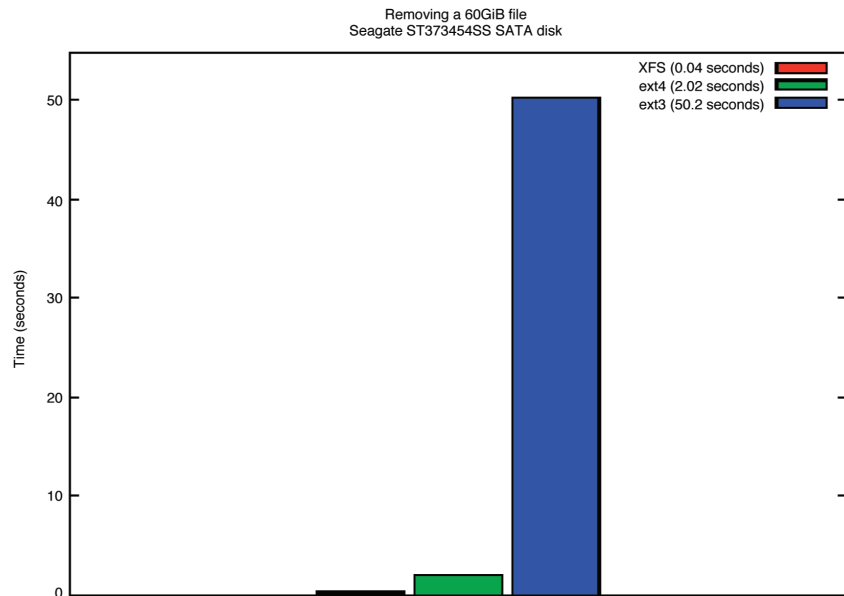


FIGURE 1: TIME SPENT REMOVING A VERY LARGE FILE

Inodes and Extended Attributes

The XFS inode consists of three parts: the inode core, the data fork, and the optional attribute fork. The inode core contains traditional UNIX inode metadata such as owner and group, number of blocks, timestamps, and a few XFS-specific additions such as project ID. The data fork contains the previously mentioned extent descriptors or the root of the extent map. The optional attribute fork contains the so-called extended attributes. The concept of extended attributes is not part of the Posix file system interface but is supported by all modern operating systems and file systems with slightly differing semantics. In Linux, extended attributes are simple name/value pairs assigned to a file that can be listed and read or written one attribute at a time. Extended attributes are used internally by Linux to implement access control lists (ACLs) and labels for SELinux, but they can also be used for storing arbitrary user metadata [3].

The attribute fork in XFS can either store extended attributes directly in the inode if the space required for the attributes is small enough, or use the same scheme of extent descriptors as described for the file data above to point to additional disk blocks. This allows XFS to support extended attribute sizes up to 64 kilobytes, while most other Linux file systems are limited to the size of a single disk block.

The size of the inode core is fixed, and the data and attribute forks share the remaining space in the inode, which is determined by the inode size chosen at file system creation time, ranging from 256 to 2048 bytes. For file systems that extensively use ACLs (e.g., for Windows file shares exported by Samba) or for file systems making extensive use of extended attributes, choosing a larger inode size can provide performance improvements, because this extra data can be stored in the inode and does not require reading additional data blocks.

Inodes in XFS are dynamically allocated, which means that, unlike many other Linux file systems, their location and number are not determined at mkfs time. This means that there is no need to predict the expected num-

ber of inodes when creating the file system, with the possibility of under- or overprovision. Because every block in the file system can now possibly contain inodes, an additional data structure is needed to keep track of inode locations and allocations. For this, each allocation group contains another B+ tree tracking the inodes allocated within it.

Because of this, XFS uses a sparse inode number scheme where inode numbers encode the location of the inode on disk. While this has advantages when looking up inodes, it also means that for large file systems, inode numbers can easily exceed the range encodable by a 32-bit integer. Despite Linux's having supported 64-bit-wide inode numbers for over 10 years, many user-space applications on 32-bit systems still cannot accommodate large inode numbers. Thus by default XFS limits the allocation of inodes to the first allocation groups, in order to ensure all inode numbers fit into 32 bits. This can have a significant performance impact, however, and can be disabled with the `inode64` mount option.

Directories

XFS supports two major forms of directories. If a directory contains only a few entries and is small enough to fit into the inode, a simple unsorted linear format can store all data inside the inode's data fork. The advantage of this format is that no external block is used and access to the directory is extremely fast, since it will already be completely cached in memory once it is accessed. Linear algorithms, however, do not scale to large directories with millions of entries. XFS thus again uses B+ trees to manage large directories. Compared to simple hashing schemes such as the `htree` option in `ext3` and `ext4`, a full B+ tree provides better ordering of `readdir` results and allows for returning unused blocks to the space allocator when a directory shrinks. The much improved ordering of `readdir` results can be seen in Figure 2, which compares the read rates of files in `readdir` order in a directory with 100,000 entries.

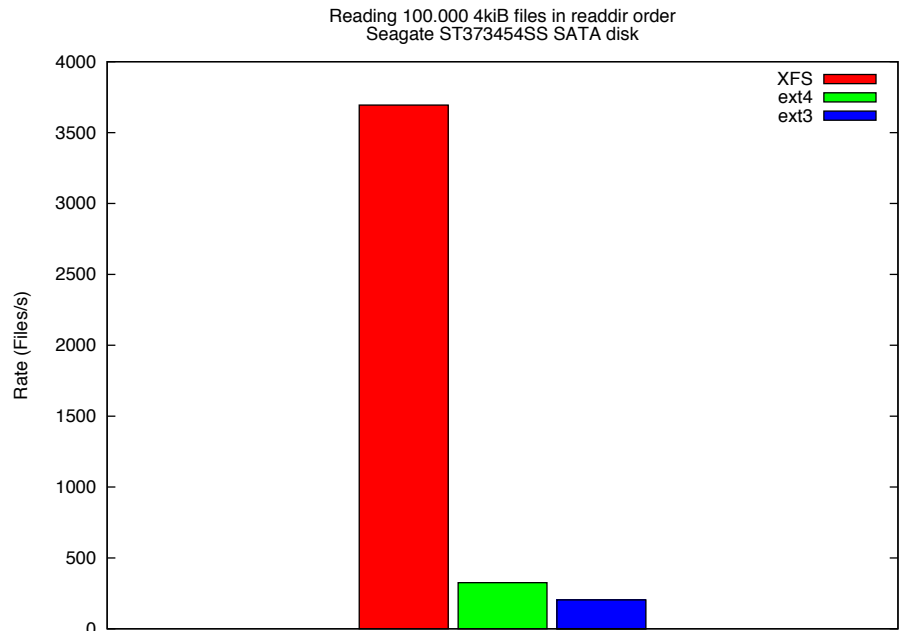


FIGURE 2: COMPARISON OF READING A LARGE (100,000 ENTRY) DIRECTORY, THEN READING EACH FILE

I/O Scalability

From day one, XFS has been designed to deal with high-performance disk subsystems, especially striped disk arrays with large aggregated bandwidth. When XFS was designed, “high performance” meant a few hundred megabytes per second, but 15 years later XFS still keeps up with aggregated bandwidth in the tens of gigabytes per second for a single file system instance [4].

To keep a RAID array busy, the file system should submit I/O requests that are large enough to span all disks in the array. In addition, I/O requests should be aligned to stripe boundaries where possible, to avoid read-modify-write cycles for common usage patterns. Because a single I/O can only be as large as a contiguous range of blocks, it is critical that files are allocated as contiguously as possible, to allow large I/O requests to be sent to the storage. The key to achieving large contiguous regions is a method known as “delayed allocation.” In delayed allocation, specific disk locations are not chosen when a buffered write is submitted; only in-memory reservations take place. Actual disk blocks are not chosen by the allocator until the data is sent to disk due to memory pressure, periodic write-backs, or an explicit sync request. With delayed allocation, there is a much better approximation of the actual size of the file when deciding about the block placement on disk. In the best case the whole file may be in memory and can be allocated in one contiguous region. In practice XFS tends to allocate contiguous regions of 50 to 100 GiB when performing large sequential I/O, even when multiple threads write to the file system at the same time [4].

While delayed allocations help with random write workloads if sufficiently large contiguous regions are filled before the flush to disk, there are still many workloads where this is not the case. To avoid fragmentation in pathological cases with random writes filling up a file very slowly (such as some HPC workloads or BitTorrent clients), XFS allows preallocation of blocks on disk to a file before actually writing to it. The preallocation just assigns data blocks to a file without touching the block contents. To avoid security problems with exposing stale data, preallocated extents are marked as unwritten, and any read from them will return zeros. Once data is written to unwritten extents, they are converted to normal extents, which incurs minimal performance overhead compared to a write to a normal allocated extent.

Figures 3 and 4 show some performance enhancement when XFS is compared to ext3, the old Linux standard file system, and to ext4, which adds delayed allocations and extents to it for sequential I/O workloads. On the other hand, Figure 5 shows that the performance for completely random I/O is not only really bad but also doesn't profit much from the XFS features.

Direct I/O

XFS provides a feature, called direct I/O, that provides the semantics of a UNIX raw device inside the file system namespace. Reads and writes to a file opened for direct I/O bypass the kernel file cache and go directly from the user buffer to the underlying I/O hardware. Bypassing the file cache offers the application full control over the I/O request size and caching policy. Avoiding the copy into the kernel address space reduces the CPU utilization for large I/O requests significantly. Thus direct I/O allows applications such as databases, which were traditionally using raw devices, to operate within the file system hierarchy.

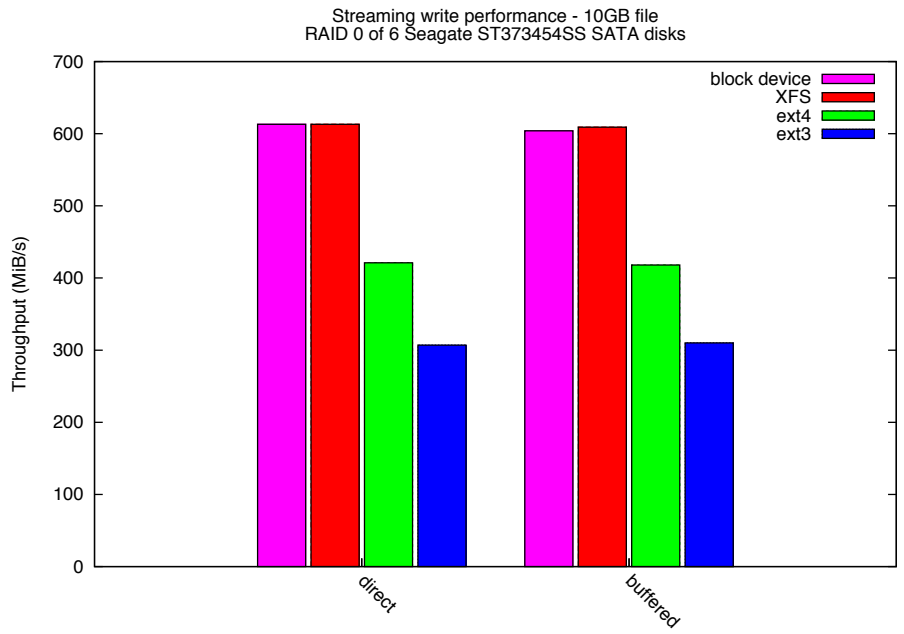


FIGURE 3: COMPARING BLOCK DEVICE, XFS, EXT4, AND EXT3 WHEN WRITING A 10 GB FILE

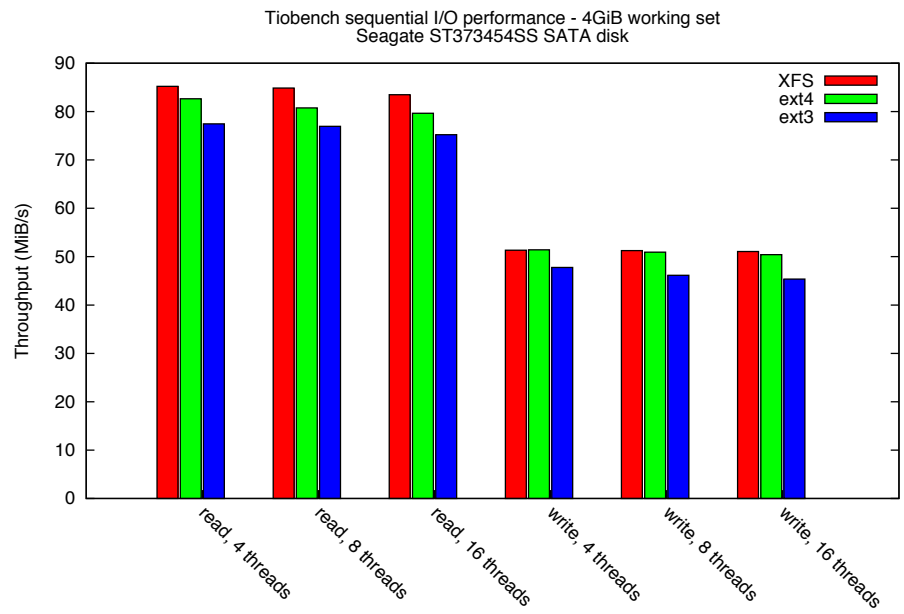


FIGURE 4: COMPARING SEQUENTIAL I/O PERFORMANCE BETWEEN XFS, EXT4, AND EXT3

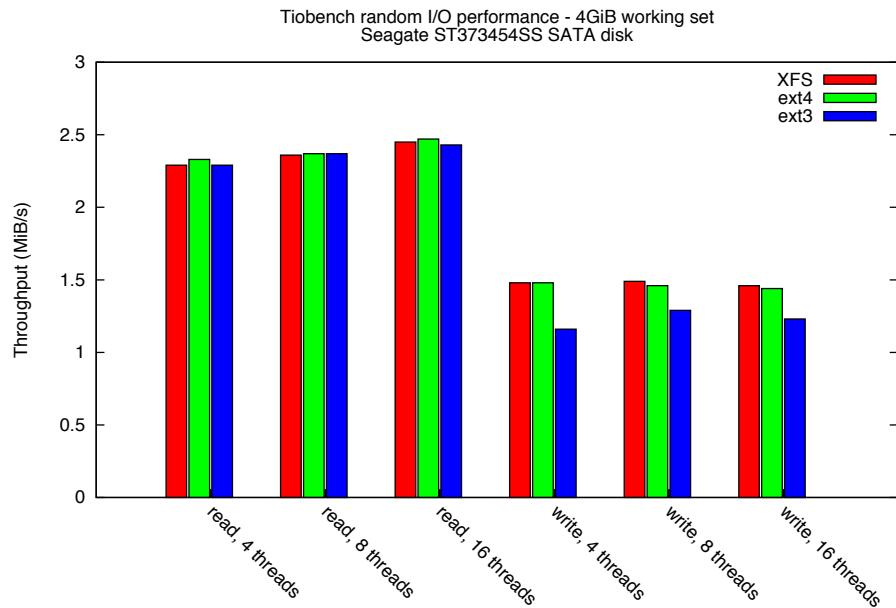


FIGURE 5: COMPARING RANDOM I/O PERFORMANCE BETWEEN XFS, EXT4, AND EXT3

Direct I/O has been adopted by all major Linux file systems, but the support outside of XFS is rather limited. While XFS guarantees the uncached I/O behavior under all circumstances, other file systems fall back to buffered I/O for many non-trivial cases such as appending writes, hole filling, or writing into preallocated blocks. A major semantic difference between direct I/O and buffered I/O in XFS is that XFS allows multiple parallel writers to files using direct I/O, instead of imposing the single-writer limit specified in Posix for buffered I/O. Serialization of I/O requests hitting the same region is left to the application, and thus allows databases to access a table in a single file in parallel from multiple threads or processes.

Crash Recovery

For today's large file systems, a full file system check on an unclean shutdown is not acceptable because it would take too long. To avoid the requirement for regular file system checks, XFS uses a write-ahead logging scheme that enables atomic updates of the file system. XFS only logs structural updates to the file system metadata, but not the actual user data, for which the Posix file system interface does not provide useful atomicity guarantees.

XFS logs every update to the file system data structures and does not batch changes from multiple transactions into a single log write, as is done by ext3. This means that XFS must write significantly more data to the log in case a single metadata structure gets modified again and again in short sequence (e.g., removing a large number of small files). To mitigate the impact of log writes to the system performance, an external log device can be used. With an external log the additional seeks on the main device are reduced, and the log can use the full sequential performance of the log device.

Unfortunately, transaction logging does not help to protect against hardware-induced errors. To deal with these problems, XFS has an offline file system checking and repair tool called `xfs_repair`. To deal with the ever growing disk sizes and worsening seek rates, `xfs_repair` has undergone a major overhaul in the past few years to perform efficient read-ahead and caching and to make use of multiple processors in SMP systems [6].

Disk Quotas

XFS provides an enhanced implementation of the BSD disk quotas. It supports the normal soft and hard limits for disk space usage and number of inodes as an integral part of the file system. Both the per-user and per-group quotas supported in BSD and other Linux file systems are supported. In addition to group quotas, XFS alternatively can support project quotas, where a project is an arbitrary integer identifier assigned by the system administrator. The project quota mechanism in XFS is used to implement directory tree quota, where a specified directory and all of the files and subdirectories below it are restricted to using a subset of the available space in the file system. For example, the sequence below restricts the size of the log files in /var/log to 1 gigabyte of space:

```
# mount -o prjquota /dev/sda6 /var
# echo 42:/var/log >> /etc/projects
# echo logfiles:42 >> /etc/projid
# xfs_quota -x -c 'project -s logfiles' /var
# xfs_quota -x -c 'limit -p bhard=1g logfiles' /var
```

Another enhancement in the XFS quota implementation is that the quota subsystem distinguishes between quota accounting and quota enforcement. Quota accounting must be turned on during mount time, while quota enforcement can be turned on and off at runtime. Using an XFS file system with quota accounting but no enforcement provides an efficient way to monitor disk usage. For this reason, XFS also accounts (but never enforces) quotas usage for the superuser.

The `xfs_quota` command [8] seen in the example above offers full access to all features in the XFS quota implementation. In addition, the standard BSD `quota` and `edquota` tools can be used to administer basic quota functionality.

Day-to-Day Use

A file system in use should be boring and mostly invisible to the system administrator and user. But to get to that state the file system must first be created. An XFS file system is created with the `mkfs.xfs` command, which is trivial to use:

```
# mkfs.xfs /dev/vg00/scratch
meta-data    =/dev/vg00/scratch  isize=256      agcount=4, agsize=1245184 blks
              =                               sectsz=512     attr=2
data         =                               bsize=4096    blocks=4980736, imaxpct=25
              =                               sunit=0       swidth=0 blks
naming       =version 2          bsize=4096    ascii-ci=0
log          =internal log      bsize=4096    blocks=2560, version=2
              =                               sectsz=512    sunit=0 blks, lazy-count=0
realtime     =none              extsz=4096    blocks=0, rtextents=0
```

As seen above, the `mkfs.xfs` command returns the geometry information for the file system to make sure all parameters are set correctly. There are not many parameters that must be manually set for normal use. For software RAID arrays, XFS already extracts the proper stripe size and alignment parameters from the underlying device, but if that is not possible (e.g., for hardware RAID controllers), the parameters can be set manually. The following example creates a file system aligned correctly for a RAID5 array with 8+1 disks and a stripe unit of 256 kiB:

```
# mkfs.xfs -d su=256k,sw=8 /dev/sdf
```

Other interesting options include using an external log device and changing the inode size. The example below creates a file system with 2 kiB-sized inodes and an external log device:

```
# mkfs.xfs -i size=2048 -l logdev=/dev/vg00/logdev /dev/vg00/home
```

For more details, see the `mkfs.xfs` man page and the XFS training material [5].

A command worth note is `xfs_fsr`. FSR stands for file system reorganizer and is the XFS equivalent to the Windows defrag tool. It allows defragmentation of the extent lists of all files in a file system and can be run in background from cron. It may also be used on a single file.

Although all normal backup applications can be used for XFS file systems, the `xfsdump` command is specifically designed for XFS backup. Unlike traditional dump tools such as `dumpe2fs` for ext2 and ext3, `xfsdump` uses a special API to perform I/O based on file handles similar to those used in the NFS over the wire protocol. That way, `xfsdump` does not suffer from the inconsistent device snapshots on the raw block device that plague traditional dump tools. The `xfsdump` command can perform backups to regular files and tapes on local and remote systems, and it supports incremental backups with a sophisticated inventory management system.

XFS file systems can be grown while mounted using the `xfs_growfs` command, but there is not yet the ability to shrink.

Conclusion

This article gave a quick overview of the features of XFS, the Linux file system for large storage systems. I hope it clearly explains why Linux needs a file system that differs from the default and also shows the benefits of a file system designed for large storage from day one.

ACKNOWLEDGMENTS

I would like to thank Eric Sandeen for reviewing this article carefully.

REFERENCES

- [1] Adam Sweeney et al., “Scalability in the XFS File System,” *Proceedings of the USENIX 1996 Annual Technical Conference*.
- [2] Silicon Graphics Inc., XFS Filesystem Structure, 2nd edition, http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf.
- [3] Linux attr(5) man page: <http://linux.die.net/man/5/attr>.
- [4] Dave Chinner and Jeremy Higdon, “Exploring High Bandwidth Filesystems on Large Systems,” *Proceedings of the Ottawa Linux Symposium 2006*: <http://oss.sgi.com/projects/xfs/papers/ols2006/ols-2006-paper.pdf>.
- [5] Silicon Graphics Inc., XFS Overview and Internals: <http://oss.sgi.com/projects/xfs/training/index.html>.
- [6] Dave Chinner and Barry Naujok, Fixing XFS Filesystems Faster: http://mirror.linux.org.au/pub/linux.conf.au/2008/slides/135-fixing_xfs_faster.pdf.
- [7] Dr. Stephen Tweedie, “EXT3, Journaling Filesystem,” transcript of a presentation at the Ottawa Linux Symposium 2000: <http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html>.
- [8] `xfs_quota(8)`—Linux manpage: http://linux.die.net/man/8/xfs_quota.