XFS for Linux

Christoph Hellwig

Russell Cattelan

Steve Lord

Jim Mostek

Table of Contents

Abstract

The limitations of traditional file systems were becoming evident as new application demands arose and larger hardware became available for SGI's MIPS-based systems and later for Intel-based Linux systems. This paper describes advanced and unique features of SGI's XFS filesystem for IRIX and Linux, lessons learned during development and porting of XFS and explains the differences between the IRIX and Linux implementations of XFS.

Introduction to XFS

In the early 1990s, SGI realized its existing file system, EFS (Extent File System) would be inadequate to support the new application demands arising from the increased disk capacity, bandwidth, and parallelism available on its systems. Applications in film and video, supercomputing, and huge databases all required performance and capacities beyond what EFS, with a design similar to the Berkeley Fast File System, could provide. EFS limitations were similar to those found in Linux file systems until a few years ago: small file system sizes (8 gigabytes), small file sizes (2 gigabytes), statically allocated metadata, and slow recovery times using fsck. To address these issues in EFS, in 1994 SGI released an advanced, journaled file system on IRIX; this file system was called XFS. Since that time, XFS has proven itself in production as a fast, highly scalable file system suitable for computer systems ranging from the desktop to supercomputers. To help address these same issues in Linux SGI has made XFS technology available for Linux under the GNU General Public License (GPL).

Features

XFS uses B+ trees extensively in place of traditional linear file system structures. B+ trees provide an efficient indexing method that is used to rapidly locate free space, to index directory entries, to manage file extents, and to keep track of the locations of file index information within the file system. XFS is a fully 64-bit file system. Most of the global counters in the system are 64-bits in length, as are the addresses used for each disk block and the unique number assigned to each file (the inode number). A single file system can theoretically be as large as 18 million terabytes. The file system is partitioned into regions called Allocation Groups (AG). Like UFS cylinder groups, each AG manages its own free space and inodes. The primary purpose of Allocation Groups is to provide scalability and parallelism within the file system. This partitioning also limits the size of the structures needed to track this information and allows the internal pointers to be 32-bits. AGs typically range in size from 0.5 to 4GB. Files and directories are not limited to allocating space within a single AG.

XFS has a variety of sophisticated support utilities to enhance its usability. These include fast mkfs (make a file system), dump and restore utilities for backup, xfsdb (XFS debug), xfscheck (XFS check), and xfsrepair to perform file system checking and repairing. The xfs fsr utility defragments existing XFS file systems. The xfs bmap utility can be used to interpret the metadata layouts for an XFS file system. The growfs utility allows XFS file systems to be enlarged on-line.

Architecture

The high level structure of XFS is similar to a conventional file system with the addition of a transaction manager. XFS supports all of the standard Unix file interfaces and is entirely POSIX- and XPG4-compliant. It sits below the vnode interface in the IRIX kernel and takes full advantage of services provided by the kernel, including the buffer/page cache, the directory name lookup cache, and the dynamic vnode cache.

XFS is modularized into several parts, each of which is responsible for a separate piece of the file system's functionality. The central and most important piece of the file system is the space manager. This module manages the file system free space, the allocation of inodes, and the allocation of space within individual files. The I/O manager is responsible for satisfying file I/O requests and depends on the space manager for allocating and keeping track of space for files. The directory manager implements the XFS file system name space. The buffer cache is used by all of these pieces to cache the contents of frequently accessed blocks from the underlying volume in memory. It is an integrated page and file cache shared by all file systems in the kernel. The transaction manager is used by the other pieces of the file system to make all updates to the metadata of the file system atomic. This enables the quick recovery of the file system after a crash. While the XFS implementation is modular, it is also large and complex. The current implementation is over 110,000 lines of C code (not including the buffer cache or user-level XFS utilities); in contrast, the EFS implementation is approximately 12,000 lines.

Journaling

XFS journals metadata updates by first writing them to an in-core log buffer, then asynchronously writing log buffers to the on-disk log. The on-disk log is a circular buffer: new log entries are written to the head of the log, and old log entries are removed from the tail once the inplace metadata updates occur. After a crash, the on-disk log is read by the recovery code which is called during a mount operation. XFS metadata modifications use transactions: create, remove, link, unlink, allocate, truncate, and rename operations all require transactions. This means the operation, from the standpoint of the file system on-disk metadata, either never starts or always completes. These operations are never partially completed on-disk: they either happened or they didn't. Transactional semantics are required for databases, but until recently have not been considered necessary for file systems. This is likely to change, as huge disks and file systems require the fast recovery and good performance journaling can provide. An important aspect of journaling is write-ahead logging: metadata objects are pinned in kernel memory while the transaction is being committed to the on-disk log. The metadata is unpinned once the in-core log has been written to the on-disk log. Note that multiple transactions may be in each in-core log buffer. Multiple in-core log buffers allow for transactions when another buffer is being written. Each transaction requires space reservation from the log system (i.e., the maximum number of blocks this transaction may need to write). All metadata objects modified by an operation, e.g., create, must be contained in one transaction.

Porting XFS to Linux

The vnode/vfs interface in IRIX

The vnode/vfs file system interface was developed in the mid-80s to allow the UNIX kernel to support multiple file systems simultaneously. Up to that time, UNIX kernels typically supported a single file system that was bolted directly into the kernel internals. With the advent of local area networks in the mid-80s, file sharing across networks became possible, and it was necessary to allow multiple file system types to be installed into the kernel. The vnode/vfs interface separates the file-system-independent vnode from the file-system-dependent inode. This separation allows new file systems to re-use existing file-system-independent code, and, at least in theory, to be developed independently from the internal kernel data structures. IRIX and XFS use the following major structures to interface between the file system and the rest of the IRIX OS components:

- vfs Virtual File System structure.
- vnode Virtual node (as opposed to inode)
- bhv desc behaviors are used for file system stacking
- buf used as an interface to store data in memory (to and from disk)
- xfs mount top-level per XFS file system structure
- xfs inode top-level per XFS file structure.

The vnode structure points at the first behavior in the chain of file systems handling the file associated with this vnode. The behavior also points to the function vector, xfs vnodeops, which contains all the file-system-specific routines at the file level. In IRIX, the vnodeops contains more than 57 routines which can be invoked on a "file". These routines cover many functions such as create, remove, read, write, open, close, and others.

Mapping the vnode/vfs interface to Linux

Changing XFS to fit directly into the Linux VFS interface would require significant changes to a large portion of the XFS codebase. The current source code organization would need to be significantly changed and code charing between the IRIX and Linux versions of XFS would become much more difficult. The alternative is to integrate the vnode and vfs object as private file-system-dependent data in the struct inode and struct super block data in Linux. This approach introduces a translation layer between the XFS code and the Linux VFS interface which translate Linux VFS calls into the equivalent XFS vnode operations. The XFS vnode itself is attached to the private data area of the Linux inode, while the XFS vfs object is attached to the private data area of the Linux superblock.

In the initial Linux port of XFS the vnode and vfs operations remained almost unchanged from IRIX and the translation layer, called linvfs, had to do a certain amount of argument and semantics remapping. For example in IRIX the read/write entry points use the uio structures that allow to pass multiple I/O requests in one call to the filesystems while the Linux read/write entry points use a much simpler scheme with one I/O request per call. In later revisions of XFS for Linux some vnode/vfs operations where changed to fit the Linux model better. For example the lookup, create and rename VOPs now pass down the Linux dentry structure that describes a directory entry in all it’s details instead of just the name as in IRIX. This allows getting rid of superfluous internal lookup calls and access/race checks already handled in the generic Linux code. A result of these changes is that more than 2,000 lines of code that were required on IRIX could be removed from the Linux XFS.

Another example are the already mentioned read/write entry points that got simplified to match their Linux counterparts.

fcntl Versus ioctl in IRIX and Linux

In IRIX, XFS supports approximately 20 special fcntl interfaces used for space pre-allocation, extent retrieval, extended file information, etc. In addition, IRIX XFS has about a dozen special system call interfaces, all implemented via the special syssgi system call. These interfaces are used for operations such as growing the file system or retrieving internal file system information.

The Linux file system interface has no fcntl operation. The only supported fcntl calls on Linux are file locking calls. We proposed to the Linux community that a fnctl file operation be added. After extensive discussion, it was decided to use the existing ioctl operation, linvfs_ioctl, and all of the fcntl and syssgi usages have been converted into ioctls. A shortcoming to the ioctl approach is in the semantics of an ioctl to block or character special devices which reside within the file system: In these cases, the device driver's ioctl routine will be used rather than the file system's. Outside of that, porting the fcntl and syssgi interfaces to ioctl's has been straightforward.

IRIX XFS creds and Linux

In older UNIX systems, the file system code used the current process's data structures to determine the user's credentials such as uid, gid, capabilities, etc. The VFS/vnode interface removed this code and introduced a cred structure which is passed to certain file system operations such as create and lookup. The file system uses this information to determine permissions and ownership.

XFS was written using the VOP/vnode interface, so it regularly uses cred structures. One of the more prevalent cred usages on IRIX XFS is get_current_cred, which returns this structure for the current process.

Linux is similar to older UNIX implementations in that file systems are expected to look directly at the task structure to determine the current process's credentials. Linux does not utilize a cred structure.

In porting XFS to Linux, we first attempted to map the various cred fields onto the corresponding task fields. This had the undesired side-effect of producing code that utilized a cred pointer that in actuality was pointing at a task. This was determined to be unacceptable.

We then considered implementing a complete cred infrastructure, which would include a pool of active creds, cred setup, teardown, lookup, etc. It was determined that this would require too much overhead.

In looking at the Linux code, we saw that all of the access/permission work occurs above the file system dependent code, so having a cred is important only on creation. We then examined our own internal usage of cred fields in XFS, and found that more often than not, a cred was passed down through a VOP_, and never used. The few places that did use a cred field were changed to use the current task structure in Linux.

Early versions of the XFS Linux port still passed a cred address on the VOPs, but we changed the linvfs later to always pass NULL into the cred arguments.

In addition to these cred changes, we have removed many access checks from the XFS code since these are now performed at a higher layer and are redundant in the file system dependent code

XFS Caching and I/O

When XFS was first implemented within IRIX, the buffer cache was enhanced in a number of ways to better support XFS, both for better file I/O performance and for better journaling performance. The IRIX implementation of XFS depends on this buffer cache functionality for several key facilities.

First, the buffer cache allows XFS to store file data which has been written by an application without first allocating space on disk. The routines which flush delayed writes are prepared to call back into XFS, when necessary, to get XFS to assign disk addresses for such blocks when it is time to flush the blocks to disk. Since delayed allocation means that XFS can determine if a large number of blocks have been written before it allocates space, XFS is able to allocate large extents for large files, without having to reallocate or fragment storage when writing small files. This facility allows XFS to optimize transfer sizes for writes, so that writes can proceed at close to the maximum speed of the disk, even if the application does its write operations in small blocks. In addition, if a file is removed and its written data is still in delayed allocation extents, the data can be discarded without ever allocating disk space.

Second, the buffer cache provides a reservation scheme, so that blocks with delayed allocation will not result in deadlock. If too much of the available memory is used for delayed allocation, a deadlock on the memory occurs when trying to do conversion from delayed to real allocations. The deadlock can occur since the conversion requires metadata reads and writes which need available memory.

Third, the buffer cache and the interface to disk drivers support the use of a single buffer object to refer to as much as an entire disk extent, even if the extent is very large and the buffered pages in memory are not contiguous. This is important for high performance, since allocating, initializing, and processing a control block for each disk block in, for example, a 7 MB HDTV video frame, would represent a large amount of processor overhead, particularly when one considers the cost of cache misses on modern processors. XFS has been able to deliver 7 GB/second from a single file on an SGI Origin 2000 system, so the overhead of processing millions of control blocks per second is of practical significance.

Fourth, the buffer cache supports "pinning" buffered storage in memory, which means that the affected buffers will not be written to disk until they have been "unpinned". XFS uses a write-ahead log protocol for metadata writes, which means XFS writes a log entry containing the desired after-image before updating the actual on disk metadata. On recovery, XFS just applies after-images from the log (in case some of the metadata writes were not completed). In order to avoid having to force the log before updating metadata, XFS "pins" modified metadata pages in memory. Such pages must count against the memory reservation (just as do delayed allocation pages). XFS pins a metadata page before updating it, logs the updates, and then unpins the page when the relevant log entries have been written to disk. Since the log is usually written lazily, this in effect provides group commit of metadata updates.

The pagebuf Module

Our approach to porting XFS has included adding pagebuf, a layered buffer cache module on top of the Linux page cache. This allows XFS to act on extent-sized aggregates. Key to this approach is the pagebuf structure, which is the major structure of the pagebuf layer. The pagebuf objects implemented by this module include a list of physical pages associated with the pagebuf, plus the device information needed to perform I/O.

In earlier versions of XFS for Linux we were experimenting with a new device request interface, so that we can queue one of these pagebuf objects directly to a device, rather than having to create and queue a large number of single-block buffer_head objects for each logical I/O request. These extensions have been superceeded by the Linux 2.5 block layer rewrite that allows the submission of multi-page bio requests; current XFS versions for Linux 2.4 create and queue buffer_head objects to perform pagebuf I/O.

A key goal for the layered buffer cache module is that its objects be strictly temporary, so that they are discarded when released by the file system, with all persistent data held purely in the page cache. This avoids creating yet another class of permanent system object, with separate locking and resource management issues. The IRIX buffer cache implementation has about 11000 lines of very complex code. By relying purely on the page cache for buffering, we avoid most of the complexity, particularly in regard to locking and resource management, of hybrid page and buffer caches, at the cost of having to pay careful attention to efficient algorithms for assembling large buffers from pages.

Delayed Allocation of Disk Space for Cached Writes

Allocating space when appending to a file slows down writes, since reliable metadata updates (to record extent allocations) result in extra writes. Also, incremental allocations can produce too-small extents, if new extents are allocated each time a small amount of data is appended to a file (as when many processes append to a log file). Delayed allocation reserves disk space for a write but does not allocate any particular space; it simply buffers the write in the page cache. Later, when pages are flushed to disk, the page writing path must ask the file system to do the actual allocation. Also, to allow for optimal extent allocations and optimal write performance, the page writing path must collect adjacent dirty pages ("cluster" them) and write them as a unit.

Since allocation of disk space may be required in the page writing path when delayed allocation is present, and such allocation may require the use of temporary storage for metadata I/O operations, some provision must be made to avoid memory deadlocks. The delayed allocation path for writes must make use of a main memory reservation system, which will limit the aggregate amount of memory used for dirty pages for which disk space has not been allocated, so that there will always be some minimum amount of space free to allow allocations to proceed. Any other non-preemptible memory allocations, such as kernel working storage pages, must be counted against the reservation limit, so that the remaining memory is genuinely available.

File I/O

Early versions of XFS for Linux used an I/O path different from the normal Linux filesystems, basically a stripped down version of the IRIX XFS I/O code sitting ontop of the pagebuf layer. In the XFS/Linux 1.2 release the buffered I/O path has been completely rewritten to use the generic Linux I/O path as much as possible but still providing XFS-unique features such as delayed allocated writes and clustered writeout.

XFS is now using the generic read/write entry points for pagecache-based filesystems (generic_file_read/generic_file_write) but wraps them with XFS-specific functionality such as dealing with DMAPI callbacks and XFS-specific locking. This means all hard work for file I/O is done by the address_space operations where XFS against uses the generic versions for the readpage (read pages into memory), prepare_write and commit_write (copy file data into the pagecache and mark it for flushing) operations. The writepage operation that is responsible for flushing pagecache data to disk is the heart of the XFS I/O path and completely different from the generic code to handle delayed allocated writes and clustered writeout.

A problem in Linux 2.4 is that the buffer layer directly writes out the buffers on it's LRU list to disk without any filesystem interaction, which makes the delayed disk block allocation and clustered writeout features of XFS impossible.

To address this issue the Linux buffer layer has been modified to call back into the filesystems writepage method for a buffer marked for delayed allocation instead of directly writing it out. These changes are localized to one file (fs/buffer.c) and provide XFS-compatible semantics with minimal intrusion.

Linux 2.5 already performs all pagecache writeout through the filesystems writepage (or writepages) entry points so no modification was nessecary.

Direct I/O

Small files which are frequently referenced are best kept in cache. Huge files, such as image and streaming media files and scientific data files, are best not cached, since blocks will always be replaced before being reused. Direct I/O is raw I/O for files: I/O directly to or from user buffers, with no data copying. The page cache must cooperate with direct I/O, so that any pages, which are cached and are modified, are read from memory, and so that writes update cached pages.

Direct I/O and raw I/O avoid copying, by addressing user pages directly. The application promises not to change the buffer during a write. The physical pages are locked in place for the duration of the I/O, via Linux kernel methods (kiobufs in 2.4, get_user_pages in 2.5).

Any dirty pages in the page cache must be flushed to disk before issuing direct I/O. The normal case will find no pages in the cache, and this can be efficiently tested by checking the inode. Once the pagebuf is assembled, the I/O path is largely common with the normal file I/O path, except that the write is never delayed and allocation is never delayed.

Direct I/O is indicated at open( ) time by using the O_DIRECT flag. Usually the needed space for the file is pre-allocated using an XFS ioctl call to ensure maximum performance.

Unlike other Linux filesystems XFS allows multiple O_DIRECT writes to the same inode to happen in parallel.

Volume Management Layers

The integration of existing Linux volume managers with the XFS file system has created some issues for the XFS port to Linux.

Traditional Linux file systems have been written to account for the requirements of the block device interface, ll_rw_block( ). ll_rw_block accepts a list of fixed size I/O requests. For any given block device on a system, the basic unit of I/O operation is set when the device is opened. This size is then a fixed length of I/O for that device. The current implementations of Linux volume managers have keyed off this fixed size I/O and utilize an I/O dispatcher algorithm.

By using a fixed I/O length, the amount of "math" that is needed is significantly less than what it would be if the I/O length were not fixed. All I/O requests from a file system will be of the same size, as both metadata and user

data is of fixed size. Therefore, all underlying devices of a logical volume must accept I/O requests of the same size. All that the volume manager needs to do for any I/O request is to determine which device in the logical volume the I/O should go to and recalculate the start block of the new device. Each I/O request is directed wholly to a new device.

The XFS file system, however, does not assume fixed size I/O. In an XFS file system, metadata can be anywhere from 512 bytes to over 8 Kbytes. The basic minimum I/O size for user data is set at file system creation time, with a typical installation using 4 Kbytes. One of the XFS design goals was to aggregate I/O together, creating large sequential I/O.

This feature of XFS created a problem for Linux volume managers, since the XFS file system can hand an I/O request off to a block device driver specifying the start position and length, which is not always fixed. A logical volume manager is just another block device to XFS, and a logical volume manager working in conjunction with XFS needs to be able to handle whatever size I/O request XFS desires, to some reasonable limit.

One of the options to address this problem in XFS is to change the on disk format of the file system to use a fixed size. This would render the Linux version of XFS incompatible with the current IRIX implementations, however, and so it was deemed unacceptable, just as making different versions of NFS would be unacceptable.

The Linux 2.4 version of XFS working around the problem of variable I/O request size by opening a device with the minimum I/O size needed: 512 bytes and performing operations in multiples of this size anyway unless the underlying device is in a blacklist of volume manages that can’t handle these I/O request.

In Linux 2.5 the new block layer interface allows to submit variable-sized requests and the burden of splitting them up is up to the actual volume managers.

Moving XFS to Open Source

For XFS to be a viable alternative file system for the open source community, it was deemed essential that XFS be released with a license at least compatible with the GNU General Public License (GPL).

The IRIX operating system in which XFS was originally developed has evolved over a long period of time, and includes assorted code bases with a variety of associated third party license agreements. For the most part these agreements are in conflict with the terms and conditions of the GNU General Public License.

The initial XFS project was an SGI initiative that started with a top-to-bottom file system design rather than an extension of an existing file system. Based upon the assertions of the original developers and the unique features of XFS, there was a priori a low probability of overlap between the XFS code and the portions of IRIX to which third-party licenses might apply. However it was still necessary to establish that the XFS source code to be open sourced was free of all encumbrances, including any associated with terms and conditions of third party licenses applying to parts of IRIX.

SGI’s objectives were:

- to ensure the absence of any intellectual property infringements
- to establish the likely derivation history to ensure the absence of any code subject to third party terms and conditions

This was a major undertaking; as the initial release of buildable XFS open source contained some 400 files and 199,000 lines of source. The process was long, but relatively straightforward, and encumbrance relief was usually by removal of code. The encumbrance review was a combined effort for SGI's Legal and Engineering organizations. The comments here will be confined to the technical issues and techniques used by the engineers.

The Encumbrance Review Process

We were faced with making comparisons across several large code bases, and in particular UNIX System V Release 4.2-MP, BSD4.3 NET/2, BSD4.4-lite and the open source version of XFS. We performed the following tasks:

- Historical survey

  We contacted as many as possible of the original XFS developers and subsequent significant maintainers, and asked a series of questions. This information was most useful as guideposts or to corroborate conclusions from the other parts of the review.

- Keyword search (all case insensitive)

  In each of the non-XFS code bases, we searched for keywords associated with unique XFS concepts or technologies (e.g. journal, transaction, etc.). In the XFS code base, we searched for keywords associated with ownership, concepts and technologies in the non-XFS code bases (e.g. att, berkeley, etc.).

- Literal copy check

  Using a specially built tool, we compared every line of each XFS source file against all of the source in the non-XFS code bases. The comparison ignored white space, and filtered out some commonly occurring strings (e.g. matching "i++;" is never going to be helpful).

- Symbol matching

  We developed tools to post-process the ASCII format databases from cscope to generate lists of symbols and their associated generic type (function, global identifier, macro, struct, union, enum, struct/union/enum member, typedef, etc.). In each XFS source file the symbols were extracted and compared against all symbols found in all the non-XFS code bases. A match occurred when the same symbol name and type was found in two different source files. Some post-processing of the symbols was done to include plausible name transformations, e.g. adding an "xfs_" prefix, or removal of all underscores, etc.

- Prototype matching

  Starting with a variant of the mkproto tool, we scanned the source code to extract ANSI C prototypes. Based on some equivalence classes, "similar" types were mapped to a smaller number of base types, and then the prototypes compared. A match occurred when the type of the function and the number and type of the arguments agreed.

- Check for similarity of function, design, concept or implementation.

  This process is based upon an understanding, and a study, of the source code. In the XFS code, for each source file, or feature implemented in a source file, or group of source files implementing a feature, it was necessary to conduct a review of the implementation of any similar source file or feature in each of the non-XFS code bases. The objective of this review is to determine if an issue of potential encumbrance arises as a consequence of similarity in the function, implementation with respect to algorithms, source code structure, etc.

- Check for evidence of license agreements.

  We examined the XFS code (especially in comments) to identify any references to relevant copyrights or license agreements.

In all of the steps above, the outcome was a list of possible matches. For each match, it was necessary to establish in the context of the matches (in one or more files), if there was a real encumbrance issue. We used a modified version of the tkdiff tool to graphically highlight the areas of the "match" without the visual confusion of all of the minutiae of the line-by-line differences. However, the classification of the matches was ultimately a manual process, based on the professional and technical skills of the engineers.

Encumbrance Relief

Especially in view of the size of the XFS source, a very small number of real encumbrance issues were identified. In all cases the relief was relatively straightforward, with removal of code required for IRIX, but not for Linux, being the most common technique.