State of the Art: Where we are with the Ext3 filesystem

Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya
IBM Linux Technology Center
{cmm, theotso, pbadari}@us.ibm.com, suparna@in.ibm.com
Andreas Dilger, Alex Tomas,
Cluster Filesystem Inc.
adilger@clusterfs.com, alex@clusterfs.com

Abstract

The ext2 and ext3 filesystems on Linux® are used by a very large number of users. This is due to its reputation of dependability, robustness, backwards and forwards compatibility, rather than that of being the state of the art in filesystem technology. Over the last few years, however, there has been a significant amount of development effort towards making ext3 an outstanding filesystem, while retaining these crucial advantages. In this paper, we discuss those features that have been accepted in the mainline Linux 2.6 kernel, including directory indexing, block reservation, and online resizing. We also discuss those features that have been implemented but are yet to be incorporated into the mainline kernel: extent maps, delayed allocation, and multiple block allocation. We will then examine the performance improvements from Linux 2.4 ext3 filesystem to Linux 2.6 ext3 filesystem using industry-standard benchmarks features. Finally, we will touch upon some potential future work which is still under discussion by the ext2/3 developers.

1 Introduction

Although the ext2 filesystem[4] was not the first filesystem used by Linux and while other filesystems have attempted to lay claim to being the native Linux filesystem (for example, when Frank Xia attempted to rename xiafs to linuxfs), nevertheless most would consider the ext2/3 filesystem as most deserving of this distinction. Why is this? Why have so many system administrations and users put their trust in the ext2/3 filesystem?

There are many possible explanations, including the fact that the filesystem has a large and diverse developer community. However, in our opinion, robustness (even in the face of hardware-induced corruption) and backwards compatibility are among the most important reasons why the ext2/3 filesystem has a large and loyal user community. Many filesystems have the unfortunate attribute of being *fragile*. That is, the corruption of a single, unlucky, block can be magnified to cause a loss of far larger amounts of data than might be expected. A fundamental design principle of the ext2/3 filesystem is to avoid fragile data structures by limiting the damage that could be caused by the loss of a single critical block.

This has sometimes led to the ext2/3 filesystem's reputation of being a little boring, and perhaps not the fastest or the most scalable filesystem on the block, but which is one of the most dependable. Part of this reputation can be attributed to the extremely conservative design of the ext2 filesystem [4], which had been extended to add journaling support in 1998, but which otherwise had very few other modern filesystem features. Despite its age, ext3 is actually growing in popularity among enterprise users/vendors because of its robustness, good recoverability, and expansion characteristics. The fact that `e2fsck` is able to recover from very severe data corruption scenarios is also very important to ext3's success.

However, in the last few years, the ext2/3 development community has been working hard to demolish the first part of this common wisdom. The initial outline of plans to "modernize" the ext2/3 filesystem was documented in a 2002 Freenix Paper [15]. Three years later, it is time to revisit those plans, see what has been accomplished, what still remains to be done, and what further extensions are now under consideration by the ext 2/3 development community.

This paper is organized into the following sections. First, we describe about those features which have already been implemented and which have been integrated into the mainline kernel in Section 2. Second, we discuss those features which have been implemented, but which have not yet been integrated in mainline in Section 3 and Section 4. Next, we examine the performance improvements on ext3 filesystem during the last few years in Section 5. Finally, we will discuss some potential future work in Section 6.

2 Features found in Linux 2.6

The past three years have seen many discussions of ext2/3 development. Some of the planned features [15] have been implemented and integrated into the mainline kernel during these three years, including directory indexing, reservation based block allocation, online resizing, extended attributes, large inode support, and extended attributes in large inode. In this section, we will give an overview of the design and the implementation for each feature.

2.1 Directory indexing

Historically, ext2/3 directories have used a simple linked list, much like the BSD Fast Filesystem. While it might be expected that the O(n) lookup times would be a significant performance issue, the Linux VFS-level directory cache mitigated the O(n) lookup times for many common workloads. However, ext2's linear directory structure did cause significant performance problems for certain applications, such as web caches and mail systems using the Maildir format.

To address this problem, various ext2 developers, including Daniel Phillips, Theodore Ts'o, and Stephen Tweedie, discussed using a B-tree data structure for directories. However, standard B-trees had numerous characteristics that were at odds with the ext2 design philosophy of simplicity and robustness. For example, XFS's B-tree implementation was larger than all of ext2 or ext3's source files combined. In addition, users of other filesystems using B-trees had reported significantly increased potential for data loss caused by the corruption of a high-level node in the filesystem's B-tree.

To address these concerns, we designed a radically simplified tree structure that was specifically optimized for filesystem directories[10].

This is in contrast to the approach used by many other filesystems, including JFS, Reiserfs, XFS, and HFS, which use a general-purpose B-tree. Ext2's scheme, which we dubbed "HTree", uses 32-bit hashes for keys, where each hash key references a range of entries stored in a leaf block. Since internal nodes are only 8 bytes, HTrees have a very high fanout factor (over 500 blocks can be referenced using a 4K index block), two levels of index nodes are sufficient to support over 16 million 52-character filenames. To further simplify the implementation, HTrees are constant depth (either one or two levels). The combination of the high fanout factor and the use of a hash of the filename, plus a filesystem-specific secret to serve as the search key for the HTree, avoids the need for the implementation to do balancing operations.

We maintain forwards compatibility in old kernels by clearing the `EXT3_INDEX_FL` whenever we modify a directory entry. In order to preserve backwards compatibility, leaf blocks in HTree are identical to old-style linear directory blocks, and index blocks are prefixed with an 8-byte data structure that makes them appear to non-HTree kernels as deleted directory entries. An additional advantage of this extremely aggressive attention towards backwards compatibility is that HTree directories are extremely robust. If any of the index nodes are corrupted, the kernel or the filesystem consistency checker can find all of the directory entries using the traditional linear directory data structures.

Daniel Phillips created an initial implementation for the Linux 2.4 kernel, and Theodore Ts'o significantly cleaned up the implementation and merged it into the mainline kernel during the Linux 2.5 development cycle, as well as implementing e2fsck support for the HTree data structures. This feature was extremely well received, since for very large directories, performance improvements were often better by a factor of 50-100 or more.

While the HTree algorithm significantly improved lookup times, it could cause some performance regressions for workloads that used `readdir()` to perform some operation of all of the files in a large directory. This is caused by `readdir()` returning filenames in a hash-sorted order, so that reads from the inode table would be done in a random order. This performance regression can be easily fixed by modifying applications to sort the directory entries returned by `readdir()` by inode number. Alternatively, an `LD_PRELOAD` library can be used, which intercepts calls to `readdir()` and returns the directory entries in sorted order.

One potential solution to mitigate this performance issue, which has been suggested by Daniel Phillips and Andreas Dilger, but not yet implemented, involves the kernel choosing free inodes whose inode numbers meet a property that groups the inodes by their filename hash. Daniel and Andreas suggest allocating the inode from a range of inodes based on the size of the directory, and then choosing a free inode from that range based on the filename hash. This should in theory reduce the amount of thrashing that results when accessing the inodes referenced in the directory in readdir order. In it is not clear that this strategy will result in a speedup, however; in fact it could increase the total number of inode blocks that might have to be referenced, and thus make the performance of `readdir()` + `stat()` workloads worse. Clearly, some experimentation and further analysis is still needed.

2.2 Improving ext3 scalability

The scalability improvements in the block layer and other portions of the kernel during 2.5 development uncovered a scaling problem for

ext3/JBD under parallel I/O load. To address this issue, Alex Tomas and Andrew Morton worked to remove a per-filesystem superblock lock (`lock_super()`) from ext3 block allocations [13].

This was done by deferring the filesystem's accounting of the number of free inodes and blocks, only updating these counts when they are needed by `statfs()` or `umount()` system call. This lazy update strategy was enabled by keeping authoritative counters of the free inodes and blocks at the per-block group level, and enabled the replacement of the filesystem-wide `lock_super()` with fine-grained locks. Since a spin lock for every block group would consume too much memory, a hashed spin lock array was used to protect accesses to the block group summary information. In addition, the need to use these spin locks was reduced further by using atomic bit operations to modify the bitmaps, thus allowing concurrent allocations within the same group.

After addressing the scalability problems in the ext3 code proper, the focus moved to the journal (JBD) routines, which made extensive use of the big kernel lock (BKL). Alex Tomas and Andrew Morton worked together to reorganize the locking of the journaling layer in order to allow as much concurrency as possible, by using a fine-grained locking scheme instead of using the BKL and the per-filesystem journal lock. This fine-grained locking scheme uses a new per-bufferhead lock (`BH_JournalHead`), a new per-transaction lock (`t_handle_lock`) and several new per-journal locks (`j_state_lock`, `j_list_lock`, and `j_revoke_lock`) to protect the list of revoked blocks. The locking hierarchy (to prevent deadlocks) for these new locks is documented in the `include/linux/jbd.h` header file.

The final scalability change that was needed was to remove the use of `sleep_on()` (which is only safe when called from within code running under the BKL) and replacing it with the new `wait_event()` facility.

These combined efforts served to improve multiple-writer performance on ext3 noticeably: ext3 throughput improved by a factor of 10 on SDET benchmark, and the context switches are dropped significantly [2, 13].

2.3 Reservation based block allocator

Since disk latency is the key factor that affects the filesystem performance, modern filesystems always attempt to layout files on a filesystem contiguously. This is to reduce disk head movement as much as possible. However, if the filesystem allocates blocks on demand, then when two files located in the same directory are being written simultaneously, the block allocations for the two files may end up getting interleaved. To address this problem, some filesystems use the technique of *preallocation*, by anticipating which files will likely need allocate blocks and allocating them in advance.

2.3.1 Preallocation background

In ext2 filesystem, preallocation is performed on the actual disk bitmap. When a new disk data block is allocated, the filesystem internally preallocates a few disk data blocks adjacent to the block just allocated. To avoid filling up filesystem space with preallocated blocks too quickly, each inode is allowed at most seven preallocated blocks at a time. Unfortunately, this scheme had to be disabled when journaling was added to ext3, since it is incompatible with journaling. If the system were to crash before the unused preallocated blocks could be reclaimed, then during system recovery, the ext3

journal would replay the block bitmap update change. At that point the inode's block mapping could end up being inconsistent with the disk block bitmap. Due to the lack of full forced fsck for ext3 to return the preallocated blocks to the free list, preallocation was disabled when the ext3 filesystem was integrated into the 2.4 Linux kernel.

Disabling preallocation means that if multiple processes attempted to allocate blocks to two files in the same directory, the blocks would be interleaved. This was a known disadvantage of ext3, but this short-coming becomes even more important with extents (see Section 3.1) since extents are far more efficient when the file on disk is contiguous. Andrew Morton, Mingming Cao, Theodore Ts'o, and Badari Pulavarty explored various possible ways to add preallocation to ext3, including the method that had been used for preallocation in ext2 filesystem. The method that was finally settled upon was a reservation-based design.

2.3.2 Reservation design overview

The core idea of the reservation based allocator is that for every inode that needs blocks, the allocator reserves a range of blocks for that inode, called a reservation window. Blocks for that inode are allocated from that range, instead of from the whole filesystem, and no other inode is allowed to allocate blocks in the reservation window. This reduces the amount of fragmentation when multiple files are written in the same directory simultaneously. The key difference between reservation and preallocation is that the blocks are only reserved in memory, rather than on disk. Thus, in the case the system crashes while there are reserved blocks, there is no inconsistency in the block group bitmaps.

The first time an inode needs a new block, a block allocation structure, which describes the reservation window information and other block allocation related information, is allocated and linked to the inode. The block allocator searches for a region of blocks that fulfills three criteria. First, the region must be near the ideal "goal" block, based on ext2/3's existing block placement algorithms. Secondly, the region must not overlap with any other inode's reservation windows. Finally, the region must have at least one free block. As an inode keeps growing, free blocks inside its reservation window will eventually be exhausted. At that point, a new window will be created for that inode, preferably right after the old with the guide of the "goal" block.

All of the reservation windows are indexed via a per-filesystem red-black tree so the block allocator can quickly determine whether a particular block or region is already reserved by a particular inode. All operations on that tree are protected by a per-filesystem global spin lock.

Initially, the default reservation window size for an inode is set to eight blocks. If the reservation allocator detects the inode's block allocation pattern to be sequential, it dynamically increases the window size for that inode. An application that knows the file size ahead of the file creation can employ an ioctl command to set the window size to be equal to the anticipated file size in order to attempt to reserve the blocks immediately.

Mingming Cao implemented this reservation based block allocator, with help from Stephen Tweedie in converting the per-filesystem reservation tree from a sorted link list to a red-black tree. In the Linux kernel versions 2.6.10 and later, the default block allocator for ext3 has been replaced by this reservation based block allocator. Some benchmarks, such as tiobench and dbench, have shown significant improvements on sequential writes and subsequent sequential reads with this reservation-based block

allocator, especially when a large number of processes are allocating blocks concurrently.

2.3.3 Future work

Currently, the reservation window only lasts until the last process writing to that file closes. At that time, the reservation window is released and those blocks are available for reservation or allocation by any other inode. This is necessary so that the blocks that were reserved can be released for use by other files, and to avoid fragmentation of the free space in the filesystem.

However, some files, such as log files and UNIX® mailbox files, have a *slow growth* pattern. That is, they grow slowly over time, by processes appending a small amount of data, and then closing the file, over and over again. For these files, in order to avoid fragmentation, it is necessary that the reservation window be preserved even after the file has been closed.

The question is how to determine which files should be allowed to retain their reservation window after the last close. One possible solution is to tag the files or directories with an attribute indicating that they contain files that have a slow growth pattern. Another possibility is to implement heuristics that can allow the filesystem to automatically determines which file seems to have a slow growth pattern, and automatically preserve the reservation window after the file is closed.

If reservation windows can be preserved in this fashion, it will be important to also implement a way for preserved reservation windows to be reclaimed when the filesystem is fully reserved. This prevents an inode that fails to find a new reservation from falling back to no-reservation mode too soon.

2.4 Online resizing

The online resizing feature was originally developed by Andreas Dilger in July of 1999 for the 2.0.36 kernel. The availability of a Logical Volume Manager (LVM), motivated the desire for on-line resizing, so that when a logical volume was dynamically resized, the filesystem could take advantage of the new space. This ability to dynamically resize volumes and filesystems is very useful in server environments, where taking downtime for unmounting a filesystem is not desirable. After missing the code freeze for the 2.4 kernel, the ext2online code was finally included into the 2.6.10 kernel and e2fsprogs 1.36 with the assistance of Stephen Tweedie and Theodore Ts'o.

2.4.1 The online resizing mechanism

The online resizing mechanism, despite its seemingly complex task, is actually rather simple in its implementation. In order to avoid a large amount of complexity it is only possible to increase the size of a filesystem while it is mounted. This addresses the primary requirement that a filesystem that is (nearly) full can have space added to it without interrupting the use of that system. The online resizing code depends on the underlying block device to handle all aspects of its own resizing prior to the start of filesystem resizing, and does nothing itself to manipulate the partition tables of LVM/MD block devices.

The ext2/3 filesystem is divided into one or more block allocation groups of a fixed size, with possibly a partial block group at the end of the filesystem [4]. The layout of each block group (where the inode and block allocation bitmaps and the inode table are stored) is kept in the group descriptor table. This table is stored at the start of at the first block group, and

consists of one or more filesystem blocks, depending on the size of the filesystem. Backup copies of the group descriptor table are kept in more groups if the filesystem is large enough.

There are three primary phases by which a filesystem is grown. The first, and simplest, is to expand the last partial block group (if any) to be a full block group. The second phase is to add a new block group to an existing block in the group descriptor table. The third phase is to add a new block to the group descriptor table and add a new group to that block. All filesystem resizes are done incrementally, going through one or more of the phases to add free space to the end of the filesystem until the desired size is reached.

2.4.2 Resizing within a group

For the first phase of growth, the online resizing code starts by briefly locking the superblock and increasing the total number of filesystem blocks to the end of the last group. All of the blocks beyond the end of the filesystem are already marked as "in use" by the block bitmap for that group, so they must be cleared. This is accomplished by the same mechanism that is used when deleting a file - `ext3_free_blocks()` and can be done without locking the whole filesystem. The online resizer simply pretends that it is deleting a file that had allocated all of the blocks at the end of the filesystem, and `ext3_free_blocks()` handles all of the bitmap and free block count updates properly.

2.4.3 Adding a new group

For the second phase of growth, the online resizer initializes the next group beyond the end of the filesystem. This is easily done because this area is currently unused and unknown to the filesystem itself. The block bitmap for that group is initialized as empty, the superblock and group descriptor backups (if any) are copied from the primary versions, and the inode bitmap and inode table are initialized. Once this has completed successfully the online resizing code briefly locks the superblock to increase the total and free blocks and inodes counts for the filesystem, add a new group to the end of the group descriptor table, and increase the total number of groups in the filesystem by one. Once this is completed the backup superblock and group descriptors are updated in case of corruption of the primary copies. If there is a problem at this stage, the next e2fsck will also update the backups.

The second phase of growth will be repeated until the filesystem has fully grown, or the last group descriptor block is full. If a partial group is being added at the end of the filesystem the blocks are marked as "in use" before the group is added. Both first and second phase of growth can be done on any ext3 filesystem with a supported kernel and suitable block device.

2.4.4 Adding a group descriptor block

The third phase of growth is needed periodically to grow a filesystem over group descriptor block boundaries (at multiples of 16 GB for filesystems with 4 KB blocksize). When the last group descriptor block is full, a new block must be added to the end of the table. However, because the table is contiguous at the start of the first group and is normally followed immediately by the block and inode bitmaps and the inode table, the online resize code needs a bit of assistance while the filesystem is unmounted (offline) in order to maintain compatibility with older kernels. Either at `mke2fs`

time, or for existing filesystems with the assistance of the `ext2prepare` command, a small number of blocks at the end of the group descriptor table are reserved for online growth. The total amount of reserved blocks is a tiny fraction of the total filesystem size, requiring only a few tens to hundreds of kilobytes to grow the filesystem 1024-fold.

For the third phase, it first gets the next reserved group descriptor block and initializes a new group and group descriptor beyond the end of the filesystem, as is done in second phase of growth. Once this is successful, the superblock is locked while reallocating the array that indexes all of the group descriptor blocks to add another entry for the new block. Finally, the superblock totals are updated, the number of groups is increased by one, and the backup superblock and group descriptors are updated.

The online resizing code takes advantage of the journaling features in ext3 to ensure that there is no risk of filesystem corruption if the resize is unexpectedly interrupted. The ext3 journal ensures strict ordering and atomicity of filesystem changes in the event of a crash - either the entire resize phase is committed or none of it is. Because the journal has no rollback mechanism (except by crashing) the resize code is careful to verify all possible failure conditions prior to modifying any part of the filesystem. This ensures that the filesystem remains valid, though slightly smaller, in the event of an error during growth.

2.4.5 Future work

Future development work in this area involves removing the need to do offline filesystem manipulation to reserve blocks before doing third phase growth. The use of Meta Block Groups [15] allows new groups to be added to the filesystem without the need to allocate contiguous blocks for the group descriptor table. Instead the group descriptor block is kept in the first group that it describes, and a backup is kept in the second and last group for that block. The Meta Block Group support was first introduced in the 2.4.25 kernel (Feb. 2004) so it is reasonable to think that a majority of existing systems could mount a filesystem that started using this when it is introduced.

A more complete description of the online growth is available in [6].

2.5 Extended attributes

2.5.1 Extended attributes overview

Many new operating system features (such as access control lists, mandatory access controls, Posix Capabilities, and hierarchical storage management) require filesystems to be able associate a small amount of custom metadata with files or directories. In order to implement support for access control lists, Andreas Gruenbacher added support for extended attributes to the ext2 filesystems. [7]

Extended attributes as implemented by Andreas Gruenbacher are stored in a single EA block. Since a large number of files will often use the same access control list, as inherited from the directory's default ACL as an optimization, the EA block may be shared by inodes that have identical extended attributes.

While the extended attribute implementation was originally optimized for use to store ACL's, the primary users of extended attributes to date have been the NSA's SELinux system, Samba 4 for storing extended attributes from Windows clients, and the Lustre filesystem.

In order to store larger EAs than a single filesystem block, work is underway to store

large EAs in another EA inode referenced from the original inode. This allows many arbitrary-sized EAs to be attached to a single file, within the limitations of the EA interface and what can be done inside a single journal transaction. These EAs could also be accessed as additional file forks/streams, if such an API were added to the Linux kernel.

2.5.2 Large inode support and EA-in-inode

Alex Tomas and Andreas Dilger implemented support for storing the extended attribute in an expanded ext2 inode, in preference to using a separate filesystem block. In order to do this, the filesystem must be created using an inode size larger than the default 128 bytes. Inode sizes must be a power of two and must be no larger than the filesystem block size, so for a filesystem with a 4 KB blocksize, inode sizes of 256, 512, 1024, 2048, or 4096 bytes are valid. The 2 byte field starting at offset 128 (`i_extra_size`) of each inode specifies the starting offset for the portion of the inode that can be used for storing EA's. Since the starting offset must be a multiple of 4, and we have not extended the fixed portion of the inode beyond `i_extra_size`, currently `i_extra_size` is 4 for all filesystems with expanded inodes. Currently, all of the inode past the initial 132 bytes can be used for storing EAs. If the user attempts to store more EAs than can fit in the expanded inode, the additional EAs will be stored in an external filesystem block.

Using the EA-in-inode, a very large (seven-fold improvement) difference was found in some Samba 4 benchmarks, taking ext3 from last place when compared to XFS, JFS, and Reiserfs3, to being clearly superior to all of the other filesystems for use in Samba 4. [5] The in-inode EA patch started by Alex Tomas and Andreas Dilger was re-worked by Andreas Gruenbacher. And the fact that this feature was such a major speedup for Samba 4, motivated it being integrated into the mainline 2.6.11 kernel very quickly.

3 Extents, delayed allocation and extent allocation

This section and the next (Section 4) will discuss features that are currently under development, and (as of this writing) have not been merged into the mainline kernel. In most cases patches exist, but they are still being polished, and discussion within the ext2/3 development community is still in progress.

Currently, the ext2/ext3 filesystem, like other traditional UNIX filesystems, uses a direct, indirect, double indirect, and triple indirect blocks to map file offsets to on-disk blocks. This scheme, sometimes simply called an indirect block mapping scheme, is not efficient for large files, especially large file deletion. In order to address this problem, many modern filesystems (including XFS and JFS on Linux) use some form of extent maps instead of the traditional indirect block mapping scheme.

Since most filesystems try to allocate blocks in a contiguous fashion, extent maps are a more efficient way to represent the mapping between logical and physical blocks for large files. An *extent* is a single descriptor for a range of contiguous blocks, instead of using, say, hundreds of entries to describe each block individually.

Over the years, there have been many discussions about moving ext3 from the traditional indirect block mapping scheme to an extent map based scheme. Unfortunately, due to the complications involved with making an incompatible format change, progress on an actual implemention of these ideas had been slow.

Alex Tomas, with help from Andreas Dilger, designed and implemented extents for ext3. He posted the initial version of his extents patch on August, 2003. The initial results on file creation and file deletion tests inspired a round of discussion in the Linux community to consider adding extents to ext3. However, given the concerns that the format changes were ones that all of the ext3 developers will have to support on a long-term basis, and the fact that it was very late in the 2.5 development cycle, it was not integrated into the mainline kernel sources at that time.

Later, in April of 2004, Alex Tomas posted an updated extents patch, as well as additional patches that implemented delayed allocation and multiple block allocation to the ext2-devel mailing list. These patches were reposted in February 2005, and this re-ignited interest in adding extents to ext3, especially when it was shown that the combination of these three features resulted in significant throughput improvements on some sequential write tests.

In the next three sections, we will discuss how these three features are designed, followed by a discussion of the performance evaluation of the combination of the three patches.

3.1 Extent maps

This implementation of extents was originally motivated by the problem of long truncate times observed for huge files.[1] As noted above, besides speeding up truncates, extents help improve the performance of sequential file writes since extents are a significantly smaller amount of metadata to be written to describe contiguous blocks, thus reducing the filesystem overhead.

Most files need only a few extents to describe their logical-to-physical block mapping, which can be accommodated within the inode or a single extent map block. However, some extreme cases, such as sparse files with random allocation patterns, or a very badly fragmented filesystem, are not efficiently represented using extent maps. In addition, allocating blocks in a random access pattern may require inserting an extent map entry in the middle of a potentially very large data representation.

One solution to this problem is to use a tree data structure to store the extent map, either a B-tree, B+ tree, or some simplified tree structure as was used for the HTree feature. Alex Tomas's implementation takes the latter approach, using a constant-depth tree structure. In this implementation, the extents are expressed using a 12 byte structure, which include a 32-bit logical block number, a 48-bit physical block number, and a 16-bit extent length. With 4 KB blocksize, a filesystem can address up to 1024 petabytes, and a maximum file size of 16 terabytes. A single extent can cover up to 2^{16} blocks or 256 MB.[2]

The extent tree information can be stored in the inode's `i_data` array, which is 60 bytes long. An attribute flag in the inode's `i_flags` word indicates whether the inode's `i_data` array should be interpreted using the traditional indirect block mapping scheme, or as an extent data structure. If the entire extent information can be stored in the `i_data` field, then it will be treated as a single leaf node of the extent tree; otherwise, it will be treated as the root node of inode's extent tree, and additional filesystem blocks serve as intermediate or leaf nodes in the extent tree.

At the beginning of each node, the `ext3_ext_header` data structure is 12 bytes long,

[1] One option to address the issue is performing asynchronous truncates, however, while this makes the CPU cycles to perform the truncate less visible, excess CPU time will still be consumed by the truncate operations.

[2] Currently, the maximum block group size given a 4 KB blocksize is 128 MB, and this will limit the maximum size for a single extent.

and contains a 16-bit magic number, 2 16-bit integers containing the number of valid entries in the node, and the maximum number of entries that can be stored in the node, a 16-bit integer containing the depth of the tree, and a 32-bit tree generation number. If the depth of the tree is 0, then root inode contains leaf node information, and the 12-byte entries contain the extent information described in the previous paragraph. Otherwise, the root node will contain 12-byte intermediate entries, which consist of 32-bit logical block and a 48-bit physical block (with 16 bits unused) of the next index or leaf block.

3.1.1 Code organization

The implementation is divided into two parts: Generic extents support that implements initialize/lookup/insert/remove functions for the extents tree, and VFS support that allows methods and callbacks like `ext3_get_block()`, `ext3_truncate()`, `ext3_new_block()` to use extents.

In order to use the generic extents layer, the user of the generic extents layer must declare its tree via an `ext3_extents_tree` structure. The structure describes where the root of the tree is stored, and specifies the helper routines used to operate on it. This way one can root a tree not only in `i_data` as described above, but also in a separate block or in EA (Extended Attributes) storage. The helper routines described by struct `ext3_extents_helpers` can be used to control the block allocation needed for tree growth, journaling metadata, using different criteria of extents mergability, removing extents etc.

3.1.2 Future work

Alex Tomas's extents implementation is still a work-in-progress. Some of the work that needs to be done is to make the implementation independent of byte-order, improving the error handling, and shrinking the depth of the tree when truncated the file. In addition, the extent scheme is less efficient than the traditional indirect block mapping scheme if the file is highly fragmented. It may be useful to develop some heuristics to determine whether or not a file should use extents automatically. It may also be desirable to allow block-mapped leaf blocks in an extent-mapped file for cases where there is not enough contiguous space in the filesystem to allocate the extents efficiently.

The last change would necessarily change the on-disk format of the extents, but it is not only the extent format that has been changed. For example, the extent format does not support logical block numbers that are greater than 32 bits, and a more efficient, variable-length format would allow more extents to be stored in the inode before spilling out to an external tree structure.

Since deployment of the extent data structure is disruptive because it involved an non-backwards-compatible change to the filesystem format, it is important that the ext3 developers are comfortable that the extent format is flexible and powerful enough for present and future needs, in order to avoid the need for additional incompatible format changes.

3.2 Delayed allocation

3.2.1 Why delayed allocation is needed

Procrastination has its virtues in the ways of an operating system. Deferring certain tasks un-

til an appropriate time often improves the overall efficiency of the system by enabling optimal deployment of resources. Filesystem I/O writes are no exception.

Typically, when a filesystem `write()` system call returns success, it has only copied the data to be written into the page cache, mapped required blocks in the filesystem and marked the pages as needing write out. The actual write out of data to disk happens at a later point of time, usually when writeback operations are clustered together by a background kernel thread in accordance with system policies, or when the user requests file data to be synced to disk. Such an approach ensures improved I/O ordering and clustering for the system, resulting in more effective utilization of I/O devices with applications spending less time in the `write()` system call, and using the cycles thus saved to perform other work.

Delayed allocation takes this a step further, by deferring the allocation of new blocks in the filesystem to disk blocks until writeback time [12]. This helps in three ways:

- Reduces fragmentation in the filesystem by improving chances of creating contiguous blocks on disk for a file. Although preallocation techniques can help avoid fragmentation, they do not address fragmentation caused by multiple threads writing to the file at different offsets simultaneously, or files which are written in a non-contiguous order. (For example, the `libfd` library, which is used by the GNU C compiler will create object files that are written out of order.)
- Reduces CPU cycles spent in repeated `get_block()` calls, by clustering allocation for multiple blocks together. Both of the above would be more effective when combined with a good multi-block allocator.
- For short lived files that can be buffered in memory, delayed allocation may avoid the need for disk updates for metadata creation altogether, which in turn reduces impact on fragmentation [12].

Delayed allocation is also useful for the Active Block I/O Scheduling System (ABISS) [1], which provides guaranteed read/write bit rates for applications that require guaranteed real-time I/O streams. Without delayed allocation, the synchronous code path for `write()` has to read, modify, update, and journal changes to the block allocation bitmap, which could disrupt the guaranteed read/write rates that ABISS is trying to deliver.

Since block allocation is deferred until background writeback when it is too late to return an error to the caller of `write()`, the `write()` operation requires a way to ensure that the allocation will indeed succeed. This can be accomplished by carving out, or reserving, a claim on the expected number of blocks on disk (for example, by subtracting this number from the total number of available blocks, an operation that can be performed without having to go through actual allocation of specific disk blocks).

Repeated invocations of `ext3_get_block()/ext3_new_block()` is not efficient for mapping consecutive blocks, especially for an extent based inode, where it is natural to process a chunk of contiguous blocks all together. For this reason, Alex Tomas implemented an extents based multiple block allocation and used it as a basis for extents based delayed allocation. We will discuss the extents based multiple block allocation in Section 3.3.

3.2.2 Extents based delayed allocation implementation

If the delayed allocation feature is enabled for an ext3 filesystem and a file uses extent maps, then the address space operations for its inode are initialized to a set of ext3 specific routines that implement the write operations a little differently. The implementation defers allocation of blocks from `prepare_write()` and employs extent walking, together with the multiple block allocation feature (described in the next section), for clustering block allocations maximally into contiguous blocks.

Instead of allocating the disk block in `prepare_write()`, the the page is marked as needing block reservation. The `commit_write()` function calculates the required number of blocks, and reserves them to make sure that there are enough free blocks in the filesystem to satisfy the write. When the pages get flushed to disk by `writepage()` or `writepages()`, these functions will walk all the dirty pages in the specified inode, cluster the logically contiguous ones, and submit the page or pages to the bio layer. After the block allocation is complete, the reservation is dropped. A single block I/O request (or `BIO`) is submitted for write out of pages processed whenever a new allocated extent (or the next mapped extent if already allocated) on the disk is not adjacent to the previous one, or when `writepages()` completes. In this manner the delayed allocation code is tightly integrated with other features to provide best performance.

3.3 Buddy based extent allocation

One of the shortcomings of the current ext3 block allocation algorithm, which allocates one block at a time, is that it is not efficient enough for high speed sequential writes. In one experiment utilizing direct I/O on a dual Opteron workstation with fast enough buses, fiber channel, and a large, fast RAID array, the CPU limited the I/O throughput to 315 MB/s. While this would not be an issue on most machines (since the maximum bandwidth of a PCI bus is 127 MB/s), but for newer or enterprise-class servers, the amount of data per second that can be written continuously to the filesystem is no longer limited by the I/O subsystem, but by the amount of CPU time consumed by ext3's block allocator.

To address this problem, Alex Tomas designed and implemented a multiple block allocation, called mballoc, which uses a classic buddy data structure on disk to store chunks of free or used blocks for each block group. This buddy data is an array of metadata, where each entry describes the status of a cluster of 2^n blocks, classified as free or in use.

Since block buddy data is not suitable for determining a specific block's status and locating a free block close to the allocation goal, the traditional block bitmap is still required in order to quickly test whether a specific block is available or not.

In order to find a contiguous extent of blocks to allocate, mballoc checks whether the goal block is available in the block bitmap. If it is available, mballoc looks up the buddy data to find the free extent length starting from the goal block. To find the real free extent length, mballoc continues by checking whether the physical block right next to the end block of the previously found free extent is available or not. If that block is available in the block bitmap, mballoc could quickly find the length of the next free extent from buddy data and add it up to the total length of the free extent from the goal block.

For example, if block M is the goal block and

is claimed to be available in the bitmap, and block M is marked as free in buddy data of order n, then initially the free chunk size from block M is known to be 2^n. Next, mballoc checks the bitmap to see if block $M + 2^n + 1$ is available or not. If so, mballoc checks the buddy data again, and finds that the free extent length from block $M + 2^n + 1$ is k. Now, the free chunk length from goal block M is known to be $2^n + 2^k$. This process continues until at some point the boundary block is not available. In this manner, instead of testing dozens, hundreds, or even thousands of blocks' availability status in the bitmap to determine the free blocks chunk size, it can be enough to just test a few bits in buddy data and the block bitmap to learn the real length of the free blocks extent.

If the found free chunk size is greater than the requested size, then the search is considered successful and mballoc allocates the found free blocks. Otherwise, depending on the allocation criteria, mballoc decides whether to accept the result of the last search in order to preserve the goal block locality, or continue searching for the next free chunk in case the length of contiguous blocks is a more important factor than where it is located. In the later case, mballoc scans the bitmap to find out the next available block, then, starts from there, and determines the related free extent size.

If mballoc fails to find a free extent that satisfies the requested size after rejecting a predefined number (currently 200) of free chunks, it stops the search and returns the best (largest) free chunk found so far. In order to speed up the scanning process, mballoc maintains the total number of available blocks and the first available block of each block group.

3.3.1 Future plans

Since in ext3 blocks are divided into block groups, the block allocator first selects a block group before it searches for free blocks. The policy employed in mballoc is quite simple: to try the block group where the goal block is located first. If allocation from that group fails, then scan the subsequent groups. However, this implies that on a large filesystem, especially when free blocks are not evenly distributed, CPU cycles could be wasted on scanning lots of almost full block groups before finding a block group with the desired free blocks criteria. Thus, a smarter mechanism to select the right block group to start the search should improve the multiple block allocator's efficiency. There are a few proposals:

1. Sort all the block groups by the total number of free blocks.
2. Sort all the groups by the group fragmentation factor.
3. Lazily sort all the block groups by the total number of free blocks, at significant change of free blocks in a group only.
4. Put extents into buckets based on extent size and/or extent location in order to quickly find extents of the correct size and goal location.

Currently the four options are under evaluation though probably the first one is a little more interesting.

3.4 Evaluating the extents patch set

The initial evaluation of the three patches (extents, delayed allocation and extent allocation) shows significant throughput improvements, especially under sequential tests. The

tests show that the extents patch significantly reduces the time for large file creation and removal, as well as file rewrite. With extents and extent allocation, the throughput of Direct I/O on the aforementioned Opteron-based workstation is significantly improved, from 315 MB/sec to 500MB/sec, and the CPU usage is significantly dropped from 100% to 50%. In addition, extensive testing on various benchmarks, including dbench, tiobench, FFSB [11] and sqlbench [16], has been done with and without this set of patches. Some initial analysis indicates that the multiple block allocation, when combined with delayed allocation, is a key factor resulting in this improvement. More testing results can be obtained from `http://www.bullopensource.org/ext4`

4 Improving ext3 without changing disk format

Replacing the traditional indirect block mapping scheme with an extent mapping scheme, has many benefits, as we have discussed in the previous section. However, changes to the on-disk format that are not backwards compatible are often slow to be adopted by users, for two reasons. First of all, robust e2fsck support sometimes lags the kernel implementation. Secondly, it is generally not possible to mount the filesystem with an older kernel once the filesystem has been converted to use these new features, preventing rollback in case of problems.

Fortunately, there are a number of improvements that can be made to the ext2/3 filesystem without making these sorts of incompatible changes to the on-disk format.

In this section, we will discuss a few of features that are implemented based on the current ext3 filesystem. Section 4.1 describes the effort to reduce the usage of bufferheads structure in ext3; Section 4.2 describes the effort to add delayed allocation without requiring the use of extents; Section 4.3 discusses the work to add multiple block allocation; Section 4.4 describes asynchronous file unlink and truncate; Section 4.5 describes a feature to allow more than 32000 subdirectories; and Section 4.6 describes a feature to allow multiple threads to concurrently create/rename/link/unlink files in a single directory.

4.1 Reducing the use of bufferheads in ext3

Bufferheads continue to be heavily used in Linux I/O and filesystem subsystem, even though closer integration of the buffer cache with the page cache since 2.4 and the new block I/O subsystem introduced in Linux 2.6 have in some sense superseded part of the traditional Linux buffer cache functionality.

There are a number of reasons for this. First of all, the buffer cache is still used as a metadata cache. All filesystem metadata (superblock, inode data, indirect blocks, etc.) are typically read into buffer cache for quick reference. Bufferheads provide a way to read/write/access this data. Second, bufferheads link a page to disk block and cache the block mapping information. In addition, the design of bufferheads supports filesystem block sizes that do not match the system page size. Bufferheads provide a convenient way to map multiple blocks to a single page. Hence, even the generic multi-page read-write routines sometimes fall back to using bufferheads for fine-graining or handling of complicated corner cases.

Ext3 is no exception to the above. Besides the above reasons, ext3 also makes use of bufferheads to enable it to provide ordering guarantees in case of a transaction commit. Ext3's or-

dered mode guarantees that file data gets written to the disk before the corresponding metadata gets committed to the journal. In order to provide this guarantee, bufferheads are used as the mechanism to associate the data pages belonging to a transaction. When the transaction is committed to the journal, ext3 uses the bufferheads attached to the transaction to make sure that all the associated data pages have been written out to the disk.

However, bufferheads have the following disadvantages:

- All bufferheads are allocated from the "buffer_head" slab cache, thus they consume low memory[3] on 32-bit architectures. Since there is one bufferhead (or more, depending on the block size) for each filesystem page cache page, the bufferhead slab can grow really quickly and consumes a lot of low memory space.
- When bufferheads get attached to a page, they take a reference on the page. The reference is dropped only when VM tries to release the page. Typically, once a page gets flushed to disk it is safe to release its bufferheads. But dropping the bufferhead, right at the time of I/O completion is not easy, since being in interrupt handler context restricts the kind of operations feasible. Hence, bufferheads are left attached to the page, and released later as and when VM decides to re-use the page. So, it is typical to have a large number of bufferheads floating around in the system.
- The extra memory references to bufferheads can impact the performance of memory caches, the Translation Lookaside Buffer (TLB) and the Segment Lookaside Buffer[4] (SLB). We have observed that when running a large NFS workload, while the ext3 journaling thread kjournald() is referencing all the transactions, all the journal heads, and all the bufferheads looking for data to flush/clean it suffers a large number of SLB misses with the associated performance penalty. The best solution for these performance problems appears to be to eliminate the use of bufferheads as much as possible, which reduces the number of memory references required by kjournald().

To address the above concerns, Badari Pulavarty has been working on removing bufferheads usage from ext3 from major impact areas, while retaining bufferheads for uncommon usage scenarios. The focus was on elimination of bufferhead usage for user data pages, while retaining bufferheads primarily for metadata caching.

Under the writeback journaling mode, since there are no ordering requirements between when metadata and data gets flushed to disk, eliminating the need for bufferheads is relatively straightforward because ext3 can use most recent generic VFS helpers for writeback. This change is already available in the latest Linux 2.6 kernels.

For ext3 ordered journaling mode, however, since bufferheads are used as linkage between pages and transactions in order to provide flushing order guarantees, removal of the use of bufferheads gets complicated. To address this issue, Andrew Morton proposed a new ext3 journaling mode, which works without bufferheads and provides semantics that are somewhat close to that provided in ordered mode[9]. The idea is that whenever there is a transaction commit, we go through all the dirty inodes and

[3] Low memory is memory that can be directly mapped into kernel virtual address space, i.e. 896MB, in the case of IA32

[4] The SLB is found on the 64-bit Power PC.

dirty pages in that filesystem and flush every one of them. This way metadata and user data are flushed at the same time. The complexity of this proposal is currently under evaluation.

4.2 Delayed allocation without extents

As we have discussed in Section 3.2, delayed allocation is a powerful technique that can result in significant performance gains, and Alex Tomas's implementation shows some very interesting and promising results. However, Alex's implementation only provide delayed allocation when the ext3 filesystem is using extents, which requires an incompatible change to the on-disk format. In addition, like past implementation of delayed allocation by other filesystems, such as XFS, Alex's changes implement the delayed allocation in filesystem-specific versions of `prepare_write()`, `commit_write()`, `writepage()`, and `writepages()`, instead of using the filesystem independent routines provided by the Linux kernel.

This motivated Suparna Bhattacharya, Badari Pulavarty and Mingming Cao to implement delayed allocation and multiple block allocation support to improve the performance of the ext3 to the extent possible without requiring any on-disk format changes.

Interestingly, the work to remove the use of bufferheads in ext3 implemented most of the necessary changes required for delayed allocation, when bufferheads are not required. The `nobh_commit_write()` function, delegates the task of writing data to the `writepage()` and `writepages()`, by simply marking the page as dirty. Since the `writepage()` function already has to handle the case of writing a page which is mapped to a sparse memory-mapped files, the `writepage()` function already handles block allocation by calling the filesystem specific `get_block()` function. Hence, if the `nobh_prepare_write` function were to omit call `get_block()`, the physical block would not be allocated until the page is actually written out via the `writepage()` or `writepages()` function.

Badari Pulavarty implemented a relatively small patch as a proof-of-concept, which demonstrates that this approach works well. The work is still in progress, with a few limitations to address. The first limitation is that in the current proof-of-concept patch, data could be dropped if the filesystem was full, without the `write()` system call returning -ENOSPC.[5] In order to address this problem, the `nobh_prepare_write` function must note that the page currently does not have a physical block assigned, and request the filesystem reserve a block for the page. So while the filesystem will not have assigned a specific physical block as a result of `nobh_prepare_write()`, it must guarantee that when `writepage()` calls the block allocator, the allocation must succeed.

The other major limitation is, at present, it only worked when bufferheads are not needed. However, the `nobh` code path as currently present into the 2.6.11 kernel tree only supports filesystems when the ext3 is journaling in writeback mode and not in ordered journaling mode, and when the blocksize is the same as the VM pagesize. Extending the `nobh` code paths to support sub-pagesize blocksizes is likely not very difficult, and is probably the appropriate way of addressing the first part of this shortcoming.

[5] The same shortcoming exists today if a sparse file is memory-mapped, and the filesystem is full when `writepage()` tries to write a newly allocated page to the filesystem. This can potentially happen after user process which wrote to the file via `mmap()` has exited, where there is no program left to receive an error report.

However, supporting delayed allocation for ext3 ordered journaling using this approach is going to be much more challenging. While metadata journaling alone is sufficient in writeback mode, ordered mode needs to track I/O submissions for purposes of waiting for completion of data writeback to disk as well, so that it can ensure that metadata updates hit the disk only after the corresponding data blocks are on disk. This avoids potential exposures and inconsistencies without requiring full data journaling[14].

However, in the current design of generic multi-page writeback routines, block I/O submissions are issued directly by the generic routines and are transparent to the filesystem specific code. In earlier situations where bufferheads were used for I/O, filesystem specific wrappers around generic code could track I/O through the bufferheads associated with a page and link them with the transaction. With the recent changes, where I/O requests are built directly as multi-page bio requests with no link from the page to the bio, this no longer applies.

A couple of solution approaches are under consideration, as of the writing of this paper:

- Introducing yet another filesystem specific callback to be invoked by the generic multi-page write routines to actually issue the I/O. ext3 could then track the number of in-flight I/O requests associated with the transaction, and wait for this to fall to zero at journal commit time. Implementing this option is complicated because the multi-page write logic occasionally falls back to the older bufferheads based logic in some scenarios. Perhaps ext3 ordered mode writeback would need to provide both the callback and the page bufferhead tracking logic if this approach is employed.
- Find a way to get ext3 journal commit to effectively reuse a part the fsync/O_SYNC implementation that waits for writeback to complete on the pages for relevant inodes, using a radix-tree walk. Since the journal layer is designed to be unaware of filesystems [14], this could perhaps be accomplished by associating a (filesystem specific) callback with journal commit, as recently suggested by Andrew Morton[9].

It remains to be seen which approach works out to be the best, as development progresses. It is clear that since ordered mode is the default journaling mode, any delayed allocation implementation must be able to support it.

4.3 Efficiently allocating multiple blocks

As with the Alex Tomas's delayed allocation patch, Alex's multiple block allocator patch relies on an incompatible on-disk format change of the ext3 filesystem to support extent maps. In addition, the extent-based mballoc patch also required a format change in order to store data for the buddy allocator which it utilized. Since oprofile measurements of Alex's patch indicated the multiple block allocator seemed to be responsible for reducing CPU usage, and since it seemed to improve throughput in some workloads, we decided to investigate whether it was possible to obtain most of the benefits of a multiple block allocator using the current ext3 filesystem format. This seemed to be a reasonable approach since many of the advantages of supporting Alex's mballoc patch seemed to derive from collapsing a large number of calls to `ext3_get_block()` into much fewer calls to `ext3_get_blocks()`, thus avoiding excess calls into the journaling layer to record changes to the block allocation bitmap.

In order to implement a multiple-block allocator based on the existing block allocation

bitmap, Mingming Cao first changed `ext3_new_block()` to accept a new argument specifying how many contiguous blocks the function should attempt to allocate, on a best efforts basis. The function now allocates the first block in the existing way, and then continues allocating up to the requested number of adjacent physical blocks at the same time if they are available.

The modified `ext3_new_block()` function was then used to implement ext3's `get_blocks()` method, the standardized filesystem interface to translate a file offset and a length to a set of on-disk blocks. It does this by starting at the first file offset and translating it into a logical block number, and then taking that logical block number and mapping it to a physical block number. If the logical block has already been mapped, then it will continue mapping the next logical block until the requisite number of physical blocks have been returned, or an unallocated block is found.

If some blocks need to be allocated, first `ext3_get_blocks()` will look ahead to see how many adjacent blocks are needed, and then passes this allocation request to `ext3_new_blocks()`, searches for the requested free blocks, marks them as used, and returns them to `ext3_get_blocks()`. Next, `ext3_get_blocks()` will update the inode's direct blocks, or a single indirect block to point at the allocated blocks.

Currently, this `ext3_get_blocks()` implementation does not allocate blocks across an indirect block boundary. There are two reasons for this. First, the `JBD` journaling requests the filesystem to reserve the maximum of blocks that will require journaling, when a new transaction handle is requested via `ext3_journal_start()`. If we were to allow a multiple block allocation request to span an indirect block boundary, it would be difficult to predict how many metadata blocks may get dirtied and thus require journaling. Secondly, it would be difficult to place any newly allocated indirect blocks so they are appropriately interleaved with the data blocks.

Currently, only the Direct I/O code path uses the `get_blocks()` interfaces; the `mpage_writepages()` function calls `mpage_writepage()` which in turn calls `get_block()`. Since only a few workloads (mainly databases) use Direct I/O, Suparna Bhattacharya has written a patch to change `mpage_writepages()` use `get_blocks()` instead. This change should be generically helpful for any filesystems which implement an efficient `get_blocks()` function.

Draft patches have already been posted to the ext2-devel mailing list. As of this writing, we are trying to integrate Mingming's `ext3_get_blocks()` patch, Suparna Bhattacharya's `mpage_writepage()` patch and Badari Pulavarty's generic delayed allocation patch (discussed in Section 4.2) in order to evaluate these three patches together using benchmarks.

4.4 Asynchronous file unlink/truncate

With block-mapped files and ext3, truncation of a large file can take a considerable amount of time (on the order of tens to hundreds of seconds if there is a lot of other filesystem activity concurrently). There are several reasons for this:

- There are limits to the size of a single journal transaction (1/4 of the journal size). When truncating a large fragmented file, it may require modifying so many block bitmaps and group descriptors that it forces a journal transaction to close out, stalling the unlink operation.

- Because of this per-transaction limit, truncate needs to zero the [dt]indirect blocks starting from the end of the file, in case it needs to start a new transaction in the middle of the truncate (ext3 guarantees that a partially-completed truncate will be consistent/completed after a crash).
- The read/write of the file's [dt]indirect blocks from the end of the file to the beginning can take a lot of time, as it does this in single-block chunks and the blocks are not contiguous.

In order to reduce the latency associated with large file truncates and unlinks on the Lustre® filesystem (which is commonly used by scientific computing applications handling very large files), the ability for ext3 to perform asynchronous unlink/truncate was implemented by Andreas Dilger in early 2003.

The delete thread is a kernel thread that services a queue of inode unlink or truncate-to-zero requests that are intercepted from normal `ext3_delete_inode()` and `ext3_truncate()` calls. If the inode to be unlinked/truncated is small enough, or if there is any error in trying to defer the operation, it is handled immediately; otherwise, it is put into the delete thread queue. In the unlink case, the inode is just put into the queue and the delete thread is woke up, before returning to the caller. For the truncate-to-zero case, a free inode is allocated and the blocks are moved over to the new inode before waking the thread and returning to the caller. When the delete thread is woke up, it does a normal truncate of all the blocks on each inode in the list, and then frees the inode.

In order to handle these deferred delete/truncate requests in a crash-safe manner, the inodes to be unlinked/truncated are added into the ext3 orphan list. This is an already existing mechanism by which ext3 handles file unlink/truncates that might be interrupted by a crash. A persistent singly-linked list of inode numbers is linked from the superblock and, if this list is not empty at filesystem mount time, the ext3 code will first walk the list and delete/truncate all of the files on it before the mount is completed.

The delete thread was written for 2.4 kernels, but is currently only in use for Lustre. The patch has not yet been ported to 2.6, but the amount of effort needed to do so is expected to be relatively small, as the ext3 code has changed relatively little in this area.

For extent-mapped files, the need to have asynchronous unlink/truncate is much less, because the number of metadata blocks is greatly reduced for a given file size (unless the file is very fragmented). An alternative to the delete thread (for both files using extent maps as well as indirect blocks) would be to walk the inode and pre-compute the number of bitmaps and group descriptors that would be modified by the operation, and try to start a single transaction of that size. If this transaction can be started, then all of the indirect, double indirect, and triple indirect blocks (also referenced as [d,t] indirect blocks) no longer have to be zeroed out, and we only have to update the block bitmaps and their group summaries, reducing the amount of I/O considerably for files using indirect blocks. Also, the walking of the file metadata blocks can be done in forward order and asynchronous readahead can be started for indirect blocks to make more efficient use of the disk. As an added benefit, we would regain the ability to undelete files in ext3 because we no longer have to zero out all of the metadata blocks.

4.5 Increased nlinks support

The use of a 16-bit value for an inode's link count (`i_nlink`) limits the number of hard links on an inode to 65535. For directories, it

starts with a link count of 2 (one for "." and one for "..") and each subdirectory has a hard link to its parent, so the number of subdirectories is similarly limited.

The ext3 implementation further reduced this limit to 32000 to avoid signed-int problems. Before indexed directories were implemented, the practical limit for files/subdirectories was about 10000 in a single directory.

A patch was implemented to overcome this subdirectory limit by not counting the subdirectory links after the counter overflowed (at 65000 links actually); instead, a link count of one is stored in the inode. The ext3 code already ignores the link count when determining if a directory is full or empty, and a link count of one is otherwise not possible for a directory.

Using a link count of one is also required because userspace tools like "find" optimize their directory walking by only checking a number of subdirectories equal to the link count minus two. Having a directory link count of one disables that heuristic.

4.6 Parallel directory operations

The Lustre filesystem (which is built on top of the ext3 filesystem) has to meet very high goals for concurrent file creation in a single directory (5000 creates/second for 10 million files) for some of its implementations. In order to meet this goal, and to allow this rate to scale with the number of CPUs in a server, the implementation of parallel directory operations (pdirops) was done by Alex Tomas in mid 2003. This patch allows multiple threads to concurrently create, unlink, and rename files within a single directory.

There are two components in the pdirops patches: one in the VFS to lock individual entries in a directory (based on filesystem preference), instead of using the directory inode semaphore to provide exclusive access to the directory; the second patch is in ext3 to implement proper locking based on the filename.

In the VFS, the directory inode semaphore actually protects two separate things. It protects the filesystem from concurrent modification of a single directory and it also protects the dcache from races in creating the same dentry multiple times for concurrent lookups. The pdirops VFS patch adds the ability to lock individual dentries (based on the dentry hash value) within a directory to prevent concurrent dcache creation. All of the places in the VFS that would take `i_sem` on a directory instead call `lock_dir()` and `unlock_dir()` to determine what type of locking is desired by the filesystem.

In ext3, the locking is done on a per-directory-leaf-block basis. This is well suited to the directory-indexing scheme, which has a tree with leaf blocks and index blocks that very rarely change. In the rare case that adding an entry to the leaf block requires that an index block needs locking the code restarts at the top of the tree and keeps the lock(s) on the index block(s) that need to be modified. At about 100,000 entries, there are 2-level index blocks that further reduce the chance of lock collisions on index blocks. By not locking index blocks initially, the common case where no change needs to be made to the index block is improved.

The use of the pdirops VFS patch was also shown to improve the performance of the tmpfs filesystem, which needs no other locking than the dentry locks.

5 Performance comparison

In this section, we will discuss some performance comparisons between the ext3 filesys-

tem found on the 2.4 kernel and the 2.6 kernel. The goal is to evaluate the progress ext3 has made over the last few years. Of course, many improvements other than the ext3 specific features, for example, VM changes, block I/O layer re-write, have been added to the Linux 2.6 kernel, which could affect the performance results overall. However, we believe it is still worthwhile to make the comparison, for the purpose of illustrating the improvements made to ext3 on some workload(s) now, compared with a few years ago.

We selected linux 2.4.29 kernel as the baseline, and compared it with the Linux 2.6.10 kernel. Linux 2.6.10 contains all the features discussed in Section 2, except the EA-in-inode feature, which is not relevant for the benchmarks we had chosen. We also performed the same benchmarks using a Linux 2.6.10 kernel patched with Alex Tomas' extents patch set, which implements extents, delayed allocation, and extents-based multiple block allocation. We plan to run the same benchmarks against a Linux 2.6.10 kernel with some of the patches described in Section 4 in the future.

In this study we chose two benchmarks. One is tiobench, a benchmark testing filesystem sequential and random I/O performance with multiple running threads. Another benchmark we used is filemark, a modified postmark[8] benchmark which simulates I/O activity on a mail server with multiple threads mode. Filemark was used by Ray Bryant when he conducted filesystem performance study on Linux 2.4.17 kernel three years ago [3].

All the tests were done on the same 8-CPU 700 MHZ Pentium III system with 1 GB RAM. All the tests were run with ext3's writeback journaling mode enabled. When running tests with the extents patch set, the filesystem was mouted with the appropriate mount options to enable the extents, multiple block allocation, and delayed allocation features. These test runs

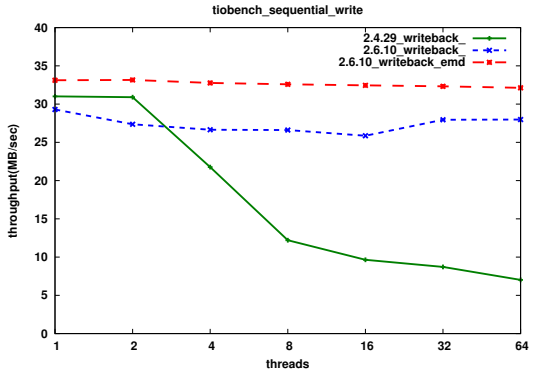

Figure 1: tiobench sequential write throughput results comparison

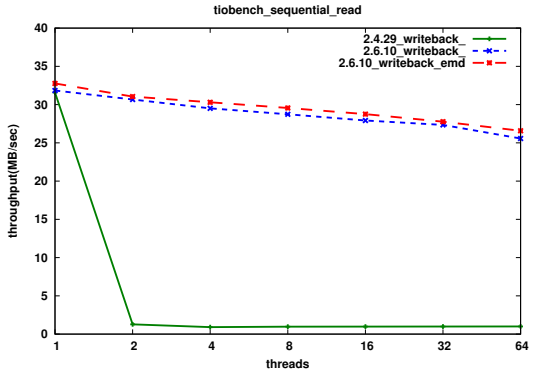

Figure 2: tiobench sequential read throughput results comparison

are shown as "`2.6.10_writeback_emd`" in the graphs.

5.1 Tiobench comparison

Although there have been a huge number of changes between the Linux 2.4.29 kernel to the Linux 2.6.10 kernel could affect overall performance (both in and outside of the ext3 filesystem), we expect that two ext3 features, removing BKL from ext3 (as described in Section 2.2) and reservation based block allocation (as described in Section 2.3) are likely to significantly impact the throughput of the tiobench

benchmark. In this sequential write test, multiple threads are sequentially writing/allocating blocks in the same directory. Allowing allocations concurrently in this case most likely will reduces the CPU usage and improves the throughput. Also, with reservation block allocation, files created by multiple threads in this test could be more contiguous on disk, and likely reduce the latency while writing and sequential reading after that.

Figure 1 and Figure 2 show the sequential write and sequential read test results of the tiobench benchmark, on the three selected kernels, with threads ranging from 1 to 64. The total files size used in this test is 4GB and the blocksize is 16348 byte. The test was done on a single 18G SCSI disk. The graphs indicate significant throughput improvement from the 2.4.29 kernel to the Linux 2.6.10 kernel on this particular workload. Figure 2 shows the sequential read throughput has been significantly improved from Linux 2.4.29 to Linux 2.6.10 on ext3 as well.

When we applied the extents patch set, we saw an additional 7-10% throughput improvement on tiobench sequential write test. We suspect the improvements comes from the combination of delayed allocation and multiple block allocation patches. As we noted earlier, having both features could help lay out files more contiguously on disk, as well as reduce the times to update the metadata, which is quite expensive and happens quite frequently with the current ext3 single block allocation mode. Future testing are needed to find out which feature among the three patches (extents, delayed allocation and extent allocation) is the key contributor of this improvement.

5.2 Filemark comparison

A Filemark execution includes three phases: creation, transaction, and delete phase. The

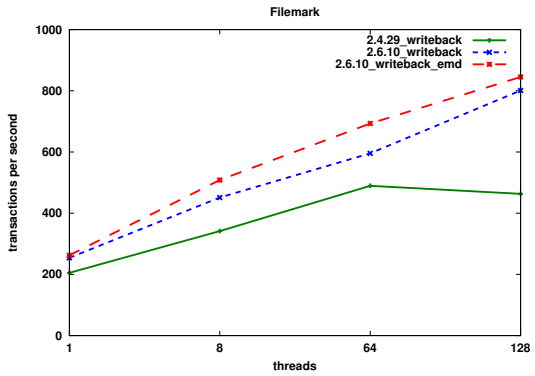

Figure 3: Filemark benchmark transaction rate comparison

transaction phase includes file read and append operations, and some file creation and removal operations. The configuration we used in this test is the so called “medium system” mentioned in Bryant’s Linux filesystem performance study [3]. Here we run filemark with 4 target directories, each on a different disk, 2000 subdirectories per target directory, and 100,000 total files. The file sizes ranged from 4KB to 16KB and the I/O size was 4KB. Figure 3 shows the average transactions per second during the transaction phase, when running Filemark with 1, 8, 64, and 128 threads on the three kernels.

This benchmark uses a varying number of threads. We therefore expected the scalability improvements to the ext3 filesystem in the 2.6 kernel should improve Linux 2.6’s performance for this benchmark. In addition, during the transaction phase, some files are deleted soon after the benchmark creates or appends data to those files. The delayed allocation could avoid the need for disk updates for metadata changes at all. So we expected Alex’s delayed allocation to improve the throughput on this benchmark as well.

The results are shown in Figure 3. At 128 threads, we see that the 2.4.29 kernel had sig-

nificant scalability problems, which were addressed in the 2.6.10 kernel. At up to 64 threads, there is approximately a 10% to 15% improvement in the transaction rate between Linux 2.4.29 and Linux 2.6.10. With the extents patch set applied to Linux 2.6.10, the transaction rate is increased another 10% at 64 threads. In the future, we plan to do futher work to determine how much of the additional 10% improvement can be ascribed to the different components of the extents patch set.

More performance results, both of the benchmark tests described above, and additional benchmark tests expected to be done before the 2005 OLS conference can be found at `http://ext2.sourceforge.net/ols05-testing`.

6 Future Work

This section will discuss some features that are still on the drawing board.

6.1 64 bit block devices

For a long time the Linux block layer limited the size of a single filesystem to 2 TB ($2^{32} *$ 512-byte sectors), and in some cases the SCSI drivers further limited this to 1TB because of signed/unsigned integer bugs. In the 2.6 kernels there is now the ability to have larger block devices and with the growing capacity and decreasing cost of disks the desire to have larger ext3 filesystems is increasing. Recent vendor kernel releases have supported ext3 filesystems up to 8 TB and which can theoretically be as large as 16 TB before it hits the 2^{32} filesystem block limit (for 4 KB blocks and the 4 KB `PAGE_SIZE` limit on i386 systems). There is also a page cache limit of 2^{32} pages in an address space, which are used for buffered block devices. This limit affects both ext3's internal metadata blocks, and the use of buffered block devices when running e2fsprogs on a device to create the filesystem in the first place. So this imposes yet another 16TB limit on the filesystem size, but only on 32-bit architectures.

However, the demand for larger filesystems is already here. Large NFS servers are in the tens of terabytes, and distributed filesystems are also this large. Lustre uses ext3 as the backend storage for filesystems in the hundreds of terabytes range by combining dozens to hundreds of individual block devices and smaller ext3 filesystems in the VFS layer, and having larger ext3 filesystems would avoid the need to artificially fragment the storage to fit within the block and filesystem size limits.

Extremely large filesystems introduce a number of scalability issues. One such concern is the overhead of allocating space in very large volumes, as described in Section 3.3. Another such concern is the time required to back up and perform filesystem consistency checks on very large filesystems. However, the primier issue with filesystems larger than 2^{32} filesystem blocks is that the traditional indirect block mapping scheme only supports 32-bit block numbers. The additional fact that filling such a large filesystem would take many millions of indirect blocks (over 1% of the whole filesystem, at least 160 GB of just indirect blocks) makes the use of the indirect block mapping scheme in such large filesystems undesirable.

Assuming a 4 KB blocksize, a 32-bit block number limits the maximum size of the filesystem to 16 TB. However, because the superblock format currently stores the number of block groups as a 16-bit integer, and because (again on a 4 KB blocksize filesystem) the maximum number of blocks in a block group is 32,768 (the number of bits in a single 4k block, for the block allocation bitmap), a combination of

these constraints limits the maximum size of the filesystem to 8 TB.

One of the plans for growing beyond the 8/16 TB boundary was to use larger filesystem blocks (8 KB up to 64 KB), which increases the filesystem limits such as group size, filesystem size, maximum file size, and makes block allocation more efficient for a given amount of space. Unfortunately, the kernel currently limits the size of a page/buffer to virtual memory's page size, which is 4 KB for i386 processors. A few years ago, it was thought that the advent of 64-bit processors like the Alpha, PPC64, and IA64 would break this limit and when they became commodity parts everyone would be able to take advantage of them. The unfortunate news is that the commodity 64-bit processor architecture, x86_64, also has a 4 KB page size in order to maintain compatibility with its i386 ancestors. Therefore, unless this particular limitation in the Linux VM can be lifted, most Linux users will not be able to take advantage of a larger filesystem block size for some time.

These factors point to a possible paradigm shift for block allocations beyond the 8 TB boundary. One possibility is to use only larger extent based allocations beyond the 8 TB boundary. The current extent layout described in Section 3.1 already has support for physical block numbers up to 2^{48} blocks, though with *only* 2^{32} blocks (16 TB) for a single file. If, at some time in the future larger VM page sizes become common, or the kernel is changed to allow buffers larger than the the VM page size, then this will allow filesystem growth up to 2^{64} bytes and files up to 2^{48} bytes (assuming 64 KB blocksize). The design of the extent structures also allows for additional extent formats like a full 64-bit physical and logical block numbers if that is necessary for 4 KB `PAGE_SIZE` systems, though they would have to be 64-bit in order for the VM to address files and storage devices this large.

It may also make sense to restrict inodes to the first 8 TB of disk, and in conjunction with the extensible inode table discussed in Section 6.2 use space within that region to allocate all inodes. This leaves the > 8 TB space free for efficient extent allocations.

6.2 Extensible Inode Table

Adding an dynamically extensible inode table is something that has been discussed extensively by ext2/3 developers, and the issues that make adding this feature difficult have been discussed before in [15]. Quickly summarized, the problem is a number of conflicting requirements:

- We must maintain enough backup metadata about the dynamic inodes to allow us to preserve ext3's robustness in the presence of lost disk blocks as far as possible.
- We must not renumber existing inodes, since this would require searching and updating all directory entries in the filesystem.
- Given the inode number the block allocation algorithms must be able to determine the block group where the inode is located.
- The number of block groups may change since ext3 filesystems may be resized.

Most obvious solutions will violate one or more of the above requirements. There is a clever solution that can solve the problem, however, by using the space counting backwards from $2^{31} - 1$, or "negative" inode. Since the number of block groups is limited by $2^{32}/(8 * \mathit{blocksize})$, and since the maximum number of inodes per block group is also the same as the maximum number of blocks per block group

is $(8 * blocksize)$, and if inode numbers and block numbers are both 32-bit integers, then the number of inodes per block group in the "negative" inode space is simply $(8 * blocksize)$ - *normal-inodes-per-blockgroup*. The location of the inode blocks in the negative inode space are stored in a reserved inode.

This particular scheme is not perfect, however, since it is not extensible to support 64 bit block numbers unless inode numbers are also extended to 64 bits. Unfortunately, this is not so easy, since on 32-bit platforms, the Linux kernel's internal inode number is 32 bits. Worse yet, the `ino_t` type in the `stat` structure is also 32 bits. Still, for filesystems that are utilizing the traditional 32 bit block numbers, this is still doable.

Is it worth it to make the inode table extensible? Well, there are a number of reasons why an extensible inode table is interesting. Historically, administrators and the `mke2fs` program have always over-allocated the number of inodes, since the number of inodes can not be increased after the filesystem has been formatted, and if all of the inodes have been exhausted, no additional files can be created even if there is plenty of free space in the filesystem. As inodes get larger in order to accommodate the `EA-in-inode` feature, the overhead of over-allocating inodes becomes significant. Therefore, being able to initially allocate a smaller number of inodes and adding more inodes later as needed is less wasteful of disk space. A smaller number of initial inodes also makes the the initial `mke2fs` takes less time, as well as speeding up the `e2fsck` time.

On the other hand, there are a number of disadvantages of an extensible inode table. First, the "negative" inode space introduces quite a bit of complexity to the inode allocation and read/write functions. Second, as mentioned earlier, it is not easily extensible to filesystems that implement the proposed 64-bit block number extension. Finally, the filesystem becomes more fragile, since if the reserved inode that describes the location of the "negative" inode space is corrupted, the location of all of the extended inodes could be lost.

So will extensible inode tables ultimately be implemented? Ultimately, this will depend on whether an ext2/3 developer believes that it is worth implementing — whether someone considers extensible inode an "itch that they wish to scratch". The authors believe that the benefits of this feature only slightly outweigh the costs, but perhaps not by enough to be worth implementing this feature. Still, this view is not unanimously held, and only time will tell.

7 Conclusion

As we have seen in this paper, there has been a tremendous amount of work that has gone into the ext2/3 filesystem, and this work is continuing. What was once essentially a simplified BSD FFS descendant has turned into an enterprise-ready filesystem that can keep up with the latest in storage technologies.

What has been the key to the ext2/3 filesystem's success? One reason is the forethought of the initial ext2 developers to add compatibility feature flags. These flags have made ext2 easily extensible in a variety of ways, without sacrificing compatibility in many cases.

Another reason can be found by looking at the company affiliations of various current and past ext2 developers: Cluster File Systems, Digeo, IBM, OSDL, Red Hat, SuSE, VMWare, and others. Different companies have different priorities, and have supported the growth of ext2/3 capabilities in different ways. Thus, this diverse and varied set of developers has allowed the ext2/3 filesystem to flourish.

The authors have no doubt that the ext2/3 filesystem will continue to mature and come to be suitable for a greater and greater number of workloads. As the old Frank Sinatra song stated, "The best is yet to come."

Patch Availability

The patches discussed in this paper can be found at `http://ext2.sourceforge.net/ols05-patches`.

Acknowledgments

The authors would like to thank all ext2/3 developers who make ext2/3 better, especially grateful to Andrew Morton, Stephen Tweedie, Daniel Phillips, and Andreas Gruenbacher for many enlightening discussions and inputs.

We also owe thanks to Ram Pai, Sonny Rao, Laurent Vivier and Avantika Mathur for their help on performance testing and analysis, and to Paul Mckenney, Werner Almesberger, and David L Stevens for paper reviewing and refining. And lastly, thanks to Gerrit Huizenga who encouraged us to finally get around to submitting and writing this paper in the first place. :-)

References

[1] ALMESBERGER, W., AND VAN DEN BRINK, B. Active block i/o scheduling systems (abiss). In *Linux Kongress* (2004).

[2] BLIGH, M. Re: 2.5.70-mm1, May, 2003. `http://marc.theaimsgroup.com/?l=linux-mm&m=105418949116972&w=2`.

[3] BRYANT, R., FORESTER, R., AND HAWKES, J. Filesystem performance and scalability in linux 2.4.17. In *USENIX Annual Technical Conference* (2002).

[4] CARD, R., TWEEDIE, S., AND TS'O, T. Design and implementation of the second extended filesystem. In *First Dutch International Symposium on Linux* (1994).

[5] CORBET, J. Which filesystem for samba4? `http://lwn.net/Articles/112566/`.

[6] DILGER, A. E. Online resizing with ext2 and ext3. In *Ottawa Linux Symposium* (2002), pp. 117–129.

[7] GRUENBACHER, A. Posix access control lists on linux. In *USENIX Annual Technical Conference* (2003), pp. 259–272.

[8] KATCHER, J. Postmark a new filesystem benchmark. Tech. rep., Network Appliances, 2002.

[9] MORTON, A. Re: [ext2-devel] [rfc] adding 'delayed allocation' support to ext3 writeback, April, 2005. `http://marc.theaimsgroup.com/?l=ext2-devel&m=111286563813799&w=2`.

[10] PHILLIPS, D. A directory index for ext2. In *5th Annual Linux Showcase and Conference* (2001), pp. 173–182.

[11] RAO, S. Re: [ext2-devel] re: Latest ext3 patches (extents, mballoc, delayed allocation), February, 2005. `http://marc.theaimsgroup.com/?l=ext2-devel&m=110865997805872&w=2`.

[12] SWEENY, A. Scalability in the xfs file system. In *USENIX Annual Technical Conference* (1996).

[13] TOMAS, A. Speeding up ext2, March, 2003. http://lwn.net/Articles/25363/.

[14] TWEEDIE, S. Ext3 journalling filesystem. In *Ottawa Linux Symposium* (2000).

[15] TWEEDIE, S., AND TS'O, T. Y. Planned extensions to the linux ext2/3 filesystem. In *USENIX Annual Technical Conference* (2002), pp. 235–244.

[16] VIVIER, L. Filesystems comparison using sysbench and mysql, April, 2005. http://marc.theaimsgroup.com/?l=ext2-devel&m=111505444514651&w=2.

Legal Statement