Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Review Committee

Proceedings Formatting Team

The new ext4 filesystem: current status and future plans

Avantika Mathur, Mingming Cao, Suparna Bhattacharya
IBM Linux Technology Center
mathur@us.ibm.com, cmm@us.ibm.com, suparna@in.ibm.com
Andreas Dilger, Alex Tomas
Cluster Filesystem Inc.
adilger@clusterfs.com, alex@clusterfs.com
Laurent Vivier
Bull S.A.S.
laurent.vivier@bull.net

Abstract

Ext3 has been the most widely used general Linux® filesystem for many years. In keeping with increasing disk capacities and state-of-the-art feature requirements, the next generation of the ext3 filesystem, ext4, was created last year. This new filesystem incorporates scalability and performance enhancements for supporting large filesystems, while maintaining reliability and stability. Ext4 will be suitable for a larger variety of workloads and is expected to replace ext3 as the "Linux filesystem."

In this paper we will first discuss the reasons for starting the ext4 filesystem, then explore the enhanced capabilities currently available and planned for ext4, discuss methods for migrating between ext3 and ext4, and finally compare ext4 and other filesystem performance on three classic filesystem benchmarks.

1 Introduction

Ext3 has been a very popular Linux filesystem due to its reliability, rich feature set, relatively good performance, and strong compatibility between versions. The conservative design of ext3 has given it the reputation of being stable and robust, but has also limited its ability to scale and perform well on large configurations.

With the pressure of increasing capabilities of new hardware and online resizing support in ext3, the requirement to address ext3 scalability and performance is more urgent than ever. One of the outstanding limits faced by ext3 today is the 16 TB maximum filesystem size. Enterprise workloads are already approaching this limit, and with disk capacities doubling every year and 1 TB hard disks easily available in stores, it will soon be hit by desktop users as well.

To address this limit, in August 2006, we posted a series of patches introducing two key features to ext3: larger filesystem capacity and extents mapping. The patches unavoidably change the on-disk format and break forwards compatibility. In order to maintain the stability of ext3 for its massive user base, we decided to branch to ext4 from ext3.

The primary goal of this new filesystem is to address scalability, performance, and reliability issues faced by ext3. A common question is why not use XFS or start an entirely new filesystem from scratch? We want to give the large number of ext3 users the opportunity to easily upgrade their filesystem, as was done from ext2 to ext3. Also, there has been considerable investment in the capabilities, robustness, and reliability of ext3 and e2fsck. Ext4 developers can take advantage of this previous work, and focus on adding advanced features and delivering a new scalable enterprise-ready filesystem in a short time frame.

Thus, ext4 was born. The new filesystem has been in mainline Linux since version 2.6.19. As of the writing of this paper, the filesystem is marked as developmental, titled ext4dev, explicitly warning users that it is not ready for production use. Currently, extents and 48-bit block numbers are included in ext4, but there are many new filesystem features in the roadmap that will be discussed throughout this paper. The current ext4 development git tree is hosted at `git://git.kernel.org/`

pub/scm/linux/kernel/git/tytso/ext4. Up-to-date ext4 patches and feature discussions can be found at the ext4 wiki page, http://ext4.wiki.kernel.org.

Some of the features in progress could possibly continue to change the on-disk layout. Ext4 will be converted from development mode to stable mode once the layout has been finalized. At that time, ext4 will be available for general use by all users in need of a more scalable and modern version of ext3. In the following three sections we will discuss new capabilities currently included in or planned for ext4 in the areas of scalability, fragmentation, and reliability.

2 Scalability enhancements

The first goal of ext4 was to become a more scalable filesystem. In this section we will discuss the scalability features that will be available in ext4.

2.1 Large filesystem

The current 16 TB filesystem size limit is caused by the 32-bit block number in ext3. To enlarge the filesystem limit, the straightforward method is to increase the number of bits used to represent block numbers and then fix all references to data and metadata blocks.

Previously, there was an extents[3] patch for ext3 with the capacity to support 48-bit physical block numbers. In ext4, instead of just extending the block numbers to 64-bits based on the current ext3 indirect block mapping, the ext4 developers decided to use extents mapping with 48-bit block numbers. This both increases filesystem capacity and improves large file efficiency. With 48-bit block numbers, ext4 can support a maximum filesystem size up to $2^{(48+12)} = 2^{60}$ bytes (1 EB) with 4 KB block size.

After changing the data block numbers to 48-bit, the next step was to correct the references to metadata blocks correspondingly. Metadata is present in the superblock, the group descriptors, and the journal. New fields have been added at the end of the superblock structure to store the most significant 32 bits for block-counter variables, s_free_blocks_count, s_blocks_count, and s_r_blocks_count, extending them to 64 bits. Similarly, we introduced new 32-bit fields at the end of the block group descriptor structure to store the most significant bits of 64-bit values for bitmaps and inode table pointers.

Since the addresses of modified blocks in the filesystem are logged in the journal, the journaling block layer (JBD) is also required to support at least 48-bit block addresses. Therefore, JBD was branched to JBD2 to support more than 32-bit block numbers at the same time ext4 was forked. Although currently only ext4 is using JBD2, it can provide general journaling support for both 32-bit and 64-bit filesystems.

One may question why we chose 48-bit rather than full 64-bit support. The 1 EB limit will be sufficient for many years. Long before this limit is hit there will be reliability issues that need to be addressed. At current speeds, a 1 EB filesystem would take 119 years to finish one full e2fsck, and 65536 times that for a 2^{64} blocks (64 ZB) filesystem. Overcoming these kind of reliability issues is the priority of ext4 developers before addressing full 64-bit support and is discussed later in the paper.

2.1.1 Future work

After extending the limit created by 32-bit block numbers, the filesystem capacity is still restricted by the number of block groups in the filesystem. In ext3, for safety concerns all block group descriptors copies are kept in the first block group. With the new uninitialized block group feature discussed in section 4.1 the new block group descriptor size is 64 bytes. Given the default 128 MB(2^{27} bytes) block group size, ext4 can have at most $2^{27}/64 = 2^{21}$ block groups. This limits the entire filesystem size to $2^{21} * 2^{27} = 2^{48}$ bytes or 256TB.

The solution to this problem is to use the *metablock group* feature (META_BG), which is already in ext3 for all 2.6 releases. With the META_BG feature, ext4 filesystems are partitioned into many metablock groups. Each metablock group is a cluster of block groups whose group descriptor structures can be stored in a single disk block. For ext4 filesystems with 4 KB block size, a single metablock group partition includes 64 block groups, or 8 GB of disk space. The metablock group feature moves the location of the group descriptors from the congested first block group of the whole filesystem into the first group of each metablock group itself. The backups are in the second and last group of each metablock group. This increases the 2^{21} maximum

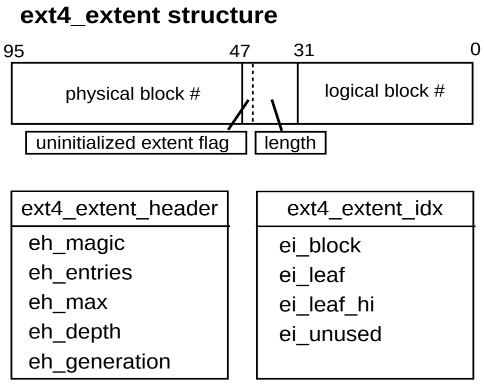

Figure 1: Ext4 extents, header and index structures

block groups limit to the hard limit 2^{32}, allowing support for the full 1 EB filesystem.

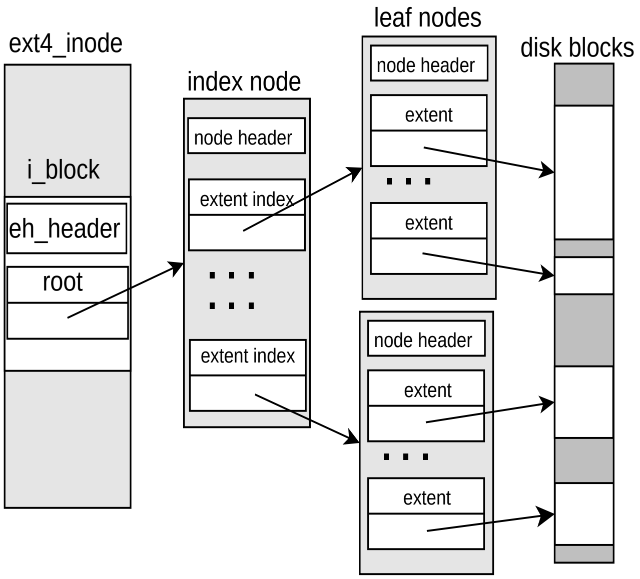

Figure 2: Ext4 extent tree layout

2.2 Extents

The ext3 filesystem uses an indirect block mapping scheme providing one-to-one mapping from logical blocks to disk blocks. This scheme is very efficient for sparse or small files, but has high overhead for larger files, performing poorly especially on large file delete and truncate operations [3].

As mentioned earlier, extents mapping is included in ext4. This approach efficiently maps logical to physical blocks for large contiguous files. An *extent* is a single descriptor which represents a range of contiguous physical blocks. Figure 1 shows the extents structure. As we discussed in previously, the physical block field in an extents structure takes 48 bits. A single extent can represent 2^{15} contiguous blocks, or 128 MB, with 4 KB block size. The MSB of the extent length is used to flag uninitialized extents, used for the preallocation feature discussed in Section 3.1.

Four extents can be stored in the ext4 inode structure directly. This is generally sufficient to represent small or contiguous files. For very large, highly fragmented, or sparse files, more extents are needed. In this case a constant depth extent tree is used to store the extents map of a file. Figure 2 shows the layout of the extents tree. The root of this tree is stored in the ext4 inode structure and extents are stored in the leaf nodes of the tree.

Each node in the tree starts with an extent header (Figure 1), which contains the number of valid entries in the node, the capacity of entries the node can store, the depth of the tree, and a magic number. The magic number can be used to differentiate between different versions of extents, as new enhancements are made to the feature, such as increasing to 64-bit block numbers.

The extent header and magic number also add much-needed robustness to the on-disk structure of the data files. For very small filesystems, the block-mapped files implicitly depended on the fact that random corruption of an indirect block would be easily detectable, because the number of valid filesystem blocks is a small subset of a random 32-bit integer. With growing filesystem sizes, random corruption in an indirect block is by itself indistinguishable from valid block numbers.

In addition to the simple magic number stored in the extent header, the tree structure of the extent tree can be verified at runtime or by e2fsck in several ways. The ext4_extent_header has some internal consistency (eh_entries and eh_max) that also depends on the filesystem block size. eh_depth decreases from the root toward the leaves. The ext4_extent entries in a leaf block must have increasing ee_block numbers, and must not overlap their neighbors with ee_len. Similarly, the ext4_extent_idx also needs increasing ei_block values, and the range of blocks that an index covers can be verified against the actual range of blocks in the extent leaf.

Currently, extents mapping is enabled in ext4 with the *extents* mount option. After the filesystem is mounted, any new files will be created with extent mapping. The benefits of extent maps are reflected in the performance evaluation Section 7.

2.2.1 Future work

Extents are not very efficient for representing sparse or highly fragmented files. For highly fragmented files, we could introduce a new type of extent, a block-mapped extent. A different magic number, stored in the extent header, distinguishes the new type of leaf block, which contains a list of allocated block numbers similar to an ext3 indirect block. This would give us the increased robustness of the extent format, with the block allocation flexibility of the block-mapped format.

In order to improve the robustness of the on-disk data, there is a proposal to create an "extent tail" in the extent blocks, in addition to the extent header. The extent tail would contain the inode number and generation of the inode that has allocated the block, and a checksum of the extent block itself (though not the data). The checksum would detect internal corruption, and could also detect misplaced writes if the block number is included therein. The inode number could be used to detect corruption that causes the tree to reference the wrong block (whether by higher-level corruption, or misplaced writes). The inode number could also be used to reconstruct the data of a corrupted inode or assemble a deleted file, and also help in doing reverse-mapping of blocks for defragmentation among other things.

2.3 Large files

In Linux, file size is calculated based on the i_blocks counter value. However, the unit is in sectors (512 bytes), rather than in the filesystem block size (4096 bytes by default). Since ext4's i_blocks is a 32-bit variable in the inode structure, this limits the maximum file size in ext4 to $2^{32} * 512$ bytes = 2^{41} bytes = 2 TB. This is a scalability limit that ext3 has planned to break for a while.

The solution for ext4 is quite straightforward. The first part is simply changing the i_blocks units in the ext4 inode to filesystem blocks. An ROCOMPAT feature flag HUGE_FILE is added in ext4 to signify that the i_blocks field in some inodes is in units of filesystem block size. Those inodes are marked with a flag EXT4_HUGE_FILE_FL, to allow existing inodes to keep i_blocks in 512-byte units without requiring a full filesystem conversion. In addition, the i_blocks variable is extended to 48 bits by using some of the reserved inode fields. We still have the limitation of 32 bit logical block numbers with the current extent format, which limits the file size to 16TB. With the flexible extents format in the future (see Section 2.2.1), we may remove that limit and fully use the 48-bit i_blocks to enlarge the file size even more.

2.4 Large number of files

Some applications already create billions of files today, and even ask for support for trillions of files. In theory, the ext4 filesystem can support billions of files with 32-bit inode numbers. However, in practice, it cannot scale to this limit. This is because ext4, following ext3, still allocates inode tables statically. Thus, the maximum number of inodes has to be fixed at filesystem creation time. To avoid running out of inodes later, users often choose a very large number of inodes up-front. The consequence is unnecessary disk space has to be allocated to store unused inode structures. The wasted space becomes more of an issue in ext4 with the larger default inode. This also makes the management and repair of large filesystems more difficult than it should be. The uninitialized group feature (Section 4.1) addresses this issue to some extent, but the problem still exists with aged filesystems in which the used and unused inodes can be mixed and spread across the whole filesystem.

Ext3 and ext4 developers have been thinking about supporting dynamic inode allocation for a while [9, 3]. There are three general considerations about the dynamic inode table allocation:

- Performance: We need an efficient way to translate inode number to the block where the inode structure is stored.
- Robustness: e2fsck should be able to locate inode table blocks scattered across the filesystem, in the case the filesystem is corrupted.
- Compatibility: We need to handle the possible inode number collision issue with 64-bit inode numbers on 32-bit systems, due to overflow.

These three requirements make the design challenging.

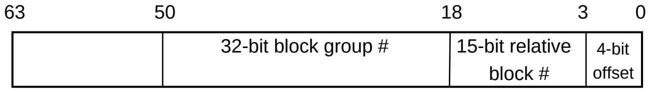

Figure 3: 64-bit inode layout

With dynamic inode tables, the blocks storing the inode structure are no longer at a fixed location. One way to efficiently map the inode number to the block storing the corresponding inode structure, is encoding the block number into the inode number directly, similar to what is done in XFS. This implies the use of 64-bit inode numbers. The low four to five bits of the inode number store the offset bits within the inode table block. The rest store the 32-bit block group number as well as 15-bit relative block number within the group, shown in Figure 3. Then, a cluster of contiguous inode table blocks (ITBC) can be allocated on demand. A bitmap at the head of the ITBC would be used to keep track of the free and used inodes, allowing fast inode allocation and deallocation.

In the case where the filesystem is corrupted, the majority of inode tables could be located by checking the directory entries. To further address the reliability concern, a magic number could be stored at the head of the ITBC, to help e2fsck to recognize this metadata block.

Relocating inodes becomes tricky with this block-number-in-inode-number proposal. If the filesystem is resized or defragmented, we may have to change the location of the inode blocks, which would require changing all references to that inode number. The proposal to address this concern is to have a per-group “inode exception map” that translates an old block/inode number into a new block number where the relocated inode structure is actually stored. The map will usually be empty, unless the inode was moved.

One concern with the 64-bit inode number is the possible inode number collision with 32-bit applications, as applications might still be using 32-bit stat() to access inode numbers and could break. Investigation is underway to see how common this case is, and whether most applications are currently fixed to use the 64-bit stat64(). One way to address this concern is to generate 32-bit inode numbers on 32-bit platforms. Seventeen bits is enough to represent block group numbers on 32-bit architectures, and we could limit the inode table blocks to the first 2^{10} blocks of a block group to construct the 32-bit inode number. This way user applications will be ensured of getting unique inode numbers on 32-bit platforms. For 32-bit applications running on 64-bit platforms, we hope they are fixed by the time ext4 is in production, and this only starts to be an issue for filesystems over 1TB in size.

In summary, dynamic inode allocation and 64-bit inode numbers are needed to support large numbers of files in ext4. The benefits are obvious, but the changes to the on-disk format may be intrusive. The design details are still under discussion.

2.5 Directory scalability

The maximum number of subdirectories contained in a single directory in ext3 is 32,000. To address directory scalability, this limit will be eliminated in ext4 providing unlimited sub-directory support.

In order to better support large directories with many entries, the directory indexing feature[6] will be turned on by default in ext4. By default in ext3, directory entries are still stored in a linked list, which is very inefficient for directories with large numbers of entries. The directory indexing feature addresses this scalability issue by storing directory entries in a constant depth HTree data structure, which is a specialized BTree-like structure using 32-bit hashes. The fast lookup time of the HTree significantly improves performance on large directories. For directories with more than 10,000 files, improvements were often by a factor of 50 to 100 [3].

2.5.1 Future work

While the HTree implementation allowed the ext2 directory format to be improved from linear to a tree search compatibly, there are also limitations to this approach. The HTree implementation has a limit of 510 * 511 4 KB directory leaf blocks (approximately 25M 24-byte filenames) that can be indexed with a 2-level tree. It would be possible to change the code to allow a 3-level HTree. There is also currently a 2 GB file size limit on directories, because the code for using the high 32-bits for i_size on directories was not implemented when the 2 GB limit was fixed for regular files.

Because the hashing used to find filenames in indexed directories is essentially random compared to the linear order in which inodes are allocated, we end up doing random seeks around the disk when accessing many inodes in a large directory. We need to have readdir in hash-index order because directory entries might be moved during the split of a directory leaf block, so to satisfy POSIX requirements we can only safely walk the directory in hash order.

To address this problem, there is a proposal to put the whole inode into the directory instead of just a directory entry that references a separate inode. This avoids the need to seek to the inode when doing a readdir, because the whole inode has been read into memory already in the readdir step. If the blocks that make up the directory are efficiently allocated, then reading the directory also does not require any seeking.

This would also allow dynamic inode allocation, with the directory as the "container" of the inode table. The inode numbers would be generated in a similar manner as previously discussed (Section 2.4), so that the block that an inode resides in can be located directly from the inode number itself. Hard linked files imply that the same block is allocated to multiple directories at the same time, but this can be reconciled by the link count in the inode itself.

We also need to store one or more file names in the inode itself, and this can be done by means of an extended attribute that uses the directory inode number as the EA name. We can then return the name(s) associated with that inode for a single directory immediately when doing readdir, and skip any other name(s) for the inode that belong to hard links in another directory. For efficient name-to-inode lookup in the directory, we would still use a secondary tree similar to the current ext3 HTree (though it would need an entry per name instead of per directory block). But because the directory entries (the inodes themselves) do not get moved as the directory grows, we can just use disk block or directory offset order for readdir.

2.6 Large inode and fast extended attributes

Ext3 supports different inode sizes. The inode size can be set to any power-of-two larger than 128 bytes size up to the filesystem block size by using the *mke2fs -I [inode size]* option at format time. The default inode structure size is 128 bytes, which is already crowded with data and has little space for new fields. In ext4, the default inode structure size will be 256 bytes.

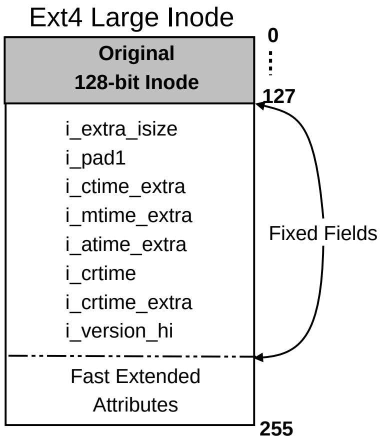

Figure 4: Layout of the large inode

In order to avoid duplicating a lot of code in the kernel and e2fsck, the large inodes keep the same fixed layout for the first 128-bytes, as shown in Figure 4. The rest of the inode is split into two parts: a fixed-field section that allows addition of fields common to all inodes, such as nanosecond timestamps (Section 5), and a section for fast extended attributes (EAs) that consumes the rest of the inode.

The fixed-field part of the inode is dynamically sized, based on what fields the current kernel knows about. The size of this area is stored in each inode in the i_extra_isize field, which is the first field beyond the original 128-byte inode. The superblock also contains two fields, s_min_extra_isize and i_want_extra_isize, which allow down-level kernels to allocate a larger i_extra_isize than it would otherwise do.

The s_min_extra_isize is the guaranteed minimum amount of fixed-field space in each inode. s_want_extra_isize is the desired amount of fixed-field space for new inode, but there is no guarantee that this much space will be available in every inode. A ROCOMPAT feature flag EXTRA_ISIZE indicates whether these superblock fields are valid. The ext4 code will soon also be able to expand i_extra_isize dynamically as needed to cover the fixed fields, so long as there is space available to store the fast EAs or migrate them to an external EA block.

The remaining large inode space may be used for storing EA data inside the inode. Since the EAs are already in memory after the inode is read from disk, this avoids a costly seek to external EA block. This can greatly improve the performance of applications that are using EAs, sometimes by a factor of 3–7 [4]. An external EA block is still available in addition to the fast EA space, which allows storing up to 4 KB of EAs for each file.

The support for fast EAs in large inodes has been available in Linux kernels since 2.6.12, though it is rarely used because many people do not know of this capability at mke2fs time. Since ext4 will have larger inodes, this feature will be enabled by default.

There have also been discussions about breaking the 4 KB EA limit, in order to store larger or more EAs. It is likely that larger single EAs will be stored in their own inode (to allow arbitrary-sized EAs) and it may also be that for many EAs they will be stored in a directory-like structure, possibly leveraging the same code as regular ext4 directories and storing small values inline.

3 Block allocation enhancements

Increased filesystem throughput is the premier goal for all modern filesystems. In order to meet this goal, developers are constantly attempting to reduce filesystem fragmentation. High fragmentation rates cause greater disk access time affecting overall throughput, and increased metadata overhead causing less efficient mapping.

There is an array of new features in line for ext4, which take advantage of the existing extents mapping and are aimed at reducing filesystem fragmentation by improving block allocation techniques.

3.1 Persistent preallocation

Some applications, like databases and streaming media servers, benefit from the ability to preallocate blocks for a file up-front (typically extending the size of the file in the process), without having to initialize those blocks with valid data or zeros. Preallocation helps ensure contiguous allocation as far as possible for a file (irrespective of when and in what order data actually gets written) and guaranteed space allocation for writes within the preallocated size. It is useful when an application has some foreknowledge of how much space the file will require. The filesystem internally interprets the preallocated but not yet initialized portions of the file as zero-filled blocks. This avoids exposing stale data for each block until it is explicitly initialized through a subsequent write. Preallocation must be persistent across reboots, unlike ext3 and ext4 block reservations [3].

For applications involving purely sequential writes, it is possible to distinguish between initialized and uninitialized portions of the file. This can be done by maintaining a single high water mark value representing the size of the initialized portion. However, for databases and other applications where random writes into the preallocated blocks can occur in any order, this is not sufficient. The filesystem needs to be able to identify ranges of uninitialized blocks in the middle of the file. Therefore, some extent based filesystems, like XFS, and now ext4, provide support for marking allocated but uninitialized extents associated with a given file.

Ext4 implements this by using the MSB of the extent length field to indicate whether a given extent contains uninitialized data, as shown in Figure 1. During reads, an uninitialized extent is treated just like a hole, so that the VFS returns zero-filled blocks. Upon writes, the extent must be split into initialized and uninitialized extents, merging the initialized portion with an adjacent initialized extent if contiguous.

Until now, XFS, the other Linux filesystem that implements preallocation, provided an ioctl interface to applications. With more filesystems, including ext4, now providing this feature, a common system-call interface for fallocate and an associated inode operation have been introduced. This allows filesystem-specific implementations of preallocation to be exploited by applications using the posix_fallocate API.

3.2 Delayed and multiple block allocation

The block allocator in ext3 allocates one block at a time during the write operation, which is inefficient for larger I/O. Since block allocation requests are passed through the VFS layer one at a time, the underlying ext3 filesystem cannot foresee and cluster future requests. This also increases the possibility of file fragmentation.

Delayed allocation is a well-known technique in which block allocations are postponed to page flush time, rather than during the write() operation [3]. This method provides the opportunity to combine many block allocation requests into a single request, reducing possible fragmentation and saving CPU cycles. Delayed allocation also avoids unnecessary block allocation for short-lived files.

Ext4 delayed allocation patches have been implemented, but there is work underway to move this support to the VFS layer, so multiple filesystems can benefit from the feature.

With delayed allocation support, multiple block allocation for buffered I/O is now possible. An entire extent, containing multiple contiguous blocks, is allocated at once rather than one block at a time. This eliminates multiple calls to ext4_get_blocks and ext4_new_blocks and reduces CPU utilization.

Ext4 multiple block allocation builds per-block group free extents information based on the on-disk block bitmap. It uses this information to guide the search for free extents to satisfy an allocation request. This free extent information is generated at filesystem mount time and stored in memory using a buddy structure.

The performance benefits of delayed allocation alone are very obvious, and can be seen in Section 7. In a previous study [3], we have seen about 30% improved throughput and 50% reduction in CPU usage with the combined two features. Overall, delayed and multiple block allocation can significantly improve filesystem performance on large I/O.

There are two other features in progress that are built on top of delayed and multiple block allocation, trying to further reduce fragmentation:

- In-core Preallocation: Using the in-core free extents information, a more powerful in-core block preallocation/reservation can be built. This further improves block placement and reduces fragmentation with concurrent write workloads. An inode can have a number of preallocated chunks, indexed by the logical blocks. This improvement can help HPC applications when a number of nodes write to one huge file at very different offsets.
- Locality Groups: Currently, allocation policy decisions for individual file are made independently. If the allocator had knowledge of file relationship, it could intelligently place related files close together, greatly benefiting read performance. The locality groups feature clusters related files together by a given attribute, such as SID or a combination of SID and parent directory. At the deferred page-flush time, dirty pages are written out by groups, instead of by individual files. The number of non-allocated blocks are tracked at the group-level, and upon flush time, the allocator can try to preallocate enough space for the entire group. This space is shared by the files in the group for their individual block allocation. In this way, related files are place tightly together.

In summary, ext4 will have a powerful block allocation scheme that can efficiently handle large block I/O and reduce filesystem fragmentation with small files under multi-threaded workloads.

3.3 Online defragmentation

Though the features discussed in this section improve block allocation to avoid fragmentation in the first place, with age, the filesystem can still become quite fragmented. The ext4 online defragmentation tool, e4defrag, has been developed to address this. This tool can defragment individual files or the entire filesystem. For each file, the tool creates a temporary inode and allocates contiguous extents to the temporary inode using multiple block allocation. It then copies the original file data to the page cache and flushes the dirty pages to the temporary inode's blocks. Finally, it migrates the block pointers from the temporary inode to the original inode.

4 Reliability enhancements

Reliability is very important to ext3 and is one of the reasons for its vast popularity. In keeping with this reputation, ext4 developers are putting much effort into maintaining the reliability of the filesystem. While it is relatively easy for any filesystem designer to make their fields 64-bits in size, it is much more difficult to make such large amounts of space actually usable in the real world.

Despite the use of journaling and RAID, there are invariably corruptions to the disk filesystem. The first line of defense is detecting and avoiding problems proactively by a combination of robust metadata design, internal redundancy at various levels, and built-in integrity checking using checksums. The fallback will always be doing integrity checking (fsck) to both detect and correct problems that will happen anyway.

One of the primary concerns with all filesystems is the speed at which a filesystem can be validated and recovered after corruption. With reasonably high-end RAID storage, a full fsck of a 2TB ext3 filesystem can take between 2 to 4 hours for a relatively "clean" filesystem. This process can degrade sharply to many days if there are large numbers of shared filesystem blocks that need expensive extra passes to correct.

Some features, like extents, have already added to the robustness of the ext4 metadata as previously described. Many more related changes are either complete, in progress, or being designed in order to ensure that ext4 will be usable at scales that will become practical in the future.

4.1 Unused inode count and fast e2fsck

In e2fsck, the checking of inodes in pass 1 is by far the most time consuming part of the operation. This requires reading all of the large inode tables from disk, scanning them for valid, invalid, or unused inodes, and then verifying and updating the block and inode allocation bitmaps. The uninitialized groups and inode table high watermark feature allows much of the lengthy pass 1 scanning to be safely skipped. This can dramatically reduce the total time taken by e2fsck by 2 to 20 times, depending on how full the filesystem is. This feature can be enabled at mke2fs time or using tune2fs via the *-O uninit_groups* option.

With this feature, the kernel stores the number of unused inodes at the end of each block group's inode table. As a result, e2fsck can skip both reading these blocks from disk, and scanning them for in-use inodes. In order to ensure that the unused inode count is safe to use by e2fsck, the group descriptor has a CRC16 checksum added to it that allows validation of all fields therein.

Since typical ext3 filesystems use only in the neighborhood of 1% to 10% of their inodes, and the inode allocation policy keeps a majority of those inodes at the start of the inode table, this can avoid processing a large majority of the inodes and speed up the pass 1 processing. The kernel does not currently increase the unused inodes count, when files are deleted. This counter is updated on every e2fsck run, so in the case where a block group had many inodes deleted, e2fsck will be more efficient in the next run.

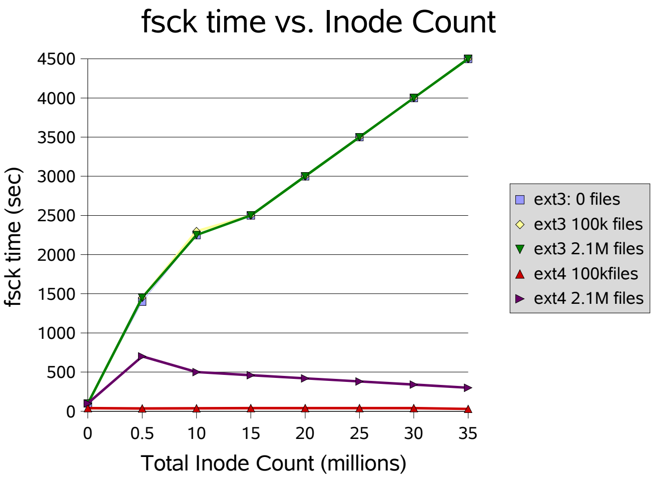

Figure 5: e2fsck performance improvement with uninitialized block groups

Figure 5 shows that e2fsck time on ext3 grows linearly with the total number of inodes in filesystem, regardless of how many are used. On ext3, e2fsck takes the same amount of time with zero used files as with 2.1 million used files. In ext4, with the unused inode high watermark feature, the e2fsck time is only dependent on the number of used inodes. As we can see, fsck of an ext4 filesystem with 100 000 used files takes a fraction of the time ext3 takes.

In addition to the unused inodes count, it is possible for mke2fs and e2fsck to mark a group's block or inode bitmap as uninitialized, so that the kernel does not need to read them from disk when first allocating from the group. Similarly, e2fsck does not need to read these bitmaps from disk, though this does not play a major role in performance improvements. What is more significant is that mke2fs will not write out the bitmaps or inode tables at format time if the *mke2fs -O lazy_bg* feature is given. Writing out the inode tables can take a significant amount of time, and has been known to cause problems for large filesystems due to the amount of dirty pages this generates in a short time.

4.2 Checksumming

Adding metadata checksumming into ext4 will allow it to more easily detect corruption, and behave appropriately instead of blindly trusting the data it gets from disk. The group descriptors already have a checksum added, per the previous section. The next immediate target for checksumming is the journal, because it has such a high density of important metadata and is constantly being written to, so has a higher chance of wearing out the platters or seeing other random corruption.

Adding checksumming to the ext4 journal is nearly complete [7]. In ext3 and ext4, each journal transaction has a header block and commit block. During normal journal operation the commit block is not sent to the disk until the transaction header and all metadata blocks which make up that transaction have been written to disk [8]. The next transaction needs to wait for the previous commit block to hit to disk before it can start to modify the filesystem.

With this two-phase commit, if the commit block has the same transaction number as the header block, it should indicate that the transaction can be replayed at recovery time. If they don't match, the journal recovery is ended. However, there are several scenarios where this can go wrong and lead to filesystem corruption.

With journal checksumming, the journal code computes a CRC32 over all of the blocks in the transaction (including the header), and the checksum is written to the commit block of the transaction. If the checksum does not match at journal recovery time, it indicates that one or more metadata blocks in the transaction are corrupted or were not written to disk. Then the transaction (along with later ones) is discarded as if the computer had crashed slightly earlier and not written a commit block at all.

Since the journal checksum in the commit block allows detection of blocks that were not written into the journal, as an added bonus there is no longer a need for having a two-phase commit for each transaction. The commit block can be written at the same time as the rest of the blocks in the transaction. This can actually speed up the filesystem operation noticeably (as much as 20% [7]), instead of the journal checksum being an overhead.

There are also some long-term plans to add checksumming to the extent tail, the allocation bitmaps, the inodes, and possibly also directories. This can be done efficiently once we have journal checksumming in place. Rather than computing the checksum of filesystem metadata each time it is changed (which has high overhead for often-modified structures), we can write the metadata to the checksummed journal and still be confident that it is valid and correct at recovery time. The blocks can have metadata-specific checksums computed a single time when they are written into the filesystem.

5 Other new features

New features are continuously being added to ext4. Two features expected to be seen in ext4 are nanosecond timestamps and inode versioning. These two features provide precision when dealing with file access times and tracking changes to files.

Ext3 has second resolution timestamps, but with today's high-speed processors, this is not sufficient to record multiple changes to a file within a second. In ext4, since we use a larger inode, there is room to support nanosecond resolution timestamps. High 32-bit fields for the atime, mtime and ctime timestamps, and also a new crtime timestamp documenting file creation time, will be added to the ext4 inode (Figure 4). 30 bits are sufficient to represent the nanosecond field, and the remaining 2 bits are used to extend the epoch by 272 years.

The NFSv4 clients need the ability to detect updates to a file made at the server end, in order to keep the client side cache up to date. Even with nanosecond support for ctime, the timestamp is not necessarily updated at the nanosecond level. The ext4 inode versioning feature addresses this issue by providing a global 64-bit counter in each inode. This counter is incremented whenever the file is changed. By comparing values of the counter, one can see whether the file has been updated. The counter is reset on file creation, and overflows are unimportant, because only equality is being tested. The i_version field already present in the 128-bit inode is used for the low 32 bits, and a high 32-bit field is added to the large ext4 inode.

6 Migration tool

Ext3 developers worked to maintain backwards compatibility between ext2 and ext3, a characteristic users appreciate and depend on. While ext4 attempts to retain compatibility with ext3 as much as possible, some of the incompatible on-disk layout changes are unavoidable. Even with these changes, users can still easily upgrade their ext3 filesystem to ext4, like it is possible from ext2 to ex3. There are methods available for users to try new ext4 features immediately, or migrate their entire filesystem to ext4 without requiring back-up and restore.

6.1 Upgrading from ext3 to ext4

There is a simple upgrade solution for ext3 users to start using extents and some ext4 features without requiring a full backup or migration. By mounting an existing ext3 filesystem as ext4 (with extents enabled), any new files are created using extents, while old files are still indirect block mapped and interpreted as such. A flag in the inode differentiates between the two formats, allowing both to coexist in one ext4 filesystem. All new ext4 features based on extents, such as preallocation and multiple block allocation, are available to the new extents files immediately.

A tool will also be available to perform a system-wide filesystem migration from ext3 to ext4. This migration tool performs two functions: migrating from indirect to extents mapping, and enlarging the inode to 256 bytes.

- Extents migration: The first step can be performed online and uses the defragmentation tool. During the defragmentation process, files are changed to extents mapping. In this way, the files are being converted to extents and defragmented at the same time.
- Inode migration: Enlarging the inode structure size must be done offline. In this case, data is backed up, and the entire filesystem is scanned and converted to extents mapping and large inodes.

For users who are not yet ready to move to ext4, but may want to in the future, it is possible to prepare their ext3 filesystem to avoid offline migration later. If an ext3 filesystem is formatted with a larger inode structure, 256 bytes or more, the fast extended attribute feature (Section 2.6) which is the default in ext4, can be used instantly. When the user later wants to upgrade to ext4, then other ext4 features using the larger inode size, such as nanosecond timestamps, can also be used without requiring any offline migration.

6.2 Downgrading from ext4 to ext3

Though not as straightforward as ext3 to ext4, there is a path for any user who may want to downgrade from ext4 back to ext3. In this case the user would remount the filesystem with the *noextents* mount option, copy all files to temporary files and rename those files over the original file. After all files have been converted back to indirect block mapping format, the INCOMPAT_EXTENTS flag must be cleared using tune2fs, and the filesystem can be re-mounted as ext3.

7 Performance evaluation

We have conducted a performance evaluation of ext4, as compared to ext3 and XFS, on three well-known filesystem benchmarks. Ext4 was tested with extents and delayed allocation enabled. The benchmarks in this analysis were chosen to show the impact of new changes in ext4. The three benchmarks chosen were: Flexible Filesystem Benchmark (FFSB) [1], Postmark [5], and IOzone [2]. FFSB, configured with a large file workload, was used to test the extents feature in ext4. Postmark was chosen to see performance of ext4 on small file workloads. Finally, we used IOzone to evaluate overall ext4 filesystem performance.

The tests were all run on the 2.6.21-rc4 kernel with delayed allocation patches. For ext3 and ext4 tests, the filesystem was mounted in writeback mode, and appropriate extents and delayed allocation mount options were set for ext4. Default mount options were used for XFS testing.

FFSB and IOzone benchmarks were run on the same 4-CPU 2.8 Ghz Intel(R) Xeon(tm) System with 2 GB of RAM, on a 68GB ultra320 SCSI disk (10000 rpm). Postmark was run on a 4-CPU 700 MHz Pentium(R) III system with 4 GB of RAM on a 9 GB SCSI disk (7200 rpm). Full test results including raw data are available at the ext4 wiki page, `http://ext4.wiki.kernel.org`.

7.1 FFSB comparison

FFSB is a powerful filesystem benchmarking tool, that can be tuned to simulate very specific workloads. We have tested multithreaded creation of large files. The test

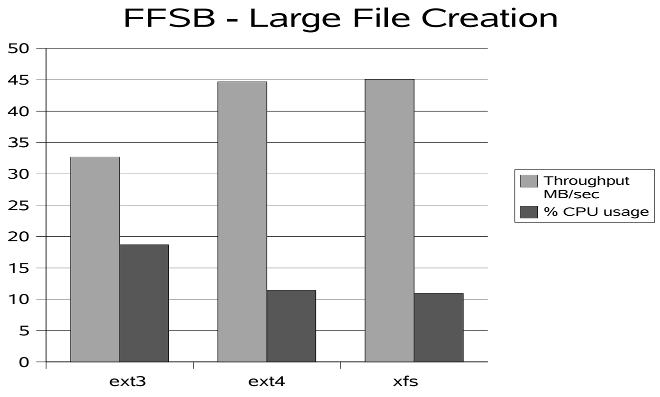

Figure 6: FFSB sequential write comparison

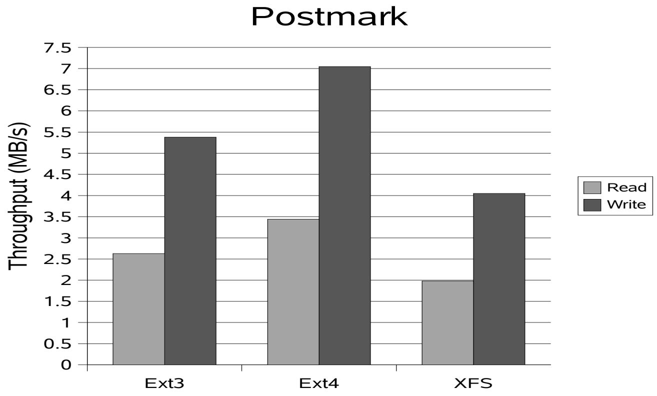

Figure 7: Postmark read write comparison

runs 4 threads, which combined create 24 1-GB files, and stress the sequential write operation.

The results, shown in Figure 6, indicate about 35% improvement in throughput and 40% decrease in CPU utilization in ext4 as compared to ext3. This performance improvement shows a diminishing gap between ext4 and XFS on sequential writes. As expected, the results verify extents and delayed allocation improve performance on large contiguous file creation.

7.2 Postmark comparison

Postmark is a well-known benchmark simulating a mail server performing many single-threaded transactions on small to medium files. The graph in Figure 7 shows about 30% throughput gain with with ext4. Similar percent improvements in CPU utilization are seen, because metadata is much more compact with extents. The write throughput is higher than read throughput because everything is being written to memory.

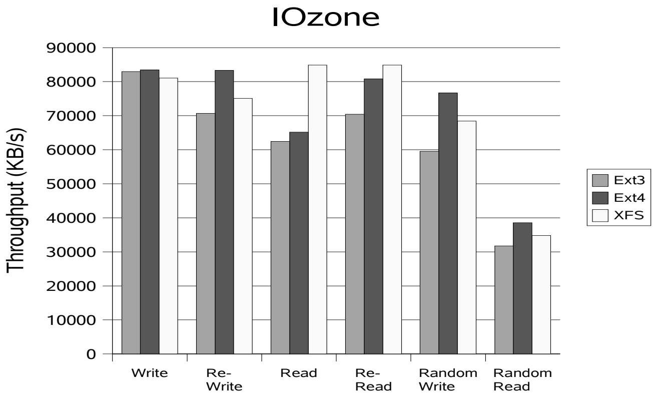

Figure 8: IOzone results: throughput of transactions on 512 MB files

These results show that, aside from the obvious performance gain on large contiguous files, ext4 is also a good choice on smaller file workloads.

7.3 IOzone comparison

For the IOzone benchmark testing, the system was booted with only 64 M of memory to really stress disk I/O. The tests were performed with 8 MB record sizes on various file sizes. Write, rewrite, read, reread, random write, and random read operations were tested. Figure 8 shows throughput results for 512 MB sized files. Overall, there is great improvement between ext3 and ext4, especially on rewrite, random-write and reread operations. In this test, XFS still has better read performance, while ext4 has shown higher throughput on write operations.

8 Conclusion

As we have discussed, the new ext4 filesystem brings many new features and enhancements to ext3, making it a good choice for a variety of workloads. A tremendous amount of work has gone into bringing ext4 to Linux, with a busy roadmap ahead to finalize ext4 for production use. What was once essentially a simple filesystem has become an enterprise-ready solution, with a good balance of scalability, reliability, performance and stability. Soon, the ext3 user community will have the option to upgrade their filesystem and take advantage of the newest generation of the ext family.

Acknowledgements

The authors would like to extend their thanks to Jean-Noël Cordenner and Valérie Clément, for their help on performance testing and analysis, and development and support of ext4.

We would also like to give special thanks to Andrew Morton for supporting ext4, and helping to bring ext4 to mainline Linux. We also owe thanks to all ext4 developers who work hard to make the filesystem better, especially: Ted T'so, Stephen Tweedie, Badari Pulavarty, Dave Kleikamp, Eric Sandeen, Amit Arora, Aneesh Veetil, and Takashi Sato.

Finally thank you to all ext3 users who have put their faith in the filesystem, and inspire us to strive to make ext4 better.

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Lustre is a trademark of Cluster File Systems, Inc.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document is provided "AS IS," with no express or implied warranties. Use the information in this document at your own risk.

References

[1] Ffsb project on sourceforge. Technical report. http://sourceforge.net/projects/ffsb.

[2] Iozone. Technical report. http://www.iozone.org.

[3] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Ottawa Linux Symposium*, 2005.

[4] Jonathan Corbet. Which filesystem for samba4? Technical report. `http://lwn.net/Articles/112566/`.

[5] Jeffrey Katcher. Postmark a new filesystem benchmark. Technical report, Network Appliances, 2002.

[6] Daniel Phillips. A directory index for ext2. In *5th Annual Linux Showcase and Conference*, pages 173–182, 2001.

[7] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci Dusseau. Iron file systems. In *SOSP'05*, pages 206–220, 2005.

[8] Stephen Tweedie. Ext3 journalling filesystem. In *Ottawa Linux Symposium*, 2000.

[9] Stephen Tweedie and Theodore Y Ts'o. Planned extensions to the linux ext2/3 filesystem. In *USENIX Annual Technical Conference*, pages 235–244, 2002.