Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Ext4 block and inode allocator improvements

Aneesh Kumar K.V
IBM Linux Technology Center
aneesh.kumar@in.ibm.com

Mingming Cao
IBM Linux Technology Center
cmm@us.ibm.com

Jose R Santos
IBM Linux Technology Center
jrs@us.ibm.com

Andreas Dilger
Sun Microsystems, Inc
adilger@sun.com

Abstract

File systems have conflicting needs with respect to block allocation for small and large files. Small related files should be nearby on disk to maximize disk track cache and avoid seeking. To avoid fragmentation, files that grow should reserve space. The new multiple block and delayed allocators for Ext4 try to satisfy these requirements by deferring allocation until flush time, packing small files contiguously, and allocating large files on RAID device-aligned boundaries with reserved space for growth. In this paper, we discuss the new multiple block allocator for Ext4 and compare the same with the reservation-based allocation. We also discuss the performance advantage of placing the meta-data blocks close together on disk so that meta-data intensive workloads seek less.

1 Introduction

The Ext4 file system was forked from Ext3 file system about two years ago to address the capability and scalability bottleneck of the Ext3 file system. Ext3 file system size is hard limited to 16 TB on x86 architecture, as a consequence of 32-bit block numbers. This limit has already been reached in the enterprise world. With the disk capacity doubling every year and with increasing needs for larger file systems to store personal digital media, desktop users will very soon want to remove the limit for Ext3 file system. Thus, the first change made in the Ext4 file system is to lift the maximum file system size from 2^{32} blocks(16 TB with 4 KB blocksize) to 2^{48} blocks.

Extent mapping is also used in Ext4 to represent new files, rather than the double, triple indirect block mapping used in Ext2/3. Extents are being used in many modern file systems[1], and it is well-known as a way to efficiently represent a large contiguous file by reducing the meta-data needed to address large files. Extents help improve the performance of sequential file read/writes since extents are a significantly smaller amount of metadata to be written to describe contiguous blocks, thus reducing the file system overhead. It also greatly reduces the time to truncate a file as the meta-data updates are reduced. The extent format supported by Ext4 is shown in Fig 1 and Fig 2. A full description of Ext4 features can be found in [2].

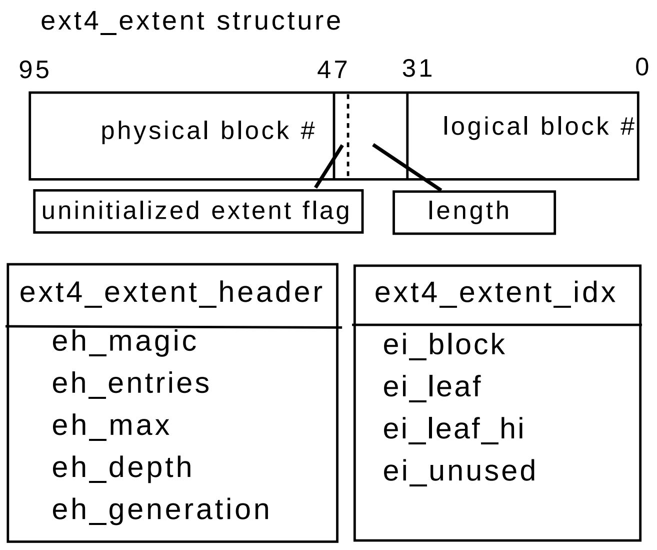

Figure 1: Ext4 extents, header and index structures

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 file system, are not efficiently represented

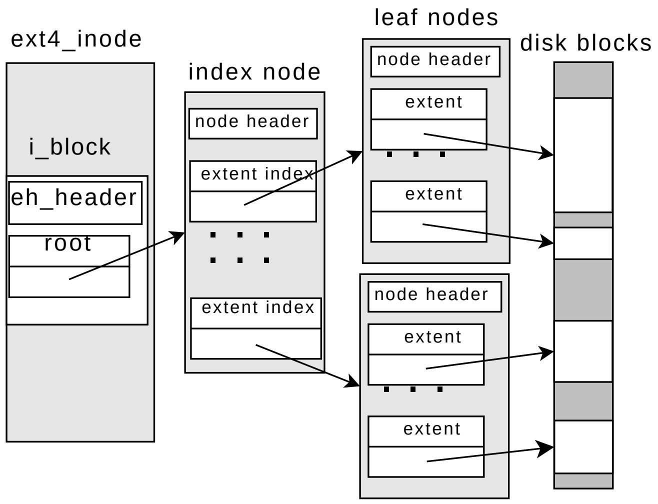

Figure 2: Ext4 extent tree layout

using extent maps. In Ext4, it is more important to have an advanced block allocator to reduce file fragmentation. A well-designed block allocator should pack small files close to each other for locality and also place large files contiguously on disk to improve I/O performance.

The block allocator in the Ext3 file system does limited work to reduce file fragmentation and maintains group locality for small files under the same directory. The Ext3 allocator tries to allocate multiple blocks on a best effort basis but does not do a smart search to find a better location to place a large file. It is also unaware of the relationship between different files, thus it is quite possible to place small related files far apart, causing extra seeks when loading a large number of related files. Ext4 block allocation tries to address these two problems with the new multiple block allocator while still making it possible to use the old block allocator. Multiple block allocation can be enabled or disabled by mount option `-o mballoc` and `-o nomballoc`, respectively. The current development version of Ext4 file system enables the multiple block allocator by default.

While a lot of work has gone into the block allocator, one must not forget that the inode allocator is what determines where blocks start getting allocated to begin with. While disk platter density has increased dramatically over the years, the old Ext3 inode allocator does little to ensure data and meta-data locality in order to take advantage of that density and avoid large seeks. Ext4 has begun exploring the possibilities of compacting data to increase locality and reduce seeks, which ultimately leads to better performance. Removal of restrictions in Ext4's block groups has made it possible to rethink meta-data placement and inode allocation in ways not possible in Ext3.

This paper is organized into the following sections. In Section 2.1 and 2.2, we give some background on Ext3 block allocation principles and current limitations. In Section 2.3, we give a detailed description of the new multiple block allocator and how it addresses the limitations facing the Ext3 file system. Section 2.4 compares the Ext3 and Ext4 block allocators with some performance data we collected.

Finally, we discuss the inode allocator changes made in Ext4. We start with Section 4.1, describing the Ext3 inode allocation policy, the impact that the inode allocation has on overall file system performance, and how changing the concept of a block group can lead to performance improvements. In Section 4.3, we explore a new inode allocator that uses this new concept in meta-data allocation to manipulate block allocation and data locality on disk. Later, we examine the performance improvements in the Ext4 file system due to the changing of block group concept and the new inode allocator. Finally, we will discuss some potential future work in Section 4.5.

2 Ext4 Multiple Blocks Allocator

In this section we give some background on the Ext2/3 block allocation before we discuss the new Ext4 block allocator. We refer to the old block allocator as the Ext3 block allocator and the new multiple block allocator as the Ext4 block allocator for simplicity.

2.1 Ext3 block allocator

Block allocation is the heart of a file system design. Overall, the goal of file system block allocation is to reduce disk seek time by reducing file system fragmentation and maintaining locality for related files. Also, it needs to scale well on large file systems and parallel allocation scenarios. Here is a short summary of the strategy used in the Ext3 block allocator:

To scale well, the Ext3 file system is partitioned into 128 MB block group chunks. Each block group maintains a single block bitmap to describe data block availability inside this block group. This way allocation on different block groups can be done in parallel.

When allocating a block for a file, the Ext3 block allocator always starts from the block group where the inode structure is stored to keep the meta-data and data blocks close to each other. When there are no free blocks available in the target block group it will search for a free block from the rest of the block groups. Ext3 always tries to keep the files under the same directory close to each other until the parent block group is filled.

To reduce large file fragmentation Ext3 uses a goal block to hint where it should allocate the next block from. If the application is doing a sequential I/O the target is to get the block following the last allocated block. When the target block is not available it will search further to find a free extent of at least 8 blocks and starts allocation from there. This way the resulting allocations will be contiguous.

In case of multiple files allocating blocks concurrently, the Ext3 block allocator uses block reservation to make sure that subsequent requests for blocks for a particular file get served before it is interleaved with other files. Reserving blocks which can be used to satisfy the subsequent requests enable the block allocator to place blocks corresponding to a file nearby. There is a per-file reservation window, indicating the range of disk blocks reserved for this file. However, the per-file reservation window is done purely in memory. Each block allocation will first check the file's own reservation window before starts to find unreserved free block on bitmap. A per-file system red-black tree is used to maintain all the reservation windows and to ensure that when allocating blocks using bitmap, we don't allocate blocks out of another file's reservation window.

2.2 Problems with Ext3 block allocator

Although block reservation makes it possible to allocate multiple blocks at a time in Ext3, this is very limited and based on best effort basis. Ext3 still uses the bitmap to search for the free blocks to reserve. The lack of free extent information across the whole file system results in poor allocation pattern for multiple blocks since the allocator searches for free blocks only inside the reservation window.

Another disadvantage of the Ext3 block allocator is that it doesn't differentiate allocation for small and large files. Large directories, such as /etc, contain large numbers of small configuration files that need to be

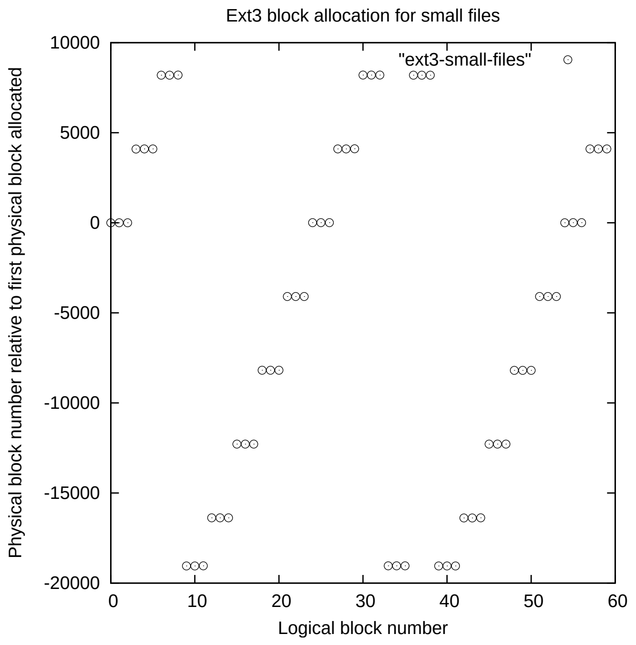

Figure 3: Ext3 block allocator for small files

read during boot. If the files are placed far apart on the disk the bootup process would be delayed by expensive seeks across the underlying device to load all the files. If the block allocator could place these related small files closer it would be a great benefit to the read performance.

We used two test cases to illustrate the performance characteristic of Ext3 block allocator for small and large files, as shown in Fig 3 and Fig 4. In the first test we used one thread to sequentially create 20 small files of 12 KB. In the second test, we create a single large file and multiple small files in parallel. The large file is created by a single thread, while the small files are created by another thread in parallel. The graph is plotted with the logical block number on the x-axis and the physical block number on the y-axis. To better illustrate the locality, the physical block number plotted is calculated by subtracting the first allocated block number from the actual allocated block number. With regard to small files, the 4^{th} logical block number is the first logical block number of the second file. This helps to better illustrate how closely the small files are placed on disk.

Since Ext3 simply uses a goal block to determine where to place the new files, small files are kept apart by Ext3 allocator intentionally to avoid too much fragmentation in case the files are large files. This is caused by lack

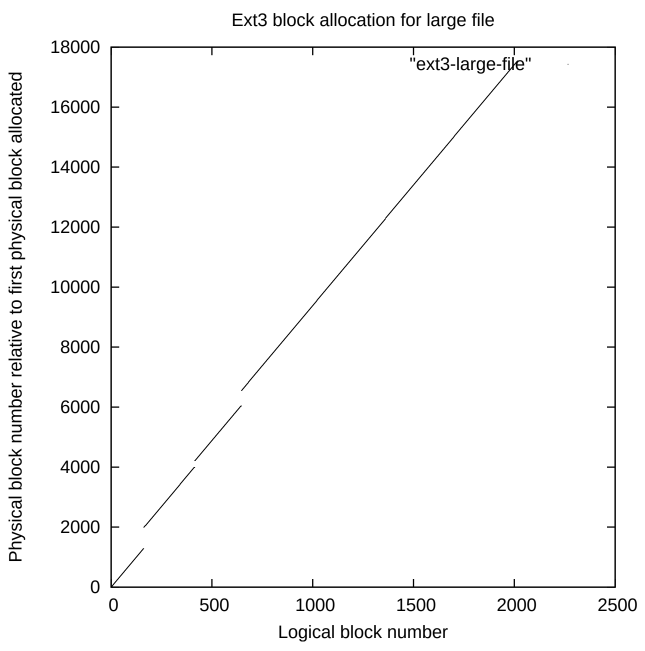

Figure 4: Ext3 block allocator for large file

of information that those small files are generated by the same process and therefore should be kept close to each other. As we see in Fig [3], the locality of those small files are bad, though the files themselves are not fragmented.

Fig [4] illustrates the fragmentation of a large file in Ext3. Because Ext3 lacks knowledge that the large file is unrelated to the small files, the allocations for the large file and the small files are fighting for free spaces close to each other, even though the block reservation helped reduce the fragmentation to some extent already. A better solution is to keep the large file allocation far apart from unrelated allocation at the very beginning to avoid interleaved fragmentation.

2.3 Ext4 Multiple block allocator

The Ext4 multiple block allocator tries to address the Ext3 block allocator limitation discussed above. The main goal is to provide better allocation for small and large files. This is achieved by using a different strategy for different allocation requests. For a relatively small allocation request, Ext4 tries to allocate from a per-CPU locality group, which is shared by all allocations under the same CPU, in order to try to keep these small files close to each other. A large allocation request is allocated from per-file preallocation first. Like Ext3 reservation, Ext4 maintains an in-memory preallocation range for each file, and uses that to solve the fragmentation issues caused by concurrent allocation.

The Ext4 multiple block allocator maintains two preallocated spaces from which block requests are satisfied: A per-inode preallocation space and a per-CPU locality group prealloction space. The per-inode preallocation space is used for larger request and helps in making sure larger files are less interleaved if blocks are allocated at the same time. The per-CPU locality group preallocation space is used for smaller file allocation and helps in making sure small files are placed closer on disk. Which preallocation space to use depends on the total size derived out of current file size and allocation request size. The allocator provides a tunable `/prof/fs/ext4/<partition>/stream_req` that defaults to 16. If the total size is less than `stream_req` blocks, we use per-CPU locality group preallocation space.

While allocating blocks from the inode preallocation space, we also make sure we pick the blocks in such a way that random writes result in less fragmentation. This is achieved by storing the logical block number as a part of preallocation space and using the value in determining the physical block that needs to be allocated for a subsequent request.

If we can't allocate blocks from the preallocation space, we then look at the per-block-group buddy cache. The buddy cache consists of multiple free extent maps and a group bitmap. The extent map is built by scanning all the free blocks in a group on the first allocation. While scanning for free blocks in a block group we consider the blocks in the preallocation space as allocated and don't count them as free. This is needed to make sure that when allocating blocks from the extent map, we don't allocate blocks from a preallocation space. Doing so can result in file fragmentation. The free extent map obtained by scanning the block group is stored in a format, as shown in Fig 5, called a buddy bitmap. We also store the block group bitmap along with the extent map. This bitmap differs from the on-disk block group bitmap in that it considers blocks in preallocation space as allocated. The free extent information and the bitmap is then stored in the page cache of an in-core inode, and is indexed with the group number. The page containing the free extent information and bitmap is calculated as shown in Fig 6.

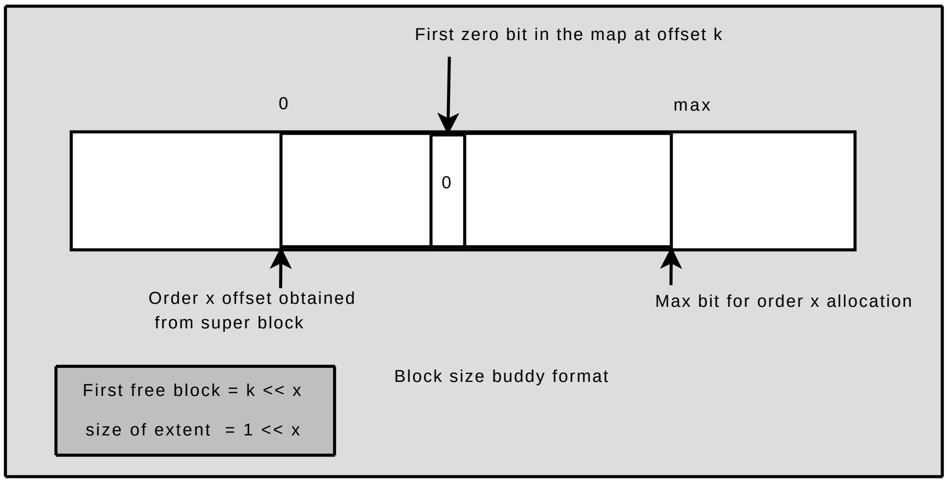
First zero bit in the map at offset k
0
max
0
Order x offset obtained from super block
Max bit for order x allocation
First free block = k << x
size of extent = 1 << x
Block size buddy format

Figure 5: Ext4 buddy cache layout

```
/*
 * the buddy cache inode stores the block bitmap
 * and buddy information in consecutive blocks.
 * So for each group we need two blocks.
 */
block = group * 2;
pnum = block / blocks_per_page;
poff = block % blocks_per_page;

page = find_get_page(inode->i_mapping, pnum);
.......
if (!PageUptodate(page)) {
      ext4_mb_init_cache(page, NULL);
......
```

Figure 6: Buddy cache inode offset mapping

Before allocating blocks from the buddy cache, we normalize the request. This helps prevent heavy fragmentation of the free extent map, which groups free blocks in power of 2 size. The extra blocks allocated out of the buddy cache are later added to the preallocation space so that the subsequent block requests are served from the preallocation space. In the case of large files, the normalization follows a list of heuristics based on file size. For smaller files, we normalize the block request to stripe size if specified at mount time or to `s_mb_group_prealloc`. `s_mb_group_prealloc` defaults to 512 and can be configured via `/proc/fs/ext4/<partition/group_prealloc`

Searching for blocks in buddy cache involves:

- Search for requested number of blocks in the extent map.
- If not found, and if the request length is same as stripe size, search for free blocks in stripe size aligned chunks. Searching for stripe aligned chunks results in better allocation on RAID setup.
- If not found, search for free blocks in the bitmap and use the best extent found.

Each of these searches starts with the block group in which the goal block is found. Not all block groups are used for the buddy cache search. If we are looking for blocks in the extent map, we only look at block groups that have the requested order of blocks free. When searching for stripe-size-aligned free blocks we only look at block groups that have stripe size chunks of free blocks. When searching for free blocks in the bitmap, we look at block groups that have the requested number of free blocks.

2.4 Ext4 multiple block allocator performance advantage

The performance advantage of the multiple block allocator related to small files is shown in Fig 7. The blocks are closer because they are satisfied from the same locality group preallocation space.

The performance advantage of multiple block allocator related to large files is shown in Fig 8. The blocks are closer because they are satisfied from the inode-specific preallocation space.

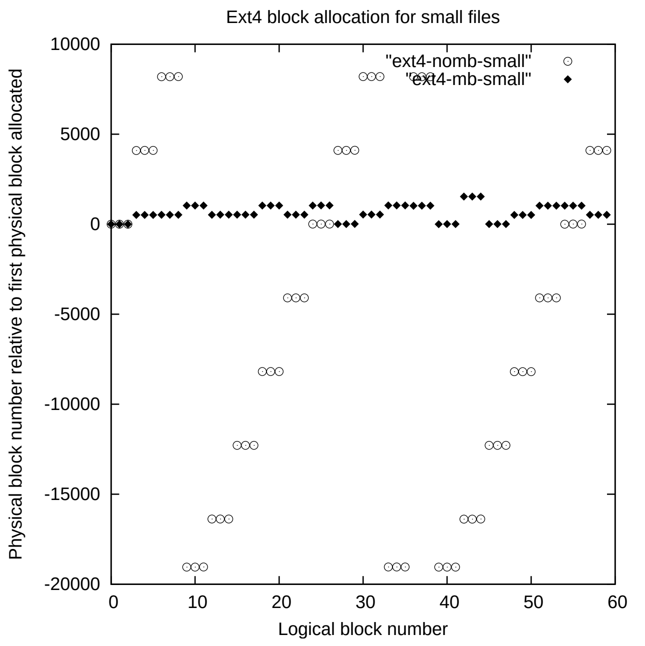

Figure 7: Multiple block allocator small file performance

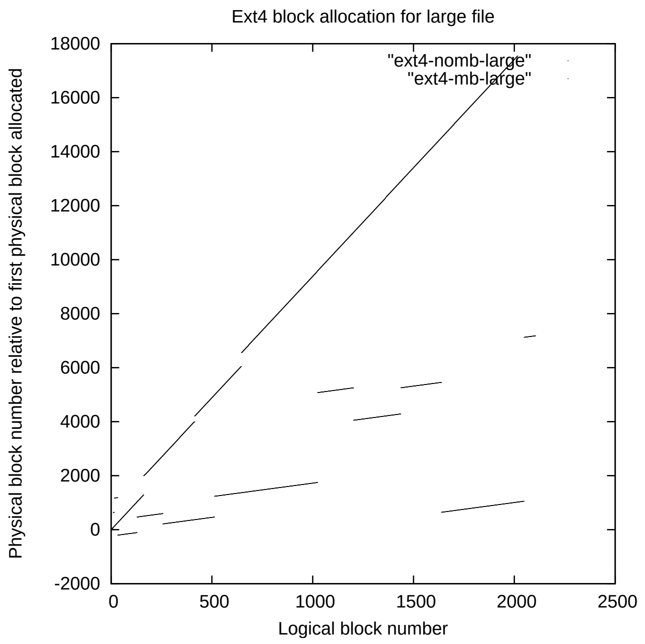

Figure 8: Multiple block allocator large file performance

Table 1 shows the compilebench[5] number comparing Ext4 and Ext3 allocators with the `data=ordered` mount option. Compilebench tries to age a file system by simulating some of the disk IO common in creating, compiling, patching, stating and reading kernel trees. It indirectly measures how well file systems can maintain directory locality as the disk fills up and directories age

Test	Ext4 allocator	Ext3 allocator
intial create	20.44 MB/s	19.64 MB/s
create total	11.81 MB/s	8.12 MB/s
patch total	4.95 MB/s	3.66 MB/s
compile total	16.66 MB/s	12.57 MB/s
clean total	247.54 MB/s	82.57 MB/s
read tree total	7.06 MB/s	6.99 MB/s
read compiled tree total	9.06 MB/s	10.40 MB/s
delete tree total	7.06 seconds	16.66 seconds
delete compiled tree total	9.64 seconds	22.21 seconds
stat tree total	5.31 seconds	13.39 seconds
stat compiled tree total	5.63 seconds	14.70 seconds

Table 1: Compliebench numbers for Ext4 and Ext3 allocator

2.5 Evolution of an Allocator

The mballoc allocator included in Ext4 is actually the third generation of this allocation engine written by Alex Tomas (Zhuravlev). The first two versions of the allocator were focused mainly on large allocations (1 MB at a time), while the third generation also works to improve small files allocation.

Even at low I/O rates, the single-block allocation used by Ext3 does not necessarily make a good decision for inter-block allocations. The Linux VFS layer splits any large I/O submitted to the kernel into page-sized chunks and forces single-block allocations by the file system without providing any information about the rest of the outstanding I/O on that file. The block found for the first allocation is usually the first free block in the block group, and no effort is made to find a range of free blocks suitable for the amount of data being written to the file. This leads to poor allocation decisions, such as selecting free block ranges that are not large enough for even a single `write()` call.

At very high I/O rates (over 1 GB/s) the single-block allocation engine also becomes a CPU bottleneck because every block allocation traverses the file system allocator, scans for a free block, and locks to update data structures. With mballoc, one test showed an improvement from 800 MB/s to 1500 MB/s on the same system by reducing the CPU cost per allocated block.

What is critical to the success of mballoc is the ability to make smart allocation decisions based on as many blocks of file data as possible. This necessitated the development of delayed allocation for normal I/O. When mballoc has a good idea that a file is small or large, and how many data blocks to allocate, it can make good decisions.

The first version of mballoc used the buddy allocator only to allocate contiguous chunks of blocks, and would align the allocation to the start of a free extent. While this still leads to performance gains due to avoided seeks and aggregation of multiple allocations, it is not optimal in the face of RAID devices that have lower overhead when the I/O is properly aligned on RAID disk and stripe-wide boundaries.

The second version of mballoc improved the buddy allocator to align large allocations to RAID device boundaries. This is easily done directly from the buddy bitmap for RAID geometries that use power-of-two numbers of data disks in each stripe by simply stopping search for free bits in the buddy bitmap at the RAID boundary size.

Avoiding the read-modify-write cycle for RAID systems can more than double performance because the costly synchronous read is avoided. In RAID devices that have a read cache that is aligned to the stripe boundaries, doing a misaligned read will double the amount of data read from disk and fill two cachelines.

The Ext4 superblock now has fields that store the RAID geometry at mke2fs time or with tune2fs. It is likely that the complex RAID geometry probing done at mkfs time for XFS will also be adopted by mke2fs in order to populate these fields automatically.

During testing for the current third version of mballoc the performance of small file I/O was investigated, and it was seen that the aggregate performance can be improved dramatically only when there are no seeks between the blocks allocated to different files. As a result the group allocation mechanism is used to pack small allocations together without any free space in between. In the past there was always space reserved or left at the end of each file "just in case" there were more blocks to allocate. For small files, this forces a seek after each read or write. With delayed allocation, the size of a small file is known at allocation time and there is no longer a need for a gap after each file.

The current mballoc allocator can align allocations to a RAID geometry that is not power-of-two aligned, though it is slightly more expensive.

More work still remains to be done to optimize the allocator. In particular, mballoc keeps the "scan groups for good allocations" behaviour of the Ext2/3 block allocator. Implementing an in-memory list for more optimal free-extent searching, as XFS does, would further reduce the cost of searching for free extents.

Also, there is some cost for initializing the buddy bitmaps. Doing this at mount time, as the first version of mballoc did, introduces an unacceptable mount delay for very large file systems. Doing it at first access adds latency and hurts performance for the first uses of the file system. Having a thread started at mount time to scan the groups and do buddy initialization asynchronously among other things, would help avoid both issues.

3 Delayed allocation

Because Ext4 uses extent mapping to efficiently represent large files, it is natural to process a multiple block allocation together, rather than one block allocation at a time as done in Ext3. Because block allocation requests for buffered I/O are passed through the VFS layer one at a time at the `write_begin` time, the underlying Ext3 file system cannot foresee and cluster future requests. Thus, delayed allocation is being proposed multiple times to enable multiple block allocation for buffered I/O.

Delayed allocation, in short, defers block allocations from `write()` operation time to page flush time. This method provides multiple benefits: it increases the opportunity to combine many block allocation requests into a single request reducing fragmentation and saving CPU cycles, and avoids unnecessary block allocation for short-lived files.

In general, with delayed allocation, instead of allocating the disk block in `write_begin`, the VFS just does a plain block look up. For those unmapped buffer heads, it calculates the required number of blocks to reserve (including data blocks and meta-data blocks), reserves them to make sure that there are enough free blocks in the file system to satisfy the write. After that is done, it marks the buffer heads as delayed allocated(`BH_DELAY`). No block allocation is done at that moment. Later, 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 attempts to perform cluster block allocation for those buffer heads marked as `BH_DELAY` all together, then submit the page or pages to the bio layer. After the block allocation is complete, the unused block reservation is returned back to the file system.

The current implementation of delayed allocation is mostly done in the VFS layer, hoping that multiple file systems, such as Ext2 and 3, can benefit from the feature. There are new address space operations added for Ext4 for delayed allocation mode, providing call back functions for `write_begin()`, `write_end()` and `writepage()` for delayed allocation, with updated block allocation/reservation/lookup functions. The current Ext4 delayed allocation only supports `data=writeback` journalling mode. In the future, there are plans to add delayed support for `data=ordered` journalling mode.

4 FLEX_BG and the Inode allocator

4.1 The old inode allocator

The inode allocator in an Ext2/3 file system uses the block groups as the vehicle to determine where new inodes are placed on the storage media. The Ext2/3/4 file system is divided into small groups of blocks with the block group size determined by the amount of blocks that a single bitmap can handle. In a 4 KB block file system, a single block bitmap can handle 32768 blocks for a total of 128 MB per block group. This means that for every 128 MB, there will be meta-data blocks (block/inode bitmaps and inode table blocks) interrupting the contiguous flow of blocks that can be used to allocate data.

The Orlov directory inode allocator [6] tries to maximize the chances of getting large block allocations by reducing the chances of getting an inode allocated in a block group with a low free block count. The Orlov allocator also tries to maintain locality of related data (i.e. files in the same directory) as much as possible. The Orlov allocator does this by looking at the ratio of free blocks, free inodes, and number of directories in a block group to find the best suitable placement of a directory inode. Inode allocations that are not directories are handled by a second allocator that starts its free inode search from the block group where the parent directory is located and attempts to place the inodes with the same parent inode in the same block group. While this approach works very well given the block group design of Ext2/3/4, it does have some limitations:

- A 1 TB file system has around 8192 block groups. Searching through that many block groups on a heavily used file system can become expensive.
- Orlov can place a directory inode in a random location that is physically far away from its parent directory.
- Orlov's calculations to find a single 128 MB block group are expensive in this multi-terabyte world we live in.
- Given that hard disk seek times are not expected to improve much during the next couple of years, while at the same time seeing big increases in capacity and interface throughput, the Orlov allocator does little to improve data locality.

Of course the real problem is not Orlov itself, but the restrictions imposed on it by the size of a block group. One solution around this limitation is to implement multi-block bitmaps. The drawback with this implementation is that things like handling bad blocks inside one of those bitmap or inode tables becomes very complicated when searching for free blocks to replace the bad ones. A simpler solution is to get rid of the notion that meta-data need to be located within the file system block group.

4.2 FLEX_BG

Simply put, the new `FLEX_BG` feature removes the restriction that the bitmaps and inode tables of a particular block group MUST be located within the block range of that block group. We can remove this restriction thanks to e2fsprogs being a robust tool at finding errors through fsck. While activating the `FLEX_BG` feature flag itself doesn't change anything in the behavior of Ext4, the feature does allow mke2fs to allocate bitmaps and inode tables in ways not possible before. By tightly allocating bitmaps and inode tables close together, one could essentially build a large virtual block group that gets around some of the size limitations of regular block groups.

The new extent feature benefits from the new meta-data allocation by moving meta-data blocks that would otherwise prevent the availability of contiguous free blocks on the storage media. By moving those blocks to the beginning of a large virtual block group, the chances of allocating larger extents are improved. Also, having the meta-data for many block groups contiguous on disk avoids seeking for meta-data intensive workloads including e2fsck.

4.3 FLEX_BG inode allocator

Given that we can now have a larger collection of blocks that can be called a block group, making the kernel take advantage of this capability was the obvious next step. Because the Orlov directory inode allocator owed some design decisions to the size limitations of traditional block groups, a new inode allocator would need to employ different techniques in order to take better advantage of the on disk meta-data allocation. Some of the ways the new inode allocator differs from the old allocator are:

- The allocator always tries to fill a virtual block group to a certain free block ratio before attempting to allocate on another group. This is done to improve data locality on the disc by avoiding seeks as much as possible.

- Directories are treated the same as regular inodes. Given that these virtual block groups can now handle multiple gigabytes worth of free blocks, spreading directories across block groups not only does not provide the same incentive, it could actually hurt performance by allowing larger seeks on relatively small amounts of data.

- The allocator may search backwards for suitable block groups. If the current group does not have enough free blocks, it may try the previous group first just in case space was freed up.

- The allocator reserves space for file appends in the group. If all the blocks in a group are used, appending blocks to an inode in that group could mean that the allocation could happen at a large offset within the storage media increasing seek times. This block reservation is not used unless the last group has been used past its reserved ratio.

The size of a virtual group is always a power-of-two multiple of a *normal* block group in size, and it is specified at mke2fs time. The size is stored in the super block in to control how to build the in-memory free inode and block structure that the algorithm uses to determine the utilization of the virtual block group. Because we look at the Ext4 block group descriptors only when a suitable virtual group is found, this algorithm requires fewer endian conversions than the traditional Orlov allocator on big endian machines.

One big focus of this new inode allocator is to maintain data and meta-data locality to reduce seek time when compared to the old allocator. Another very important area of focus is reduction in allocation overhead. By not handling directory inodes differently, we remove a lot of the complexity from the old allocator while the smaller number of virtual block groups makes searching for adequate block groups easier.

Now that a group of inode tables is treated as if it were a single large inode table, the new allocator also benefits from another new feature found in Ext4. Uninitialized block groups mark inode tables as uninitialized when they are not in use and thus skips reading those inode tables at fsck time providing significant fsck speed improvements. Because the allocator only uses an inode table when the previous table on the same virtual group is full, fewer inode tables get initialized resulting in fewer inode tables that need to be loaded and improved fsck times.

4.4 Performance results

We used the FFSB[4] benchmark with a profile that executes a combination of small file reads, writes, creates, appends, and deletes. This helps simulate a meta-data heavy workload. A Fibre Channel disk array was used as a storage media for the FFSB test with 1 GB of fast write cache. The benchmark was run with a `FLEX_BG` virtual block group of 64 packed groups and it is compared to a regular EXT4 file system mounted with both mballoc and delalloc. Note that 64 packed groups is not the optimum number for every hardware configuration, but this number provided very good overall performance for workloads on the hardware available. Further study is needed to determine the right size for a particular hardware configuration or workload.

Op	**Ext4**	**Ext4(flex_bg)**
read	67937 ops	73056 ops
write	98488 ops	104904 ops
create	1086604 ops	1228395 ops
append	31903 ops	33225 ops
delete	59503 ops	70075 ops
Total ops/s	**4477.37 ops/s**	**5026.78 ops/s**

Table 2: FFSB small meta-data FibreChannel(1 thread)-FLEX_BG with 64 block groups

In Table 2, the single threaded results show a 10% overall improvement in operations-per-second throughput. The 16 threaded results in Table 3 show an even better improvements of 18% over the regular EXT4 allocation. The results also show that there is 5.6% better scalability from 1 to 16 threads when using FLEX_BG grouping compared to the normal allocation.

Op	**Ext4**	**Ext4(flex_bg)**
read	96778 ops	119135 ops
write	143744 ops	174409 ops
create	1584997ops	1937469 ops
append	46735 ops	56409 ops
delete	93333 ops	113598 ops
Total ops/s	**6514.51 ops/s**	**7968.24 ops/s**

Table 3: FFSB small meta-data (16 threads)- FLEX_BG with 64 block groups

To see the overall effect of the new allocator on more complex directory layouts, we used Compilebench[5], which generates a specified number of Linux kernel tree like directories with files that represent file sizes in the actual kernel tree. In table 4, we see the overall results for the benchmark are better when using grouping and the new inode allocator. One exception is both the "read tree" and the "read compiled tree" which show slightly slower results. It shows that there is still some room for improvement in the meta-data allocation and the inode allocator.

Test	**Ext4**	**Ext4(flex_bg)**
intial create	73.38 MB/s	81.09 MB/s
create total	69.90 MB/s	73.32 MB/s
patch total	21.73 MB/s	21.85 MB/s
compile total	125.52 MB/s	127.26 MB/s
clean total	921.01 MB/s	973.91 MB/s
read tree total	18.73 MB/s	18.43 MB/s
read compiled tree total	35.59 MB/s	33.91 MB/s
delete tree total	4.18 seconds	3.70 seconds
delete compiled tree total	4.11 seconds	3.90 seconds
stat tree total	2.68 seconds	2.59 seconds
stat compiled tree total	3.20 seconds	2.86 seconds

Table 4: Compliebench FiberChannel - FLEX_BG with 64 block groups

4.5 Future work

The concept of grouping meta-data in EXT4 is still in its infancy and there is still a lot of room for improvement. We expect to gain performance out of the EXT4 file system by looking at the layout of the meta-data. Because the definition of the FLEX_BG feature removes meta-data placement restrictions, this allows the implementation of virtual groups to be fluid without sacrificing backward compatibility.

Other optimizations that are worth exploring are placement of meta-data in the center of the virtual group instead of the beginning to reduce the worst case seek scenario. Because the current implementation just focuses on inodes as a way to manipulate the block allocator, future implementations could go further still and make the various block allocators FLEX_BG grouping aware. Because the right size of groups depends on things related to the disk itself, making mke2fs smarter to automatically set the virtual group size base on media size, media type and RAID configuration will help users deploy this feature.

Shorter-term fixes for removing file system size limitations are in the works. The current code stores the sum of free blocks and inodes of all the groups that build a virtual block group in an in-memory data structure. This means that if the data structure exceeds page size, the allocation would fail, in which case we revert back to the old allocator on very large file systems. While this is a performance feature, storing all this information in memory may be overkill for very large file systems. Building this on top of an LRU scheme removes this limitation and saves kernel memory.

5 Conclusion

The main motivation for forking the Ext4 file system from the Ext3 file system was to break the relatively small file system size limit. This satisfies the requirements for larger files, large I/O, and a large number of files. To maintain the Ext4 file system as a general purpose file system on a desktop, it is also important to make sure the Ext4 file system performs better on small files.

We have discussed the work related to block allocation and inode allocation in the Ext4 file system and how to satisfy the conflicting requirements of making Ext4 a high performance general purpose file system for both the desktop and the server. The combination of preallocation, delayed allocation, group preallocation, and multiple block allocation greatly help reduce fragmentation issues occurring on large file allocation and poor locality issues on small files that have been seen in Ext3 file system. By tightly allocating bitmap and inode tables close together with `FLEX_BG`, one could essentially build a large virtual block group that increases the likelyhood of allocating large chunks of extents and allows Ext4 file system to handles better on meta-data-intensive workload. Future work, including support for delayed allocation for ordered mode journalling and online defragmentation, will help to future reduce file fragmentation issues.

Acknowledgements

We would like to thank many Ext3/4 developers for their contribution to Ext4 file system, especially grateful to Alex Thomas, Theodore Tso, Dave Kleikamp, Eric Sandeen, Jan Kara, Josef Bacik, Amit Arora, Avantika Mathur, and Takashi Sato.

We owe thanks to Jean-Noël Cordenner and Valérie Clément, for their help on performance testing and analysis, and development and support of the Ext4 file system.

We owe thanks to Eric Sandeen for his careful review on an earlier draft of this manuscript.

Legal Statement

References

[1] BEST, S. JFS overview `http://jfs.sourceforge.net/project/pub/jfs.pdf`.

[2] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A AND VIVER, L. The New ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium* (2007). `http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf`

[3] BRYANT, R., FORESTER, R., HAWKES, J. Filesystem Performance and Scalability in Linux 2.4.17 . In *USENIX Annual Technical Conference, Freenix Track* (2002). `http://www.usenix.org/event/usenix02/tech/freenix/full_papers/bryant/bryant_html/`

[4] Ffsb project on sourceforge. Tech. rep. `http://sourceforge.net/projects/ffsb`.

[5] Compilebench Tech. rep. `http://oss.oracle.com/~mason/compilebench`.

[6] COBERT, J. The Orlov block allocator. `http://lwn.net/Articles/14633/`.