Taking Linux Filesystems to the Space Age: Space Maps in Ext4

Saurabh Kadekodi
Spring Computing Pvt. Ltd.
saurabhkadekodi@gmail.com

Shweta Jain
Clogeny Technologies Pvt. Ltd.
atewhs.jain@gmail.com

Abstract

With the ever increasing filesystem sizes, there is a constant need for faster filesystem access. A vital requirement to achieve this is efficient filesystem metadata management.

The bitmap technique currently used to manage free space in Ext4 is faced by scalability challenges owing to this exponential increase. This has led us to re-examine the available choices and explore a radically different design of managing free space called Space Maps.

This paper describes the design and implementation of space maps in Ext4. The paper also highlights the limitations of bitmaps and does a comparative study of how space maps fare against them. In space maps, free space is represented by extent based red-black trees and logs. The design of space maps makes the free space information of the filesystem extremely compact allowing it to be stored in main memory at all times. This significantly reduces the long, random seeks on the disk that were required for updating the metadata. Likewise, analogous on-disk structures and their interaction with the in-memory space maps ensure that filesystem integrity is maintained. Since seeks are the bottleneck as far as filesystem performance is concerned, their extensive reduction leads to faster filesystem operations. Apart from the allocation/deallocation improvements, the log based design of Space Maps helps reduce fragmentation at the filesystem level itself. Space Maps uplift the performance of the filesystem and keep the metadata management in tune with the highly scalable Ext4.

1 Introduction

Since linux kernel 2.6.28, Ext4 has been included in the mainstream and has become the default filesystem with most distributions. In a very short span of time, it has grown in popularity as well as stability. Over its predecessor Ext3, Ext4 brings many new features like scalability, delayed allocation, multiple-block allocation, improved timestamps[1] among others. One of its most important features is its use of extents.

1.1 Impact of extents

An extent is a combination of two integers, the first is the start block number and the second is the number of contiguous physical blocks ahead of the start block.

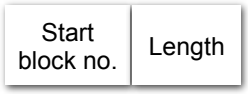

Figure 1: Extent

Inodes in Ext4 no longer use the indirect block mapping scheme. Instead they have extents which are used to denote range of contiguous physical blocks owned by a file. Huge files are split into several extents. Four extents can be stored in the Ext4 inode structure directly and if more extents are required (eg. in case of very large, highly fragmented or sparse files) they are stored on disk in the form of an Htree. Extents offer multiple advantages. With extents, the amount of metadata to be written to describe contiguous blocks is much lesser than that required by the double/triple indirect mapping scheme. This results in improved performance for sequential read/writes. They also greatly reduce the time to truncate a file as also the CPU usage[2]. Moreover, extents encourage continuous layouts on the disk, resulting in lesser fragmentation[4]. Extents have been shown to bring about a 25% throughput gain in large sequential I/O workloads when compared with Ext3. Tests conducted using the Postmark benchmark, which simulates a mail server, with creation and deletion of a large

number of small to medium files also showed similar performance gain[2][10].

The crux of features like delayed allocation and persistent preallocation is the extent. Preallocation deals with pre-allocating/reserving contiguous disk blocks for a file without actually writing to them immediately. Due to this, the file remains in a more contiguous state, enhancing the read performance of the filesystem. Moreover, it also helps in reducing fragmentation to a great extent. With preallocation, the applications are assured that they will always get the space they need, avoiding situations where the filesystem gets full in the middle of an important operation[4].

With delayed allocation, block allocations are postponed to page flush time, rather than writing them to disk immediately. As a result there are no block allocations for short lived files, and several individual block allocation requests can be clubbed into one request[2]. Since the exact number of blocks required are known at flush time, the allocator tends to assign a contiguous chunk rather than satisfy short term needs, which probably would have split the file.

1.2 Extents in free space management

Ext4 also introduced extents in its free space management technique. To avoid changing the on-disk layout, extents were maintained only in-memory, eventually relying on the on-disk bitmap blocks. One of the problems faced by the Ext3 block allocator, which allocated one block at a time, was that it did not perform well for high speed sequential writes[5]. Alex Tomas addressed this problem and implemented *mballoc* - the multiple block allocator. Mballoc uses the classic *buddy* data structure.

Whenever a process requests for blocks, the allocator refers the bitmap to find if the goal block is available or not. After this point is where the traditional balloc and mballoc differ. Balloc would have returned the status of just one block and this would have continued for every block that the process requires. Mballoc, on the other hand constructs a buddy data structure as soon as it fetches the bitmap in memory. Buddy is simply an array of metadata, where each entry describes the status of a cluster of nth power of 2 blocks, classified as free or in use[5]. After the allocator confirms the availability of the goal block from the bitmap, it refers the buddy to find the free extent length starting from the goal block, and if found, returns the extent to the requesting process. In case the extent is not of appropriate length, the allocator continues to search for the best suitable extent. If after searching for a stipulated time, a larger extent is not found, then mballoc returns the largest extent found in that search.

Both, the bitmap and the buddy are maintained in the page cache of an in-core inode[3]. Before flushing the bitmap to disk, information from the buddy is reflected in the bitmap and the buddy is discarded.

The bitmap and buddy combination enabled mballoc to speed up the allocation process. The combination of delayed allocation and multiple block allocation have been shown to significantly reduce CPU usage and improve throughput on large I/O. Performance testing shows a 30% throughput gain for large sequential writes and 7% to 10% improvement on small files (e.g., those seen in mail-server-type workloads)[2]. The credit for this goes to mballoc's ability to report free space in the form of extents. However, this mechanism still raises certain issues:

1. Even though the buddy scheme of Ext4 is more efficient at finding contiguous free space than the bitmap-scanning scheme of Ext3, the overhead of fetching and flushing bitmaps is still involved. Updating the bitmaps in-memory is fast, seeking and fetching them is the bottleneck.

2. Initializing the buddy bitmaps entails some cost[3]. Every time a bitmap is fetched into memory, there is an extra overhead of constructing the buddy.

3. Usually, only one structure is used to define the free space status of the filesystem. However in case of mballoc, both the buddy and the bitmap are used. Both these structures have to be updated on every allocation/deallocation. This introduces redundancy.

4. The buddy technique consumes twice the amount of memory as compared to only bitmaps. Thus, lesser number of bitmaps can reside in memory, resulting in more seeks.

5. Whenever preallocation has to be discarded, there is a comparison done between the buddy and the on-disk bitmaps. The on-disk bitmaps need to be referred to find out the exact chunk of space utilized. This leads to even more seeks.

6. Finally, when a filesystem has significantly aged, the buddy structure will be of little use as the available disk space may hardly be available contiguously. In such cases we currently have to rely on the primitive bitmap technique which is inefficient and slow.

The above points clearly indicate that optimizations are possible in the Ext4 block allocator. Something like an in-memory list for more optimal free extent searching, would further assist mballoc[3].

The underlying phenomenon of the success of extents, is that the filesystem usually deals with blocks in chunks, unlike inodes which are dealt with individually. Creation/deletion of files/directories mostly involves dealing with chunk of blocks. It thus seems natural to represent their free space status, also in the form of chunks. We take this idea further and explore a technique that is based entirely on extents. This technique is called **Space Maps**.

2 Design Details

In implementing this technique, there is a change to Ext4's on-disk layout. In Ext4 without space maps, for 4K block size, each 128MB chunk is called a blockgroup and each blockgroup has a bitmap. For space maps, we have combined a number of such block groups, amounting to 8GB space, calling it a metablockgroup, and each metablockgroup has a space map. The 8GB size is the default size of a metablockgroup for a 4K block size filesystem. The metablockgroup size is tunable at mkfs time.

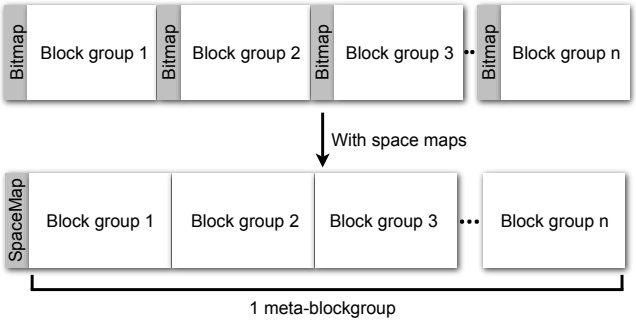

Figure 2: Metablockgroup

A space map consists of a red-black tree and a log. The nodes of the tree are extents of free space, sorted by offset and the log maintains records of recent allocations and frees. The tree and the log together provide the free space information.

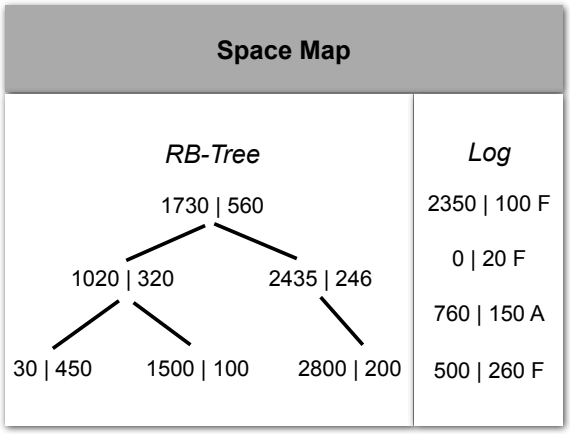

Figure 3: Structure of space maps

Bitmaps have a linear relationship with the size of the filesystem. This is not true with extent based techniques. Their size changes dynamically depending on the state of the filesystem. The tree has as many nodes as there are number of free space fragments (holes) in the metablockgroup. An experiment was conducted to get an idea of the space required by the space maps. *E2freefrag*[9] (a free space extent measuring utility) was executed on a 12GB Ext3 partition which was more than 90% full. Totalling the extents of various sizes, there were in all 1175 extents. Ext3 is good at avoiding fragmentation, but Ext4 is even better. Even then, as a safe cut-off mark, let us assume 2000 fragments in one metablockgroup i.e. 8GB, and calculate the space required for space maps. Here, one space map would require 2000 entries * 20 bytes per tree node entry = approximately only 40KB for the tree and let us keep aside 8KB for the log. Hence, space maps consume 48KB for 1 metablockgroup. This means that for an 8TB filesystem, where bitmaps would have required 256MB, space maps require merely 48MB i.e. only around 1/5th of the space required for bitmaps. This significant reduction in size enables space maps to reside entirely in memory at all times. To find free space, the allocator has to refer only the in memory structures and update them. This eliminates the huge I/O traffic from reading and writing bitmaps, which resulted from the fact that only a limited number could reside in memory at any given time.

The space maps are initialized at mkfs time. When the

filesystem is mounted, each space map is read into memory. They persist in memory for the duration that the filesystem remains mounted. During this period they keep their interaction with on-disk structures to a minimum, such that filesystem integrity is maintained. On unmounting, space maps are flushed back to disk.

The detailed description of the tree and the log along with a reasoning of why they are chosen is given below.

2.1 In-memory structures

2.1.1 Red black tree

The red black tree[7] of the space map, as described above, consists of nodes which are extents of free space, sorted by offset. The red black property of the tree helps the tree adjust itself if it is skewing to any one side. This limits the height of the tree making searches efficient. The tree is the primary structure denoting the free spaces in a particular metablockgroup, while log is a secondary structure, temporarily noting recent operations.

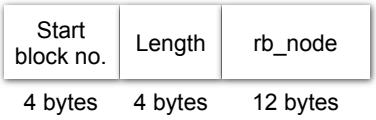

Figure 4: In-memory tree node

The tree node is 20 bytes in size. The start block number is relative to the first block of the metablockgroup to which the space map belongs. Hence, just 4 bytes suffice for the start block number and length fields.

2.1.2 Log

The log being an append-only structure, insertions to the log are very fast. Thus all operations are initially noted in the log, and then depending on their nature, they are either reflected in the tree instantly or in a delayed manner. The log assists the RB tree in maintaining a consistent state of the filesystem. Another reason for choosing the log is its ability to retain frees. In bitmaps, once the requested deallocation was performed, the freed space was forgotten. Due to this, an allocation following the free would be searched completely independent of the recently done free. This involved the tedious task of searching the bitmaps again, possibly from a completely different part of the platter. Moreover this also increased the chances of holes in the filesystem. In such a situation, the log acts as a scratchpad noting the recently done frees which can directly be used to satisfy the upcoming allocation requests, if any. Additionally, as allocations following frees fill up the recent frees from the log itself they help reduce holes in the filesystem. The working section along with an example will explain the behaviour of the log in much more detail.

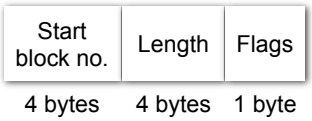

Figure 5: In-memory log entry

As each space map has a log, here too the start block number is relative to the start of the metablockgroup. The flags field is one byte with one of its bits denoting the type of the operation viz. allocation or free. Thus, a log entry is totally 9 bytes.

2.2 On-disk structures

2.2.1 Tree

For persistence across boots, the in-memory tree is stored on disk as an array of extents. At mount time, the extents from the on-disk tree are read to form the in-memory tree. The on-disk tree is updated only under two circumstances; when the filesystem is unmounting or when the on-disk log gets full.

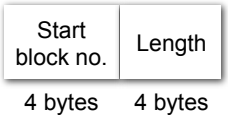

Figure 6: On-disk tree node

As shown in the preceeding figure, one on-disk tree entry consists of just one extent. Its size is 8 bytes.

2.2.2 Log

To avoid inconsistency in case of a crash, the operations noted in the in-memory log are also noted in the on-disk log in a transactional manner. As the log is an append-only structure only the last block of the on-disk log is required in memory. The exact operation is discussed in detail in the working of the technique.

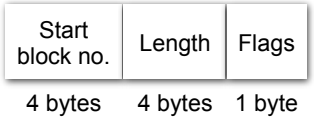

Figure 8: On-disk log entry

The on-disk log structure is same as that of the in-memory log.

3 Working

3.1 Allocation

The first flowchart outlines the allocation procedure.

3.2 Deallocation

The second flowchart explains the process of freeing space.

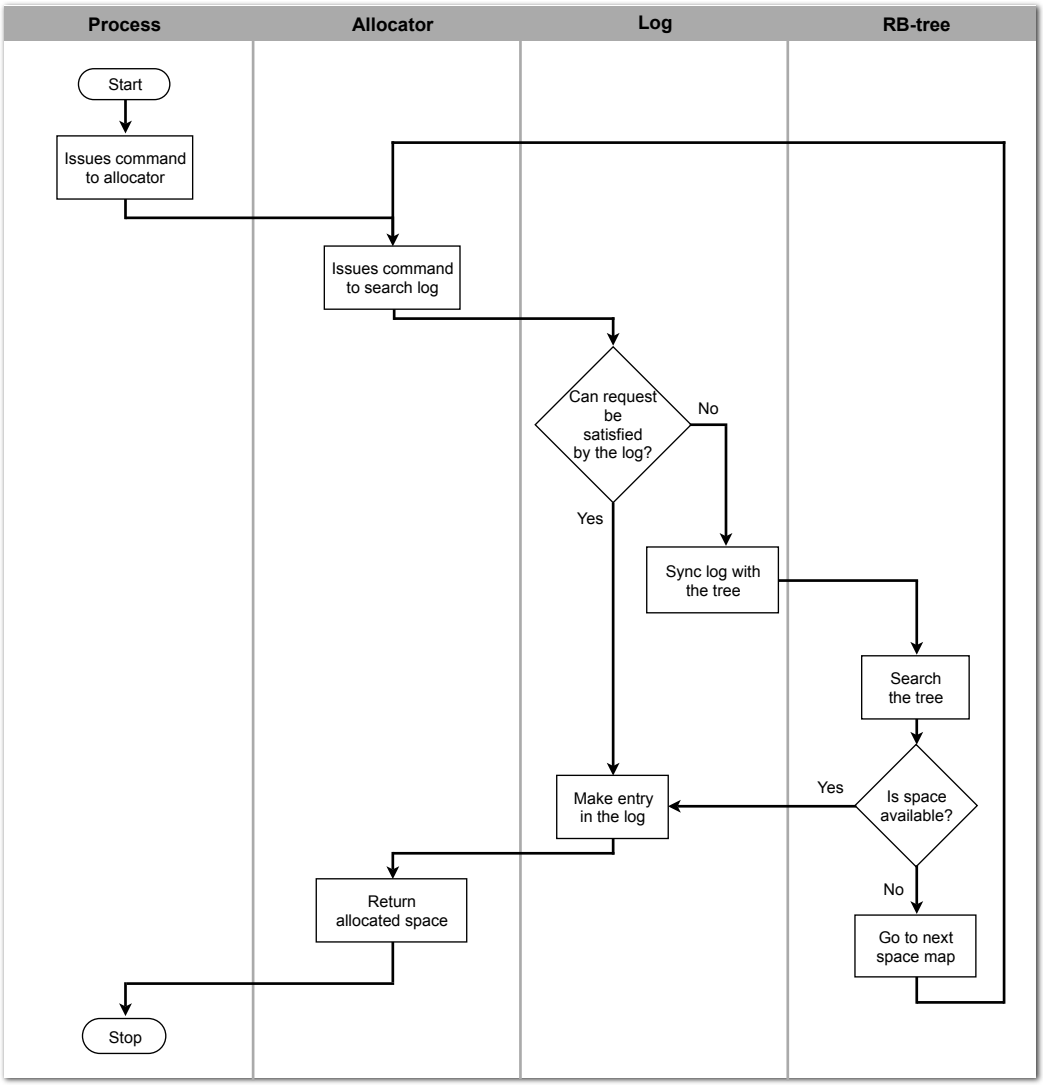

Figure 7: Allocation flowchart

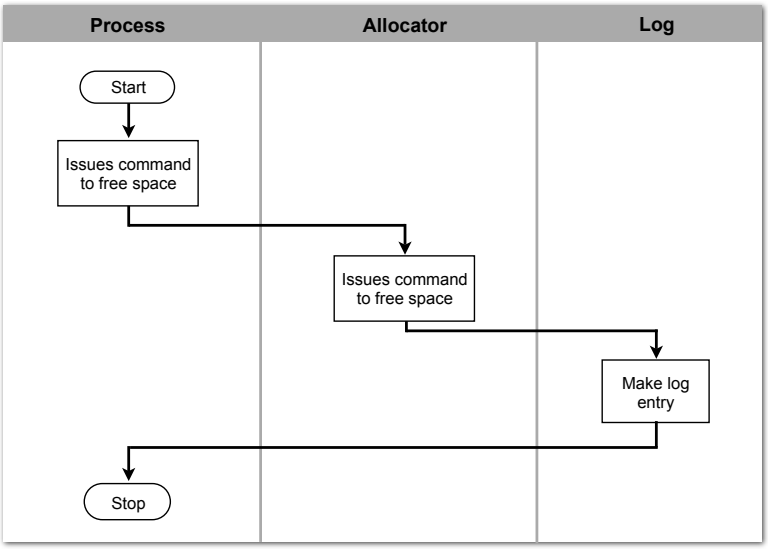

Figure 9: Free flowchart

The example below better illustrates the working of the system:

- Consider a newly made filesystem. As explained earlier, the log (left figure) is empty and the tree (right figure) consists of a single node. For the sake of this example, let us assume that a metablockgroup consists of 5000 blocks. The single node of the tree indicates that the entire metablockgroup is free.

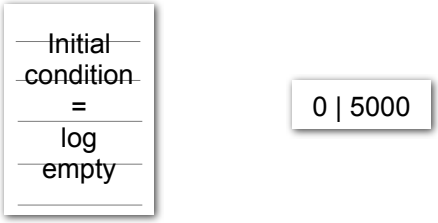

Figure 10: First scenario

- Suppose a process requests for 100 blocks. In this case, the allocator first searches the in-memory log of the goal metablockgroup for any recently done free operations that can satisfy the request. Since the log is empty, the tree is searched. As it is able to find a suitable extent, the process is allocated 100 blocks starting from, say, block number 0. Corresponding entries are made in the in-memory and on-disk logs in a single transaction. Note that neither logs are synced with the trees immediately.

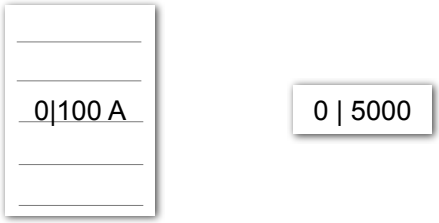

Figure 11: Second scenario

- Now, suppose there is a request for allocation of 150 blocks. As there is no entry in the log that can satisfy the request, the tree has to be searched. But since the tree does not reflect the updated state of the filesystem, we first need to sync the in-memory log with the in-memory tree. The in-memory log is then nullified. Here, the on-disk log is not synchronized with the on-disk tree as it would result in rewriting the entire on-disk tree. Writing the entire tree to disk every time any operation takes place would result in a lot of unnecessary writes to the filesystem. The beauty of this design is that as the on-disk structures are meant only to maintain space maps across boots, and are not referred for any allocation/deallocation, it suffices to just make a note of the operations somewhere on disk. The log serves this purpose. Hence, only the last

block of the append-only on-disk log needs to be in memory at all times to which we append entries of operations. Assume that the allocator assigns 150 blocks starting from block number 450. Entries in the logs are made accordingly. Even if the in-memory structures and on-disk structures are different, the in-memory log + in-memory tree = on-disk log + on-disk tree; thus maintaining consistency. If there is a crash at this point, replaying the on-disk log onto the on-disk tree will give us the exact state of the filesystem as it was before the crash.

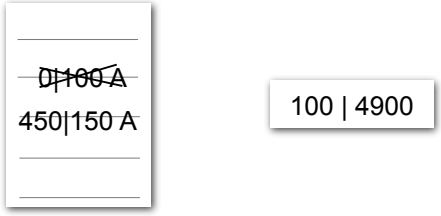

Figure 12: Third scenario

- Another allocation request of 200 blocks is tackled in a similar fashion.

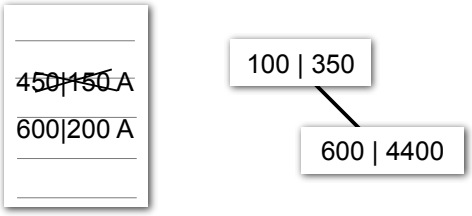

Figure 13: Fourth scenario

- Suppose now there is a request to free 150 blocks starting from block number 650. For a free request, ideally, there should be no need to search anything at all. All that is required is that the free space manager is informed about the blocks that are freed. Here, the bitmap technique faces a problem. In bitmaps, the specific bits of the particular bitmap block (in whose block group the file resided) are to be nullified. This causes a lot of seeks. This is where the log plays its most important role. The fastest way of informing space maps about a free is simply appending an entry to the logs. That is exactly what happens in space maps. So here, only the logs are updated with the entry suggesting that 150 blocks from block number 650 are freed. This maintains perfect locality of appends and results in very fast, virtually seekless operations. Furthermore, the deletion of a large, sparse or fragmented file requires many bitmaps to reside in memory. As against this, in space maps only the last block of the on-disk log (to which the appends are to be made) needs to be in memory. Hence, for an 8GB metablockgroup, where in the worst case (i.e. if the file/directory being deleted had occupied blocks in all 64 block groups comprising that metablockgroup) the bitmap technique would have required fetching all 64 bitmap blocks in memory, space maps require only 1 block (viz. the last block of the on-disk log) in memory. This not only reduces the memory consumption but also speeds up deallocation process.

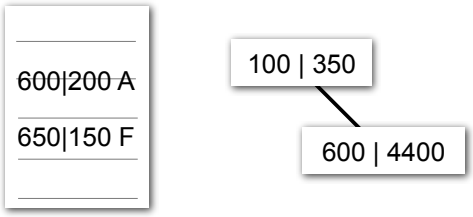

Figure 14: Fifth scenario

- Suppose there is another request for 150 blocks. In this case, the in-memory log does have a recent free which can satisfy the request. The 150 blocks are allocated from the log itself. This not only prevents another sync with the tree and a scan of the whole tree for an appropriate chunk, but also fills up a hole, thereby reducing potential free space fragmentation. The two entries for allocation and free are purged in the log itself.

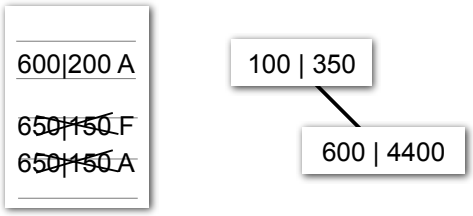

Figure 15: Sixth scenario

All further allocations and deallocations continue to

happen in the above-mentioned way.

Since logs are append-only, they can keep on filling indefinitely. The log sizes cannot be kept infinite and thus the design has to incorporate the handling of logs getting filled up completely. Consider the following two scenarios:

- *In-memory log gets full.* In this case, before the next entry to the log is made, the log is synchronized with the in-memory tree and nullified. Thus, the filesystem is still consistent.

- *On-disk log gets full.* In this case, the in-memory log is first synchronized with the in-memory tree. The in-memory tree showing the then most up-to-date condition of the filesystem overwrites the on-disk tree. After this, both the logs are nullified.

4 Evaluation

Space maps were put to test using some standard benchmarks. In the tests below, Ext4 (as of kernel 2.6.33.2) is compared with Ext4 (of the same kernel version) with space maps implemented. To really stress the allocator, the tests were conducted on a 50GB partition with memory size as 384MB. Also the filesystem was made with 1K block size to increase the number of bitmaps to simulate the behaviour of large filesystems.

4.1 Small file handling using postmark

Postmark[10] is a tool that simulates the behaviour of a web server, typically a mail server. Thus its tests involves creation and deletion of a large number of small files. In the test below, postmark was run 5 times. Each time 100000 files were added to the previous test, starting with 100000 files.

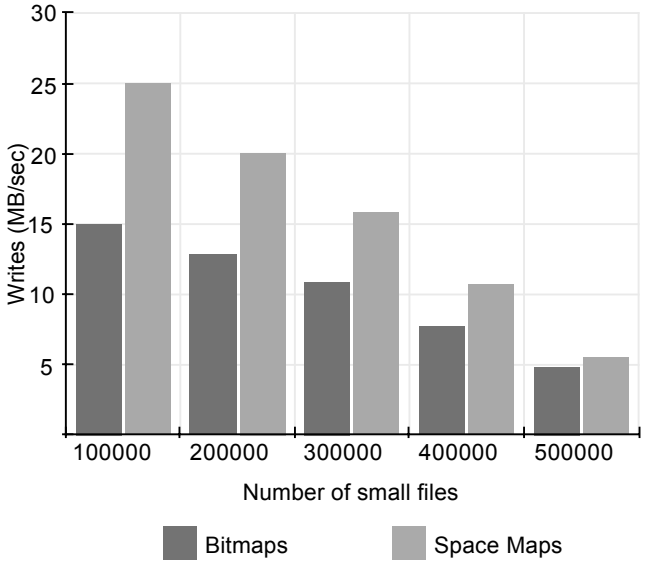

Figure 16: Graph of postmark

As predicted, better extents result in faster file writes. There is a stark difference in the speed of allocation initially between space maps and bitmaps, but the difference gradually reduces as the number of files goes up. Even then, at all times space maps allocation speed is higher than that of bitmaps.

4.2 Simultaneous small file and large file creation

Mballoc has the ability to treat large and small files differently. In order to stress mballoc, a test was conducted in which 5 linux kernels were untarred in different directories and 5 movie files (typically in the range of 700MB) were copied simultaneously. Operations like these jumble the allocator with large file and small file requests at the same time. In such scenarios, mballoc tends to make a lot of seeks. This is evident from the results below. The following statistics were taken when performing the tests on a newly made filesystem. The tool used below to measure seek count is called *seekwatcher*[8] by Chris Mason. It uses the output of the *blktrace*[11] utility and constructs a graph with time on the x-axis. It uses matplotlib to build the graphs.

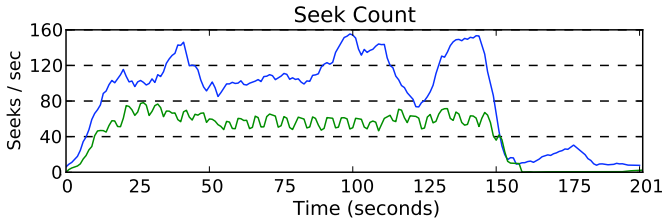

Figure 17: Stress test seek comparison run 1

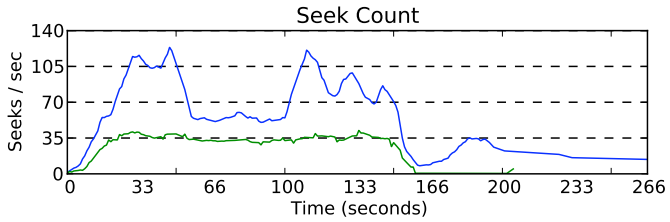

Figure 18: Stress test seek comparison run 2

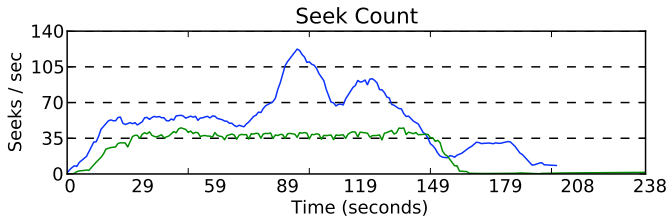

Figure 19: Stress test seek comparison run 3

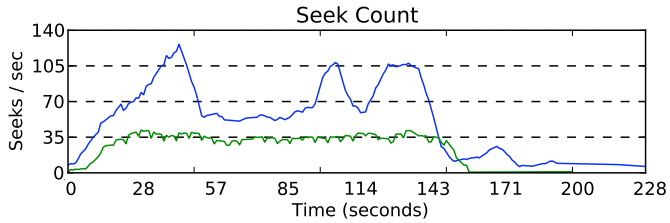

Figure 20: Stress test seek comparison run 4

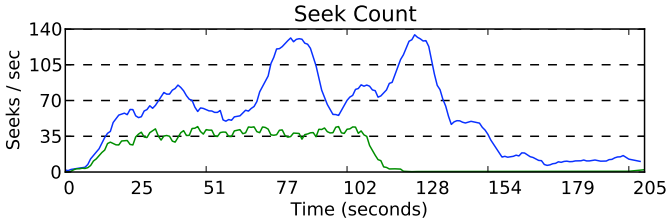

Figure 21: Stress test seek comparison run 5

The above test was conducted 5 times consequtively. As clearly visible in all the operations the seek count when allocation was done using space maps is less than half of the seeks required by the Ext4 that used bitmaps.

4.3 Free space fragmentation using e2freefrag

The following test measures the number and size of the fragments of free space in the filesystem. The test is just an extension to the previously performed simultaneous large and small file creation executed a total of 7 times. Fragmentation was measured at the end of each iteration. In Ext4 with bitmaps, we measure this attribute with *e2freefrag*[9], whereas in Ext4 with space maps, extents were nothing but the nodes of the trees. As clear below, the number of free space fragments of the filesystem go on reducing as the filesystem fills up. This is because the nodes of the tree get filled very efficiently resulting in lesser nodes, and thus lesser fragments.

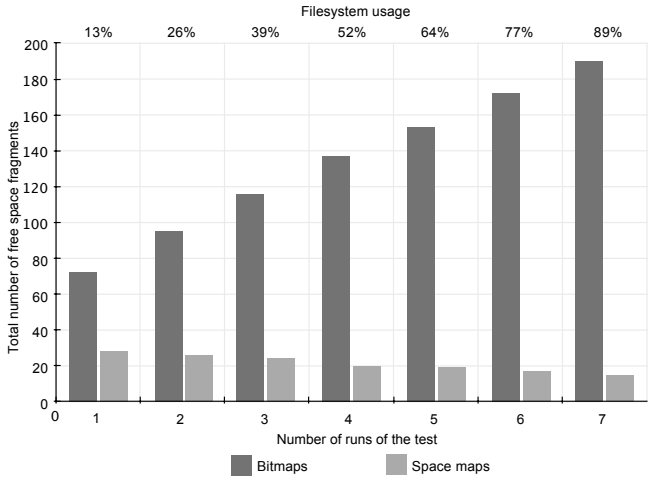

Figure 22: Graph of number of free space fragments

After having seen the number of fragments, let us have a look at the nature of the fragments in terms of their

size. In general, the larger the extents of free space, the more chances of a future file residing contiguously on disk. The results show that even when the filesystem is 89% full, the extents of free space greater than 1GB are around 74%, whereas in bitmaps it falls down drastically to 36%. This confirms that the more extent oriented information is available, the more efficiently allocations can be carried out with minimum free space fragmentation.

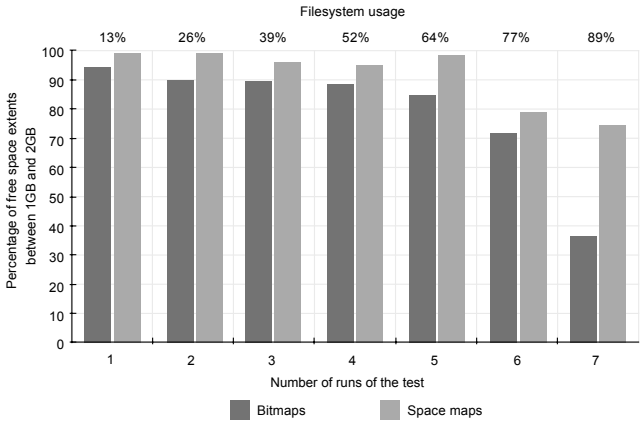

Figure 23: Graph of e2freefrag

4.4 File fragmentation using filefrag[12]

As stated earlier, the allocator tends to give better and more contiguous space to files if it recieves better extents. The graph below completely supports the claim. To measure the effects of file fragmentation, a test was conducted which involved copying of large 1GB files to a 10GB partition. The partition was made using the default flexible block group parameter of 16 block groups. Even then, the average number of fragments shown by files allocated using space maps are 1/5th of those allocated using bitmaps.

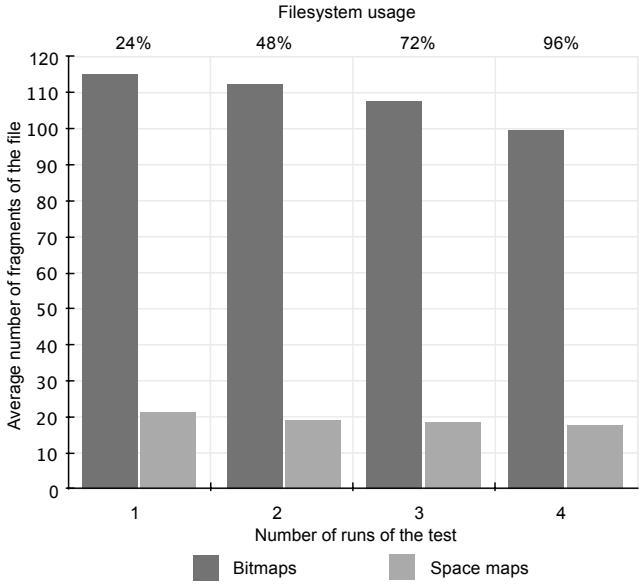

Figure 24: Graph of filefrag

5 Other benefits

- During mkfs, until the new uninitialized block groups feature was incorporated, all the bitmaps had to be invalidated to indicate that the entire filesystem was free. With uninitialized block groups you can now just set a field in the group descriptors indicating that the bitmap for this block group is invalid[2]. The actual bitmap block is nullified only when it is about to be used. This has significantly sped up the process of formatting Ext4. With space maps, the mkfs process involves just the entry of 1 node per metablockgroup indicating that the entire metablockgroup is free. This is as fast as the uninitilized block groups feature if not faster, as there are lesser number of metablockgroups than the group descriptors in a filesystem. Along with that, this completely takes care of marking the entire metablockgroup free as against just the invalidation in the group descriptor. So there is no extra operation required before using the particular metablockgroup for the first time and it is completely ready for usage.
- If the filesystem is made with 1K block size, then there are 16 times more bitmaps than the same filesystem made with the default 4K block size. Even in this case, as the space maps parameter for size of metablockgroup is tunable, the number and size of space maps can remain the same.

- Ext4 has done remarkably well to avoid fragmentation mainly due to the use of persistent preallocation. Even though this is the case, when the inode preallocation runs out of preallocated space, it may have to place a part of the file at a different offset. While doing this, fast access to flexible extents (extents not limited by block group size and/or not just as powers of 2) of free space eases the job of the allocator and results in lesser file fragments.

- Intelligent allocator heuristics will result in the reduction of the size of the tree as the filesystem goes on filling up. This will in-turn increase the allocation speed as tree lookups will be faster.

6 Limitations

- Every time the filesystem is mounted, the on-disk trees are read and the RB tree is constructed. This is time consuming as compared with the current bitmap technique as nothing has to be initialized with respect to bitmaps. Also while umounting, the trees have to again be traversed and stored onto disk in the form of a list of extents. That too is more time consuming as compared with the current scenario.

- In cases of extreme fragmentation (say every alternate disk block is empty) the memory consumption due to space maps will be higher than that of bitmaps.

7 Future enhancements

- One of the further optimizations could be the intelligent separation of the space maps based on file sizes. If we have separate space maps for large and small files, then the log entries in those space maps (for frees) will be of similar nature. Thus more allocation requests can be satified by the log itself without having to rely on the tree. This will result in maximum utilization of the log design.

- Another enhancement can be in designing a more efficient log. Currently, the log is simply an array of extents. Advanced data structures can enable much faster lookups of the log, resulting in even faster allocations/deallocations.

8 Related work

8.1 Space maps in ZFS

The concept of Space Maps is not new. ZFS, a solaris filesystem, also has the idea of space maps but the mechanism of implementation varies. Each virtual device on ZFS is divided into metaslabs, each having its own space map. Metaslab of ZFS is analogous to the metablock-group of Ext4. However, in case of ZFS, space map is simply a log of allocations and frees as they happen. Due to the use of the log, ZFS also benefits from the perfect locality of appends. Appends are made for allocations as well as frees.

Whenever the allocator wants to search for free space in a particular metaslab, it reads its space map and replays the allocations and frees into an in-memory AVL tree of free space, sorted by offset. At this time, it also purges any allocation-free pairs that cancel out. The on-disk space map is then overwritten with this smaller, in-memory version[6].

8.2 Comparison with space maps in Ext4

- In ZFS, space map is nothing but an on-disk log of allocations and frees. Also, an AVL tree is used which is the only in-memory structure. As against this, the Ext4 implementation has an RB tree along with an in-memory log helping reduce fragmentation and speed-up deallocations.

- Another difference is that the ZFS allocator has to update the tree for each and every request whether it is an allocation or a free. There can be cases when the entire tree needs to be reshuffled often. In a case where there are allocations following several frees and if the allocations can be satisfied by the recent frees, the Ext4 space maps can answer the request from the in-memory log itself instead of having to sync the tree time and again.

9 Conclusion

Space Maps demonstrate all the qualities essential for supporting fast free space allocation and deallocation in large filesystems common today. Finding free space can now be done entirely in memory and requires very little involvement of the disk. Space maps provide great

scalability and are proved to maintain filesystem consistency. We believe that improvements in space maps will further bring in optimizations and lift the current performance of this technique even higher.

10 Acknowledgements

We wish to thank the following people for their contributions; Ms. Pavneet Kaur, who shared her thoughts during the development of space maps. Mr. Anuj Kolekar who assisted during the testing phase. Mr. Kalpak Shah, our mentor whose ideas and timely criticism helped us bring this to fruition. Mr. Jeff Bonwick, who laid the seed of the idea. Last, but not the least, we wish to thank the anonymous reviewers, who patiently read our paper and gave us valuable inputs which helped make significant improvements in this paper.

References

[1] Mathur A., Cao M., Bhattacharya S., Dilger A., Tomas A and Viver L., *The New ext4 filesystem: current status and future plans.* (2007) [Online] Available: `http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf`

[2] Avantika Mathur, Mingming Cao and Andreas Dilger, *ext4: the next generation of the ext3 file system* (2007) [Online] Available: `http://www.usenix.org/publications/login/2007-06/openpdfs/mathur.pdf`

[3] Aneesh Kumar K.V, Mingming Cao, Jose R Santos and Andreas Dilger, *Ext4 block and inode allocator improvements* (2008) [Online] Available: `http://www.linuxsymposium.org/archives/OLS/Reprints-2008/kumar-reprint.pdf`

[4] *Ext4* (2010) [Online] Available: `http://kernelnewbies.org/Ext4`

[5] Mingming Cao, et.al. *State of the Art: Where we are with the Ext3 filesystem* (2005) [Online] Available: `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`

[6] Jeff Bonwick, *Space Maps* (2007) [Online] Available: `http://blogs.sun.com/bonwick/entry/space_maps`

[7] Rob Landley, *Red-black tree* Available: Linux kernel documentation

[8] Chris Mason, *Seekwatcher* [Online] Available: `http://oss.oracle.com/~mason/seekwatcher`

[9] Rupesh Thakare, Andreas Dilger, Kalpak Shah, *e2freefrag* [Online] Available: `http://manpages.ubuntu.com/manpages/lucid/en/man8/e2freefrag.8.html`

[10] Jeffrey Katcher, *PostMark: A New File System Benchmark* [Online] Available: `http://communities.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp-tr3022.pdf`

[11] Jens Axboe, Alan D. Brunelle and Nathan Scott, *blktrace User Guide* [Online] Available: `http://pdfedit.petricek.net/bt/file_download.php?file_id=17&type=bug`

[12] Theodore Tso, *filefrag* Available: Linux kernel documentation

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Programme Committee

Proceedings Committee