Chunkfs: Using divide–and–conquer to improve file system reliability and repair

Val Henson
Open Source Technology Center
Intel Corporation
val_henson@linux.intel.com

Arjan van de Ven
Open Source Technology Center
Intel Corporation
arjan@linux.intel.com

Amit Gud
Kansas State University
gud@cis.ksu.edu

Zach Brown
Oracle, Inc.
zach.brown@oracle.com

Abstract

The absolute time required to check and repair a file system is increasing because disk capacities are growing faster than disk bandwidth and seek time remains almost unchanged. At the same time, file system repair is becoming more common, because the per–bit error rate of disks is not dropping as fast as the number of bits per disk is growing, resulting in more errors per disk. With existing file systems, a single corrupted metadata block requires the entire file system to be unmounted, checked, and repaired—a process that takes hours or days to complete, during which time the data is completely unavailable. The resulting "fsck time crunch" is already making file systems only a few terabytes in size impractical to administrate. We propose *chunkfs*, which divides on–disk file system data into small, individually repairable fault–isolation domains while preserving normal file system semantics.

1 Introduction

Looking at industry projections for disk drive technology over the next 7 years, we see a familiar, expected trend: the capacity of individual disk enclosures will continue to double every 9–18 months[10]. File systems have successfully coped with this trend for two decades with relatively minor changes, and we might easily assume that file systems will continue to cope with exponential capacity growth for another two decades. But a closer look at three associated hardware trends sets off alarm bells (see Table 1).

	2006	2009	2013	Change
Capacity (GB)	500	2000	8000	16x
Bandwidth (Mb/s)	1000	2000	5000	5x
Seek time (ms)	8	7.2	6.5	1.2x

Table 1: Projected disk hardware trends[10].

First, disk I/O bandwidth is not keeping up with capacity. Between 2006 and 2013, disk capacity is projected to increase by 16 times, but disk bandwidth by only 5 times. As a result, it will take about 3 times longer to read an entire disk. It's as if your milkshake got 3 times bigger, but the size of your straw stayed the same. Second, seek time will stay almost flat over the next 7 years, improving by a pitiful factor of only 1.2. The performance of workloads with any significant number of seeks (i.e., most workloads) will not scale with the capacity of the disk. Third, the per–bit error rate is not improving as fast as disk capacity is growing. Simply put, every time disk capacity doubles, the per–bit error rate must be cut in half to keep overall errors per disk the same—needless to say, that is not happening. The absolute chance of an error occurring somewhere on a disk increases as the size of the disk grows.

What do these trends mean for file systems? Any operation that is O(size of file system) will take longer to complete—at least three times longer, factoring in only the effect of lower relative bandwidth. Flat seek time will further compromise the ability of the file system to scale to larger disks. The number of disk corruption events per disk will increase due to hardware–related media errors alone. The end result is that file system repair will take longer at the same that it also becomes more frequent—what we call the "fsck time crunch."

Our proposed solution, *chunkfs*, divides up the on–disk file system format into individually repairable *chunks* with strong fault isolation boundaries. Each chunk can be individually checked and repaired with only occasional, limited references to data outside of itself. Cross–chunk references, e.g., for files larger than a single chunk, are rare and follow strict rules which speed up consistency checks, such as requiring both forward and back pointers. Our measurements show that write activity already tends to be concentrated in relatively small subsets of the disk at any given time, making on–line checking, repair, and defragmentation of idle chunks ex-

ceptionally fast and simple.

2 The fsck time crunch

Most existing file systems have been designed under the assumption that media errors are rare. In the event that a media error corrupts file system metadata, the file system must be unmounted and repaired with fsck, a program that checks for and repairs file system metadata inconsistencies. At a high level, fsck traverses all file system metadata starting with the root directory, checking inodes and directories for internal consistency, rebuilding inode and block allocation maps, and checking for overall consistency between all metadata. Unfortunately, this process takes time proportional to the amount of file system metadata and involves many seeks to follow links and block pointers. With some exceptions, file system metadata grows roughly as the size of the file system, and the size of the file system grows roughly proportional to the size of disks. As disk capacity grows, fsck time grows proportionally.

Given increasing capacity, lower relative bandwidth, flat seek time, and increasing number of per–disk errors, the implications for fsck are as follows: (1) Fsck time will grow in absolute terms, to be on the order of days or weeks for "normal" file systems, (2) Larger file systems will have greater likelihood of suffering corruption of at least one piece of metadata, (3) *The fraction of time data is unavailable while running fsck may asymptotically approach unity in the absence of basic architectural changes.*

What can be done to combat this trend? Journaling file systems only speed fsck time in the case of a system crash, disconnected disk, or other interruptions in the middle of file system updates. They do not speed recovery in the case of "real" metadata corruption—a journaling file system must still read all metadata to correct any kind of error other than a half-finished update. The same goes for soft updates[11], copy–on–write[2, 8] and log–structured file systems[14], which are again designed to speed fsck after a crash or similar event.

Checksums on file system metadata, such as those added to ext3 in the IRON project[13], will detect errors and improve reliability, but won't change the need to read all file system metadata to repair consistency. File system level replication of metadata combined with checksums, such as in ZFS[2], can eliminate nearly all need to run fsck for file system corruption caused below the file system layer, but uses more disk space and does not help with corruption caused by file system bugs as the inconsistency will be present in both checksummed copies. Those inclined to dismiss file system bugs as a significant source of file system corruption are invited to consider the recent XFS bug in Linux 2.6.17[1] requiring repair via the XFS file system repair program, causing significant downtime. Even worse, the existing XFS repair program could not fix this bug, so developers had to write new repair code and distribute the new utility before the file system could be repaired. While file system bugs may be relatively rare, we must take them into consideration when the cost of such a bug is several hours or days of downtime to repair the file system.

RAID solutions can improve the reliability of the underlying disks, but corruption still occurs frequently enough that it we cannot rely on it alone when, again, the cost is hours or days of downtime. Recently the main server for kernel.org, which hosts several years' worth of Linux kernel archives, suffered file system corruption at the RAID level; running fsck on the (journaling) ext3 file system took over a week, more than the time required to restore the entire file system from backup. In addition, two of our primary design targets, desktops and laptops, so far have not been attractive markets for RAID.

To help understand why existing techniques do not solve the fsck time crunch in and of themselves, consider the case of a file with a corrupted link count. To find the true link count, you must read every directory entry and count up all the entries with links to that file. Fundamentally, answering the question of the correct value of one bit in the block allocation bitmap requires reading every piece of metadata in today's file systems.

One apparent solution is to optimize fsck—perhaps fsck only takes so long because it is (to state it baldly) poorly written. However, when it comes to optimizing fsck, most of the low–hanging fruit has already been plucked. For example, many of the more well–known inefficiencies of fsck[4] have been corrected in the ext2/3 fsck utility. And to compensate for exponential reduction in performance due to hardware trends, we would have to continuously improve fsck performance at the same exponential rate for years—an unsustainable solution.

Another workaround is to divide up the disk into many smaller file systems and manage them individually. This approach has many drawbacks. Each file system must be mounted at a particular point in the namespace, and all files in each file system must be children of the same top–level directory, constraining the organization of files. Disk space becomes fragmented, so one file system may run out of space while several others have plenty of free space. When this happens, files must be rebalanced between file systems by hand. Most existing file system code is not designed for or tested well for I/O errors, and when one file system suffers a failure, it often causes cascading failures in other supposedly independent file systems sharing the same code or system resources. Simply partitioning the disk into many small file systems produces a brittle, hard–to–manage system with somewhat lower data loss but not much improvement in data availability.

3 Designing for the next decade

We need a file system architecture that can scale into the next decade. Our goals are fast and robust recovery from file system corruption, same or reduced administrative burden, look and feel of a normal POSIX file system, strong fault detection and isolation, and no O(size of file system metadata) operations.

3.1 Repair–driven design

Traditional file system on–disk formats are designed primarily with the goal of improving the performance of the file system in normal use. We argue that the performance and reliability of file system *repair* should also be a major design goal when designing the on–disk format. Trading off some steady–state performance for advantages in repair and reliability especially makes sense for file systems, for which the ultimate benchmark is how reliably it can store and retrieve data. We describe this philosophy as *repair–driven design*.

Repair–driven design encourages redundancy and checksums in the on–disk format. Checksums, magic numbers, UUIDs, back pointers, and outright duplication of data are all good examples of repair–driven design. Optimizations intended to compress on–disk data and reduce redundancy are antithetical to repair–driven design as they increase the damage possible from each incident of file system corruption and increase the difficulty of repairing the corruption. Similarly, the on–disk format should avoid "fragile" data structures—data structures that are complex to update, highly interdependent with other data structures, and difficult to interpret and repair when partially corrupted. B–trees of all sorts, dynamic metadata allocation, complex on–disk look up structures and the like improve on–line performance but tend to perform badly in repair and recovery. A simple linear directory layout performs poorly with some workloads but is trivial to repair.

One more aspect of repair–driven design is that the on–disk format should be structured and laid out such that traversing the file system in the order needed for repair is fast.

4 Chunkfs architecture

The core idea of chunkfs is to split a file system up into many small fault isolation domains on disk—chunks, on the order of a few gigabytes in size. Each chunk has its own block number space, allocation bitmaps, superblock, and other traditional per–file system metadata. A small amount of metadata about the location and contents of the chunks is stored outside the chunks in a summary region. The file system namespace and available disk space are still shared, so that it still feels like one file system to the user and administrator.

Splitting up the file system into chunks is easy. The difficulty is in gluing it back together again to preserve the shared namespace and disk space, while keeping chunks fault–isolated from each other. The basic rules are, (1) All cross–chunk references have explicit forward and back pointers, and (2) Keep cross–chunk references to a minimum. This section describes how we use these principles to solve the major difficulties in implementing chunkfs.

4.1 Continuation inodes

The first problem we encounter is how to store files larger than the space available in a chunk. We could allocate blocks from another chunk and have direct pointers to them from our chunk, but that would make file system repair no longer local to the chunk. To understand why, think about repairing the block allocation bitmap. If only inodes inside a chunk can have pointers to blocks in that chunk, then we only have to check the inodes inside the chunk to determine whether a particular block is truly allocated or not. But if we allow inodes in other chunks to have pointers to blocks in our chunk, we have to check all inodes in all chunks and we are back to square one.

Our solution to this problem is called *continuation inodes*. We pre–allocate forward and back pointers in every inode. When a file outgrows a chunk, we allocate another inode in another chunk, mark it as a continuation inode, and fill out the forward and back pointers to create a doubly linked list. Then we allocate blocks from the continuation inode's chunk and point to them from the continuation inode. Continuation inodes are checked in an additional final pass of fsck, to check that the forward and back pointers agree. Directories that outgrow their chunk are handled in the same way as files.

The next obvious problem is hard links between directories and files in different chunks. In order for fsck to quickly check the number of links to an inode, the inode and all directory entries referencing it must be in the same chunk. While we prefer to allocate directories and the files they link to in the same chunk, eventually we have to allocate from a different chunk, creating a problem for our goal of keeping link count calculation local to a chunk. Another more direct source of cross–chunk hard links are files with hard links from multiple directories in different chunks—clearly the inode can't be in more than one chunk.

Our solution is to allocate a continuation inode for the linked-to inode in the same chunk as the directory. If the directory's chunk is full, we allocate a continuation inode for both directory and inode in a chunk with free space. An inode's link count is the sum of the link counts of all its continuation inodes (this link count can be cached in the "parent" inode).

4.2 Continuation inode implementation notes

The implementation of continuation inodes must avoid some obvious pitfalls. First, we must avoid excessive

numbers of continuation inodes, both in terms of the total number in the file system and number per file. One especially painful scenario is what we call the doubling back problem, which will happen when file grows while free space moves around the file system due to other activity. We may end up beginning allocation in chunk A, moving to chunk B, and then returning to chunk A because more space has been freed up in the meantime. If we are not careful, we could end up with one data block per inode.

Our solution is to implement sparse files and allow each continuation inode to encompass an arbitrary set of blocks in the file. This results in a maximum overhead of one continuation inode per file per chunk, which is the lowest upper bound theoretically possible (consider the case of a single file which fills the entire file system).

In order to make intra-file seek times reasonable, we may have to implement a lookup structure of some sort, depending on how fragmented files end up being in typical usage. The doubly linked list structure in each inode could be easily replaced with a tree node or hash structure to speed offset location. We could also generate an in–memory lookup structure when a file with continuation inodes is first accessed.

The size of chunks should be small to speed up per-chunk fsck, but large enough that the cross–chunk pass is still fast (i.e., continuation inodes should be rare). Two important inputs to this decision are the size of files and the number of hard links to files. By definition, large files requiring continuation inodes are rare because you cannot store many large files before you run out of space on the file system. A quick survey of several Linux developers' laptop file systems during the 2006 Linux file systems workshop[6] found that 99.5% or more of files were less than 1MB in size. We guesstimate that an optimal chunk size would be on the order of 1/100th of the total file system size. With regard to multiple hard links, keep in mind that the vast majority of files have exactly one link and will be allocated in the same chunk as the parent directory. Our informal laptop file system survey found that only about 1% of files had more than one link.

4.3 Directory hard links

We have decided to violate the strict interpretation of UNIX file system semantics in favor of reliability and simplicity and disallow multiple hard links to directories, so determining proper directory reference counts is relatively easy. The Linux VFS layer has never supported multiple directory hard links, and all modern UNIX implementations at minimum restrict such a capability to the superuser due to the damage directory cycles can cause.

4.4 File system repair

Our primary goal is to make fsck go fast. This is accomplished by checking chunks in parallel and only checking chunks that need it (i.e., only if we have some reason to suspect inconistency or corruption). During fsck, each chunk is checked in the usual way, without referring to other chunks. Metadata consistency checks that reference other chunks must be done in a final pass across all chunks; the necessary metadata from each chunk must be quickly accessible.

5 Benefits of chunkfs

The design principles of chunkfs are closely related to those of crash–only software[5]. Our goals are safe crashes, quick recovery, and strong fault isolation. Chunks suffering disk corruption ("bugs") can be unmounted ("crashed"), checked and repaired ("recovered"), and remounted, all independent of other chunks (modulo the UNIX directory structure).

5.1 Reduced fsck time

When part of the file system suffers corruption (detected either by checksum errors, safety checks in the file system code, or errors reported by the underlying storage), only the chunk containing it must be checked. Likewise, when the system crashes, only chunks with metadata being actively modified at the time of the crash must be checked before the file system can be brought back online. Since the chunks are relatively small (order of a few gigabytes), the fsck time is correspondingly short.

5.2 Reduced fsck memory requirements

Fsck sometimes requires more memory to run than is available to the system, especially early in boot. One author's backup server recently could not be fsck'd because it had so many directory entries that they could not all fit into memory at once. Chunkfs only requires enough memory to check one chunk at a time, and to make a pass over all cross–chunk references.

5.3 Strong fault boundaries

It is difficult for corruption in one chunk to cause corruption in another chunk, since no allocation bitmaps are shared and continuation inodes require agreement about forward and back pointers from the associated inode in the separate chunk. If pointers do disagree and can't be repaired, the consequences for the continuation inode in the uncorrupted chunk are at worst orphaning and a move to the `lost+found` directory.

5.4 On–line partial fsck

At any given time, a relatively small subset of a file system is write–busy—that is, the metadata is being modified. This is partly because most file systems try to keep writes grouped on disk for better performance, and partly because disks simply aren't capable of writing to the entire platter at once. We measured the distribution of metadata updates to the file system by instrumenting the ext2 file system to record which block groups had metadata updates using a one–second time

slice. We found that over a period of 50 minutes of active use of the file system on a development laptop doing web–browsing, file editing, and kernel compilation, all block groups were clean 98–100% of the time. While filling up the file system as quickly as possible with an artificially constructed workload using both `dd` and `cp -r`, all block groups were clean at least 75% of the time, and most were clean far more often. While laptop disks are not particularly high performance, these results confirm our intuition that metadata updates tend to be localized in both time and space. In other words, only a few block groups are being actively modified at any given time.

We predict that many chunks will be idle with respect to metadata writes most of the time. We can take advantage of this to incrementally check chunks on–line. Chunks that are too busy to check while on–line (a relatively small subset) can be quickly and completely checked at the next mount. We implemented a "dirty bit" indicating whether a file system is being currently modified for the ext2 file system as a proof of concept and found it to be a relatively easy task[7]. On–line repair will be more difficult and will require careful handling of open files and management of kernel structures, and may not be worth solving if the file system need be off–line for only a few minutes to complete the repair. In addition, in the event of a crash, the dirty bits indicate which chunks need to be recovered and which we can skip, shortening recovery time significantly. We can also randomly check a few chunks at every mount; over time you check everything while the incremental price is low. This kind of scrubbing is especially important given the prevalence of latent (invisible) faults and their effects on long–term data preservation[3].

5.5 Per–chunk on–disk format

Each chunk could be implemented to have different ratios of inodes to data, different block sizes, or other differences in layout, and can be initialized to a particular layout when first allocated. For example, a large file could be stored in a chunk containing exactly one inode and the associated data. File system growth is simple—add another chunk—and file system shrinkage is much easier, since the inode that "owns" any allocated block is in the same chunk (rather than located potentially anywhere on disk).

6 Drawbacks of chunkfs

Chunkfs will only work if cross–chunk references can be kept rare. Heavy fragmentation of chunk free space resulting in many continuation inodes will waste space and increase file system check and repair time. We expect to use a common tactic for controlling file system fragmentation: reserve a certain percentage of free space, avoiding worst case fragmentation and performance degradation at the cost of relatively cheap and plentiful disk space. Another mitigating factor is that the chunk structure greatly simplifies on–line defragmentation.

While most consistency checks can be done on each chunk individually, we must still check inter-chunk consistency. For example, if we check chunk A and follow a continuation inode's back pointer to the original inode in chunk B, and then check chunk B and discover that the original inode was orphaned, we will then have to free the continuation inode and associated blocks in chunk A.

The hierarchical structure of UNIX file systems complicates the goal of truly independent chunk failures, since a component in the pathname of a file in chunk A may be located in chunk B. If chunk B is corrupted, the file A is disconnected and unavailable. This can be mitigated by directory duplication, caching parent name hints in inodes, and by mechanisms to mount chunks independently or otherwise get access to all files in a chunk.

7 Status

We recently released a prototype of chunkfs using FUSE on Linux with basic functionality, and have begun work on a chunkfs fsck. Our preliminary performance measurements do not indicate any glaring performance problems as yet. The next step will be an implementation based on ext2. We are not using ext3 primarily because the main advantage of ext3 over ext2 is fast recovery after a crash; since the goal of chunkfs is extremely fast fsck, ext3's main advantage over ext2 is reduced (we only have to run fsck on the few chunks with metadata being actively modified at the time of the crash).

8 Related work

As noted in Section 2, most file systems have focused on repairing the file system after a system crash or other cause of unclean unmount. A few file systems, such as ZFS[2], have gone as far as checksumming and duplicating all file system metadata, which reduces the frequency of fsck but not the overall time. At least two file systems, ext3 and XFS, have plans underway to parallelize fsck further within the constraints of the existing on–disk format.

A method of reducing fsck time proposed in [12] involves tracking active block groups and using this information plus a few other kinds of metadata to speed up fsck. Some of the authors independently "rediscovered" and evaluated a similar solution, called per-block group dirty bits (see Section 7 in [7]). We quickly discovered that block groups were not a meaningful boundary for file system metadata, since any inode can refer to any block in the file system, and any directory entry to any inode. For example, operations that change link counts are

not helped by per-block group information. The method of working around this problem used in [12], *link tags*, led to a significant performance hit without an NVRAM write cache. In Section 6 in [7], we outlined a similar approach, *linked writes*, which creates a list of dirty *inodes* and orders writes to them such that the file system can be recovered by scanning data pointed to by the dirty inodes—a not particularly elegant approach which we did not pursue.

Many distributed file systems have mechanisms for improving fault isolation between individual servers[9, 15], mostly through replication of metadata and/or data, but none that we know of explicitly address improving file system repair time.

9 Conclusion

Chunkfs improves file system reliability, shortens repair time, and increases data availability by dividing the file system into chunks, small fault–isolation domains which can be checked and repaired almost entirely independently of other chunks. This allows fast, incremental, and partially on–line file system checking and repair. In addition, the chunkfs architecture makes many other useful file system features feasible, such as on–line resizing, on–line defragmentation, and per–chunk on–disk formats.

10 Acknowledgments

Thanks to Brian Warner, Kristal Pollack, Greg Ganger, Vijayan Prabhakaran, John Wilkes, and our anonymous reviewers for many excellent comments and suggestions on this paper. Thanks to the participants of the 2006 Linux File Systems workshop and the 2006 Hot Topics in Dependibility workshop for their inspired and wide-ranging feedback on the design of chunkfs. Finally, thanks to Theodore Y. T'so for many excellent discussions which inspired much of this work.

References

[1] SGI Developer Central Open Source | XFS. `http://oss.sgi.com/projects/xfs/faq.html#dir2`.

[2] ZFS at OpenSolaris.org. `http://www.opensolaris.org/os/community/zfs/`.

[3] M. Baker, M. Shah, D.S.H. Rosenthal, M. Roussopoulos, P. Maniatis, TJ Giuli, , and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006*, 2006.

[4] Eric J. Bina and Perry A. Emrath. A faster fsck for BSD UNIX. In *USENIX Winter Technical Conference*, pages 173–185, 1989.

[5] George Candea and Armando Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[6] Valerie Henson. The 2006 Linux File Systems Workshop. `http://lwn.net/Articles/190222/`.

[7] Valerie Henson, Zach Brown, Theodore Y. Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 file systems. In *Ottawa Linux Symposium 2006*, 2006.

[8] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[9] Minwen Ji, Edward W. Felten, Randolph Wang, and Jaswinder Pal Singh. Archipelago: An island-based file system for highly available and scalable Internet services. In *4th USENIX Windows System Symposium*, 2000.

[10] Mark Kryder. Future storage technologies: A look beyond the horizon. `http://www.snwusa.com/documents/presentations-s06/MarkKryder.pdf`.

[11] Marshall K. McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17. USENIX, 1999.

[12] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast consistency checking for the Solaris file system. In *USENIX Annual Technical Conference*, 1998.

[13] 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.

[14] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, pages 1–15, 1991.

[15] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.