A Simulated Study of File System Metadata Scalability

Jianming Wu

Computer Sciences Department
University of Wisconsin, Madison

Abstract

In this paper, we analyzed four Linux file systems for their metadata scalability. The increasing gap between the size of data set and memory capacity has placed an pressure on existing file systems. We studied the design and implementation of these file system to gain an insight of how they organize their metadata to manage raw data. Simulations are conducted to see the simulated behavior of those four file system. We simulated a simple case with only one directory and one file with various size, an emulated disk with data from previous literature and a simple calculation is presented for an aging file system.

Our results shows that not all extent-based file system always has less overheads than traditional block-based file system, and those extent-based file systems with variable-sized inode structure have good flexibility to scale, thus avoid the metadata scalability problem that has been emerging.

1 Introduction

We present a simulated study of file system scalability problem caused by its metadata. Four file systems on Linux platform: Ext3, ReiserFS, JFS and XFS are studied to check their scalability to accommodate the technology advancement in the near future. The simulated approach enables us to gain a rough idea of how each different file system scales in artificial environments and then jump to conclusions on their scalability in practice.

File system scalability is not a new topic in operating system research literature. People's demand for larger storage system has never been underrepresented. To accommodate the increasing hard disk size and workload, the file systems have evolved to be more scalable. From the traditional UNIXfile system UFS, which was modeled from the memory and could provide satisfactory performance at that time, to nowadays XFS with complex data structures and algorithms to manage terabytes hard disk spaces. From the floppy disk file system FAT12, which has only 12 bits for cluster addressing, to modern file system's 64 bit addressing capacity.

Till now, the metadata scalability of file systems caused by the technology advancement has never been well studied. Recently, Mark Kryder proposed Kryder's Law [13] stating that the capacity of hard disk drive will double annually. A study of over 70,000 sample file systems from 2000 to 2004 shows that the aggregate fullness of hard disks seldom changes [1]. Therefore, we can conclude that the data set of a computer system will also double annually. However, the Moore's Law [8] predicts the transistor density on a single integrated circuit will double in 18 months, which determines the growing rate of main memory capacity of a computer system. So the fact that the growth rate of data set outpacing that of main memory may cause scalability problem of file systems. File systems use metadata to organize and manage data on disk and in memory, which places a storage overhead in the system. This overhead may become significant with the increasing gap between data set and memory, and finally cause performance degradation.

In this paper, we analyzed four file systems on Linux platform and simulated their overheads in several scenarios. The simplest case we simulated was a single file contained in the root directory, in which we studied the relationship between file size and metadata overhead in different file systems. Then we created an emulated file system with parameters taken from [1] to simulate the synthesized overheads. In this synthesized case, we studied file size with a typical distribution and two extreme case: small and large. We also simulated a simple aging file system to see its trend of scalability.

Our results indicate that well-designed block-based file system with variable-sized inode structure, such as ReiserFS, can have less overhead than some extent-based file systems, such as JFS, in cases when file size is relatively small. Extent-based file systems have sub-linear relationship with the size of data size. In optimized case, the overhead in extent-based file systems is constant. The most important conclusion is that extent-based file systems with variable-sized inode structure, such as XFS, can scale to a large extent no matter what distribution of the file sizes is.

The rest of this paper is as follows. A brief introduction to the file system architecture in Linux will be presented in Section 2. Then, four popular file systems on Linux platform will be examined in Section 3. The simulation results are shown in Section 4. Conclusions and related work are presented in Section 5 and 6.

2 A Tour of Linux VFS

The Virtual File System (VFS) [4] is first developed in SUN Network File System to provide a common interface for the kernel to operate both network file system and local file systems. Linux adopted this design to support tens of various file systems. It is an intermediate layer placed between the kernel and the underlying file systems in an operating system to provide a common interface to the kernel for file operations. It maintains common data structures in memory to present an abstraction of all files in this system. The underlying file systems manage on-disk

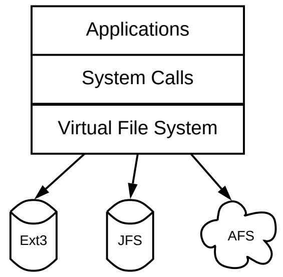


Figure 1: Virtual File System Layer

file data and structure. They implement various operations on actual storage systems. Thus, the kernel uses the same function names to call into the file systems, and VFS decides which file system specific calls will be invoked. The architecture of VFS is illustrated in Figure 1. In this figure, three underlying file systems are mounted to VFS. Two file systems (Ext3 and JFS) are typical disk file systems; while the third one, AFS, is a network file system.

When a partition is mounted, the particular file system on that partition will be registered at the VFS. On registration, the functions for file system-wide operations, `super_operations` and single file operations, `inode_operations` will be loaded into the corresponding VFS calls. Hence, the calls into VFS from kernel are directly translated into actual file system calls. Regarding data structures, VFS maintains data structures with common fields for all file systems that are essential to perform general operations. Each file system may have their own specific data fields in memory, but VFS is not aware of these data. Thus, the functions and data structures above and below VFS layer are well connected.

In the VFS layer, several data structures are designed to represent objects and improve performance. The `struct inode` is an in-memory abstract data structure to represent file objects in kernel for all file systems. Common data fields for all kinds of files, such as data block mapping, owner information and last access time, are stored in this structure. The `dentry` structure in Linux stands for directory entry, which caches mappings of file names and corresponding inodes. Thus, it provides short-cuts from file name to inode number, without traditional directory tree traversal. The `file` structure and `fdtable` structure are maintained for every process that performs file operations. The `file` structure keeps runtime properties of a file, such as the current offset. The `fdtable` structure is a file descriptor table that keeps a track of opened files and provides an process-wide index of files for users. Figure 2 shows the relationship of major data structures in VFS with Ext3 as the underlying file system [1].

As the major data structures in VFS are the same for all kinds of file systems. They have already received much attention to keep a reasonable balance between performance and memory efficiency. However, regarding file system specific metadata, most work done is performance oriented while less is done to improve memory efficiency and scalability. As the faster growth rate of hard disk drive capacity than memory capacity [13] and the stability of disk fullness [1], it is necessary to examine the overheads of these file system specific metadata to evaluate the scalability of file systems. In the next section, we will briefly revisit the design and implementation of four Linux file systems.

3 The Nuts and Bolts

In this section, we will examine the differences in the design and implementation of four popular file systems. We only focus on the metadata of data block organization as all file systems need to deal with this problem and this kind of metadata will possibly grow with regard to file size. Actual file systems have their own on-disk file layouts and data organizations. They need to implement complete on-disk file operations for the VFS. Due to there different design, the overheads of in-memory data structures are different. These overheads are mainly the data block pointers. With the growth of data set, this kind of previously underrepresented overhead should be well studied to prevent performance degradation.

[1]The inode for `/file_2` is not shown here

Table 1: A Summary of File Systems Features

	Fixed-size Inode Structure	Variable-sized Inode Structure
Block-based	Ext3	ReiserFS
Extent-based	JFS	XFS

These file systems mainly varies in the ways they organize data blocks and their inode structures. Some file system uses block pointers, in which each data block has one pointer. This is the most straightforward way to organize data blocks. Some file system uses extent-based organization, which uses a pointer and a length to represent a set of contiguous data blocks. This approach potentially reduces the overhead of data block pointers, but its space efficiency depends on the fragmentation of files. Thus, defragmentation utilities are needed to keep the disk at a relatively low fragmentation status. Some file system uses fixed size inode structure, which is easy to design and implement. However, because the number of data block per file varies, indirection technique, such as indirect pointer in Ext3 and B+ tree in JFS, is introduced, which places extra overheads in the file system. Some file system uses variable-sized inode structure, in which the data addressing part can grow as the file size grows. This approach gains advantage of flexibility in data organization, but more complex addressing mechanisms are needed to translate logic data block number to physical block number. In the rest of this section, four popular file systems in Linux: Ext3, ReiserFS, JFS and XFS, are studied. A summary of features of these four file systems is shown in Table 1. Their main features are briefly introduced and data addressing (based on Linux source code of kernel version 2.6.22) methods are examined.

3.1 Ext3

Ext3 file system is the simplest file system among all these the file systems we studied. It was designed to provide journaling feature while hold backwards

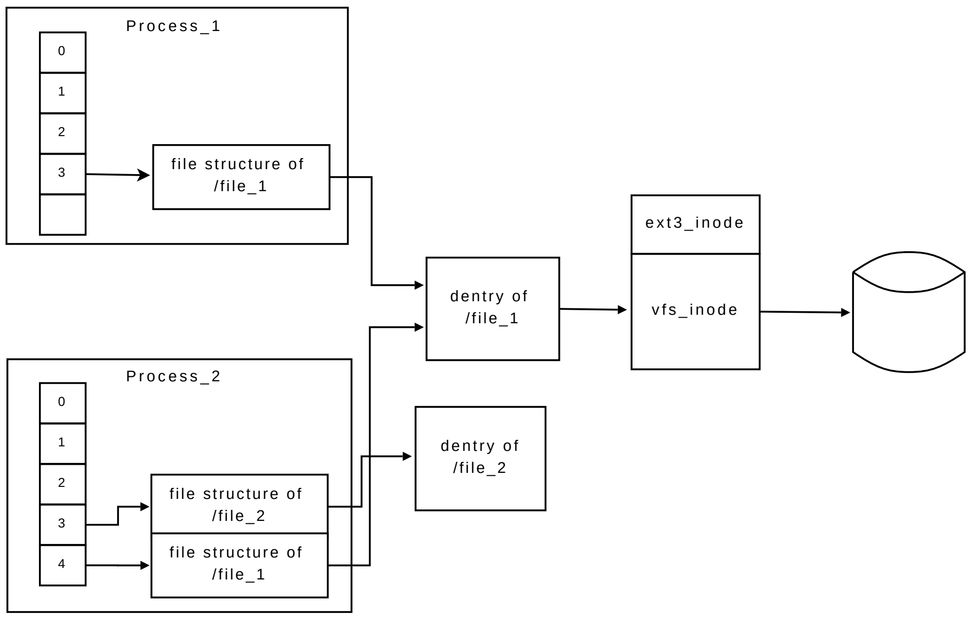


Figure 2: Major Data Structures in VFS

compatibility with Ext2 file system. So the on-disk structures in Ext3 are almost the same as Ext2 file system. Ext3 is a typical block-based file system with fixed inode structure. In the fixed inode structure of Ext3, there are 15 pointers for data block addressing. Of these pointers, 12 are direct pointers, which point to individual data blocks on disk. The 13th pointer is the indirect pointer, which points to a block of data block pointers. The 14th pointer is the double indirect pointer which points to a block of indirect pointers, and the last one is the triple indirect pointer. Therefore, the overhead of data block pointers is almost linear to the file size. For example, in a 32-bit system with block size set to be 4 KB , when the file size S KB is less than 48KB, then the overhead is 15 pointers, or 60 bytes; when S is larger than 48 but less than 4146, then the overhead is $(S/4 + 3)$ pointers, or $(S + 12)$ bytes.

The data block addressing method used in Ext3 is straightforward: based on the desired offset of logic data block number, the physical data block can be accessed by at most four disk read: for first 48 KB data, single disk read is required; for data offset larger than 48 KB but less than 4146 KB , two disk reads are required, first the indirect pointer block, then the specific data block; for data offset larger than 4146 KB but less than 4198450 KB (4 GB + 4 MB + 48 KB), three reads are needed; and for data offset larger than 4198450 KB but less than 4299165746 KB (4 TB + 4 GB + 4 MB + 48 KB), four disk reads are needed.

3.2 ReiserFS

ReiserFS [5] is a general-purpose file system with journaling. It was designed to outperform the Ext2 file system and improve disk space efficiency. As a result, ReiserFS can perform much faster than Ext2 and Ext3 when file size is relatively small, which is a great advantage in some kind of web services.

Compared to Ext3 file system, which also has journaling, ReiserFS has two notable features: balanced trees for file layout management and variable metadata structure/location. The major benefit of B+ tree is its extensibility. There is almost no limit on the numbers of objects in the B+ tree. Hence, the file system using B+ tree for file layout can extend as the tree grows, and the location of objects is no longer fixed.

Different from the inode number used in Ext3 to identify a file or directory, an object (file/directory) in ReiserFS can be referenced by multiple items. Each part of the object has a corresponding item. ReiserFS uses keys to identify items, which consists of the directory ID, the object ID, the offset with the object, and a type. ReiserFS uses these keys to locate items in the B+ tree. Thus, items within the same directory, items for the same file, are implicitly grouped together due to the characteristics of B+ tree. There are four types of items: stat item, directory item, direct item and indirect item. They are assigned a type value in this order. Thus, in the B+ tree, the metadata of an object, the stat item, always comes ahead of the object data.

Files are made up of direct item and/or indirect items. If a file is small enough (less or equal to 4048 bytes), then the whole file data is contained in the direct item. ReiserFS has especially good performance with small files as the meta-data and data are closely placed on disk. No additional seeking is needed if the file size is no larger than 4048 bytes. Otherwise, indirect items are needed and the direct item is used to store the tail part of the file data. Given a 4 KB block size, there can be 1012 pointers[2] to unformatted data blocks per indirect item. Thus, one indirect item counts up to 4048 KB of data with the block size of 4 KB . If more space needed, more indirect items will be allocated for that file. The addressing method in ReiserFS is: first find the right item covering the desired offset, then get data directly if it's in a direct item or follow a pointer to a unformatted data block if it's in an indirect item.

Another notable features of ReiserFS is that ReiserFS adds quite a few bytes to the vfs_inode structure when in memory. This small overhead of inode structure is a direct result of its simple design of data organization. The space efficiency brought by the small overhead of inode structure can be huge with the increasing data set. In section 4, there is analysis of the benefit brought by ReiserFS' small overhead of inode structure.

3.3 JFS

JFS [2] [3] is IBM's Journaled File System which provide direct support of Distributed Computing Environment (DCE). JFS separates the notion of aggregates and filesets, which stand for physical disk space storage pools and logical file namespaces. This separation helps JFS to manage resources in distributed environments. The major goal of JFS is to provide fast file system restart in the event of a system crash. Thus, JFS was designed with focus on the journaling feature.

JFS uses extent-based allocation mechanism to support large volume of disk space. Small files (less or equal to 128 Byte) can be stored in inline storage space. Due to the extent representation of data blocks, small block size is preferred (512B or 1KB) to reduce internal fragments. At the same time, the metadata to address data can be kept at a low level. JFS has a fixed number of of extents, so when more extents are needed, like Ext3, JFS will use indirect extents to accommodate the large data size. B+ tree is also used in JFS, but only for file layout, not file data layout.

JFS has a noticeably large inode structure, which is 512 bytes for on-disk structure and over 560 bytes for in-memory inode structure. Unlike Ext3, in which the location of inodes are fixed, JFS generate inodes dynamically. When more inodes are needed, JFS allocates an inode extent which is 16 KB in size, to store new inodes. The large inode structure makes JFS has a high overhead of metadata if the file number is large while the file sizes are small.

3.4 XFS

XFS [9] is the next generation local file system for SGI's workstation and server. It was designed to run on machines with gigabytes of memory and terabytes of hard disk spaces. So the scalability is a premier design goal of XFS. To achieve high scalability, XFS pervasively uses B+ trees to organize metadata. For example, the directory items, file data extents,

[2](4096-24 byte block header - 24 byte item header)/4 byte = 1012

attributes extents and even the free space extents. Due to the scalability of B+ tree, XFS achieved high scalability and performance [12].

XFS also uses extent as the allocation unit. So for file data, the extent-based representation greatly reduces the overhead of pointers, as used in Ext3. For free spaces, the extent-based management eliminate the necessary of bitmap, thus enables low metadata level when used in large disk pools. The inode structure is also variable as ReiserFS. The fixed part of inode structure is small, and the rest part depends on the numbers of extents that this inode owns. Thanks to the B+ tree, XFS has a small fixed number of extents in inode structure. If more extents needed due to the large file size, a B+ tree is created to organize those extents.

4 Simulation Result

With the trend that files are growing bigger and bigger, the affected part in file metadata is the data block management. Larger space needs more data to maintain fast access to data at any offset. So we'll focus on the data block management part of these file systems. In this section, we built an overhead simulator to simulate the overheads of the four file systems we studied in three scenarios. The first scenario is a single file with various sizes, in which the comparison of the overheads in each file system is obvious. The second scenario is a simulated disk case, in which a typical set of file servers is simulated and we could gain an idea of the overheads of these file systems in practice. The third scenario is an aging file system which evolves in its life time.

4.1 A Single-Dir-Single-File Case

In this part, a single file case will be simulated for all four file systems. We perform this experiment to show a typical overheads relationship with the size of a file.

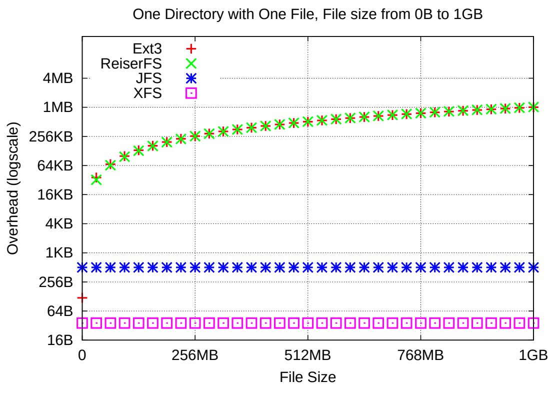
For the two block-based file systems Ext3 and ReiserFS, they should display a similar overheads/file size relationship as at least one pointer is needed for each data block in both of them. The inode size and location in Ext3 are fixed, so Ext3 uses multiple indirect pointers (indirect, double indirect and triple indirect) to organize the large number of data block pointers for huge files. This will place extra overhead in Ext3 file system. However, in ReiserFS, both inode size and location can be variable. Thus, ReiserFS uses only one level of indirect pointers for huge files. This change improves file read/write performance but from Figure 3(a) we can see the overheads of these two file systems are almost the same in general. Due to the advantage of containing all data of small files inside the inode structure, we know that it makes ReiserFS perform exceptionally well in small file operations. From Figure 3(b) we could discover that ReiserFS has less overhead than Ext3 does for small files.

For JFS and XFS, they are two extent-based file systems. In this single file case, the data blocks are composed of one single data block for the directory and the rest for the file data. Therefore, one extent is enough to cover all the data blocks in that file, regardless of the file size. So in both Figure 3(a) and Figure 3(b) we can see that these two file systems have constant small overheads when file size scales.

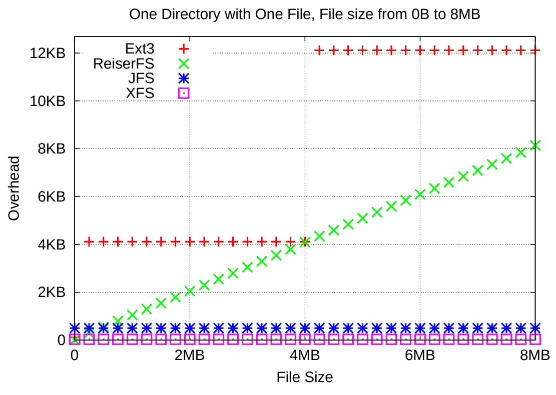
4.2 An Emulated Disk

In this part, a simulation on an emulated disk will be conducted. It is hard to find a perfect typical disk partition to perform the simulation in the real world. So we experiment with an emulated disk which has typical characteristics of real world disks. To meet our simulation requirement, four numbers of a file system should be set: the directory count, directory size distribution, file count and file size distribution.

Based on the work done by Agrawal *et al.*[1], we used the data from year 2004 to create an emulated disk. Those data in [1] were collected from over 70,000 sample file systems, so the statistical numbers can not meet in a single file system. For example, the median number of direct count in 2004 was 4 thousand and median file count was 52 thousand. From this, we can conclude the average files within a directory for a typical disk would be around 13, but the paper said in 2004 the average file count in a directory was 10.2. So we just fix the directory count,



(a) File size from 0B to 1GB



(b) File size from 0B to 8MB

Figure 3: Single-Dir-Single-File Simulation

and generate other data from their distribution.

First, we set the directory count to be 4 thousand, the median number observed in [1]. Because there are two kind of entries in a directory: the files and the subdirectories. Thus we created three distributions to approximate the distributions of files in a directory, the distributions of subdirectory and the file size distribution. In our emulation, only the total number of entries should be considered as the files and subdirectories are treated in the same way when counting the overheads. From [1], we know that the median of file count in a directory is almost 2 and the arithmetic mean is around 10.2. Combined with the shape of Figure 14, we choose Kumaraswamy distribution with $\alpha = 0.45$ and $\beta = 32.02$ to approximate the curve on Figure 14. With these two parameters well chosen, we have our median number equals 2 and arithmetic mean equals 10.2, which are similar to [1]'s data. For subdirectory count, due to its simple distribution curve, we manually set the cumulative percentages from 0 to 10 based on Figure 15. For the file size distribution, which was depicted on Figure 2 and 3 of [1], we know the median is almost 4 KB and mean is 189 KB in 2004. We used log-normal distribution curve to approximate this distribution. Two parameters: $\mu = 8.32$ and $\sigma = 2.78$ are selected based on the median and mean values for the log-normal distribution. Thus, the basic characteristics of file size distribution is well approximated by our log-normal distribution with median at 4096 and arithmetic mean at 189 KB .

The exact overheads of block-based file systems can be simulated with the emulated disk above set. However, it is difficult to measure the overheads in extent-based file systems. Because overheads in extent-based file system depend on the fragmentation of the file system. In the worst case, each extent contains only one block, then the overheads are linear to the number of total data blocks; however, in the best case, the defragmentation utility connects all the data blocks of one file into one single extent, thus the overheads are constant to a single file. Due to the fact that JFS provided defragmentation utilities [3] to manually optimize the data block layout, and XFS does this defragmentation work automatically whenever an allocation is occurred [9], we can safely assume the emulated partition is already defragmented, *i.e.*every file contains only one extent of data. This assumption does not affect the result of block-based file systems, as the aggregation of data blocks in one file will not change the number of pointers needed for that file.

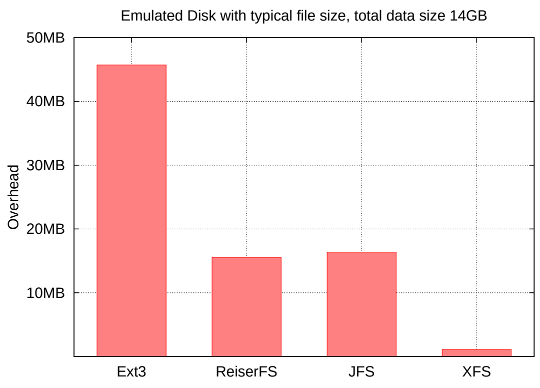


Figure 4: Overheads in an emulated disk approximating typical disks

Figure 4 shows the graph of overheads in these four file systems. This emulated disk has around 14 GB raw data. From the graph, Ext3 has the poorest space efficiency, with overheads 0.33% of the raw data. ReiserFS performs as expected, with overheads roughly 0.10% of the raw data. To our surprise, JFS also incurs high overheads ratio in our simulation and its advantage of extent-based allocation does not show any benefit against ReiserFS. We think this is largely because of the large inode size of JFS, which adds up to an inneglectible amount of overheads. XFS, as expected, has the least overheads due to its variable-sized inode structure and extent-based allocation.

To further study the effect of file size, we created two more emulated disk: one with all file sizes equal 1 KB , the other one with all file sizes equal 10 MB . The structure of the disk is set to be the same emulated disk as stated above. We select 1 KB as the representative of small files and 10 MB for large files because those two are two typical extremes of file systems, though over 100 MB may be a typical file size in some certain file systems, such as GFS [6]. The results are shown in Figure 5.

From Figure 5(a) we see ReiserFS incurs very little overhead as ReiserFS can store this small piece of data in its inline storage space. In this small file case, JFS's big inode structure places a huge overhead for the system. With the file size increasing, JFS's overhead remains the same as we assumed the disk is always defragmented. But the overhead of Ext3 and ReiserFS grows linearly with the size of data set. At the 10 MB level, from Figure 5(b), we noticed that both Ext3 and ReiserFS have higher overhead than JFS, and the gap between Ext3 and ReiserFS is decreasing.

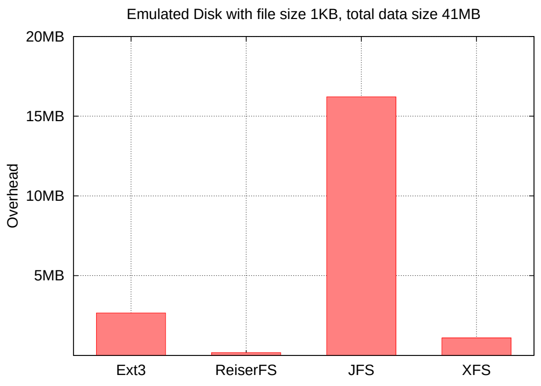
4.3 An Aging File System

The file system is not stable. Its aging will result in a larger data set and fragmented data layout. Smith *et al.*[11] has done some research into the file system aging. They created a file system aging model based on the snapshots of real file systems they collected.

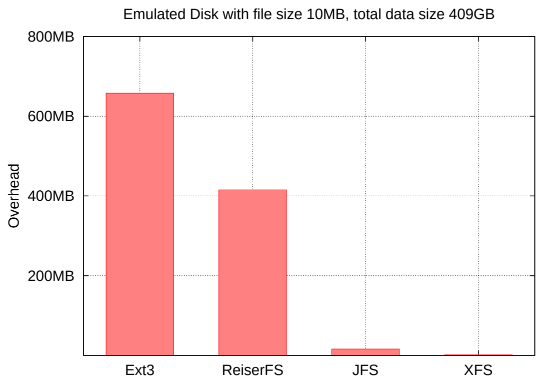
Here we have no such snapshots of read file systems. Thus, the fragmentation model can not be designed. But we can do some simple calculations on the size of data set. If a file system uses block-based allocation, such as Ext3 and ReiserFS, then the ratio of overheads to the size of data set is at least 0.1%. Given today's mainstream hard disk drive capacity 500 GB and the typical fullness ratio of 40%, we get the overhead is roughly 0.2 GB . By Kryder's Law [13] and observation of disk fullness by Agrawal *et al.*[1], we can deduct that the overhead doubles annually. Then for the memory size growth, given today's mainstream memory size 2 GB , and luckily Moore's Law could still hold in the following years, then in less than 10 years, the overhead will outnumber the memory size in block-based file systems.

5 Conclusions

Different file systems have different metadata to organize raw data. They organize data either in a block-based way or an extent-based way. The block-based allocation is a straightforward design to manage raw data as each pointer points to exactly one data block. Thus, the data access method and performance is good as there is a direct mapping of file offset and the corresponding pointer to that data block. The extent-based allocation reduces space overhead in storing so many pointers to the data block. Due to the fact that extent-based allocation eliminates the



(a) Emulated disk with file sizes equal 1 KB



(b) Emulated disk with file sizes equal 10 MB

Figure 5: Emulated disk with file sizes in two extremes

direct mapping of file offset and the corresponding pointer, the random I/O performance may not as good as the block-based counterpart. However, extent-based allocation would increase the sequential I/O performance [7]. What's more, extent-based allocation's reduction of the pointers to address file data in turn reduces the cache usage for large file servers. Thus, more and more recently file systems adopt the extent-based allocation method.

The most amazing findings in our simulation is that extent-based file systems are not always better than block-based file systems. For example, in small file case, well-designed block-based file system ReiserFS has far less overhead than extent-based file system JFS. Extent-based allocation can definitely reduce the overhead if the disk is well defragmented, however, the inode structure and other related data structure should also receive enough attentions to make it outperform traditional block-based allocation in space efficiency.

The extent-based file systems are proved to be scalable in the war of Moore's Law and Kryder's Law. Even the size of data size grows faster than the memory capacity, extent-based file systems can keep the overhead at a quasi-constant level. Especially the XFS with variable-sized inode structure, it can keep the overhead relatively low in both small file case and large file case.

Due to the increasing gap between data set and memory, more and more effort should be placed in the design of clever algorithms in file systems to keep the metadata at a low level. Human-oriented design has been changed to disk-oriented design to lower the metadata overhead and improve file system scalability. We think the extent-based file system with variable-sized inode structure will be a trend in the future file system design.

6 Related Work

Weinstock *et al.*[14] presented an analysis of system's scalability and described the factors to be considered when assessing the potential for system scalability. A definition of scalability is given and a model is presented to stand for scalability. They also analyze the restrict factors and how to solve them.

McVoy *et al.*[7] did research into the benefit of performance gains by using extent-based file systems. They observed the I/O improvement but a few unexpected problems also emerged in extent-based file system, such as the page thrashing problem.

Smith *et al.*[11] found that the file system benchmarks always perform test on a clean disk, which can not fully reflect the practical file systems' performance. Thus, they designed a file system aging model to artificially age a file system, to make it larger and fragmented. They used a series of snapshots of real file systems and interpolate these snapshots by simulating the life cycle of files. Hence, they can create an aged file system by replaying a workload similar to that experienced by a real file system over a certain period.

Prabhakaran *et al.*[10] developed two tools: Semantic Block Analysis (SBA) to help infer file systems' behavior, and Semantic Trace Playback (STP) to simulate disk traffics. Combined with these two tools, they analyzed four journaling file systems and uncovered many strengths and weaknesses of these file systems by semantic level analysis.

7 Acknowledgments

The author would like to express great thanks to Prof. Remzi Arpaci-Dusseau for his guidance throughout the whole project. Without him, the project may get stuck many times. I also want to thank Nitin Agrawal. He helped me to analyze the data of an emulated disk. Besides, I would like to express my thankfulness to Han Zhang and Gwendolyn Stockman, who helped me select the right distributions and pick up right parameters.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th Conference on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[2] S. Best. JFS Overview –How the Journaled File System cuts system restart times to the quick. Technical report, Linux Technology Center, IBM, January 2000.

[3] S. Best and D. Kleikamp. JFS Layout –How the Journaled File System handles the on-disk layout. Technical report, Linux Technology Center, IBM, 2000.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2006.

[5] F. Buchholz. The structure of the Reiser file system. 2006.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[7] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIXfile system. In *Proceedings of the USENIX Winter 1991 Technical Conference*, pages 33–43, Dallas, TX, USA, 21–25 1991.

[8] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[9] B. Naujok. XFS Filesystem Structure. Technical report, Silicon Graphics, Inc., 2006.

[10] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.

[11] K. A. Smith and M. I. Seltzer. File system aging - increasing the relevance of file system benchmarks. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 203–213, New York, NY, USA, 1997. ACM.

[12] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, 22–26 1996.

[13] C. Walter. Kryder's Law. *Scientific American*, pages 32–33, August 2005.

[14] C. B. Weinstock and J. B. Goodenough. On system scalability. Technical report, Carnegie Mellon Software Engineering Institute, March 2006.