

Leveraging Intra-object Locality with EBOFS

Sage A. Weil

University of California, Santa Cruz
sage@cs.ucsc.edu

Abstract

The current and coming generations of large distributed file systems stripe data across large numbers of object-based storage devices (OSDs). Subsequently, individual OSD workloads tend to exhibit no inter-object locality of reference. Small object sizes reduce OSD efficiency due to disk seek overheads. EBOFS, an extent and B+tree based object file system, allows arbitrarily sized objects and preserves intra-object locality of reference by allocating data contiguously on disk, and maintains high levels of contiguity even over the entire lifetime of a disk's file system, allowing OSDs to operate more efficiently and distributed file systems to maximize performance.

1. Introduction

A compelling architecture for large distributed storage systems involves the use of semi-intelligent object-based storage devices (OSDs) and the separation of metadata (namespace and hierarchy) operations from file read and write activity. OSDs are an attractive design choice because they hide the gory details of block allocation and on-disk consistency, exposing a simple interface based on object identifiers and simplifying file system design.

A consequence of this design choice is that the OSD model prevents the careful placement of file data based on directory or temporal locality, tried and true strategies used by general purpose file systems. Instead, files are striped across a large number of OSD devices, providing massively parallel and scalable access to data. The workload for an individual OSD will thus see very little or no locality of reference between different read and write requests; the only form of locality that will remain in the workload is intra-object locality, typically in the form of sequential access to object data. Many distributed file systems stripe file data across a large number of storage servers in order to increase parallel throughput and distribute load using fixed-sized blocks [3, 12, 9, 11, 1, 6]. Consequently the workload observed on individual storage devices appears random and

typically involves a seek for each data access, resulting in relatively poor efficiency.

EBOFS seeks to improve performance by storing arbitrarily sized objects and taking steps to keep their contents written contiguously on disk. A prototype implemented in the Linux 2.6 kernel shows sequential read and write throughput similar to existing general purpose file systems, and fragmentation behavior superior to block-based file systems like ext2. An allocation simulator modeling long-term file system fragmentation over the projected life span of a file system (on the order of 5 years and a few billion write or delete operations) indicates that long-term fragmentation levels are effectively bounded and that allocator effectiveness remains consistent, suggesting that EBOFS can provide superior OSD and greater overall system efficiency and throughput.

2. Background

The Storage Systems Research Center at UC Santa Cruz is currently researching a petabyte-scale (10^{15} byte) storage system designed to handle both general-purpose and scientific computing workloads by exporting a POSIX-compliant interface. This architecture will consist of tens of metadata servers (MDSs), thousands of object-based storage devices (OSDs), and potentially hundreds of thousands of clients. Intelligent OSDs (which will most likely consist of a hard disk, a commodity CPU, and a network interface) simplify file system design by handling block-level allocation internally and presenting a simple object-based interface—file data will be striped across many such objects on many OSDs. This architecture is shared by other distributed file systems like Lustre [2]. Applications of such a system currently include scientific computing environments, the Internet Archive, and large data centers, whose storage demands may well be typical of distributed file systems in a few years time.

2.1. System Architecture

In such a system metadata operations are completely decoupled from read and write operations. A client wishing to open a file will first contact the metadata server (MDS) cluster to obtain a capability (permission) and information allowing it to locate the objects containing file data in the OSD cluster. Subsequent read and write traffic then involves only the client and relevant OSDs without further MDS involvement. Such an arrangement allows maximum scalability for system throughput, since the primary bottleneck limiting I/O transfer rates becomes the network, and no single component within the system.

Each OSD manages its own on-disk data storage and exposes a simple object-based interface. Unlike conventional block interfaces used by commodity drives in a SAN, objects can be variably sized and allocation details are hidden, simplifying file system design. The T10 working group (part of the ANSI standards organization) is developing a standard interface for use with OSDs, with significant support from a hard disk industry looking to add value to their storage product lines.

2.2. OSD Workload

In an OSD based distributed file system, data for individual files will be striped and replicated across a large number of OSDs in order to achieve higher aggregate throughput (through the use of parallel prefetching) and reliability. As such, the objects stored on an individual OSD will exhibit little or no locality of reference; the workload will consist of small pieces of files originating from a large number of hosts. Many large storage systems stripe data across a large number of storage servers in a similar manner, including Swift [3], GPFS [12], Petal [9], RAMA [11], Slice [1], and Zebra [6].

In any such system each unit of allocation must be identified by some sort of identifier (object ID's in our system, block number in past systems). For large files, the list of identifiers for objects storing the content becomes large and cumbersome to deal with. The process of distributing objects across a large cluster of OSDs to balance disk utilization and properly distribute object replicas also becomes difficult. To solve these issues, our system will utilize a deterministic algorithm that generates a sequence of OSDs with the desired properties based on a single input value [8], reducing the problem of storing an object list (analogous to a block map in traditional file systems) to that of storing a single input value. Object IDs can then be based on any value unique to the file (derived from the inode number, for example). Some number of files will share the same distribution in order to simplify replication and recovery operations [15].

The size of objects stored on an individual OSD will depend on the strategy for striping files across OSDs and the overall workload of the system. File system workload studies have indicated that while most files are small, most data is contained by large files, so the OSD workload will be dominated by larger objects, whose actual size may depend on the stripe unit.

3. Related Work

An object-based storage model with independent meta-data management was originally used by Swift [3] and later by NASD [5]. Lustre is an open source file system largely based on NASD's design that is currently under development [2]. Because hardware OSDs are not yet available, current systems use existing general purpose file systems on standard hosts for object storage (Lustre uses ext3 and Linux).

Ext2 and ext3 are general purpose file systems that are somewhat typical in their resemblance to FFS, the Berkeley Fast File System derived from the original Unix file system out of Bell Labs. They allocate disk in blocks and maintain a block list in each inode, utilizing indirect, double-indirect, and triple-indirect blocks as necessary for large files. This approach is generally sufficient for general purpose file systems, but does not scale well to extremely large files or volumes, where huge lists of blocks become cumbersome and inefficient to manage. More recent file systems like XFS [13] and ReiserFS utilize extents (a block start address and length pair) in place of block lists to more succinctly describe contiguous allocations on disk. XFS and Reiser also use Btrees to manage block allocation information in place of simple bitmaps because they are more efficient to manipulate and scale well in both time and space.

Although their prevalence and wide support make the use of general file systems for object storage convenient, we believe a special purpose file system will result in better performance. Unlike traditional file systems, and object storage system has no use for a hierarchical directory structure, requiring only a single flat and homogenous object namespace (which legacy file systems are typically not optimized for). Furthermore, OSD workloads will exhibit little or no locality of reference between objects, making most file systems' attempts at clustering useless.

Feng has built an object file system called OBFS that is designed specifically to handle the kind of workload an OSD is likely to encounter in a large distributed file system [14]. OBFS expects that most objects will be equal to the stripe unit size (on the order of 1MB) and optimizes allocation for such objects to avoid file system fragmentation, reduce seek overhead, and minimize recovery times after a failure. Since most data stored lives in large files, file content will be striped across a large number of such objects,

distributed evenly across many OSDs. Such a distribution allows for extremely high throughput by exploiting the parallelism in the OSD cluster and network. An analogous approach is taken by GPFS, which uses large blocks (256 KB by default) distributed across many disks in a SAN to minimize seek overhead and maximize parallel throughput [12].

From the perspective of an individual OSD, most I/O traffic will be in the form of 1 MB chunks, and will appear completely random. This is because each OSD is handling storage for a random portion of files from random portions of the original file hierarchy. The complete lack of locality between objects means that OBFS can optimize performance based on physical disk parameters (current head position, rotational latency). However, on average, reads are random, and will involve a seek for every request.

4. Design

EBOFS is motivated by the hypothesis that eliminating all locality from OSD workload results in non-ideal aggregate throughput for individual disks and thus aggregate OSD cluster throughput when the system is under load. Instead of limiting object sizes based on the stripe unit, one might stripe a single file’s data across a smaller set of larger objects (as with RAID), exposing the OSD to additional locality of reference within individual (large) objects.

By allowing arbitrarily sized objects, EBOFS can no longer take advantage of a simplified workload and exposes itself to greater potential fragmentation by its more complicated allocation strategies, particularly as the object file system ages over time.

To minimize the impact of this choice, and to maximize file system efficiency and scalability for large disks, EBOFS utilizes extents as the underlying unit of allocation, and balanced trees (B+trees) to manage object free lists and the object lookup table.

4.1. Extents

Extents are an efficient and concise mechanism for representing contiguous sequences of blocks on disk, replacing traditional block lists (literally, lists) with shorter lists of (start, length) tuples. In a system where contiguous allocation is typical, extents are many orders of magnitude more efficient in the average case, and only slightly worse (2x larger) in the pathological worst case of fully non-contiguous allocation. Extents are also more efficient for managing free space on a disk, which in most cases is largely contiguous.

Allocations of larger extents result in lower overhead due to disk seeks (arm positioning and rotational latency), which are necessary for head positioning before starting

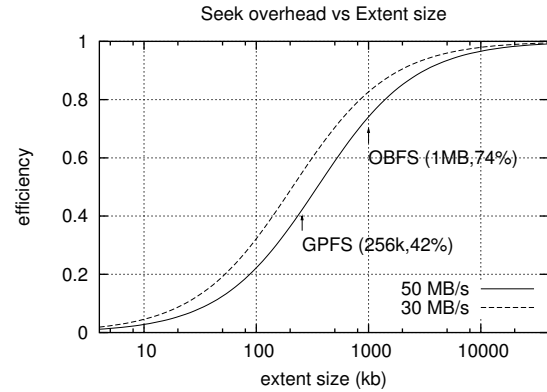


Figure 1. Seek overhead vs extent size, assuming average seek time of 7 ms and a disk transfer rate of 30 or 50 MB/second. Efficiency for 256 KB blocks (GPFS default) and OBFS large blocks (1 MB) are shown for reference.

to read or write but have a negligible effect once sequential access begins. Since OSDs see little or no locality of reference between objects, on average disk seeks for reads (writes may conceivably be written anywhere) result in a penalty (on the order of 7 ms for modern hard disks) before an extent’s data may be transferred at the disks full bandwidth (currently on the order of 30 to 50 MB/second). Figure 1 shows the average seek overhead for various extent (allocation unit) sizes, assuming the disk is able to read the entire allocation unit without interruption (by other requests, etc.). Although disk drive areal densities are expected to continue to increase, average seek times fall much more slowly, which means that the overall efficiency for any given allocation unit size is expected to drop.

4.2. B+trees

B+trees are used extensively in EBOFS to manage the free extent lists and the object lookup table, which translates object ids to locations on disk. B+trees consist of leaf nodes, which contain sorted (key, value) pairs, and index nodes, which map keys indexing child nodes to child node pointers. B+tree insertion and deletion rules ensure that they remain balanced (hence the B), providing fast (log n) lookup, insertion and deletion times.

Individual trees are identified simply by the address of the root node. Space for btree nodes in EBOFS is reserved on disk in “chunks” to enforce some locality, since (unlike objects) tree nodes are accessed/traversed in some sequence. An initial chunk is allocated when the file system is created based on an estimate of how many nodes might be necessary. If the number of free nodes gets low in a

mature file system, additional chunks can be allocated and added to the pool of available nodes. For simplicity, B+tree nodes currently occupy a single logical disk block (4 KB) and hold around 500 (key, value) pairs each.

B+trees are an attractive choice for managing index and list structures because they maintain records in sorted order and scale efficiently in both time and space. The prototype B+tree implementation used for EBOFS maps 32-bit keys to 32-bit values, which is ideal for managing extents, whose (start, length) tuple maps directly onto a (key, value) pair.

4.3. Onodes

An onode (analogous to an inode in general purpose systems) contains object metadata, which include size, a/m/ctime, permissions, and the location of data. For large objects, OBFS took the straightforward approach of using the onode number to describe the actual location of the onode on disk, and then locating it contiguous with large objects' data, not unlike embedded inodes [4]. This allowed a read to proceed with just an object lookup (implemented in OBFS as an in-memory hash), a single seek, and then a sequential read. EBOFS extends this approach to include objects of all sizes. An object lookup B+tree maps object numbers to onode identifiers, which are simply block numbers where onodes are located on disk.

In EBOFS, onode allocation metadata consist of a list of extents containing the file data. Ordinarily, the first extent immediately follows the block containing the onode, although this is not required (it might not be possible for new objects in a pathologically fragmented or nearly full file system, for instance). This results in a storage overhead of one block for every object (in addition to the average one half block wasted by the tail of the file) to store the onode. For very small objects, this strategy might be augmented to colocate object data within the same block. For the prototype, however, the coding effort to make this work was not deemed worthwhile, nor is the performance impact expected to be significant, when most data is consumed by large objects (whose overhead is relatively small).

The current implementation limits the number of extents to what will fit in the onode block. A robust implementation should probably use some sort of indirection (one or more extents describing blocks filled with extents), or more simply a variable length onode.

4.4. Allocation Strategy

EBOFS attempts to allocate objects on disk in large, contiguous and (ideally) singular extents. For small objects, this requires only small extents of free disk space, and is relatively easy. For large objects, large extents are a scarcer commodity, resulting in the fragmentation of large

objects across multiple smaller extents. In such cases, these extents should ideally be close together to minimize seek overhead during sequential reads.

In order to accomodate both types of allocation demands, EBOFS employs a "closest good fit" strategy, whereby the allocator tries to find a free extent approximately the size of that required (or possibly a bit larger) that is closest to the starting point. For subsequent extents beyond the first one allocated to an object, that starting point is the end of the last extent. For new objects, the starting point may be the current disk head position, or not be used at all.

To accomplish this, EBOFS groups free extents into a series of buckets based on their approximate size. Within each bucket, extents are sorted, allowing an extent near a particular position on disk to be found in $O(\log n)$ time. To find an extent of size s , EBOFS will search in the smallest bucket that might contain available extents. If none are found, the larger buckets are searched as well. When an extent is found, the portion of it closest to the desired starting point (either the head, the tail, or some middle portion) is removed from the free list and the remaining portions are reinserted in the appropriate buckets.

How tightly extents in the free list should be grouped into buckets is a critical design question. Grouping them more tightly will result in small "holes" in the allocated disk being filled by small objects, leaving larger extents free for large files and minimizing fragmentation. On the other hand, grouping extents too closely prevents EBOFS from finding extents that are nearby other file fragments, resulting in poor sequential read or write performance when objects inevitably do become fragmented. The effect of the bucket distribution is evaluated in 6.3.4.

5. Prototype

An in-kernel prototype of EBOFS has been implemented in the Linux 2.6 kernel, utilizing the VFS file system layer. The VFS provides an integrated page, inode and dentry cache, allowing me to implement a fully functional prototype in less than 2000 lines of code. In fact, it is worth noting that the code path for most file I/O is nearly identical for ext2, ReiserFS, OBFS, and EBOFS, as they all utilize functionality provided by VFS, differing primarily in the details related to mapping 4 KB file pages to (4 KB logical) disk blocks.

In an actual OSD device, providing a VFS interface (and thus exposing a POSIX file system interface to user space) may not actually be useful, as all I/O will be conducted via a network interface and can be more efficiently handled in-kernel without a context switch and data copying between kernel and user space. For the purposes of evaluating a prototype, however, a VFS-based implementation allows

us to compare EBOFS to traditional file systems and fairly evaluate their relative performance characteristics.

Although the current EBOFS prototype is quite stable, there are a number of items that should be augmented in a robust implementation. These include on-disk consistency and recovery, additional B+tree library generalization, and the maintenance of additional object indices.

5.1. Consistency and Recovery

The current implementation makes no particular attempt to preserve data integrity in the case of a power or system failure. Although B+tree consistency is preserved by using copy-on-write techniques when updating and flushing B+tree nodes, there is currently no integration of that consistency strategy with the allocation code to keep object data consistent. There are three general approaches that can be employed to maintain file system consistency across power failures, with varying guarantees.

The first option is to employ “soft update” techniques [10] to ensure that the on-disk image is internally consistent (or trivially correctable) at all times. This is accomplished by maintaining dependency information between object and metadata updates and controlling the order in which they are written to disk. Such an approach has the advantage of being relatively straightforward to implement in a flat namespace file system with a single set of directory and allocation structures, and has minimal impact on overall file system performance (it’s largest impact being a possible re-ordering of writes). Soft updates (currently used by EBOFS for B+tree updates) do not ensure, however, that the consistent on-disk image contains all updates processed by the OSD before power failure, as some updates will likely have existed in memory and not have been flushed to disk yet. If some (recent) data loss is acceptable, however, soft updates do avoid any requirement for time consuming file system consistency checks on recovery.

Such an approach can be augmented with the use of a log, synchronously written to disk, to ensure that recent file system changes are not lost. This approach is taken by many general-purpose file systems, including ext3, XFS, and ReiserFS. On-disk logs have significant performance disadvantages, however, as they require synchronous writes to a specific location of disk and degrade the performance of the disk scheduler. An alternative is to use a small amount of persistent, non-volatile RAM for the log, as is done by WAFL [7]. This eliminates the need for any synchronous writes, and provides a soft update approach with maximum flexibility when scheduling disk activity. The low cost of NVRAM may make this an attractive possibility for OSD devices seeking to achieve maximum performance.

An alternative strategy to avoid any data loss and maintain consistency is to simply use synchronous writes to ensure that all data reaches disk before writes are acknowledged. Synchronous metadata updates in such a case are performance killers as they require a seek to a known location on disk. Alternatively, the possible locations of new objects can be restricted to a subset of possible locations in the free list that is sufficiently large to have little impact on overall write performance but small enough to scan quickly on recovery. One possibility is to allow the placement of new onodes only in the first block of extents in the free list. This restriction allows a recovery process to scan only the first block of each free extent and be confident that all newly written objects will be found and changes to the object lookup table and free extent list reconstructed. The expense of such a strategy is a less flexible allocation strategy and synchronously updated onodes, which may require an additional seek to update object size and mtime in the onode after writing object data. This is the approach taken by OBFS, whose region-based allocation strategy limits the locations that must be scanned on recovery to a very reasonable number.

5.2. Robust B+trees

The current B+tree library maps 32-bit keys to 32-bit values. This is sufficient for managing extents on a reasonably sized volume (the current maximum volume size is 16 TB when using 4 KB blocks). However, it will not be sufficient for maintaining an object lookup table to map object ids to onode locations for a larger object id namespace, which is expected to be on the order of 128 bits. It also restricts the amount of information that can be indexed to 32-bit values, which is currently sufficient for locating information elsewhere on disk, but does not allow efficient inline storage of other information within the B+tree structure.

A robust implementation should allow variable key and value widths. This can be achieved with minimal effort by specifying key and value sizes on a per-B+tree basis and adjusting the number of records stored in a given tree node (currently a single 4 KB disk block) appropriately.

5.3. Additional Indices

A robust OSD must also support object groups and other metadata indices in order to comply with the emerging T10 working group spec. Although the generalized B+tree library should make the management of such indices relatively straightforward, evaluating any implementation details is outside the scope of this paper.

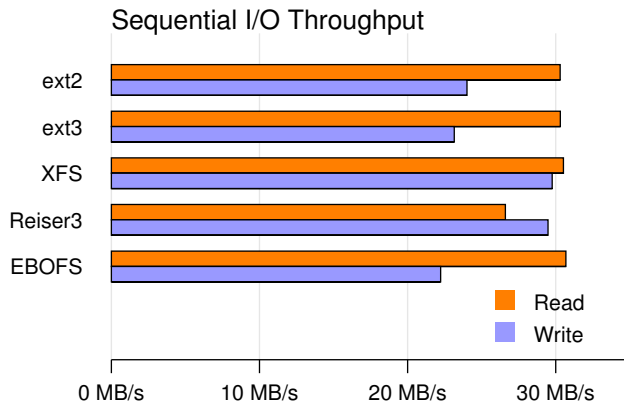


Figure 2. Sequential I/O Performance for ext2/3, ReiserFS 3, XFS, and EBOFS. Performance for ext2/3 and EBOFS is approximately equivalent due to shared use of VFS page cache facilities. XFS makes extensive page and buffer cache modifications, while Reiser3 include modest changes to the write path.

6. Evaluation

I have evaluated a number of aspects of EBOFS’s performance with benchmarks comparing it to existing general purpose file systems, and with a simulator to evaluate long term behavior of the allocation strategy and fragmentation. All benchmarks were performed on a 2.8 GHz Pentium 4 with 1 GB RAM running Linux 2.6.0 on a Maxtor 6Y250P0 250 GB ATA hard disk.

6.1. Buffer Cache Performance

The first benchmark is a measure of raw I/O performance writing to and reading from a single large object or file. The test was performed on an empty file system, which means that no difficult allocation was necessary and the write was fully contiguous. This has the effect of measuring overall performance of the page cache and buffer management code. The large size of the file (40 GB) in relation to the total memory (1 GB) ensures that any caching has minimal affect on total performance. The file systems were also mounted synchronously (`mount -o sync`).

Figure 2 shows relative read and write performance for ext2/3, ReiserFS 3, XFS, and EBOFS. The similar performance for ext2/3 and EBOFS is attributed to the nearly identical code path for both reads and writes, due to similar usage of the VFS page cache and buffer layer. XFS makes extensive modifications to the buffer and page caches to achieve higher throughput. Reiser3 makes more modest

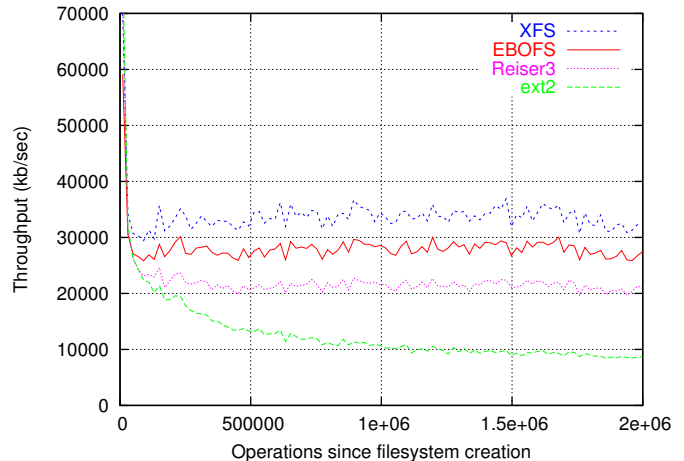


Figure 3. Aggregate read/write performance over time, measured using a workload consisting of 2 million operations (80% reads and 20% writes/deletes). Performance of ext2 degrades significantly as its block-based allocation strategy fails to avoid fragmenting files.

changes to the write code path to make the process of mapping pages to disk blocks for efficient, although I am surprised it made as much of a difference as it did. A cursory review of the code did not reveal any other obvious differences—Reiser3 still uses the VFS page cache and regular buffer management code.

These results indicate that for efficient write performance with large extents, changes to the Linux VFS page cache and buffer management are necessary. These changes and their effect on performance are considered to be independent of file system allocation strategies.

6.2. Aged file systems

Although raw sequential I/O throughput is important, a more critical aspect of overall file system performance is the effectiveness of the allocation strategy in minimizing disk seeks. This is especially important as a file system ages and a less regular layout of surviving data results in new files being written in multiple fragments on disk. In some ways, maintaining an orderly disk layout is antithetical to write efficiency, as it may require longer seeks or even the rearranging of existing data for ideal placement. Since the net impact of fragmentation is also a decrease in performance, a good allocation policy must seek to balance good immediate write performance with intelligent layout decisions to minimize fragmentation over time.

Figure 3 shows the aggregate performance of EBOFS and common general purpose file systems over time, as

measured on a 4 GB partition with a workload of approximately 80% reads/20% writes and a typical distribution of file sizes (mostly small files, most data in large files). The X axis shows time measured in disk operations (2 million in total, representing between 12 to 24 hours of continuous disk activity) and the Y axis shows average read and write throughput (in MB/second). Ext2, which allocates space on a block-by-block basis, showed the worst performance degradation due to fragmentation over time. XFS, EBOFS, and Reiser3 all showed reasonably consistent behavior. These three file systems all utilize extents and B+trees to manage disk allocation, suggesting that policies based on those data structures do a reasonably good job of avoiding excessive fragmentation. The superior overall throughput of XFS is attributed to its improved buffer and page cache performance.

6.3. Allocation Policy

The mixed read/write benchmark suggests that extent based allocation strategies can avoid long-term fragmentation better than ext2's block-based approach. Although a benchmark of two million operations completes in under 24 hours, a busy system may experience similar disk saturation at all times, but be expected to operate efficiently years later. Benchmarking a fully loaded disk to evaluate performance over its expected lifetime requires benchmarks that last for the same period of time. In order to evaluate long-term allocation behavior, I constructed a simulator that utilizes (links) the EBOFS allocation code but performs all allocation operations in memory. This allows analysis of file system fragmentation over periods of time orders of magnitude longer than real benchmarks.

6.3.1. Simulation Workload Modeling file system workload as seen by an individual OSD in a large system is a surprisingly complicated problem. The underlying difficulty is that although it is relatively straightforward to generate a realistic workload that might originate from a particular file system user or collection of users, file system operations are rarely performed atomically at the disk. Depending on the overall system load level, disk speed, host cache behavior, and disk interface characteristics, large read and write operations will be submitted to the disk simultaneously in unpredictable increments. At any point in time, there may be any number of streams of data being read or written to different objects on potentially different parts of the OSD's disk. In a large distributed file system the aggregate OSD workload thus becomes a function of the original client workload (of many thousands of individual users) and the file system's strategy for striping and replicating data across the cluster of OSDs.

Since it's impossible to characterize the workload of a system that doesn't exist yet, the simulator instead approximates a nearly worst case behavior pattern: all writes are submitted to the device in small increments, proceed in parallel, and new write streams continue to arrive at regular intervals. The result is that at any given point in time, it is likely that one or more large writes are in progress to large objects, while writes for small objects continue to arrive in parallel.

The underlying workload being serviced by the OSD is assumed to be typical: most writes are to small objects, most data is written to large objects, and most files are deleted either while they are young or never at all.

6.3.2. Long-term Performance Preliminary simulation results indicate that the level of fragmentation, as measured by the number and sizes of extents allocated, tends to level off after about 3 million write operations (on a 100 GB disk filled to around 80% capacity) and remain relatively constant thereafter. This was somewhat surprising given that the drive only reaches 80% capacity after 2 million operations; it takes very little time for the level of fragmentation to reach an approximate steady state.

Simulations run for more than a billion operations show that a variety of allocation parameters affect the level of performance and fragmentation. In all cases, however, the long-term behavior appears to be bounded. The simulation results focus on a few key measures of fragmentation, as they directly affect observed performance: the average number of extents compromising an object, and the average extent size.

Figure 4 shows the average number of extents per object over more than a billion write operations into the lifetime of the file system. The X axis shows the number of write or delete operations since file system creation, plotting using a log scale. As a point of reference, a drive with 7 ms average seek time can perform about 4.5 billion seeks per year; when you factor in the time spent actually writing data, and that as much as 80% of a typical disk workload will be reads, even a fully saturated disk will likely reach its end of lifetime (3-5 years) before then. The figure shows that higher space utilization results in greater fragmentation (more extents per object on average) but that that value never exceeds even 1.1.

The average size of extents for the same file system over the same period is between 700 and 800 KB. This value is very close to the ideal maximum (the average object size); its efficiency varies inversely with the number extents per file in Figure 4. The average extent size is less than the 1 MB allocation increment for OBFS because most files in the workload are still small; in OBFS the 1 MB blocks are only used for large files. A more useful metric is the average extent size for objects that consume more than one

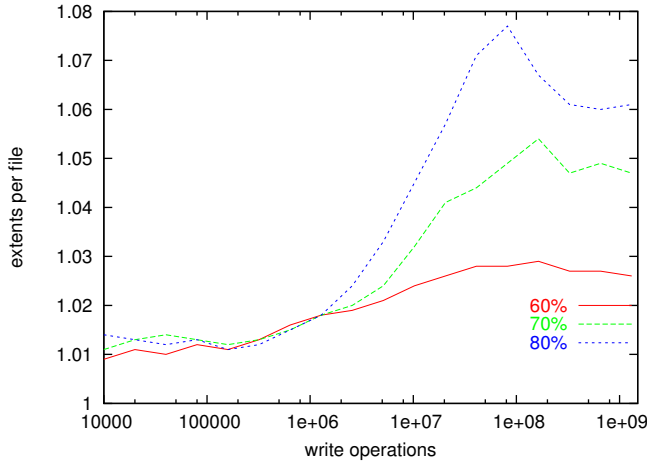


Figure 4. Number of extents per object over time. The time period shown approximates the expected lifetime of a fully saturated disk. Higher space utilization results in higher levels of fragmentation.

extent, as single extent objects are by definition ideally allocated (and average between 300 KB and 400 KB in the simulated workload). Figure 5 shows the average size of extents for multi-extent objects over the same period as Figure 4. Although the average extent size decreases while the file system is in its infancy, extent sizes level off after about 100 million operations and remain steady thereafter. (Remember, the X axis is plotted on a log scale; 100 million operations is about 1% of the interval shown.) Extent sizes at this point remain significantly larger than the fixed-size allocation blocks used in existing distributed file systems like OBFS and GPFS (whose maximum allocation units are 1 MB).

Although an asymptotic lower or upper bound for these values is not immediately apparent, the fragmentation over the lifetime of any individual OSD file system is effectively bounded by the lifetime of the disk itself. That is estimated to be on the order of a billion write operations; however, even 50 times that lifetime is only equivalent to extrapolating the figures about an inch to the right.

These results indicate that the average extent size observed after only a hundred million write operations (not that long after the disk is initially filled) does not decrease significantly with the allocation strategy employed by EBOFS. In order to verify that the distribution of extent sizes is similarly consistent over that period, Figure 6 plots a logarithmic extent size distribution over time. Each line represents a the number of extents of the specified magnitude (in 4 KB blocks). The number of extents of all sizes levels off after a few million operations and that distribution does not change significant over time, with the exception of

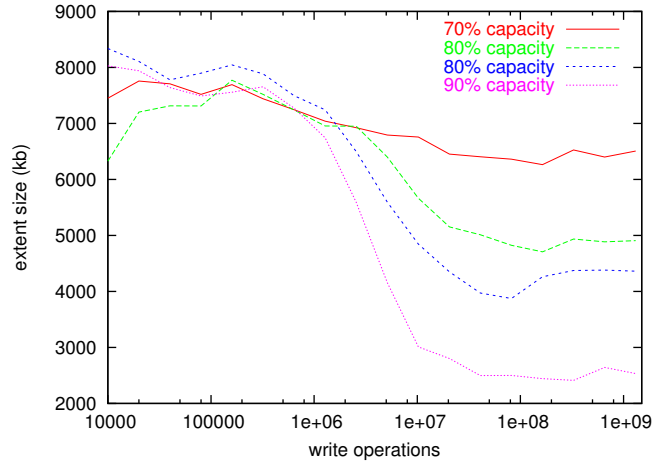


Figure 5. Average extent size of multi-extent objects for different utilizations over time. Higher utilization results in smaller extents.

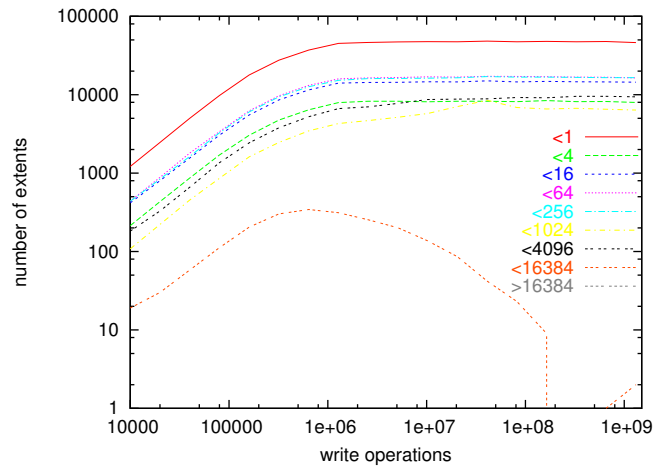


Figure 6. Extent size distribution over time. The character of the extent distribution remains relatively consistent as the extent count for all sizes approaches an upper bound.

particularly large extents (between 1024 and 4096 blocks, or 16 and 64 MB), whose data slowly migrates to slightly smaller extents (256 to 1024 blocks, or 4 to 16 MB) as the file system ages. With that exception, the character of fragmentation and distribution of remains consistent over long periods of time.

6.3.3. Preallocation Although the fragmentation of an EBOFS file system is effectively bounded over the disk lifetime for a variety of algorithm parameters we tested, those values have a significant impact on the observed per-

formance. These parameters include the aggressiveness of the preallocation code and the size of the buckets that form the free extent list.

The allocation simulator assumes a nearly worst case parallel workload in which write operations are submitted to the drive in parallel. The overall performance of the allocator is dependent on the aggressiveness with which space for objects is preallocated. If data were allocated strictly on a need basis, the fragmentation would be heavily affected by the workload as extent sizes would match the writes submitted to the disk.

To avoid this situation, EBOFS allocates extents for objects based on a minimum (required) and maximum extent size. Assuming the end of the object has not been reached, and additional storage is needed, the allocator will select the closest extent it can find that satisfies the maximum size. If such an extent cannot be found, it will allocate multiple smaller extents until the minimum requirement is met. If an extent is found that is larger than the maximum extent size, it will only allocate a portion of it that does not exceed the maximum size.

Generally speaking, the larger the minimum and maximum extent sizes, the larger the extents allocated in the file system. The trade-off is a potential write performance penalty in a parallel workload, where the amount of incoming data may flood the disk cache and require longer seeks to serve data streams in distant areas of disk.

Figure 7 shows the average number of extents per object for a range of different minimum and maximum preallocation values. Figure 8 shows the average extent size of multi-extent objects over the same range. In both cases, the minimum extent size is 1024 blocks (4 MB). As the maximum preallocation size increases, extent sizes increase, but the figures illustrate diminishing returns: in Figure 8 the extent sizes are similarly sized once the maximum preallocation unit reaches 32 MB. This is likely because as extents get larger, they effect a smaller number of objects.

On the other hand, aggressively allocating large extents results in a slight increase in the average seek distance between extents comprising large objects, as seen in Figure 9. Because large extents are typically a more scarce resource than small extents, aggressive preallocation will find fewer, larger and more distant extents. How this will translate into actual performance may depends on the nature of the workload and how much sequential I/O an OSD can ultimately perform before being interrupted with another task, although the relatively small difference between the preallocation settings tested suggests that the effect is not that significant. As overall disk capacity increases, the pool of available extents will expand and have minimal effect on the allocator.

The choice for these values in an actual OSD implementation will likely depend on the desired performance

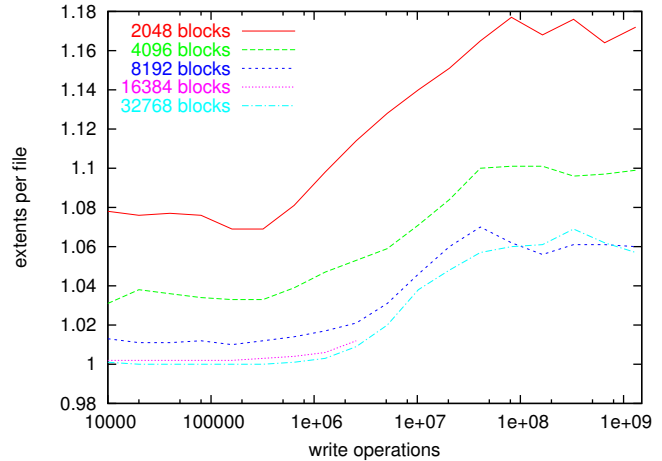


Figure 7. Average extents per object for various preallocation settings. Preallocating larger extents results in lower fragmentation, with diminishing returns.

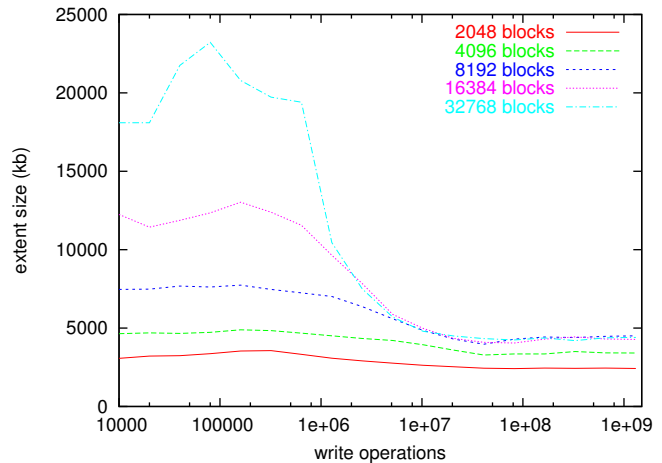


Figure 8. Average extent size of multi-extent objects for various preallocation settings. Preallocating larger extents results in larger extent sizes, with diminished returns beyond 8192 4 KB blocks (32 MB).

parameters of the drive. The values might also be varied dynamically based on the current disk workload. An OSD servicing multiple write streams but no reads might lower the allocation values at the expense of fragmentation in order to increase throughput. Alternatively, an OSD servicing multiple write streams and a varied read workload should probably increase the preallocation values to lower net fragmentation because the reads are already forcing a

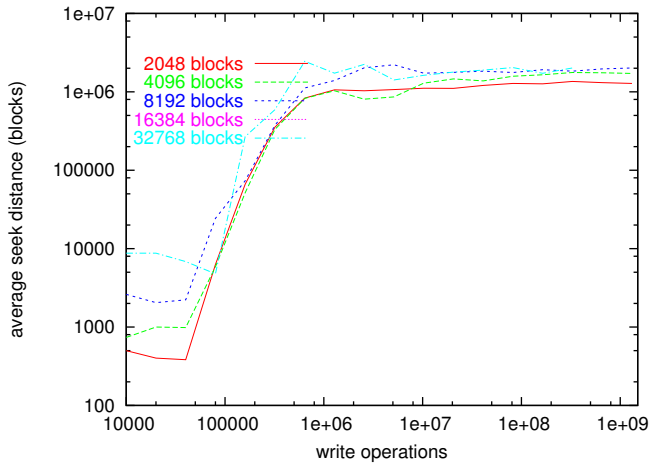


Figure 9. Average seek distance between extents in multi-extent objects for various preallocation settings. Aggressive preallocation results in slightly longer seek times between fewer, larger extents.

disk seek before each write, thereby masking the usual seek penalty for preallocating large extents for multiple writers.

6.3.4. Bucket Size The distribution of free extents across the buckets making up the free list also affect the performance of the allocator. Each bucket has a maximum extent size some multiple (the “bucket size factor”) of the previous bucket’s. Such an arrangement allows the allocator to quickly find a list of extents that are approximately the right order of magnitude in size. Bucket size factors tested range from 2 to 8. A small factor means that the list consulted by the allocator will contain extents very close to the size needed, conserving larger extents by writing small objects in “nooks and crannies”. Conversely, expanding the number of eligible free extents to include a larger range of sizes makes it easier to keep extents comprising a large object near each other.

Figures 10 and 11 indicate that the average extent size and number of extents per object do not change significantly when the bucket size factor is changed. This is due to the competing goals of finding a good fit and aggressively preallocating extents for large objects, both of which reduce overall fragmentation. Average seek distances between extents in multi-extent objects are also similar, as seen in Figure 12, although small bucket factors seem to result in marginally shorter seek distances (I do not have a good explanation for this).

The more compelling factor affecting the choice of bucket factor is the computational advantage of keeping the B+trees storing free extents small. Smaller bucket factors mean that buckets contain extents closer to the desired size,

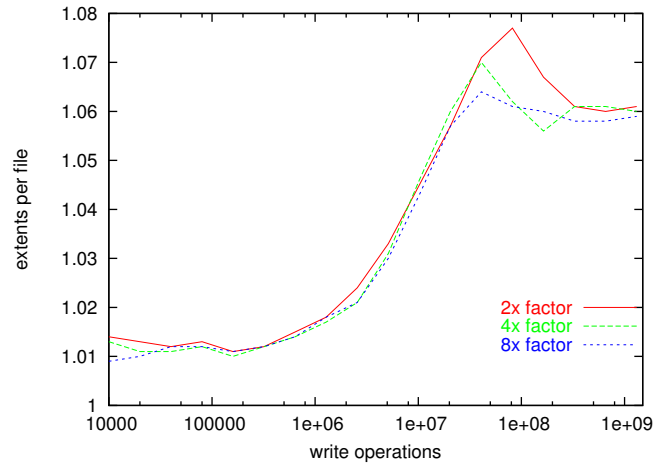


Figure 10. Average number of extents per object for various bucket size factors. Average performance is similar.

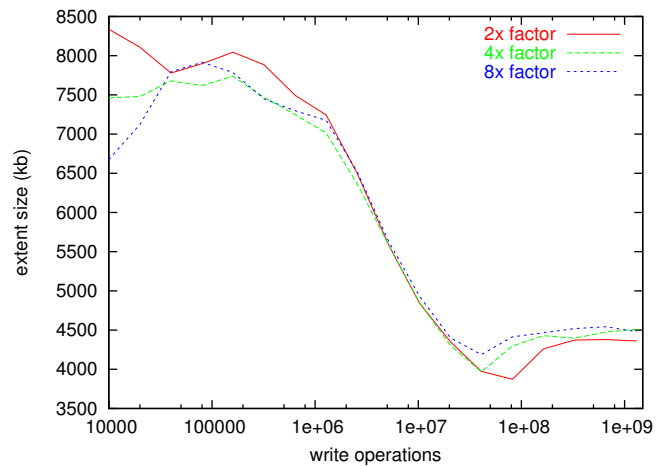


Figure 11. Average extent size for various bucket size factors. Average performance is similar.

minimizing the amount of tree traversal that is necessary to find a suitable extent for allocation.

7. Future Work

One problem with analyzing average extent size and object fragmentation is that actual performance is related to the distribution of extents over different object sizes and their relative prevalence in the OSD workload. Because most files are small and most accesses are to small files, it is probably more important that those objects remain in a single extent. Because they are much easier to allo-

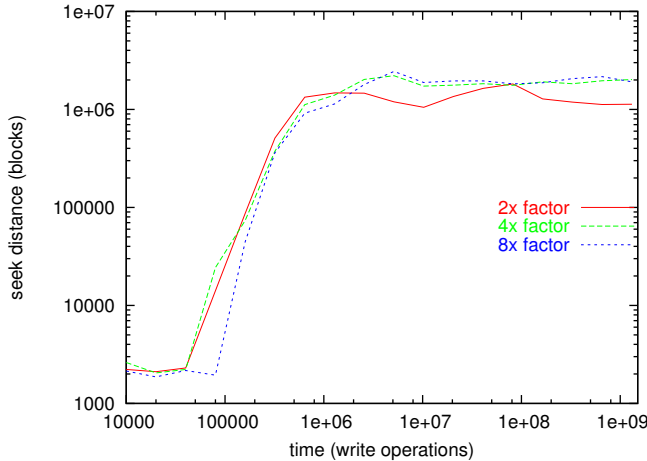


Figure 12. Average seek length between extents in individual objects for different bucket factors. A small bucket factor results in marginally better intra-object locality.

cate contiguously than large objects this is already likely to be the case, but further analysis is necessary to demonstrate that OSD performance follows fragmentation behavior over long periods of time.

The character of file system fragmentation is also highly dependent on other aspects of the OSD workload. Although the simulator tries to generate a nearly worst case, it's behavior is still relatively regular and may not properly reflect the random nature of an OSD in a large distributed environment. Read performance is similarly linked to the character of the workload, and allocation decisions might best depend on the current mixture of read and write requests being serviced by the OSD. A thorough analysis thus requires testing EBOFS and other file systems with an OSD workload based both on a representative client workload and the metadata and object distribution strategy. The effect of allocation parameters like the bucket factor and preallocation on net read and write performance should be evaluated in this context.

Finally, it is still necessary to evaluate the potential advantages of the increased contiguity of objects stored in EBOFS. Existing object storage file systems like OBFS utilize fixed-sizes extents to avoid fragmentation under the assumption that any additional performance can be achieved through greater parallelism. If EBOFS can achieve greater contiguity with minimal probability of degenerate seek performance for small or medium-size objects it is likely to perform better. However, in order to take advantage of larger objects the object data distribution in the storage system must be changed to stripe file data across a small set of large objects, instead of over a large set of small objects. This has the effect of limiting the maximum bandwidth

available for a single reader to the sum of a smaller number of OSD devices (restricting parallelism), but increasing the locality within a given OSD's workload and potentially increasing its efficiency. Depending on the demands of the overall system, this may or may not be appropriate. Striping data across a small number of devices also presents problems for certain kinds of scientific applications whose file access may be staggered regularly in a file, causing it to repeatedly access the same OSD. One possible solution might be to permute the device ordering within each data stripe.

8. Conclusions

Benchmarks indicate that EBOFS's performance is on par with other VFS-based Linux file systems, with the exception of XFS and Reiser3, which rewrite portions of the VFS page and buffer caches. A production OSD device should clearly be mindful of raw I/O performance and keep caching operations and code paths efficient.

The EBOFS allocation simulator indicates that fragmentation is limited over the expected lifetime of the file system, increasingly at an extremely slow rate. This long-term fragmentation does not have a substantially adverse effect on EBOFS's ability to allocate large extents for large files. Such contiguity in object data makes it possible for an OSD to take advantage of sequential access, the only kind of locality likely to exist in an OSD workload. Future work is required to determine the extent to which that contiguity will effect OSD and aggregate system throughput in the context of a large distributed file system.

References

- [1] D. Anderson and J. Chase. Failure-atomic access in the Slice interposed network storage system. *Cluster Computing Journal*, 5(1), 2002.
- [2] P. J. Braam. The Lustre storage architecture, 2002.
- [3] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.
- [4] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17. USENIX Association, Jan. 1997.
- [5] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

- [6] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.
- [7] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.
- [8] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.
- [9] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, Cambridge, MA, 1996.
- [10] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the Fast File System. In *Proceedings of the Freenix Track: 1999 USENIX Annual Technical Conference*, pages 1–18, June 1999.
- [11] E. L. Miller and R. H. Katz. RAMA: A file system for massively-parallel computers. In *Proceedings of the 12th IEEE Symposium on Mass Storage Systems*, pages 163–168, Apr. 1993.
- [12] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244. USENIX, Jan. 2002.
- [13] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 1–14, Jan. 1996.
- [14] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. OBFS: A file system for object-based storage devices. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004. IEEE.
- [15] Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, Apr. 2003.