2009 Sixth IFIP International Conference on Network and Parallel Computing

Design and Performance Evaluation of a Versatile Object-based File System

Qingsong Wei Zhixiang Li Rajesh Vellore Arumugam Kyawt Kyawt Khaing
Data Storage Institute, A*STAR (Agency for Science, Technology and Research), Singapore
{WEI_Qingsong, LI_Zhixiang, Rajesh, Kyawt_Kyawt_Khaing}@dsi.a-star.edu.sg

***Abstract*—The object-based storage system stripes data across large numbers of object-based storage devices (OSDs) to enable parallel data access. Subsequently, the workload presented to the individual OSD will be quite different from that of general purpose file systems. However, many distributed file systems employ general-purpose file systems as their underlying file system. This paper presents a Versatile Object-based File System (referred to as V-OBFS), an extent and B+tree based file system designed for use in OSDs. The V-OBFS implements flexible disk layout with multiple block size and differentiated free space management for both small objects and large objects. In addition, proposed V-OBFS enables fast object search and mapping between object ID and physical location with an efficient object namespace management. Our experiments show that our user-level implementation of V-OBFS can efficiently prevent file system fragmentation and outperforms Linux Ext2 and Ext3 by a factor of two or three.**

Keywords-object-based storage; file system; multiple block size; namespace

I. INTRODUCTION

Rapid advances in general-purpose communication networks have motivated the deployment of inexpensive components to build competitive cluster-based storage solutions to meet the increasing demand of scalable computing. Object-based storage system is being developed to aggregate and virtualize distributed storage resources on hundreds or thousands of commodity OSDs into a single file system image and provide low cost, large capacity and high-performance storage service [1,2]. Very large storage systems are also likely to serve different groups of users and different workloads that have different characteristics. Provision of differentiated storage service to satisfy multiple applications on a single storage platform is very desirable. Realizing this objective requires novel technologies to flexibly meet different requirements of multiple applications.

Object striping is one of the performance enhancing techniques for object-based storage system that has been widely used. Files are divided into multiple objects and distributed across OSD cluster to enable parallel data transfer, thus achieve high aggregate bandwidth. Subsequently, the workload presented to the individual OSD in this system will be quite different from that of general purpose file systems.

In practice, general-purpose file systems are often used as the storage manager inside OSD. For example, Lustre uses the Linux Ext3 file system as its storage manager [8]. File systems such as Ext2 and Ext3 are optimized for general-purpose Unix environments [6,7]. They have following disadvantages that limit their effectiveness in object-based storage system. First, reading or writing large objects costs a relatively small amount more than reading or writing small object. In addition, existing file systems such as Ext2 and Ext3 have been very successful at exploiting disk bandwidth for large object; they have failed to do so for small object activity. Second, existing general-purpose file systems utilize uniform block size (i.e. 4K bytes) for both small objects and large objects, which results in file system fragmentation. Last, many general-purpose file systems such as Ext2 and Ext3 use flat directories in a tree-like hierarchy, which results in relatively poor searching performance for directories of more than a thousand objects.

Because large-scale object-based storage systems may employ thousands of OSDs, even a small inefficiency in the storage manager can result in a significant loss of performance in the overall storage system.

Therefore, in this paper, we address the above mentioned issue by designing a Versatile Object-based File System (referred to as V-OBFS), an extent and B+tree based file system designed to be the storage manager in each OSD. The *V*-OBFS implements differentiated storage policies for different workloads in terms of disk layout and free space allocation, as well as an efficient object namespace management enabling directly access object on-disk data just with object ID. Our experiments results show that our user-level implementation of *V*-OBFS outperforms Linux kernel implementations of Ext2 and Ext3 by a factor of two or three, regardless of the object size. Compared with Linux Ext2 and Ext3, *V*-OBFS has better data layout and more efficiently manages the flat name space exported by OSDs.

The rest of this paper is organized as follows. Section 2 provides relevant background. Detailed design of *V*-OBFS is presented in Section 3. Section 4 gives evaluation results. Related works in the literature is presented in Section 5, and section 6 summarizes the paper with the conclusions.

II. BACKGROUND

In this section, we discuss two basic concepts that are essential to our work: object-based storage, and characteristics of hard disk and file system.

A. *Object-based Storage*

Object-based storage, which leverages the strengths of DAS, NAS, and SAN into single framework, have the potential to provide high system performance, scalability,

978-0-7695-3837-2/09 $26.00 © 2009 IEEE
DOI 10.1109/NPC.2009.20

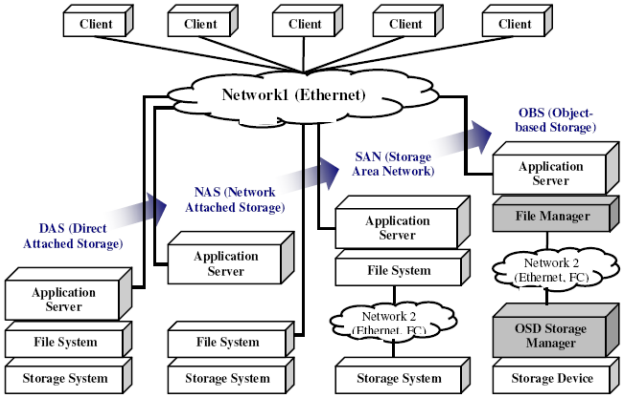


Figure 1. Object-based Storage is a new generation of network storage architecture

security, and easy data sharing across multiple platforms. By decoupling the metadata from data access and enabling the OSD to become more intelligent (e.g., self-management, self-optimization), the object-based storage system is emerging as an attractive upcoming generation network storage architecture [3, 4].

Object-based Storage is being developed to support high performance computing environments which have strong scalability and reliability requirements. To satisfy these requirements, the functionality of traditional file system has been divided into two separate logical components: a file manager and a storage manager. The file manager sitting in Metadata Server (MDS) is in charge of hierarchy management, naming and access control, while the storage manager, offloaded from host operating systems to OSD, handles the low-level storage space management and mapping between objects and physical blocks, as shown in Figure 1. Because hardware OSDs are not yet available, general-purpose file systems are often used as the storage manager in OSD. For example, Lustre uses the Linux Ext3 file system as its storage manager [8].

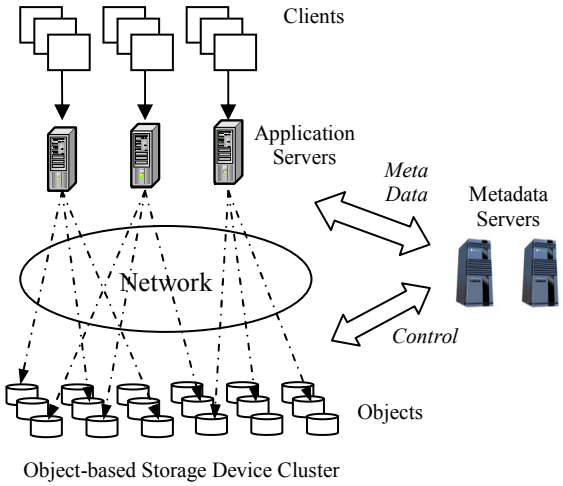


Figure 2. Object-based Storage System Architecture

Object-based storage system consists of three main components: MDS cluster, OSD cluster and Object-based Storage Clients (OSC, namely Application Servers), attached by network, as shown in Figure 2. The MDS presents a global name space for object location and secure access. Objects stored in OSD are of variable-length and can be used to store any types of data, such as database records, images or multimedia data. When a client request for a file, an application server first needs to get access capacity and the metadata from MDS. Once this is completed, object data is transferred directly between the application server and the OSD cluster in a concurrent way.

B. *Characteristics of Hard Disk and File System*

The service time can be broken into two parts: *size-dependent* (data transfer time) and *size-independent* (disk head positioning time, i.e. seek time and rotational delay). Although disk bandwidth is high, its performance is limited by its *size independent* mechanical part.

Then, disk exposes different performance on small object and large object. For small object requests, the *size-independent* part of the disk service time dominates the *size-dependent* part. In contrast, large object can achieve good performance because the *size-dependant* part is much more than *size-independent* part. Small object access then becomes very expensive. As a result, reading or writing large objects costs a relatively small amount more than reading or writing small object [12,13].

The incremental cost of reading or writing several blocks rather than just one is small; For example, a 16KB access takes less than 10%longer than an 8 KB access; A 64KB access takes less than twice as long as an 8KB access. Therefore, transferring larger quantities of useful data in fewer requests will result in a significant performance increase. The first step in allowing large I/O requests to a file is to allocate the file as contiguously as possible. This is because the size of a request to the underlying drives is limited by the range of contiguous blocks in the file being read or written.

In addition, existing file systems inside OSD such as Ext2 and Ext3 have been very successful at exploiting disk bandwidth for large object; they have failed to do so for small object activity. Existing general-purpose file systems utilize uniform block size (i.e. 4K bytes) for both small objects and large objects, which results in file system fragmentation.

These motivate us to propose a versatile object-based file system to efficiently prevent disk fragmentation by allocating objects on disk in contiguous blocks and improve performance for both small objects and large objects.

III. DESIGN OF *V*-OBFS

Enabling the storage manager with enough flexibility is our concerns, which can help OSD to satisfy performance requirement for different workloads. Thus, the consideration

and design of a versatile storage management in terms of disk layout, free space management and object metadata management should be much different from those conventional file systems.

A. Disk Layout

To allow the coexistence of multiple workloads in one storage platform, disk layout should be carefully designed to enable multiple object size, differentiated free space allocation policy and object metadata management for different applications in an OSD.

To do so, disk space is partitioned into regions called Object Groups (OG) based on block size. Groups are located in fixed positions on disk and have uniform sizes. All of the blocks in Group have the same size, but the block sizes in different Groups may be different. Each Group has its own separate data structure to manage free space allocation and object metadata within group. This mechanism not only limit the data structure within a Group and doesn't scale as the system become larger, but also provide convenience to design differentiated storage management policies for different workload. In different Object Group, block size, free space allocation policy and object metadata management may be different according to feature of workload, shown as figure 3.

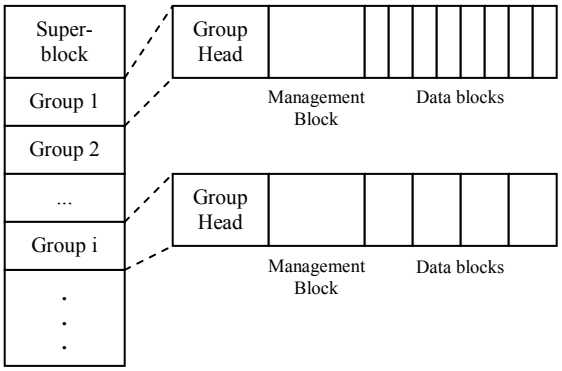


Figure 3. Flexible Disk layout with variable block sizes and space management in group

Rather than pre-allocate full disk into groups such as Ext2, Group allocation in the *V*-OBFS is on-demand and dynamic. A new Group is allocated when there are insufficient free space in current working Groups with same block size. Block size is initialized after group successfully allocated. This on-demand allocation mechanism guarantees the application with large capacity requirement to be allocated relatively larger storage space, resulting in high utilization of storage space.

Now, we support two types of block size (4Kbytes and 64Kbytes). With multiple types of block size supporting, free space can be allocated for object in corresponding Object Group. Thus, small object will be stored in 4K-Groups avoiding file system fragmentation and saving storage, and large object can be laid out contiguously in 64K-Groups. To further satisfy requirements of multiple applications, differentiated free space allocation and object metadata management policies are specified for different workload, which will be discussed in following sections.

Generally, Group consists of three areas: group header, management blocks and object data blocks.

We use extent to represent contiguous sequences of blocks on disk, replacing traditional block lists with shorter lists of (start, length) tuple. In a system where contiguous allocation is typical, extent is many orders of magnitude more efficient in the average case. Extents are also more efficient for managing free space on a disk, which in most cases is largely contiguous. Extent is widely used as allocation unit in more recent file systems like XFS [19].

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. Searching an extent based tree is more efficient than a linear bitmap scan, especially for large, contiguous allocations. Finding an extent of a given size with the B+tree indexed by free extent size, and finding an extent near a given block with the B+tree indexed by extent starting block are both O(log N) operations. B+trees will be used extensively in proposed *V*-OBFS to manage the free extent lists and name space.

To describe object, we introduce Onode, analogous to an inode in general-purpose file systems. Onode contains object metadata, which include size, time, permission and location of data.

B. Differentiated Free Space Management

We try to allocate continuous space to both of the small object and large object. This can improve read performance and decrease disk fragmentation. To do so, this paper takes different free space allocation policy for small objects and large objects.

1) 4K Group

Although poor performance of small I/O for modern disk, it does exist in many applications. Serving small I/O is inevitable for many large-scale storage systems. This paper tries to optimize small object I/O in two aspects: a) reducing disk access times; b) allocating continuous space to small object.

For 4K group, we embed object metadata with object data, and store them together and continuously. The first sector (512 bytes) will always be allocated for Onode and object data is written following the Onode, shown as figure 4. This organization extremely reduces disk access times, so as to size-independent cost and hence improve read performance.

Due to small block size, there are relatively more blocks to be managed. To allocate free space continuously and efficiently, this paper utilizes B+tree to manage the free space allocation. In the Group, the free extents are grouped into several buckets based on approximate size. Within each bucket, free extents are managed by B+tree with key of position. Given a request, it firstly selects a bucket matching

the request size, then searches the B+tree and returns an extent closest to the request position. This method may efficiently avoid allocating a large extent to a small request, so as to avoid fragmentation. In addition, it tries its best to store the object data on the close extents to reduce read seek time.

Object residing in continuous blocks make it possible to directly access object on-disk data by object ID forwarded from clients. Direct object access avoids multi-level indirection among object ID, object metadata, and object on-disk data, which involves several small disk access. This policy extremely reduces the disk access time and boost read performance. It also satisfies best-effort requests desiring low response times.

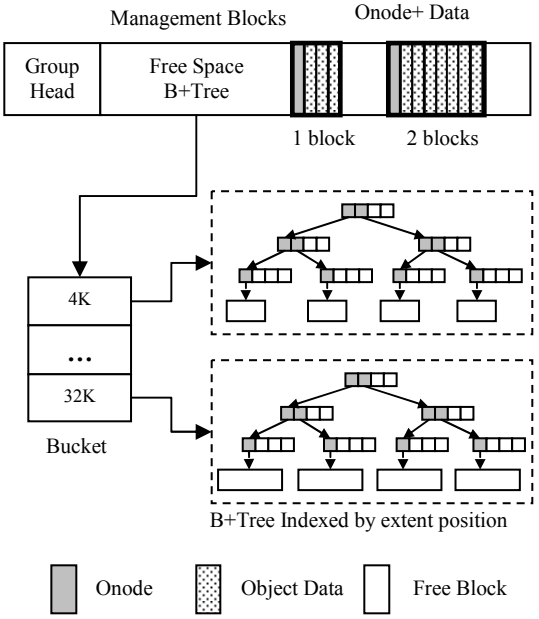


Figure 4. Free space management for 4K Group

2) 64K Group

In contrast, this paper tries to optimize free space management to increase throughput of large object I/O. To make full use of disk bandwidth, we try to allocate continuous blocks to large object.

Like a traditional file system, we pre-allocate Onode on disk same as Ext2, and use bitmap to manage free space for 64K Group, shown as figure 5. The bitmap is used to mark the unused blocks. However, unlike the traditional block-based bitmap, we use extent-based bitmap to represent the free spaces in large group. Due to large block size, there are relatively fewer blocks to be managed. In addition to easy implementation, extent-based bitmap algorithm can achieve good performance in free space search. This not only provides an easy way in maintaining the free space, but also provides an efficient scheme in searching continuous free space for allocation. In addition, this mechanism greatly reduces fragmented objects in large group as compared to other file system such as Ext2 [14].

In extent-based bitmap, *odd* number arrays records the free block address in the group while the *even* number of the bitmap arrays store the size of continuous free space (block numbers). For ease of understanding, the Figure 5 shows how the extents are embedded in the bitmap in managing the free paces in a group. For example, the Onode index 5 has 3 continuous blocks of free spaces (index 5 to index 8 are free spaces).

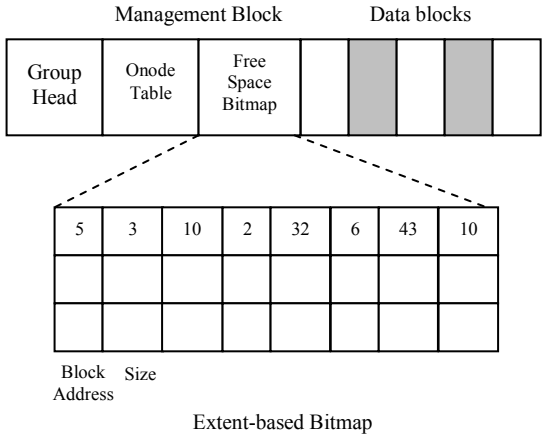


Figure 5. Free space management for 64K Group

C. *Embedding Object Metadata in B+tree Namespace*

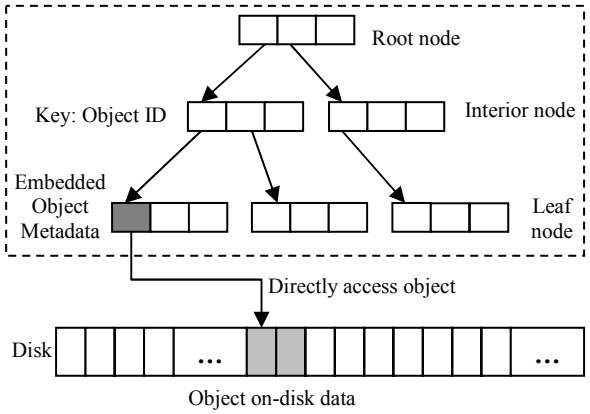
According to the T10 standard, given an object identifier, we need to retrieve the object from the disk, which involves object lookup and location. Because hundreds of thousands of objects might coexist in a single OSD, efficient organization of the flat namespace is a primary requirement.

Many general-purpose file systems such as Ext2 are extremely inefficient in managing large number of files in a directory due to the fact that they do linear search, resulting in O(n) performance on simple directory operations. Linearly seeking a given object ID in the flat namespace is time-intensive and prohibitively expensive.

To avoid this, we design a namespace B+tree for efficient object lookup, as well as a direct location mechanism by embedding object metadata in leaf node of B+tree.

In the B+tree, each valid object has an entry in form of <key, value>, in which index key is object ID and value is node address for root node and interior node. We do optimization on value in leaf node of the B+tree. To improve the performance of object on-disk location, we carefully extended the leaf node of namespace B+tree so that it can accommodate parts of metadata in addition to Onode ID, which can be used for quick location and direct access of object on-disk data by object ID, shown as figure 6 (A).

This organization extremely reduces several levels of indirection between an object ID and the corresponding data, each of which can result in a separate disk access, which occurs in mapping between file name and on-disk inode of traditional file system.



A: Object B+tree Namespace

Flag	Group Class	Group ID	Size
Block address		Onode Index	

B: Structure of Embedded Object Metadata

Figure 6. Object Direct Access by Embedding Object Metadata in Leaf Node of B+tree Namespace

The extended leaf node is designed to be a 128 bit structure, which consists of flag, group ID, Onode ID, Group class (i.e. '00' is for 4K Group and '01' is for 64K Group) and size of the object (based on block size of group), and the object on-disk address. The flag is one bit field to indicate an object is stored on either continuous blocks or fragmented blocks (i.e. '0' denotes continuous object and '1' denotes fragmented object). This is extremely useful in handling read request for continuous objects without having to read the metadata from the on-disk Onode which will involve a separate disk access, and hence decrease the read performance. Figure 6 (B) shows the embedded object metadata in leaf node of name space. With a given object ID, we can quickly seek the corresponding leave node which contains parts of metadata of the object. With the metadata, we may know whether the object is stored on continuous blocks. If it is a continuous object, we can read the object data directly with the staring address of the object and the size of the object. Enabling object direct access from object ID avoids reading object metadata from disk and hence reduces disk access times. This mechanism is extremely useful and achieves high read performance if free space is allocated continuously in most time.

For a fragmented object, because its data resides in multi-locations, we cannot determine the entire data by using above mechanism. It requires loading the metadata from the disk which will involve seek overhead. Nevertheless, our storage system seldom encounters fragmented objects since it utilizes different optimization on continuous free space allocation for small object and large object. In the namespace B+tree, several block addresses also can be maintained for direct access of fragmented object. But it will increase the size of leaf node, which brings high management cost to namespace B+tree. This is an interesting research issues which will be investigated in future.

For efficiency, the namespace B+tree is loaded into main memory and updated asynchronously.

IV. EVALUATION

We compared mean response time and throughput of proposed V-OBFS with Ext2, and Ext3 using different workloads. Our *V*-OBFS runs in the user-level and uses iSCSI commands to read or write object data to the disk, while ext2 and ext3 run in the kernel level which making use of the VFS layer.

A. Setup

We set up our test bed in a 1 GHZ PENTIUM III PC with 2GB RAM, running on Red Hat 9 Linux, Kernel 2.6.12. All the evaluation experiments are conducted on a Seagate ST318437LW SCSI hard disk. Table 1 shows the hard disk specifications. We set group size as 10 GB.

We generated various sets of *random* write and read requests using I/O meter. We compare the random writes and reads instead of the sequential write and read requests because in real life, almost all the data forwarded to the file system are random. For a set of requests, we repeat the experiment 3 times and we take the average out of the 3 results. After that, we took an average value out of the various set of random requests for a particular workload. The results are plotted in Figure 7, Figure 8, Figure 9, and Figure 10.

TABLE I. SPECIFICATION OF HARD DISK CONFIGURATION

Capacity	120GB
Interface	Utral3-SCSI Wide
Rotation Speed	7200 RPM
Single Track Seek(read/write)	0.4ms/0.5ms
Max Full Seek	14.9ms/15.5ms

B. Result

From the Figure 7 and Figure 8, we can see that mean response time of proposed *V*-OBFS, Ext3 and Ext2 increases when request size changes from 4k to 1024k. But the *V*-OBFS outperforms both Ext3 and Ext2 under different request size. When request size is small (i.e. 4k), the *V*-OBFS outperforms Linux Ext2 and Ext3 by a factor of two or three. This is because proposed *V*-OBFS embeds metadata with data to reduce disk access time for small objects.

From the Figure 9 and Figure 10, we can observe that throughput of proposed *V*-OBFS, Ext3 and Ext2 increases when request size changes from 4k to 1024k. In comparison, proposed V-OBFS delivered much better performance than ext2 and ext3 regardless of the request size. It can obviously be seen that the read performance of V-OBFS is improved greatly as compared to ext2 and ext3. This is because the location of an object is calculated instead of requiring the metadata to be loaded from the onode table and perform a

seek to the data area. When request size is large (i.e. 64k), the *V*-OBFS outperforms Linux Ext2 and Ext3 by a factor of two or three. This is because proposed *V*-OBFS can store large object on continuous disk blocks and reduce disk seek time, thus improve throughput for large objects.

The experimental results demonstrate that proposed *V*-OBFS can perform well for both large objects and small objects. With differentiated optimization on object storage management, it can achieve high performance in terms of access latency and throughput for different workload.

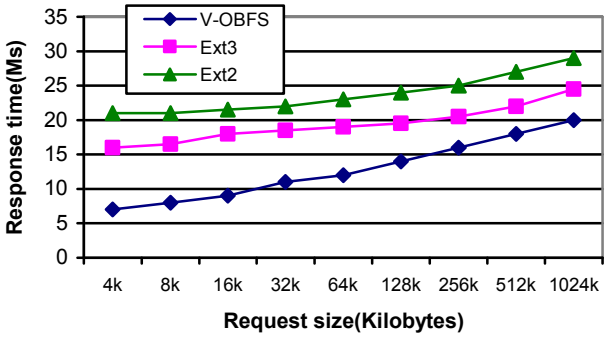


Figure 7. Response time of random read

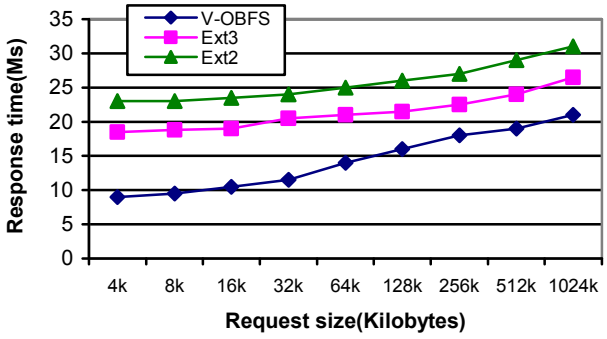


Figure 8. Response time of random write

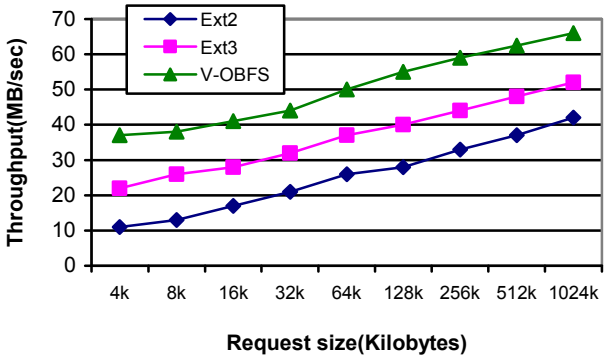


Figure 9. Throughput time of random read

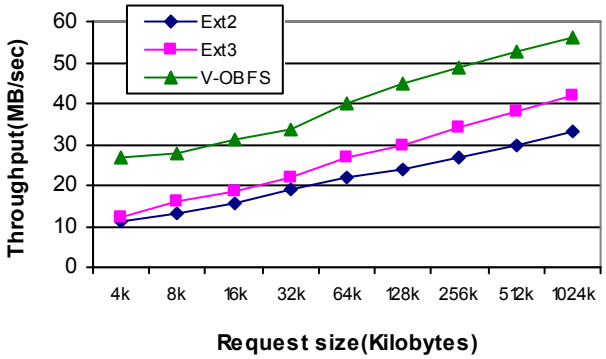


Figure 10. Throughput of random write

V. RELATED WORKS

An object-based storage model with independent metadata management was originally used by the Network Attached Secure Disks (NASD) project [15] at CMU. Object-based storage attracts extensive research and development passion from university and company. In January 2005, ANSI ratified and published T10 OSD SCSI standard [5]. Intel and University of Minnesota [18] implemented an iSCSI-OSD reference to prove the concept of T10 standard. The Storage Systems Research Center at UC Santa Cruz is currently developing a petabyte-scale object-based storage system called Ceph to handle both general-purpose and scientific computing workloads [11]. Although the standard is still under developing, the industry is already implementing systems using the object-based storage technologies. Examples include Lustre file system [8], ActiveScale storage clusters [17], StorageTank [16] and pNFS.

General file systems such as Ext2 [6] and Ext3 [7] are used as storage manager running in OSD in some object-based storage systems such as Lustre [8] and Panasas [17], failing to provide optimal performance for different workloads. They allocate disk in blocks and maintain a block list in each inode, using indirect, double-indirect, and triple-indirect blocks as necessary for large files. This approach is generally sufficient for general purpose file systems, but dose not scale well to extremely large files, where huge lists of blocks become cumbersome and inefficient to manage. More recent file systems like XFS utilize extent in place of block lists to describe contiguous allocation on disk. XFS also uses B+trees to manage block allocation information in place of simple bitmap because they are more efficient to manipulate and scale well in both time and space.

OBFS is an object-based file system developed specifically for OSDs by the University of California, Santa Cruz, Storage System Research Center [9]. OBFS currently use two block size: small blocks, roughly equivalent to the blocks in general purpose file system, and large blocks, equal to the maximum object size (the system strip unit size). OBFS separates the raw disk into regions and all of the blocks in a region have the same size. Blocks are laid out in regions that contain both the object data and onode for the

objects. Free blocks of the appropriate size are allocated sequentially.

EBOFS is another object-based file system upon OBFS [11]. EBOFS is an extent and B+tree based object file system, allow arbitrarily sized objects and preserve intra-object locality of reference by allocating data contiguously on disk, and maintain 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. EBOFS uses Buckets to allocate the closest extents from free space. But it can not keep disk layout compact and prevent disk space fragmentation.

This paper differs from the above mentioned studies in a number of ways. First, proposed *V*-OBF*S* implements flexible disk layout with multiple block size and differentiated free space management for both small objects and large objects. A second major feature of this study is that V-OBFS enables fast object search and mapping between object ID and physical location with an efficient object namespace management.

VI. CONCLUSIONS

This paper presents a Versatile Object-based File System (referred to as V-OBFS), an extent and B+tree based file system designed to be the storage manager on each OSD. The *V*-OBFS implements differentiated storage policies for different workloads in terms of disk layout and free space allocation, as well as an efficient object namespace management enabling directly access object on-disk data just with object ID. Our experiments results show that our user-level implementation of *V*-OBFS outperforms Linux kernel implementations of Ext2 and Ext3 by a factor of two or three, regardless of the object size. Compared with Linux Ext2 and Ext3, *V*-OBFS has better data layout and more efficiently manages the flat name space exported by OSDs.

REFERENCES

[1] M. Mesnier, G.R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, Vol. 41:pp. 84–90, 2003.

[2] Thomas M. Ruwart, OSD: A Tutorial on Object Storage Devices, *Proceedings of the 19th IEEE/10th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST2002)*.

[3] Alain Azagury, Vladimir Dreizin and Michael Factor, Towards an Object Store, *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST2003)*.

[4] Object-based storage: The next wave of storage technology and devices. *Intel White Paper*.

[5] Information technology–SCSI Object-Based Storage Device Commands (OSD), Oct. 2004, http://www.t10.org/ftp/t10/drafts/osd2/osd2r00.pdf.

[6] Remy Card, Theodore Ts'o, Stephen Tweedie, Design and Implementation of the Second Extended File System, *Proceedings of the First Dutch International Symposium on Linux*, ISBN 90 367 0385 9.

[7] Whitepaper: Red Hat's New Journaling File System: ext3, http://www.redhat.com/support/wpapers/redhat/ext3.

[8] Lustre: A Scalable, High-Performance File System, http://www.lustre.org/docs/whitepaper.pdf.

[9] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long, OBFS: A File System for Object-based Storage Device. *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST2004)*.

[10] S. A. Weil. Leveraging intra-object locality with EBOFS. UCSC CMPS-290S Project Report, May 2004.

[11] Sage A. Weil, Scott A. Brandt, Ethan L. Miller and Darrell D. E. Long, Ceph: A Scalable, High-Performance Distributed File System, *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, November 2006.

[12] Wei Qingsong, Lin Wujuan, Yong Khai Leong. Adaptive Replica Management for Large-scale Object-based Storage Devices, *The 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST 2006)*, May 15-18, 2006, College Park, Maryland, USA.

[13] Gregory R G, Kaashoek M F. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, CA, 1997.

[14] Wei-Khing For, Wei-Ya X. Adaptive Extents-Based File System for Object-Based Storage devices, *The 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST 2006)*, May 15-18, 2006, College Park, Maryland, USA.

[15] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. SIGPLAN Not., 33(11):92–103, 1998.

[16] J.Menon, D.A. Pease, R.Rees, L.Duyanovich, and B.Hillsberg, "IBM Storage Tank - A Heterogeneous Scalable SAN File System", IBM Systems Journal, 42(2), 2003.

[17] D. Nagle, D. Serenyi, and A. Matthews, The Panasas ActiveScale Storage Cluster – Delivering Scalable High Bandwidth Storage, Proceedings of the 2004 AGM/IEEE Conference on Supercomputing (SC2004), pg. 53, Nov. 2004.

[18] David Du, Dingshan He, Changjin Hong, Jaehoon Jeong, Vishal Kher, Yongdae Kim, Yingping Lu, Aravindan Raghuveer, Sarah Sharafkandi, Experiences Building an Object-Based Storage System based on the OSD T-10 Standard, *Proceeding of the 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST 2006)*, May 15-18, 2006, Maryland, USA.

[19] A.Sweeney, D.Douncette, W.Hu, C.Anderson, M.Nishimoto, and G.Peck. Scalability in the XFS File System. In *Proceeding of the 1996 USENIX Annual Technical Meeting Conference*, 1996.