Adaptive Extents-Based File System for Object-Based Storage Devices

Wei-Khing For Wei-Ya Xi
Data Storage Institute, Network Storage Technology Division
Email:{FOR_Wei_Khing, XI_Weiya}@dsi.a-star.edu.sg

Abstract

Object-Based Storage Systems is emerging as the storage architecture in the next wave of storage technology. An Object-Based Storage System is rquired to deal with object-based storage devices (OSD) workloads which are generally different from the general purpose file system workloads. This paper describe how we implemented a user-level Object-Based Storage Device File System (OSDFS) which utilizes extents in the traditional bitmap, an adaptive metadata updating scheme, a wasted free space data allocation scheme, extended size of an object mapping (onode ID) to include information of metadata so as to achieve a high performance and high throughput file system for OSD. Our experiments show that OSDFS is able to maintain its continuity in allocating free space for objects up to 98% of disk utilization. In addition, OSDFS is not only outperforms ext2 and ext3 in handling OSD workloads*, but also provides better performance when dealing with* general purpose file system workloads *as compared to conventional file systems.*

1. Introduction

With the recent advances in network technologies such as Gigabit fiber optic network [4] and the proliferation of wireless technologies (i.e WiFi, WiMax [1] (in the near future)), ubiquitous data accessing can be done in a much shorter time as never before. As a result, thousand tonnes of email, e-commerce transactions, multimedia files can be generated and uploaded to a network in a day, putting unprecedentedly pressure on the storage industry to develop a more efficient storage technology in managing and storing network data.

In fact, the storage industry has already moved from the old Direct Attached Storage (DAS) architecture to the Network Attached Storage (NAS) architecture and Storage Area Networks (SAN) architecture in managing network data. However, both of the NAS and SAN architectures have their own limitations [7, 14]. NAS provides file sharing for heterogeneous network platforms with the use of a file server in handling all the metadata (data that describe data), but the throughput is limited by the file server. SAN overcomes NAS's limitations by providing direct access to the storage devices. Nevertheless, SAN compromises not only its security for better performance, but also suffers compatibility drawbacks of different platforms for file sharing. As such, a next generation storage technology - Object-Based Storage System [7, 8, 14] was proposed to overcome the deficiencies in NAS and SAN.

The Object-Based Storage System has the advantages of both the SAN and NAS architectures in providing scalable, block based accessing (high performance), and secure object sharing for heterogeneous Operating System networks. Files are treated as objects and stored in Object-Based Storage Devices (OSD) [2, 6]. Like other general storage systems, the OSD has its own file system - an object-based file system in handling how the objects are being stored. A good file system not only is able to provide high performance and high throughput for the storage system, but it is also able to maintain high utilization of the storage system.

In this paper, we show that through our Object-Based Storage Device File System (OSDFS) which utilizes the well-known extents [13, 16] method, we can provide a high performance and high utilization storage architecture for OSD. In addition, we have also proposed an adaptive metadata updating scheme and extended the size of an onode ID to accommodate metadata in enhancing the file system performance. We described our OSDFS is suitable for implementation in an object-based storage system. We evaluated our OSDFS with various kind of workloads [15, 17] and showed that our OSDFS can provide high throughput and high storage utilization for object-based storage system.

The rest of the paper is structured as follow: in Section 2, we give an overview of the Object-Based Storage System and provide further details on our project. We then describe the related works in Section 3. In Section 4, we describe our OSDFS which includes its architecture, management policies and the novelty of our file system. Our evaluation results in Section 5 show that OSDFS is suitable for various kind of workloads and provides better perfor-

mance as compared to conventional file systems such as ext2 [9] and ext3 [5]. We conclude in Section 6 with some discussions on possible further works.

2. Background

In current network storage technologies, the block-based data accessing approach (i.e SAN) and the file sharing data accessing method (i.e NAS) are the two common ways for a client to retrieve data from storage devices. Block-based data accessing method provides a higher throughput than the file sharing data accessing method since there is no contention with the storage devices. However, it is less secure as compared to the file sharing method since the block-based architecture provides clients direct access to the storage devices. Moreover, unlike file sharing, servers in SAN need to have the same format with one another in order to enable data sharing in a heterogeneous platform, making block access limited in data sharing applications. With the emergence of the Object-Based Storage System, the deficiencies of SAN and NAS can be alleviated.

Figure 1 shows the architecture of an Object-based Storage System. It consists of three main components: the Object Manager(OM), the Object-Based Storage Devices(OSDs) and clients. OM is responsible for managing metadata of OSDs in a storage system. It also provides authentication when a client wants to access the data in an OSD. OSD is a device that stores the clients' data. Each of the OSDs consists of an object-based file system for managing the free spaces and the stored data. When a client needs to access to an OSD, it first needs to obtain authentication from OM. Once this is done, OM sends the metadata (i.e object mapping) of the OSD to the client. Finally, with the metadata and authentication, the client can access the OSD directly. As a result, unlike NAS and SAN, an Object-Based Storage System does not need to compromise either its performance or its security for high throughput and secure data communication between clients and OSDs.

As Object-Based Storage Systems appears likely to be the next wave of storage technologies, our project aims to build an Object-Based Storage System with OSDFS embedded in every OSD. The ultimate objective of our project is to develop a real-time large-scale distributed Object-Based Storage System capable of providing and handling millions or even billions of requests in a single day of different kind of workloads without any compromise in throughput and utilization of the storage system. To achieve this, we need to address two challenging issues in designing OSDFS:

1. *High performance and high utilization file system.* In a storage system, it is important to have a file system that is able to provide high performance and high utilization of the storage system. For this reason, we have adopted the renowned extents(index, size) method in managing the free spaces and storing index in onodes. In addition, we have also utilized an adaptive metadata updating scheme instead of a synchronous metadata updating scheme to achieve a high performance file system (More on this will be discussed in Section 4.6). Moreover, onode ID with embedded metadata is another innovative feature used in enhancing the performance of the file system (See Section 4.7).

2. *Heterogeneous workloads file system.* In order to design a file system that is suitable for various kinds of workloads, we have designed our workload based on [15, 17] (i.e INS, OSD Scientific workloads). As a result, there are five different type of workloads used in benchmarking our OSDFS. In addition, we also provide user-friendly utility for the user to configure the OSDFS in dealing with other workloads. This is extremely useful for the user who knows their workloads in advance so that OSDFS can be tuned to maximize its performance.

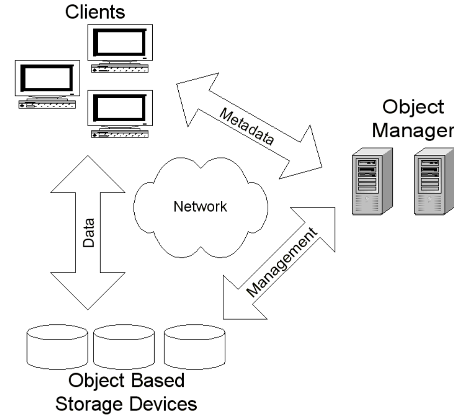

Figure 1. Object Based Storage System Architecture

3. Related Works

Although the Object-Based Storage System is a recent popular research area in network storage technology, there is only a few research focus on the design of the object-based file system as many of the Object-Based Storage System adopted general purpose file system (i.e ext2, ext3) as their object file system. However, the workloads encountered by OSDs are quite different from the general purpose file system workload. As such, designing an efficient object-based file system instead of using a general purpose file system is a crucial step in improving the performance

of the overall large-scale Object-Based Storage System.

As far as we know, only two related works [17, 18] focussed on designing and improving the performance of the object-based file system. In [17], OBFS was designed specially to hande OSD workloads. The workload was categorized into small and large objects and based on this categorization, the OBFS stored the small objects to the small region which consists of a bitmap area and the onode table and the large objects to the large region which utilize embedded onodes in reducing the seeking time of the hard disk. However, the OBFS adopted a synchronous update scheme for writing small workloads which involved a seek time to the onode table. In addition, reading data also involved a seeking distance for the hard disk to read from the onode table and then to the data area. In [18], EBOFS utilized the extents as their allocation unit and the B+ tree as their tree list in maintaining the object free list as well as their object lookup table. To reduce the hard disk's seeking overhead, EBOFS groups the free extents into a series of buckets based on the free extent size. However, how tightly extents in the free list should be grouped is a design question in the design of EBOFS and a poor grouping decision will degrade the performance of EBOFS.

In OSDFS, we proposed a new feature by embedding *extents* in the traditional bitmap area. As such, unlike [18], we avoid the grouping of free extents issue which can affect the storage system performance. On the other hand, to reduce the seeking time to the onode table as encountered in [17], we used an *adaptive metadata updating scheme* based on either the total requests size or the number of requests in updating the data to the onode. For reads, instead of requiring a seek time in reading metadata from the onode and then to the data area, we expanded the onode ID in [17] to include information on the size of the object. Hence, the location of an object can be determined by an equation instead of having to read the metadata from the hard disk.

4. Design of OSDFS

As mention previously, the main objective in designing OSDFS is to provide a high throughput, high disk utilization file system architecture for large-scale distributed storage systems in dealing with large volumes of data in a single day. In addition, our OSDFS needs not only to be designed for the OSD's workload (in the order of 1MB [18]), but to be also suitable for the general purpose file system workloads. To address the above challenges: high throughput, high disk utilization and suitable for OSD workloads as well as general purpose file system workloads, we have included in our OSDFS features such as extent-based bitmap and an adaptive metadata update scheme. Figure 2 shows the architecture of OSDFS. It consists of the following features in providing an efficient and reliable file system for OSD:

1. *An extent-based bitmap and onode.* OSDFS embeds extents in the traditional block oriented bitmap for handling free spaces. In addition, OSDFS also uses extents in storing logical block addresses in the onode to represent the location of data. (See Section Section 4.3 and 4.5 for more details)

2. *An innovation data allocation scheme.* In OSDFS, in order to maintain the high performance of the file system, we have proposed an innovative data allocation scheme which include data allocated to different region based on their wasted disk space; adaptive metadata updating scheme in improving the file system performance while minimizing possible data loss due to unexpected failures such as system crash or power failure; an efficient continuous free space searching by using extents-based bitmap (more on this will be discussed in Section 4.6).

3. *An onode ID with embedded metadata.* Similar to [17], OSDFS also uses Onode ID for object mapping. We extend the size of the Onode ID such that it is able to accommodate the size information of an object. As a result, OSDFS can improve the performance of read requests as the location of an object can be determined based on the metadata embedded in the Onode ID instead of having to read the metadata from Onode table which involve a seeking time (See Section Section 4.7) .

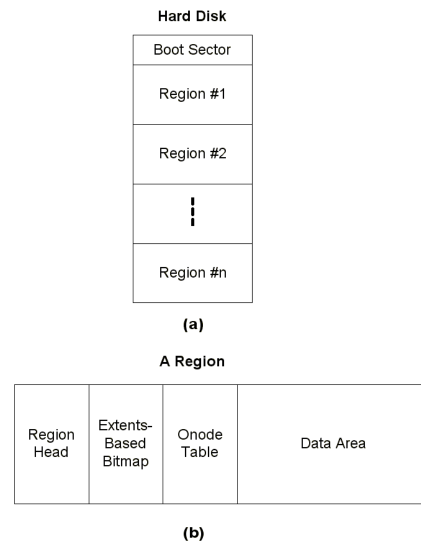

Figure 2. (a) OSDFS Architecture; (b) Structure of a Region in OSDFS

4.1. OSDFS Architecture

In each of the OSD, there is one boot sector, in which the region size, number of regions and pointer to the free region are recorded. We have divided the plain disk into regions of uniform size (i.e 256 MB). This is not only for ease of management as compared to the management of the full plain disk, but it can also provide multi variable sizes of blocks in an OSD (more on this will be discussed in next section).

In each region, there is one region head where information about the region such as region ID, free onodes, starting address of onode table and starting address of data area of eachof the region are stored. Besides, in each region, there is also an extents-based bitmap area for free spaces management as well as the onode table where metadata of the data in that region are recorded. The extents-based bitmap area and the onode table have been designed to accommodate the maximum number of objects in each region.

4.2. User Configurable Variable Size of Block

In OSDFS, each of the regions can be configured into blocks of the same size. However, different regions may have different sizes of blocks (i.e. 4 KB, 512 KB). Provisions is made for the user to be able to configure up to 3 different types of sizes of blocks based on their workloads distribution. This user-friendly feature is useful for the user who knows their workloads in advance so that they can configure the file system to perform efficiently, with higher throughputs and higher utilization especially when the file system encounters fragmentation. In general, the performance of the file system with a smaller size of blocks (i.e 4 KB) will decrease drastically when fragmentation occurs as compared to a file system that consists of a larger size of blocks (i.e 512 KB). This is because file system with a smaller size of blocks requires a larger number of fragmented free spaces to store a complete data, while a file system with a larger size of blocks only requires a few fragmented free spaces to hold this data. Hence, a file system with a smaller size of blocks needs to seek to different locations to store the data. As a result, the performance will be degraded.

In addition, with this user configurable feature, our OSDFS can be used as a general-purpose file system such as ext2, ext3 when dealing with small sizes of files while still maintaining the high utilization. In this paper, we are not focusing on the effect of the number of variable sizes of blocks on the performance of the file system when dealing with different kind of workloads. This is an interesting research issues which will be investigated in future.

4.3. Onodes

In OSDFS, the metadata of each of the files are stored in an onode table. The number of onodes and blocks are the same in each region. An onode consists of the size of a file on disk, the file size, and the o_block array where locations of the data are stored. Each o_block can store up to 110 data locations in the format of extents(lbn, size in blocks). For continuous data, only one array of o_block is needed to store the data location. More than one array is only needed in an o_block when storing the data locations of fragmented files. The o_block makes use of the ext2 direct and indirect pointer to store the data locations if the data locations of a fragmented file exceed the number of arrays in an o_block. In OSDFS, one onode is designed to be 512 Bytes.

4.4. OnodeID

The Onode ID is an identifier for a particular onode. It is used in mapping the object ID forwarded from the clients. Onode IDs are maintained in a tree list and uses in determining the location of a stored data. In OSDFS, the Onode ID is designed to be a 64-bit data. It consists of Region ID, Onode Index, Type of Region (i.e 4 KB or 512 KB) and size of the objects. We embed the metadata of an object such as the types of region that the object's data reside and the size of the objects. This is extremely useful in handling read requests without having to read the metadata from the onode table which will involve a seek delay, and hence decrease the performance of the file system. Figure 3 shows the structure of an Onode ID.

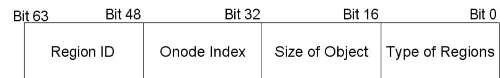

Figure 3. Structure of Onode ID with Embedded-Metadata

4.5. Extents-Based Bitmap

Like a traditional file system, OSDFS also has a bitmap area in each region. The bitmap area is used to mark the unused block. However, unlike the traditional bitmap, we use extents(index, size) to represent the free spaces in each region. This not only provides an easy way in maintaining the free spaces in a hard disk, but also provides an efficient scheme in searching continuous free space to allocate data. In addition, fragmented files in OSDFS are greatly reduced as compared to other file systems such as ext2. Tests show that OSDFS only encounters fragmentation when the disk

reaches above 95% of utilization (write data to a plain disk until 95% of the disk has been utilized).

In a bitmap, *odd* number arrays always record the free onode index in the region while the *even* number of the bitmap arrays store the continuous free spaces. For ease of understanding, Figure 4 shows how the extents(index, length) are embedded in the bitmap in managing the free spaces in a region. The onode index 30 has 5 continuous blocks of free spaces (index 30 to index 34 are the free spaces).

Onode Index	Length	30	5		

Figure 4. Extents-Based Bitmap

Figure 5 shows the bitmap operation when a *WRITE* request is forwarded to OSDFS. OSDFS scans the bitmap array to find the exact match of the continuous free space required by the object (See Figure 5(a)). If there is no exact match between the continuous free spaces and the data size, OSDFS will allocate the data to the area that has the larger continuous free spaces than the data as shown in Figure 5(b).

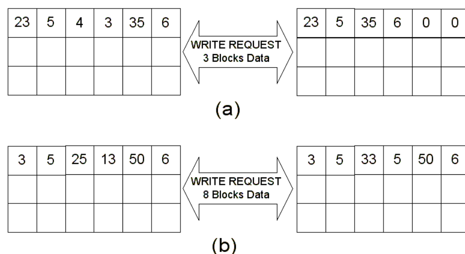

Figure 5. Bitmap Operation for *WRITE* Request

When dealing with a *DELETE* request (Figure 6), OSDFS first scans the bitmap area to check if any of the arrays in the bitmap area can be merged to form a continuous free spaces (See Figure 6(a)). If there there cannot be done, OSDFS will allocate the free spaces to the unused bitmap array as shown in Figure 6(b).

4.6. Data Allocation Strategies

A good allocation strategy and file system structure are the key components in designing a high throughput storage system. OSDFS consists of the following strategies in allocating data to provide a high throughput storage system: (1) data allocation based on wasted disk space; (2) adaptive schemes for metadata updating; (3) extents-based bitmap for continuous free space searching.

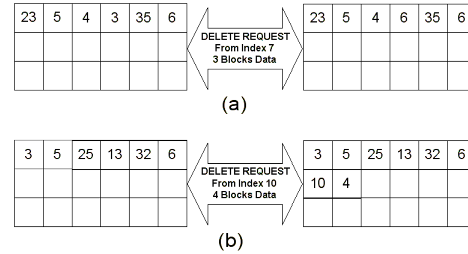

Figure 6. Bitmap Operation for Delete request

4.6.1. Data Allocation Based on Wasted Disk Space In OSDFS, data are allocated to different regions based on their *wasted disk space*. For example: consider a disk with two different types of regions, 4 KB and 512 KB. When a request requires a size larger than 508 KB but smaller than 512 KB, OSDFS will allocate this data to the 512 KB region instead of the 4 KB region. This is because 128 blocks will be required if the 4 KB region is chosen and the wasted free space will be 4 KB. If we allocate this data to the 512 KB region, the wasted memory is also 4 KB. As a result, allocation of the data to the 512 KB region is preferred as only one block is involved in the 512 KB region as compared to 128 continuous blocks in the 4 KB region. Moreover, allocation of data to the larger block-size region will provide a higher throughput as compared to that for the smaller block-size region when the file system encounters fragmentation.

4.6.2. Adaptive Scheme for Metadata Updating In order to enhance the performance of the file system while minimizing the possible loss of data in the event of an unexpected system crash or power failure, OSDFS uses an adaptive metadata updating scheme in writing metadata to the onode. OSDFS updates the metadata to the onode based on either the total size of the files it has encountered or the number of write requests it has completed in a type of region but the metadata are being buffered. If the number of requests it has encountered exceeds a preset counter, it will update all the previous buffered metadata in one time. For example, if the preset counter is set to 10 requests, OSDFS will update the previous 9 requests' metadata and the current request's metadata when OSDFS is dealing with the 10th request. On the other hand, if the total size of the files exceeds a certain threshold (i.e 100 MB), OSDFS will update the onodes which reside in its buffer even though the number of requests is less than the preset counter. This adaptive scheme provides a higher throughput than the synchronous update scheme where the file system has to update the metadata (in an onode table) while writing the data

to the hard disk. In addition, we have not adopted embedded onodes [10] as our OSDFS architecture. This is because when the data is corrupted or the storage system crashes, we can recover back the data since we know the location of the onode table.

4.6.3. Continuous Free Space Searching Using Extents As mention previously, OSDFS uses a traditional bitmap in managing free spaces. However, we utilize an extent-based bitmap instead of the traditional block-based bitmap. This is to ensure ease of searching of continuous free spaces for the incoming requests. When a request is forwarded to the file system, the file system decide which region is suitable for the incoming request based on the above wasted space allocation strategy. After the region has been selected, a check is made whether there is such an initialized region. If OSDFS does not find such a region, the file system will initialize one of the free regions to allocate the data. If there is such a region, the file system will first look for the continuous free spaces from the extent-based bitmap to allocate the data. If the continuous free spaces are not large enough to contain the data, the file system will create a new region to store the data. When there is a lack of uninitialized free region, the file system will allocate the data to the old initialized region and search for the continuous free spaces to allocate the data. In OSDFS, fragmentation occurs when the disk is almost full. The overall flow of write request is shown in Figure 7.

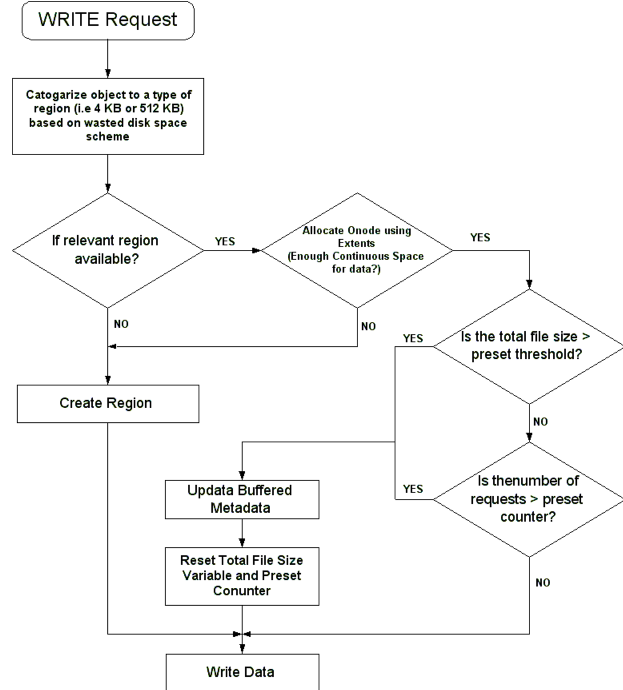

Figure 7. Write Operation with (a)*Data Allocation Based on Wasted Disk Space Scheme***; (b)***Adaptive Scheme for Metadata Updating***; (c)** *Continuous Free Space Searching Using Extents*

4.7. Data Searching Using Embedded-Metadata Onode ID

Data searching is another component that affects the performance of the file system. Because most of the read requests are random, it is impossible to predict the location of the next data. As such, minimizing the seek distance become the main design criteria in providing a high performance file system. In OSDFS, we minimize the seeking distance when dealing with read requests by using Onode ID with embedded metadata. In [17], Onode ID are used in mapping the object ID and maintained in a hash list. However, the read request will require one seek distance where the file system needs to read the metadata from the onode table and seek back to the data area. To improve the file system performance, we have redesigned the Onode ID so that it can accommodate the metadata such as the region type as well as the size of the files. As a result, the location of a file can be determined using Equation 1. By redesigning the Onode ID which includes metadata, OSDFS can avoid reading metadata from the onode and hence improve the read performance.

$$DataLBN = RegData + OIndex * SizeBlk / SecSize \quad (1)$$

where *DataLBN* is the starting address of the file, *RegData* is the staring address of the data area in a region (this value is determined when the region is initialized), *OIndex* is determined from the Onode ID (bit 32 to bit 47) (See Fiure 3), *SizeBlk* is the number of blocks that can be supported in a region (it is depends on the region type - Onode ID bit 0 to bit 15), *SecSize* is the size of one sector which is 512 Bytes.

From the Onode ID, we have the information about the size of an object. Hence, we can read the data with the information of the staring address of the object and the size of the object. However, this method is only applicable for a continuous object. For a fragmented object, because its data resides in multi-locations, we cannot determine the entire data by using the above equation. OSDFS requires loading the metadata from the onode table which will involve a seeking distance to onode table when dealing with fragmented files. Nevertheless, our file system seldom encounter fragmented objects since it utilizes the extent in allocating continuous free spaces.

Figure 8 shows the data searching of a continuous object while Figure 9 shows the data searching of a fragemented object which will involve a seeking distance for the hard disk.

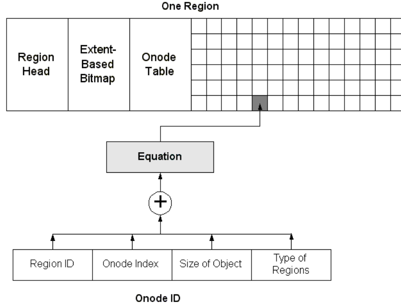

Figure 8. Data Searching of a Continuous Object

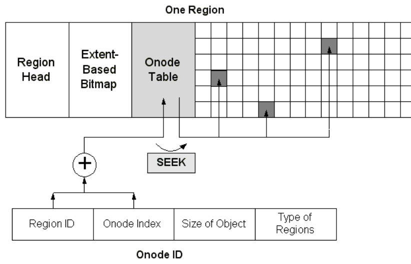

Figure 9. Data Searching of a Fragemented Object

5. Evaluation and Results

OSDFS was evaluated using different kinds of workloads such as INS, RES, WEB, NT based on [15] and Scientific Workload based on[17]. We have also compared the performance of our OSDFS with Linux ext2 and ext3. Our OSDFS runs in the user-level and uses SCSI commands to read or write data directly to the disk, while ext2 and ext3 run in the kernel level which making use of the VFS layer. We inject the workloads to the ext2 and ext3 file system using I/O meter [3]. In order to have fair comparison, we have mounted the drive using *-o sync* parameters, which is similar to [17] so that the data can write synchronously to the disk.

5.1. Experimental Test bed

We set up our test bed in a 1 GHZ PENTIUM III PC with 512 MB of RAM, running on Red Hat 9 Linux, Kernel 2.4.20. All the evaluation experiments are conducted on a Seagate ST318437LW SCSI hard disk. Table 1 shows the hard disk specifications. We have divided the plain disk into 3 GB partitions. For five of the workloads, we generated various sets of *random* write and read requests. The disk is filled up with the write requests until it is full. All the write requests are done synchronously since we mounted the disk to use synchronous I/O. After that, we read all the data we have written to the disk. 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.

Table 1. Specifications of Seagate ST318437LW SCSI Hard Disk

Formatted Capacity	18.4 GB
Head	2
Interface	Ultra3-SCSI Wide
Rotational Speed	7200 RPM
Single Track Seek (read/write)	0.4ms/0.5ms
Max Full Seek (read/write)	14.9ms/15.5ms

5.2. Results

We compare the throughput of Linux ext2 and ext3 versus our OSDFS for random writes and reads using 5 different types of workloads. 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. Figure 10 shows the throughput of write operations while Figure 11 shows the throughput of read operations of ext2, ext3 and different variable sizes of blocks of OSDFS which include 4 KB region, 4 KB and 512 KB regions, and 4 KB, 256 KB and 1 MB regions. In comparison, OSDFS delivered much better performance than ext2 and ext3 regardless of the workloads encountered. It can obviously be seen that the read performance of OSDFS 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. Even with the different variable sizes of blocks, OSDFS still can deliver the same throughput for the storage system. This is because OSDFS utilizes extents in allocating free spaces for the data.

In this paper, we do not compare the effects of different settings of the variable size of blocks to the performance of the storage system. In general, a single region will deliver slightly better write and read performance than multiple regions when the file system can maintain the continuity of its objects. This is because we utilize extents to represent the free spaces and allocate the data to these free spaces. For example, 16 blocks of 4 KB is equivalent to 1 block of 64 KB which is equal to 128 of 512 Bytes sectors. However, the performance of the file system will be greatly reduced for a single region when the file system encounters fragmentation.

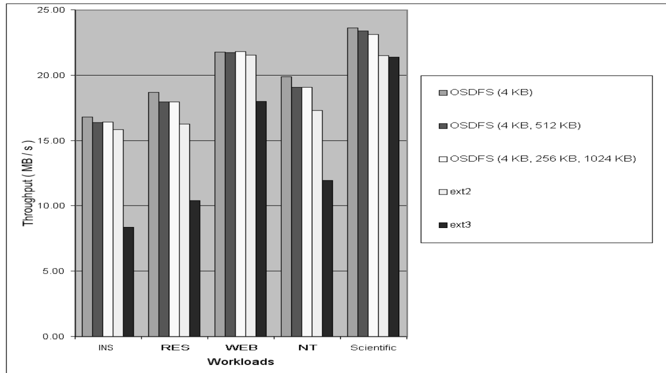

Figure 10. Performance of WRITE Request dealing with different kinds of workloads

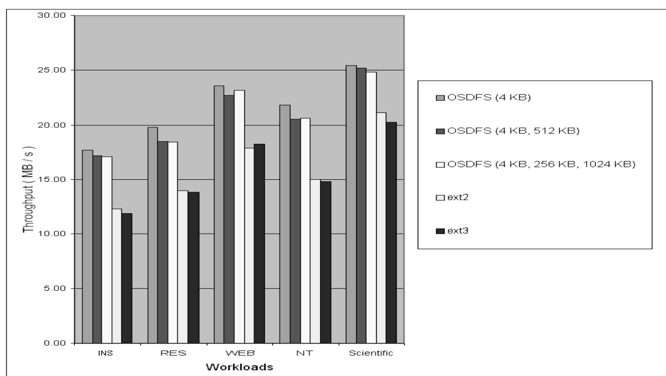

Figure 11. Performance of READ Request dealing with different kinds of workloads

6. Conclusion and Future Work

The primary objective of designing an object-based file system is to provide a high performance, high throughput and high utilization storage file system to the object-based storage devices. In this paper, we presented the implementation of OSDFS in the user-level using extents in the traditional bitmap for management and allocation of continuous free spaces in order to minimize the fragmentation of a file. In addition, we proposed an innovative data allocation strategy which includes data allocation based on wasted disk space and an adaptive metadata updating scheme. We also embed the metadata of an object into the Onode ID to enhance the file system performance. We have shown that OSDFS not only can provide better performance as compared to ext2 and ext3 when dealing with the OSD workload (i.e Scientific workload), but also outperforms ext2 and ext3 when dealing with general purpose workloads such as INS. OSDFS is suitable in supporting various kind of workloads due to its multi variable size of block regions, improved free spaces management by using extents in the traditional bitmap, improved data allocation scheme, an adaptive metadata updating scheme and data searching scheme.

In order to evaluate OSDFS more completely which include scheduler and cache buffer, we will implement our OSDFS in the kernel level by making use of VFS. In addition, investigating the effect of multiple variable sizes of blocks is another main research focus for our project. We will also be extending our work to the kernel level in order to support various applications such as content-sensitive or context-sensitive file system[11, 12]. For example, the file system must allocate continuous free spaces in large sizes of block such as 10 MB region for storing video data.

References

[1] http://www.ieee802.org/16/.
[2] http://www.intel.com/technology/computing/storage/osd/.
[3] http://www.iometer.org/.
[4] http://www.nlr.net/architecture.html.
[5] http://www.redhat.com/support/wpapers/redhat/ext3/.
[6] http://www.snia.org/tech_activities/workgroups/osd/.
[7] Object-based storage: The next wave of storage technology and devices. *Intel White Paper*.
[8] www.t10.org.
[9] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. *In Proceedings of the First Dutch International Symposium on Linux*, 1994.
[10] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk banthwidth for small files. *In Proceedings of the 1997 USENIX Technical Conference*, pages pp. 1–17, 1997.
[11] H. C. K. and C. R. H. An application of a context-aware file system. *Pervasive Ubiquitous Computing*, pages pp. 339–352, 2003.
[12] S. Khungar and J. Riekki. A context based storage system for mobile computing applications. *In Proceedings of Second European Symposium on Ambient Intelligence*, pages pp. 55–58, 2004.
[13] L. M. McVoy and S. R. Kleiman. Extent-like performance from a unix file system. *In Proceedings of the 1991 USENIX Technical Conference*, pages pp. 33–44, 1991.
[14] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, Vol. 41:pp. 84–90, 2003.
[15] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. *In Proceedings of the 2000 USENIX Technical Conference*, pages pp. 41–54, 2000.
[16] 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 Technical Conference*, pages pp. 1–14, 1996.
[17] 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 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004.
[18] S. A. Weil. Leveraging intra-object locality with ebofs. *University of California, Santa Cruz*, 2004.