MGPA: a multi-granularity space preallocation algorithm for object-based storage devices

Shuibing He, Yuanhua Yang* and Xianbin Xu

School of Computer,
Wuhan University,
Wuhan, Hubei, China
Email: heshuibing@whu.edu.cn
Email: yangyuanhua123@163.com
Email: xbxu@whu.edu.cn
*Corresponding author

Xiaohua Xu

EECS Department,
University of Toledo,
Toledo, USA
Email: xiaohua.xu@utoledo.edu

Abstract: Object-based storage systems are promising because they effectively narrow the performance disparity between processors and storage devices. To achieve high performance, object-based storage devices (OSDs) generally preallocate disk space for an object when the desired space is not allocated. However, most existing space allocation algorithms utilise fixed-size preallocation strategies to preserve space for objects, resulting in poor disk space continuity when OSDs concurrently serve multiple objects. In this work, we propose MGPA, an adaptive multi-granularity object space preallocation algorithm to improve the I/O performance of OSDs. MGPA exploits both a user-informed method and an adaptive varied-size method to preallocate disk space. In the simulation-based experimental results, we show that MGPA can significantly improve the object space continuity, which will improve the long-term I/O performance of OSDs.

Keywords: object-based storage system; OBSS; object-based storage device; OSD; space preallocation.

Reference to this paper should be made as follows: He, S., Yang, Y., Xu, X. and Xu, X. (2016) 'MGPA: a multi-granularity space preallocation algorithm for object-based storage devices', *Int. J. Embedded Systems*, Vol. 8, Nos. 2/3, pp.237–248.

Biographical notes: Shuibing He received his PhD in Computing Science from Huazhong University of Science and Technology, China in 2009. He is a Lecturer at the School of Computer, Wuhan University, China. His research areas include parallel computer architecture, distributed storage and file systems, and embedded system.

Yuanhua Yang is a PhD student at the School of Computer, Wuhan University, China. His research interests include computer architecture, distributed storage systems, embedded systems.

Xianbin Xu received his PhD in School of Computer from Wuhan University, China in 2005. He is a Professor of School of Computer, Wuhan University. His research areas include computer architecture and I/O systems.

Xiaohua Xu received his PhD in Computer Science from Illinois Institute of Technology, USA in 2012. He is currently a Research Assistant Professor in the EECS Department of University of Toledo. His research interests include cyber security, cloud computing, sensor networks, and wireless networking.

1 Introduction

Many applications in important fields, such as astronomy, molecular dynamics, and climate modelling, are becoming increasingly data-intensive. For example, in Kandemir et al. (2008), some applications will generate more than 50 GB data on disk systems while more than half of the applications' total execution time is spent on disk I/O operations. Thus, efficient I/O support is critical to improve the performance of data-intensive applications (He et al., 2013b). If we can significantly increase the efficiency of data access on disk storage systems, an application's performance can benefit from the advancements of multi-core systems, and the overall system's utilisation can be largely improved.

To meet the high I/O demands, many data-intensive applications rely on parallel storage systems to provide data accesses (He et al., 2013a, 2014; Li et al., 2010; Dong et al., 2010; Bhardwaj and Sinha, 2006). One of the parallel storage systems, object-based storage system (OBSS), has recently gained enormous popularity in the network storage area (Mesnier et al., 2005; Kang et al., 2011, 2014). This is due to the merits of OBSS in cross-platform, including data sharing, policy-based security, direct data access, and high scalability. Many OBSSs, such as lustre (Schwan, 2003), storage tank (Menon et al., 2003), PanFS (Nagle et al., 2004) and Ceph (Weil et al., 2006), have been developed by the industrial community. To regulate the progressively sophisticated object-based technology, the object-based storage interface standard (Weber, 2009) (also referred as T10 OSD standard) is being developed by the Storage Network Industry Association. With a more expressive object interface, object-based storage devices (OSDs) are promisingly gaining intelligence (He et al., 2012).

OBSS typically consists of clients, metadata servers (Hua et al., 2011), and OSDs. OSDs are the ultimate data container for user data (He and Feng, 2007, 2008). In a large scale OBSS, there may be thousands of OSDs cooperating together to support high I/O performance (Hospodor and Miller, 2004). As a result, the performance of the overall storage system can be significantly improved even if the performance improvement of a single OSD is minimal. Therefore, designing high-performance OSDs is critical to meet the increasing I/O demands of data-intensive applications.

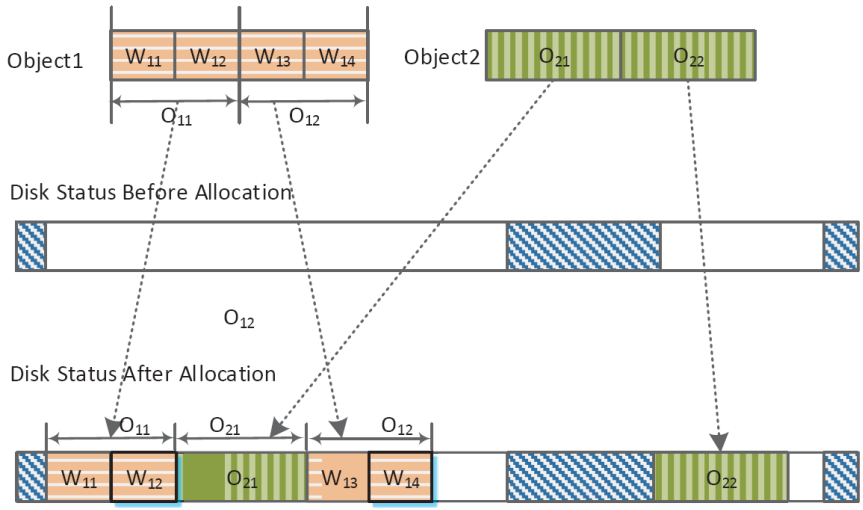
In OBSSs, each OSD relies on an object-based file system (OFS) to manage data. Existing OFSs can be divided into two categories: LOFSs and SOFSs. LOFSs adopt a local file system [such as Ext3 (Ts'o and Tweedie, 2002)] to store objects, allowing for simple design and implementation. In contrast, SOFSs use specific components to manage objects by considering the unique workload characteristic of OBSS (Wang et al., 2004b). For example, an extent-based object file system (EBOFS) (Weil, 2004) is designed to efficiently manage data in Ceph system (Weil et al., 2006). In an OFS, object space allocator is responsible for allocating disk space for objects. Because sequential accesses will reduce disk seek overhead, most allocators in both LOFSs and SOFSs attempt to allocate continuous space for user objects. To obtain high I/O performance, allocators usually preallocate free space for an object larger than the actual space needed. For example, EBOFS uses a predefined size for object space preallocation, and a local file system in LOFSs conserves a data region of approximately several kilobytes per request.

However, most available allocators in both LOFSs and SOFSs only use a single-granularity space preallocation strategy. Although these methods can help increase I/O throughput in a relatively simple environment, their effects are limited to situations with low I/O concurrency. As a shared resource, it is common in OSDs to concurrently serve multiple objects for data-intensive applications. Because OSDs need to concurrently allocate space for multiple objects, the disk space preallocated for the current object may be interleaved with the space allocated for other objects. Figure 1 describes an example of space allocation when an OFS allocates space for two objects concurrently. In this example, we assume that the OFS preallocates fixed-size space for a single request when the required space has not been allotted. For the first write request W_{11} on *Object*1, the OFS allocates the needed space. In addition, a neighbouring free space is preallocated. When the second write request W_{12} on *Object*1 arrives, its space has been allocated. However, because the OFS also allocates space for other requests on *Object*2 concurrently, the space behind W_{12} is allocated to *Object*2. In such a way, the object space continuity is largely decreased, leading to low disk I/O throughput even when the client accesses the object sequentially.

In this paper, we propose MGPA, an adaptive multi-granularity object space preallocation algorithm to improve the I/O performance for OSDs. MGPA exploits both a user-informed method and an adaptive varied-size preallocation method to preserve disk space for objects. If the size of an object can be obtained in advance, MGPA attempts to allocate a continuous disk space for the whole object with the informed size. Otherwise, MGPA selects a proper preallocation size for the object depending on its current size. In this case, MGPA divides the object size in the OSD into different ranges, each of which is configured with a preallocation granularity. When allocating space for an object, MGPA adaptively chooses various preallocation granularities depending upon the current object size.

We extended the *user object information* attribute page in the T10 OSD standard to enable our MGPA algorithm and implemented MGPA under EBOFS. Our extensive simulation-based experiments validate that MGPA can significantly improve the object space continuity when OSDs concurrently provide I/O services for multiple objects. Consequently, the long-term I/O throughput of the OSDs can be significantly improved.

Figure 1 Object space allocation example for two objects (see online version for colours)



Note: The colourful box means the space which has been allocated in disk, the white blank box means free space, while the colourful box (e.g., W_{12} and W_{14}) with blue shadow means space preallocated in the previous write request.

The rest of this paper is organised as follows. In Section 2, we present the background and the related work on object space preallocation. We describe the multi-granularity object space preallocation algorithm in Section 3. In Section 4, we use extensive experiments to evaluate our algorithm and analyse the results. Finally, we conclude this paper in Section 5.

2 Background and related work

In this section, we first provide an overview of the OBSS; then we present the related works on object space preallocation.

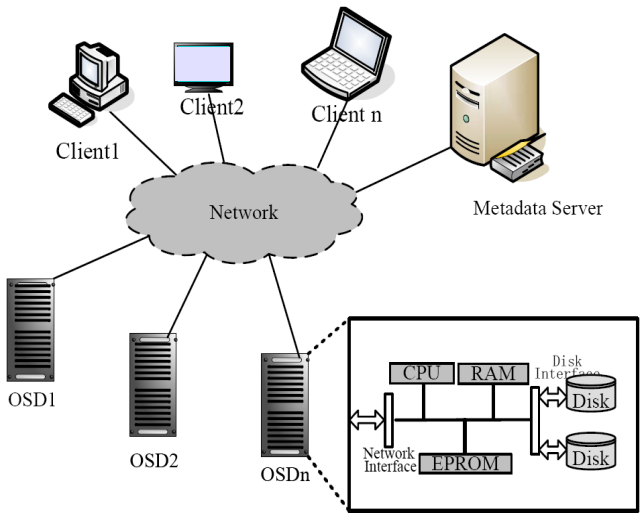
2.1 Overview of object-based storage

An object is a storage container of variable size and can be used to store any type of data. It behaves exactly like a file, which can be created and deleted, and can grow and shrink its size. An object also has numerous attributes that can describe the characteristics of the data.

The architecture of a typical OBSS is shown in Figure 2. The OBSS consists of three main parts: the clients, the metadata server (MDS), and the OSDs. The MDS provides object mapping information and authentication for clients' data accesses. When a client accesses data in an OSD, it first contacts the MDS and obtains the mapping information about the object. Then the client interacts with the OSD directly. Here the request contains an object ID, an offset within the object, attribute values, and so on. Finally, the OSD receives the object-based request and performs corresponding operations.

The OSD is one of the core components of OBSSs. It is a versatile storage device that contains CPU, memory, storage media (disk), and network interface. Generally, the OSD provides three major functions: object management, device security control, and network communication.

Figure 2 The architecture of an OBSS (see online version for colours)



The T10 OSD standard defines the OSD model and the basic command set (e.g., READ OBJECT and WRITE OBJECT command) that operate data on the OSD. Compared to a traditional file, objects provide a more expressive interface. Each object is identified by an object ID, and can be accessed by offset, length, and so on. Besides user objects, there are three kinds of objects, namely root objects, partition objects, and collection objects. The last three types of objects can help to address user objects more efficiently.

In addition to data access commands, the T10 OSD standard also defines attribute commands (e.g., GET ATTRIBUTES and SET ATTRIBUTES) that allow users to set and get attributes. An object can have many different attributes, related attributes are organised into one attribute page. Each attribute page has a unique page number, and each attribute in the page has a unique attribute number. Besides the attributes defined by the OSD standard, if necessary a user can define their flown attributes. In this

paper, we extend the *user object information* attribute page, and add two attributes – *informed-size* (Section 3.1) and *CLOSE* (Section 3.2) to support the proposed multi-granularity object space preallocation approach.

2.2 LOFS space preallocation

Because of the convenience in practical design and implementation, LOFSs usually adopt a local file system, such as Ext3 (Ts'o and Tweedie, 2002), Ext4 (KV et al., 2008), Reiserfs (Buchholz, 2006), and XFS (Mostek et al., 1999), to store objects. In this case, an object is mapped to a file, and LOFSs use the conventional preallocation methods to prevent disk fragmentation.

Conventional preallocation methods allocate some disk space to store a file prior to write requests (Powell, 1977), and these methods can be conducted in the operating system kernel or user mode. For the kernel mode, the file space allocator preallocates a fixed-size region of tens of kilobytes in addition to the size of the file write request. For example, EXT3, EXT4, and XFS adopt these methods (KV et al., 2008; Ts'o and Tweedie, 2002) to improve I/O performance. In this way, a file continually grows by processing continuous write requests until the final file is generated. While these methods are effective to reduce the disk fragmentation, it is difficult to choose suitable sizes for different files. If the size is too large, a minimal amount of continuous free space is left for the newly created file as the file system ages. If the size is too small, it is not very effective for large-size files. In contrast, MPGA uses an adaptive multiple-granularity method to preserve space for objects depending upon their growing sizes. Thus, MPGA can keep the preallocation neither conservative nor aggressive.

For the user mode, a user or an application predicts the final size of a file and preallocates enough space. For example, a POSIX standard function *fallocate*(·) is currently implemented by Lustre. In IBM (2014), IBM SAN file system has file preallocation policies specified by the administrator. It is very effective at reducing file fragments if these approaches are properly used. However, they are not feasible when the user or application is unable to predict the final size of the file. Furthermore, it is difficult to use these methods because it is problematic to initiate a preallocation with a narrow `file' interface. Compared to these methods, MGPA utilises the expressive object interface to perform informed preallocation, and also adds an adaptive strategy to preserve space for objects. Thus, MGPA is extremely applicable.

2.3 SOFS space preallocation

SOFSs use their flown data structures and disk space allocators to manage object data. Space allocator is responsible for appropriating space when clients write new data to an object and reclaiming space when the object is deleted. To maximise system efficiency and scalability, EBOFS utilises extent as an object space allocation unit (McVoy and Kleiman, 1991; Schindler et al., 2002). In addition, EBOFS adopts balanced trees (B+trees) to manage disk free space and object allocated space, thus corresponding lookup overhead can be significantly decreased.

There are numerous object create, write and delete operations during the lifetime of an object file system. In order to reduce the object fragmentation as the file system ages, EBOFS preallocates disk space for each object write request. This approach effectively reduces the number of extents within an object, and attempts to allocate writes adjacently. However, because it uses a fixed-size strategy, this approach leads to the drawback mentioned in Section 1. MPGA adopts adaptive multiple-granularity to preallocate data, allowing object space to be more continuous during the lifetime of the object file system.

3 Multi-granularity object space preallocation

Traditional preallocation methods are not effective for OSDs when they concurrently provide I/O services for multiple objects. We propose MGPA, a multi-granularity object space preallocation algorithm to improve the I/O performance of OSDs. MGPA adjusts its preallocation size depending on two policies: the informed object size and the growing object size.

3.1 User-informed preallocation

For many data intensive applications, data access patterns are generally independent of the data values and these patterns have predictable alterations (Wang and Kaeli, 2003; Zhang and Jiang, 2010). For example, the BTIO application (Wong and der Wijngaart, 2003), an I/O kernel responsible for solving block-tridiagonal matrices on a three dimensional array, has the following feature. Once the size of the array, the number of time steps, the write interval, and the number of processes are given, the I/O behaviours of the application are determined and the final size of the user object can be predicted before the program executes. Users with thorough knowledge about their application can easily predict the I/O behaviours of the applications.

The predictability of object size provides an opportunity to achieve improved object space preallocation. This is because more accurate preallocation for the whole object can largely alleviate object fragmentations when OSDs serve multiple objects concurrently. As a result, MPGA uses a user-informed preallocation policy to preserve space when the size of an object can be obtained in advance. Specially, with the expressive object interface, MGPA preallocates space for each object as follows.

1 The *user object information* attribute page for each object in the system is extended to support the preallocation method. For each object, an attribute, named *informed-size*, is added into the attribute page. Table 1 summaries the content of the extended attribute page.

2 If users know the final object size in advance when they issue CREATE OBJECT commands, users can set the *informed-size* to the desired value with the SET ATTRIBUTES command. Otherwise, the size is set to 0.

3 When a user sends a WRITE OBJECT command to the OSD, MGPA gets the *informed-size* in the attribute page with the GET ATTRIBUTES interface for the current object. If the *informed-size* is not 0, MGPA will set the preallocation size to the informed value, and tries to preserve the whole space for the current object.

Table 1 The *user object information* attribute page

Attribute number	*Length (bytes)*	*Attribute description*	*User settable*
0h	40	Page identification	No
1h	8	Partition_ID	No
2h	8	User_object_ID	No
3h	8	Informed-size	Yes
4h	8	CLOSE	Yes
5h–8h		Reserved	No
9h	Variable	Username	Yes
...	...	...	...

Notes: The extended attributes are 03h and 04h attributes, namely *informed-size* and CLOSE, and they can be set by users. Other attributes in the table are defined by the T10 OSD standard.

3.2 Adaptive varied-size space preallocation

In practice, there are a number of objects whose sizes cannot be determined in advance. Requiring users to inform all object sizes is not feasible. In this case, in order to reduce disk fragmentation and increase the object space continuity for long-term I/O operations, MGPA uses adaptive varied-size preallocation to preserve space for objects.

MGPA divides the object size in an OSD into a series of excluding ranges, each of which is configured with a corresponding preallocation granularity. The larger the range of the current object size, the more defined granularity of the object. Initially, MGPA allocates its space with a small preallocation size when an object is created. Gradually, a large preallocation space will be assigned to the current object as it grows. In such a way, an adaptive varied-size preallocation can be performed for the object during its lifetime. Because the preallocation granularity is increased gradually, both conservative and aggressive disk space allocation can be avoided.

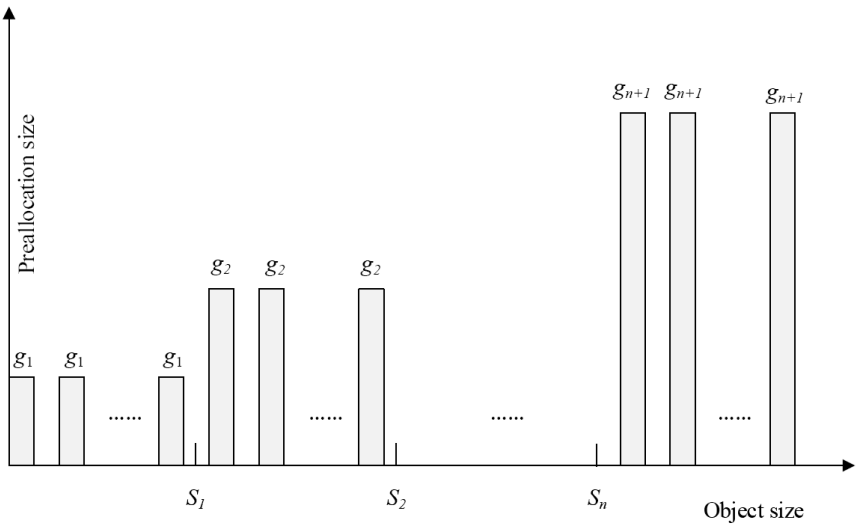
Assuming s_1, s_2, ..., and s_n are the n boundaries used to divide the size of the object, and g_1, g_2, ..., g_n, and g_{n+1} are the $n + 1$ preallocation granularity, then equation (1) calculates the preallocation size for the current object with a growing size of S_{obj}

$$prealloc\text{-}size = \begin{cases} g_1, & \text{if } S_{obj} < s_1 \\ g_{i+1}, & \text{if } s_i \le S_{obj} < s_{i+1} \\ g_{n+1}, & \text{if } S_{obj} > s_n \end{cases} \quad (1)$$

Figure 3 illustrates the space allocation process of an object with growing sizes. In this example, when the object size is below S_1, MGPA preserves space with size g_1 each time for the object. This preallocation size is repeated until the current object size is above S_1. Similarly, a preallocation size of g_2 is adapted when the object size is between S_1 and S_2. A dynamic size is used to preallocate disk space for the object as it grows.

Theoretically, the object size in an OSD can be divided into unlimited ranges. However, because practical I/O workloads of OSDs usually demonstrate some unique characteristics (Wang and Kaeli, 2003), we only use three ranges to preallocate space in our current design. The related parameters are chosen based on experiments and experience, and our observations show that they are effective in preallocating space for objects (in Section 4). We leave the more delicate design to our future work.

Figure 3 Preallocation size is adjusted as the object grows



One concern with varied-size preallocation is that the preallocated space may not actually be used by the object, leading to disk space waste and increased internal disk fragmentation. To overcome this potential drawback, MGPA releases the unused disk space when the object does not grow any longer with an attribute-synergistic method as follows.

1 A new attribute *CLOSE* is added into the *User Object Information* attribute page described in Table 1. CLOSE is a Boolean variable, which indicates whether the unused preallocation space should be reclaimed.

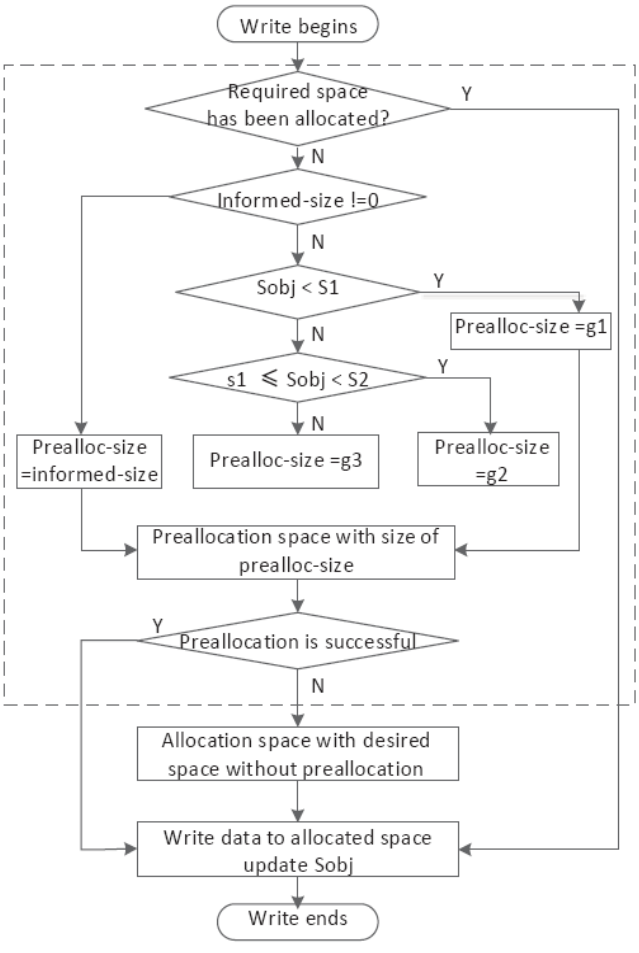
2 Once the users finish their object requests (like the file close operations in local file systems), they can set *CLOSE* to be *TRUE*.

3 If MGPA detects that the *CLOSE* value is *TRUE* through the attribute operation interface when processing object requests, it will release the unused space of the object.

3.3 Object write process

Object space preallocation occurs in the process of object write operations. Figure 4 shows the flowchart of a write operation with the proposed object space preallocation method.

Figure 4 Object write-process flowchart



First, the allocator checks if the data is written to an allocated space. If yes, the data will be directly written to the allocated space. Otherwise, MGPA is triggered to preallocate space for the current object. In this case, MGPA first determines whether the *informed-size* is 0 by getting the object's extended attributes. If it is true, MGPA will use the *informed-size* to preallocate disk space for the current request. Otherwise, MGPA will preallocate space based on the policy described in Section 3.2. In Figure 4, two sizes of s_1 and s_2 are used to divide the object size into three ranges, and three granularity of g_1 to g_3 are adopted for space preallocation respectively.

4 Evaluation

In this section, we conduct a number of experiments to evaluate the MGPA preallocation algorithm with the existing fixed-size preallocation algorithm (FSPA). Since a newly created file system has sufficient free disk space and is not sensitive to different preallocation policies, we focus on aged file systems in this paper. As EBOFS is a typical and excellent specific object file system, we evaluate different preallocation policies based on the EBOFS file system.

4.1 Experimental setup

Since benchmarking a fully loaded disk over its expected lifetime requires a long time, we use a simulator integrated with different preallocation policies to allocate disk space in our experiments. The simulator utilises the allocation code of EBOFS but performs all allocation operations in memory as in Weil (2004). This allows analysis of file system fragmentation over much longer time than real benchmarks.

Modelling file system workload as seen by an individual OSD in a large system is a surprisingly complicated problem. Since it is impossible to characterise the workload of a system that does not exist yet, the simulator instead approximates a nearly worst case behaviour 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 concurrently.

The underlying workload being serviced by the OSD is assumed to be typical: most writes are sent to small objects, most data is written to large objects, and most files are deleted either while they are young or never at all. The statistics of object operations in our workload are shown in Table 2.

Table 2 The statistic of object operations

Operation	*Number of objects*	*Total space*
Read	200,297	742.4 GB
Create/write	410,749	1,545.1 GB
Delete	388,954	1,446.2 GB

Figure 5 shows the size distribution for all objects in the object-based file system. All objects are between 0 KB and 2 GB, and a substantial amount are between 512 KB and 16 MB, occupying 85% of the overall object space. The workload characteristics are similar with those in the real traces collected from LLNL trace (Wang et al., 2004a). Figure 6 describes the number of objects existing in the object file system, and Figure 7 illustrates the occupied disk space for all objects on a 120 GB disk as the file system ages. From these figures, we can see that as users create and delete objects, the statistics of objects are changed gradually.

Figure 5 The size distribution for all objects in the file system (see online version for colours)

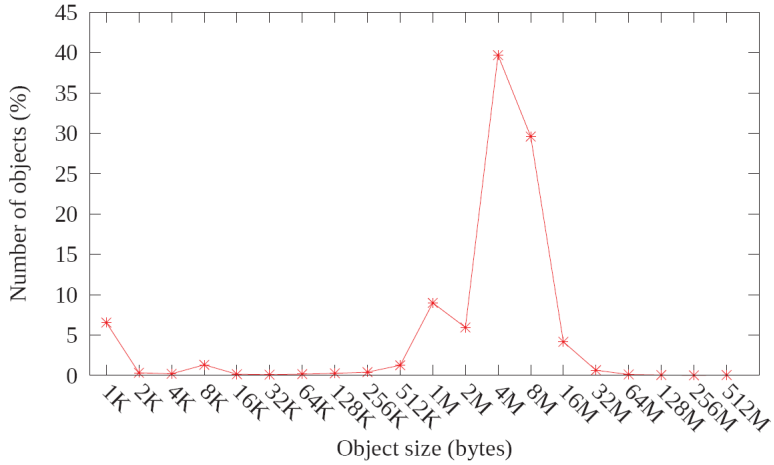


Figure 6 The number of the objects as file system ages (see online version for colours)

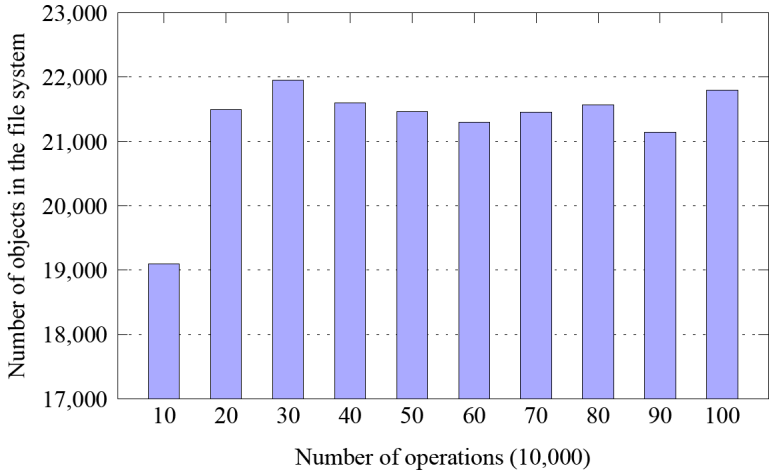
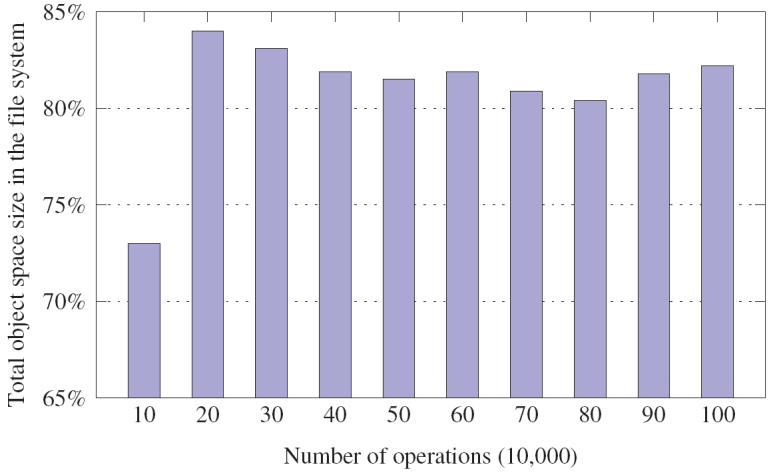


Figure 7 The occupied disk space for all objects as file system ages (see online version for colours)



4.2 Evaluation metric

We use allocated space continuity to evaluate different preallocation policies in this paper. Firstly, we introduce the concept of block layout score (Smith and Seltzer, 1996), denoted by *LS*(*Block*), to measure the space continuity for each block (or extent) in an object. Assuming $B_{i,j}$ is the j^{th} block of Object O_i, and $P(B_{i,j})$ is the physical address of $B_{i,j}$ on the disk, we calculate $LS(B_{i,j})$ as in equation (2).

$$LS(B_{i,j}) = \begin{cases} 1, & j = 0 \\ 1, & j > 0 \,\&\, P(B_{i,j-1}) + 1 = P(B_{i,j}) \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

Secondly, we adopt *LS*(*alloc*) to measure the allocated space continuity for all objects in the file system based on block layout score. Assuming object O_i has *sizeof*(O_i) blocks, by measuring all objects in the allocated space *S*, *LS*(*alloc*) can be calculated as in equation (3).

$$LS(alloc) = \frac{\sum_{O_i \in S} \sum_{j=1}^{sizeof(O_i)} LS(B_{i,j})}{\sum_{O_i \in S} sizeof(O_i)} \quad (3)$$

In our experiments, we use *LS*(*alloc*) to evaluate the behaviours of different object space preallocation algorithms. Different algorithms lead to various *LS*(*alloc*) values. Generally, an algorithm which has a value closer to 1 means it provides better I/O performance.

Table 3 The workload description in the experiments

Experiments	*Workload descriptions*
set1	All object sizes are informed
set2	No object size is informed
set3	Partial object sizes are informed

4.3 Results

We conduct three sets of experiments to compare MGPA with traditional FSPA algorithm. Table 3 lists the detailed workload configurations of each set of experiments. As described in previous sections, the user informed and varied-size method will be triggered in set1 and 2 respectively in MGPA, and both will work in set3.

4.3.1 Result of set1

Figure 8 describes the allocated space continuity of MGPA and FSPA. For the former, the object space is preallocated each time in a 'try to fit' manner. For the later, the size for space preallocation is varied between 512 KB and 16 MB. We observe that the former has a better result than the later.

The allocated space continuity is low when the size is smaller in FSPA (see the 512 KB line) because the concurrent write requests are served in an interleaved fashion. With the increase in preallocation size, the space continuity increases, showing that aggressive preallocation can alleviate fragmentation and improve disk I/O performance. FSPA nearly has the same performance as MGPA for the first 10^5 operations when the size increased up to 8 MB and 16 MB. However, with the increase in operations, its space continuity is rapidly reduced. This indicates that large-size preallocation in FSPA will lead to serious fragmentation for aged file systems. In contrast, MGPA has a stable high space continuity even for a long-term object operations.

In more detail, Figure 9 describes the number of extents when different preallocation algorithms are adopted in the experiments. There are a large number of extents in the file system when a 512 KB size is used in FSPA. When the size is 2 MB, the number of extents is significantly reduced by around 60%. This is because smaller sizes require multiple allocations to satisfy one object, resulting in more extents in the allocated space. When the size is 8 MB, fewer extents are needed to accommodate the data of an object. However, aggressive preallocation will also increase the number of small free extents, and decrease the number of large free extents. Consequently, one object is composed of more small extents. The situation worsens when the preallocation size is 16 MB.

Figure 8 The space continuity of allocated object space

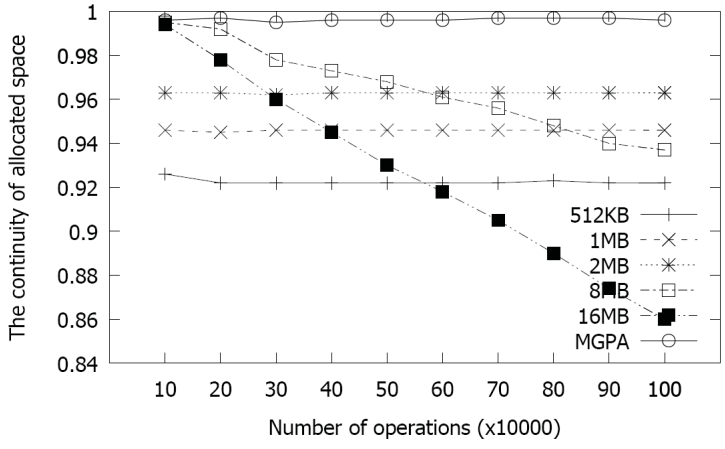
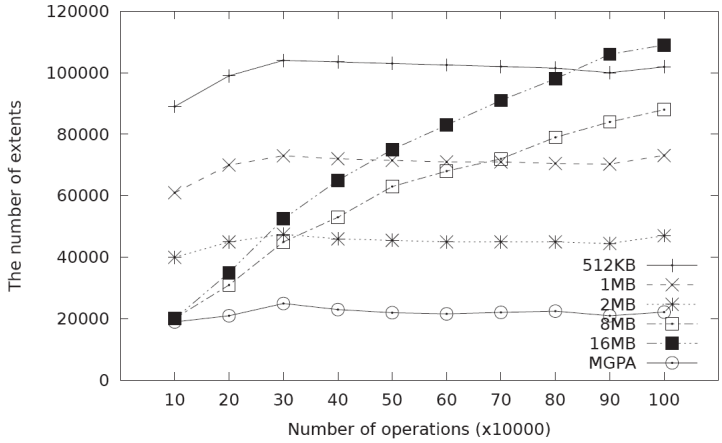


Figure 9 The number of extents in the allocated space



4.3.2 Result of set2

In these experiments, we compare the performance of MGPA and FSPA under the workloads with unknown object sizes. To show the efficiency of MGPA under a comprehensive environment, three configurations are used in the MGPA algorithm. Table 4 lists the related parameters of MGPA. For FSPA algorithm, since an extreme size, as described in Section 4.3.1, will lead to poor performance, we only depict the results of FSPA with moderate preallocation sizes of 2 MB and 8 MB.

Table 4 The settings of size-adjusted preallocation

Experiments	*Parameters (MB)*
MGPA1	s1 = 1, s2 = 16, g1 = 1, g2 = 4, g3 = 8
MGPA2	s1 = 2, s2 = 16, g1 = 2, g2 = 4, g3 = 8
MGPA3	s1 = 4, s2 = 16, g1 = 2, g2 = 4, g3 = 8

Figure 10 demonstrates the allocated space continuity under MGPA and FSPA. From the figure, we can conclude that FSPA with a large preallocation size outperforms MGPA, when the number of operations is small. For example, FSPA has a higher space continuity than MGPA when the number of operations is below $4 * 10^5$. For the proposed varied-size preallocation algorithms, MGPA1 has the lowest performance among the three MGPA algorithms. This is because more small extents are allocated for one object owing to its smaller preallocation granularity configured with a longer size range. Conversely, MGPA3 has the best performance.

However, all of them exceed FSPA as the file system ages.

Figure 11 describes the number of extents of MGPA and FSPA in the experiments. MGPA has drastically decreased extents in the system compared to FSPA as the number of operations is increased gradually. Therefore, improved I/O performance can be obtained for an aged file system.

4.3.3 Result of set3

In these experiments, only a part of the objects have informed sizes. In this case, both user informed preallocation and varied-size method will work in MGPA. Table 5 lists the ratio of objects with informed sizes. For MGPA algorithms, the related parameters for preallocation are identical to those in the third configuration in Table 4 because these parameters lead to the best performance. For FSPA algorithms, we only use preallocation sizes of 2 MB and 8 MB.

Table 5 The settings in the experiments

Experiments	*Configurations*
MGPA1	The sizes of 30% objects are informed
MGPA2	The sizes of 50% objects are informed
MGPA3	The sizes of 70% objects are informed

Figures 12 and 13 display the allocated space continuity and the number of extents of MGPA and FSPA respectively. Similar to the results in experiment set2, MGPA has a better performance than FSPA. MGPA1 has the lowest performance among the three MGPA algorithms, and MGPA3 is the optimal one. From the three sets of experiments, we can observe that MGPA is able to reduce the number of extents in the file system, thus decreasing the extents per object. As the allocated space continuity is significantly increased, the I/O performance of object accesses can be improved.

5 Conclusions and future work

In this paper, we propose MGPA, an adaptive multi-granularity object space preallocation algorithm to improve disk I/O performance of OSDs. Our extensive simulation-based experiments validate that MGPA can significantly improve object space continuity, when the system concurrently serves multiple objects. Therefore, the long-term I/O performance of OSDs can be considerably improved.

Though MGPA demonstrates its advantages in the simulations under EBOFS, it is also promising under other object file systems. In the future, we will extend MGPA to other file systems and test its behaviours. Moreover, the performance of MGPA is sensitive to the parameters configured in the algorithm and dependent on the characteristics of I/O workload. We will conduct further research on the delicate design of object size division and preallocation granularity selection.

Figure 10 The space continuity of allocated object space

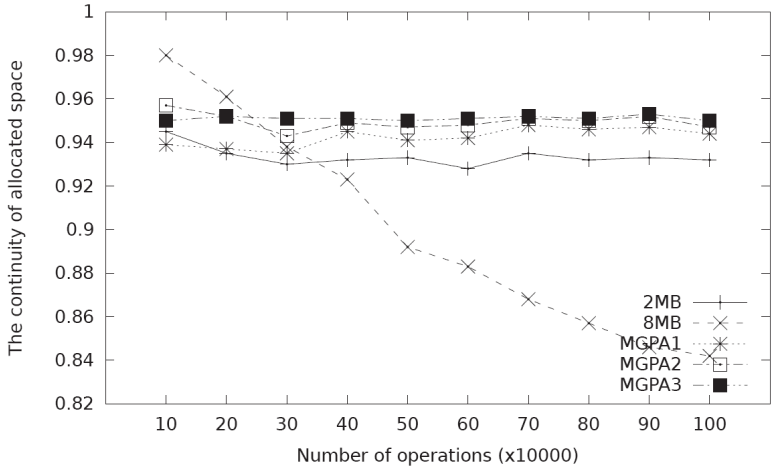


Figure 11 The number of extents in the allocated space

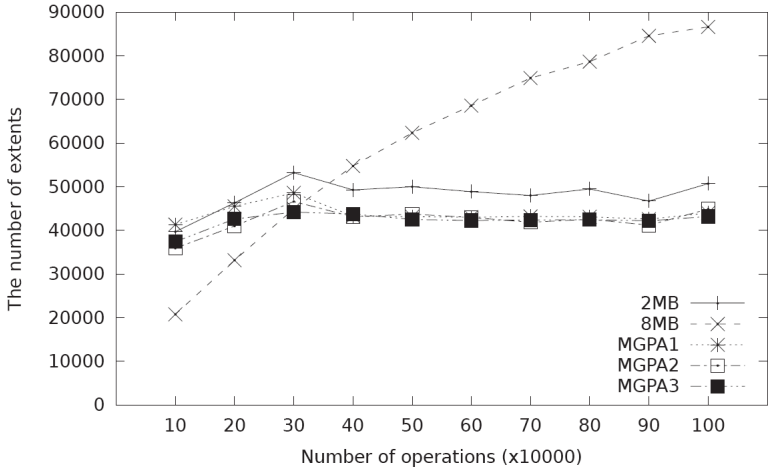


Figure 12 The space continuity of allocated object space

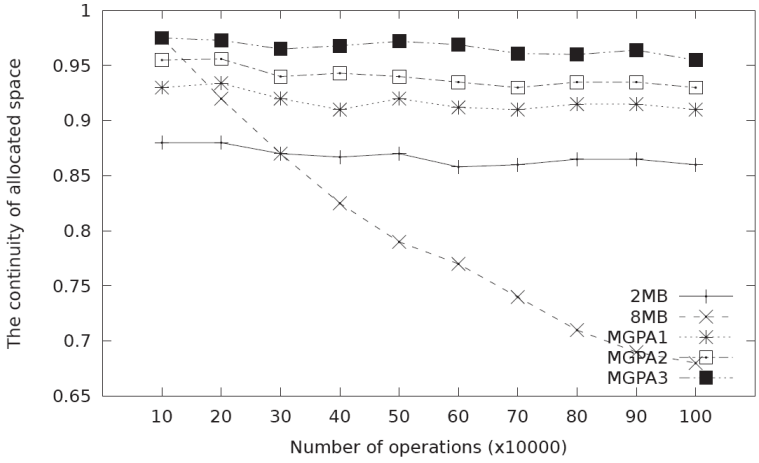
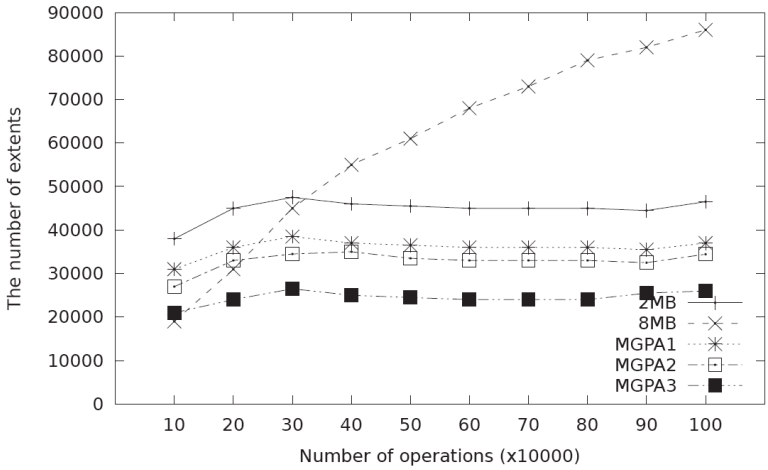


Figure 13 The number of extents in the allocated space



Acknowledgements

This research was supported in part by the Natural Science Foundation of Hubei Province, China (Grant No. 2014CFB239), and National Science Foundation under grant NSF ECCS-1310551.

References

Bhardwaj, D. and Sinha, M.K. (2006) 'GridFS: highly scalable I/O solution for clusters and computational grids', *International Journal of Computational Science and Engineering*, Vol. 2, Nos. 5/6, pp.287–291.

Buchholz, F. (2006) *The Structure of the Reiser File System*, Technical Report Technical Document.

Dong, B., Li, X., Xiao, L. and Ruan, L. (2010) 'An optimal candidate selection model for self-acting load balancing of parallel file system', *International Journal of High Performance Computing and Networking*, Vol. 7, No. 2, pp.123–128.

He, S. and Feng, D. (2007) 'Implementation and performance evaluation of an object-based storage device', *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, pp.129–136.

He, S. and Feng, D. (2008) 'Design of an object-based storage device based on I/O processor', *ACM SIGOPS Operating Systems Review*, Vol. 42, No. 6, pp.30–35.

He, S., Sun, X-H. and Feng, B. (2014) 'S4D-cache: smart selective SSD cache for parallel I/O systems', *Proceedings of the International Conference on Distributed Computing Systems*.

He, S., Sun, X-H. and Yin, Y. (2013a) 'BPS: a performance metric of I/O system', *Proceedings of the International Workshop on High Performance Data Intensive Computing (HPDIC)*, pp.1954–1962.

He, S., Sun, X-H., Feng, B., Huang, X. and Feng, K. (2013b) 'A cost-aware region-level data placement scheme for hybrid parallel I/O systems', *Proceedings of the IEEE International Conference on Cluster Computing*.

He, S., Xu, X. and Yang, Y. (2012) 'OASA: an active storage architecture for object-based storage system', *International Journal of Computational Intelligence Systems*, Vol. 5, No. 6, pp.1173–1183.

Hospodor, A. and Miller, E. (2004) 'Interconnection architectures for petabyte-scale high-performance storage systems', *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp.273–281.

Hua, Y., Zhu, Y., Jiang, H., Feng, D. and Tian, L. (2011) 'Supporting scalable and adaptive metadata management in ultralarge-scale file systems', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 4, pp.580–593.

IBM (2014) *SAN File System V2.2.2* [online] http://publib.boulder.ibm.com/infocenter/tssfsv21/v1r0m0/index.jsp?.

Kandemir, M., Son, S.W. and Karakoy, M. (2008) 'Improving I/O performance of applications through compiler-directed code restructuring', *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pp.159–174.

Kang, Y., Pitchumani, R., Marlette, T. and Miller, E.L. (2014) 'Muninn: a versioning flash key-value store using an object-based storage model', *Proceedings of International Conference on Systems and Storage*, pp.1–11.

Kang, Y., Yang, J. and Miller, E.L. (2011) 'Object-based SCM: an efficient interface for storage class memories', *Proceedings of the 2011 IEEE 27th Symposium onMass Storage Systems and Technologies*.

KV, A., Cao, M., Santos, J. and Dilger, A. (2008) 'Ext4 block and inode allocator improvements', *Proceedings of the Linux Symposium*, pp.263–274.

Li, C., Zhou, K. and Feng, D. (2010) 'Capturing the object behaviour for storage system evaluation', *International Journal of High Performance Computing and Networking*, Vol. 6, Nos. 3/4, pp.226– 233.

McVoy, L. and Kleiman, S. (1991) 'Extent-like performance from a unix file system', *Proceedings of the USENIX Winter Conference*, pp.33–43.

Menon, J., Pease, D., Rees, R., Duyanovich, L. and Hillsberg, B. (2003) 'IBM storage tank – a heterogeneous scalable SAN file system', *IBM Systems Journal*, Vol. 42, No. 2, pp.250–267.

Mesnier, M., Ganger, G. and Riedel, E. (2005) 'Object-based storage: pushing more functionality into storage', *IEEE Potentials*, Vol. 23, No. 2, pp.31–34.

Mostek, J., Earl, W. and Koren, D. (1999) 'Porting the SGI XFS file system', *Proceedings of the Freenix Track: USENIX Annual Technical Conference*.

Nagle, D., Serenyi, D. and Serenyi, D. (2004) 'The Panasas active Scale storage cluster: delivering scalable high bandwidth storage', *Proceedings of the ACM/IEEE Conference on Supercomputing*.

Powell, M. (1977) 'The DEMOS file system', *ACM SIGOPS Operating Systems Review*, Vol. 11, No. 5, pp.33–42.

Schindler, J., Griffin, J., Lumb, C. and Ganger, G. (2002) 'Track-aligned extents: matching access patterns to disk drive characteristics', *Proceedings of the USENIX Conference on File and Storage Technologies*, pp.259–274.

Schwan, P. (2003) 'Lustre: building a file system for 1000-node clusters', *Proceedings of the Linux Symposium*, Vol. 2003.

Smith, K. and Seltzer, M. (1996) 'A comparison of FFS disk allocation policies', *Proceedings of the USENIX Conference*, pp.15–26.

Ts'o, T. and Tweedie, S. (2002) 'Planned extensions to the Linux EXT2/EXT3 file system', *Proceedings of the Freenix Track: USENIX Annual Technical Conference*, pp.235–244.

Wang, F., Brandt, S.A., Miller, E.L. and Long, D.D.E. (2004a) 'OBFS: a file system for object-based storage devices', *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp.283–300.

Wang, F., Xin, Q., Hong, B., Brandt, S., Miller, E., Long, D. and McLarty, T. (2004b) 'File system workload analysis for large scale scientific computing applications', *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp.139–152.

Wang, Y. and Kaeli, D. (2003) 'Profile-guided I/O partitioning', *Proceedings of the 17th Annual International Conference on Supercomputing*, pp.252– 260.

Weber, R. (2009) *Information Technology – SCSI Object-based Storage Device Commands-2 (OSD-2), Revision 5*, Technical Report INCITS Technical Committee T10/1729-D.

Weil, S. (2004) *Leveraging Intra-Object Locality with EBOFS*, Technical Report UCSC cmps-290s Project Report.

Weil, S., Brandt, S., Miller, E., Long, D. and Maltzahn, C. (2006) 'Ceph: a scalable, high-performance distributed file system', *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pp.307–320.

Wong, P. and der Wijngaart, R. (2003) *NAS Parallel Benchmarks I/O Version 2.4*, Technical Report NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002.

Zhang, X. and Jiang, S. (2010) 'Interference removal: removing interference of disk access for MPI programs through data replication', *Proceedings of the 24th ACM International Conference on Supercomputing*, pp.223–232.