Future Generation Computer Systems 56 (2016) 746–758

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Oasis: An active storage framework for object storage platform

Yulai Xie [a], Dan Feng [a,*], Yan Li [b], Darrell D.E. Long [b]

[a] School of Computer, Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, PR China
[b] Jack Baskin School of Engineering, University of California, Santa Cruz, CA 95064, USA

HIGHLIGHTS

- We provide transparent and multi-granularity processing for function execution.
- We provide digital signature scheme for ensuring the code integrity.
- We can integrate the Oasis framework into the Kinetic cloud platform.
- We implement and evaluate the adaptive computation partition method in Oasis.

ARTICLE INFO

Article history:
Received 26 June 2015
Received in revised form
3 August 2015
Accepted 19 August 2015
Available online 28 August 2015

Keywords:
Storage system
Object-based storage
Active storage
Kinetic
Adaptive computation partition

ABSTRACT

The network bottleneck incurred by big data process and transfer has increasingly become a severe problem in today's data center and cloud. Exploring and exploiting the advantages of both the scalable object storage architecture and intelligent active storage technology are one of the ways to address this challenge. In this paper, we present the design and performance evaluation of Oasis, an active storage framework for object-based storage platform such as Seagate Kinetic. The basic idea behind Oasis is to leverage the OSD's processing capability to run data intensive applications locally. In contrast with previous work, Oasis has the following advantages. First, Oasis enables users to transparently process the OSD object and supports different processing granularity. Second, Oasis can ensure the integrity of execution code using signature scheme and provide the access control for the code execution in the OSD by enhancing the existing OSD security protocol. Third, Oasis can partition the computation task between host and OSD dynamically according to the OSD workload status. Our work on Oasis can be integrated into Kinetic object storage platform seamlessly. Experimental results on widely-used real world applications demonstrate the performance and efficiency of our system.

1. Introduction

Today's data center and cloud have to process, transfer and analyze the ever-increasing data deluge. According to the study from International Data Corporation (IDC) [1,2], at least 800 Exabytes of data were created and analyzed every year, and this number is still increasing. The big commercial companies like IBM, EMC, Google, Amazon, also have to handle PB (even EB) data every day to support their online service [3–5]. As the data center and cloud are usually networked architecture that is constructed via interconnecting a large number of servers, it is critical to avoid the network bottleneck on the interconnect incurred by the big data transfer so as to provide real-time service to the users.

Processing the data where they are (i.e., Active Storage [6,7]) has been a simple but popular idea to solve this problem. By exploiting the processing power of the storage device, active storage is not only able to filter data and reduce the bandwidth requirement on the network, but also provide aggregation processing capabilities through the parallelism of the disks. A number of research institutions have developed active storage systems such as database machine [8], IDISK [9], Active Disk [6], and IRAM [10]. However, these approaches do not get wide deployment. A critical reason is that different approaches need different interface extension. In addition, most of the systems need big modification to the file system, which is not acceptable for the commercial systems.

Efficient active storage scheme needs to execute *application function* (application-specific code that can be downloaded and executed on the user data, e.g., compression, classification, etc.) inside the storage device and answer the questions, such as "how to transparently execute application function and how to guarantee

* Corresponding author.
E-mail address: dfeng@hust.edu.cn (D. Feng).

http://dx.doi.org/10.1016/j.future.2015.08.011
0167-739X/© 2015 Elsevier B.V. All rights reserved.

the execution of application function inside storage device is safe?'' These problems are more challenging to address, especially the data deluge is becoming more serious and the data sharing is more frequently. Specifically, we need to address the following challenges.

Challenge 1: transparent and multi-granularity process. In most cases, a user of an active storage product does not wish to remember the details of application functions: how these application functions are programmed, the execution parameters or the identifiers of the application functions. How to accurately schedule the application function to execute but makes it transparent to the users simultaneously presents a big challenge to address. In addition, a user might want to encrypt anything from a file to a whole directory that contains hundreds of thousands of files. In many cases, encrypting each file one by one is not an efficient solution. Especially in the case of data deluge, processing data in the batch mode is critical. So providing multi-granularity process is also necessary.

Challenge 2: security. The execution of an application function must be safe. The reason is obvious, unsafe execution can result in the wrong results, e.g., a program accesses an invalid memory space. In addition, a user may not want the code that he downloaded to the storage device be used by other users, so access control should be considered.

Challenge 3: resource contention. The users always care about the response time, especially for the online real-time service. However, as we are always in the data share circumstance (e.g., cloud), resource contention issue has become increasingly serious. This can significantly impact the system performance if multiple application functions are running in the same storage devices at the same time. How to avoid the active storage devices heavily-loaded in a maximum extent is a big challenge to address.

More recently Seagate introduced Kinetic object device that includes an internal processor [11]. The Panasas has also continually promoted the standardization of object-based storage specification and has developed an OSD initiator included in the Linux 2.6.30 kernel [12]. Motivated by these, and also to address the challenges above, we propose *Oasis*, an active storage framework for object storage platform. The basic idea behind Oasis is to leverage the OSD's processing power to run data intensive applications. The rationale comes from the basic understanding between the properties of object-based storage and active storage. On one hand, object storage devices can manage the location of object data itself, which makes it possible to process the object data inside OSD. On the other hand, the *application function* performed in OSD can be taken as a kind of object. Thus it is much easier to process it by using the existing object commands, object security scheme, and object management methods. In addition, as the object has attributes that contain rich semantic information, the user can control the execution of *application function* through the expressive object interface that OSD exports. Specifically, we make the following contributions.

First, (for challenges 1 and 2), Oasis frees users from needing to remember the details of *application functions* and enables users to transparently process the OSD object. In addition, Oasis supports different processing granularity (from a single object to all the objects in the OSD) by extending the OSD object attribute. Moreover, Oasis uses signature scheme to ensure the integrity of execution code and provides the access control for the execution of the application function in the OSD by enhancing the existing OSD security protocol, thus preventing unauthorized users from intentionally destroying the system.

Second, (for challenge 3), Oasis can monitor system resources (e.g., CPU utilization and network traffic status) and partition the computation workload between host and OSD dynamically according to their workload status. This can maximumly alleviate the system traffic and avoid degrading the whole system performance.

Third, we have implemented the Oasis components in a software layer, with only slight modification to the current object file system. The extensive experiments examine the Oasis performance and scalability by using typical real-world applications. Our work on Oasis can be integrated into the emerging products such as Kinetic object storage platform.

The rest of the paper is organized as follows. We summarize background and related work in Section 2 and elaborate the design and implementation of Oasis in Section 3. In Section 4, we compare Oasis with the recent RPC-based active storage approach. In Section 5, we evaluate the implementation of Oasis. In Section 6, we describe how Oasis can be integrated into the Kinetic platform and analyze the performance result. In Section 7, we discuss the issues on interface and data consistency in Oasis. In Section 8, we conclude the paper.

2. Background and related work

In this section, we first give an overview of object-based storage and Kinetic object storage platform. Then, we present the related work on active storage and then motivate our research.

2.1. Object-based storage and Kinetic

With the rapidly escalating storage requirements of enterprises, object-based storage [13] has emerged as one of the most promising technical solutions to next-generation storage systems in the past few years. It offloads storage management functions from the host operating system to the intelligent object-based storage device (OSD) that manages its own storage space and exports an expressive object interface. Object-based storage systems, such as Lustre [14], Panasas [15] and Ceph [16] that combine the advantages of NAS [17] and SAN [18], can provide high throughput, reliability, availability and scalability. The widely used storage media, such as storage class memory [19,20], also has a better performance and reliability with object-based storage technology.

Seagate, the hard disk manufacturer, has also proposed a Kinetic open storage platform that adopts the key/value object interface [11]. It is a device-based ethernet-attached storage platform that eliminates the traditional storage server infrastructure. The client API provides a series of command interface (such as PUT, GET, and DELETE) to access the objects in the Kinetic device. It can also restrict the operations a user can perform by using Access Control Lists (ACLs). It aims to achieve benefits in the performance, cost, and manageability for the emerging application demands.

2.2. Existing active storage approaches

Active storage [6,7,21–25], which enables computation inside storage devices, has long been an important way to make device intelligent and optimize the system performance. In the earliest work, researchers developed various database machines [8] to increase the performance of database application by exploiting the processing power within the disk arm. These machines failed to gain wide acceptance as they used non-commodity hardware and the performance gains were limited. With the development of the VLSI (Very Large Scale Integrated circuit) technology that makes it possible for the disk drive to have more powerful processing capability, researchers proposed the active disk project [6,7] to re-examine the database machine work. By partitioning applications (e.g., data mining, image processing) into the host and disk portions, the Active Disk system is able to obtain higher throughput and less response time. In response to the storage and computational demand for DSS (Decision Support Systems) and data warehousing workloads, Keeton et al. [9] presented a computer architecture that utilizes "intelligent" disks, which exploit the low-cost embedded processing capability and improve cost–performance by offloading general-purpose computation from expensive desktop processors. The MapReduce [26,27] software framework also employs a concept similar to active storage. It splits a large dataset into many pieces and distributes them into many commodity-hardware computers that then process the data locally, and merges the results into the output. In recent work, Son et al. [25] proposed to deploy active storage technology on parallel I/O software stack by extending the MPI-IO interface, and Chen et al. [28,29] further improved their work by exploring the issues on resource contention and data dependency. Another important trend is that SSD (but not HDD) has been gradually suggested to do the active storage work because of the high performance on random writes and multiple I/O channels to flash memory [30,31].

Most of the above work is built on the storage systems based on the block-level interface. Since object-based storage technology may be the next wave in the storage field, a lot of studies have gradually focused on building the active storage system on object-based storage platforms. Huston et al. [32] presented *diamond*, an active storage architecture designed to address the issue of searching non-indexed data from the massive storage system. This system uses the concept of object-based storage, such as object attributes, to perform semantic filter processing in the device. Piernas et al. [33] presented an active storage framework for Lustre [14], which is implemented in user space and proves to be faster and more portable than the previous kernel-space version [34]. Our work is closely related to Devulapalli et al.'s [35] and Runde et al.'s [36]. The former uses sandbox to isolate the execute code and directly schedules the code to execute by specifying the method id. The latter further employs multiple execution engines (e.g., C and Java) to support code execution. However, they do not take into account transparent and multi-granularity processing, concrete methods to enable the code integrity and access control for the code execution, or the load balance management.

In our prior work [37], we gave a basic implementation and evaluation of the Oasis framework. In this work, we further design the signature scheme to ensure the integrity of the execution code, and propose to enable efficient computation workload partition between host and OSD. The adaptive computation workload partition is similar to the previous work [32,38] that enables dynamic function placement among different processing nodes, but is more straight-forward and easier to implement. Then, we make a detailed comparison between Oasis and the recent RPC-based active storage approach [36]. In addition, we discuss how to integrate Oasis with the promising Seagate Kinetic object storage platform and analyze the performance results.

3. Oasis design and implementation

In this section, first we will state the overall architecture of Oasis. Then we will elaborate the details on design and implementations of Oasis.

3.1. Architecture overview

Fig. 1 shows the architecture of Oasis. As depicted in Fig. 1, Oasis consists of five major modules, namely, the Object Command Handler, the Object Filesystem, the Association Check, the Function Scheduler and the Function Object Run-time Environment. *Object Command Handler* gets and analyzes OSD commands from the client and forwards them to the *Object Filesystem* that is responsible

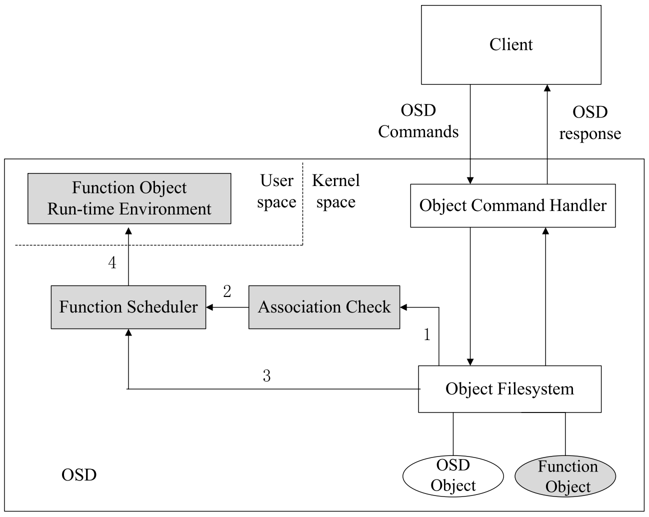


Fig. 1. Architecture of Oasis.

for reading and writing the object data, as well as performing the management of OSD objects and function objects that represent the offloaded application functions. *Association Check* is responsible for checking whether there exists any function object associated with the OSD object that is being read or written, reading the function object ID and parameters from the OSD objects' attributes if the association exists, and then passing these information to the *Function Scheduler* which is responsible for scheduling the related function objects to execute. Currently, the *Function Scheduler* performs schedule work on a first come first serve basis. However, the *Function Scheduler* also monitors the OSD system resource (e.g., CPU utilization) and can send the computation tasks to the client whenever the OSD workload is already very heavy. The *Function Object Run-time Environment* provides the necessary platform (e.g., virtual machine) for the execution of function objects in the user space. Oasis provides access control for the execution of the function objects by simply extending the Permission Bit Mask of the capability (see Section 3.4). The execution results will be written to the local disk or returned to the client. Note that Oasis does not have to modify object interface between OSD and the client. It also has no impact on any server components (e.g., metadata server) in a distributed system.

3.2. Function object

According to the OSD specification [39], the OSD objects are either used for storing user data (i.e., user object) or used for addressing and retrieving user data (i.e., root object, partition object and collection object). The function object is suggested to hold the offloaded application function (e.g., compression, classification, etc.).

A function object is identified by a function object ID and also contains attributes that describe the basic information of the function object (e.g., creation time, access time, etc.). All the function objects are motivated to be executed in OSD to perform operations or analyses on user objects. A function object can be written using the C programming language or a cross-platform language such as Tcl/Python script or Java, and the OSD needs to implement the script interpreter or virtual machine to execute the corresponding functions. Fig. 2 illustrates a piece of C and Java code for a function object that performs data filtering respectively. Both of them retrieve the data from the input stream, process the data and pass the result to the output stream. Except for the input stream and output stream, such as a file, a buffer or a pipe, there is no other way to communicate with the outside operation system.

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
 int main(int argc, char *argv[])
{
 char ch[50];
 long content=0;
 FILE *instream;
 FILE *outstream;

 if((outstream=fopen(argv[2],
 “w+” ))==NULL)
  return -1;
 if((instream=fopen(argv[1], “r” ))!=NULL)
   {
    do{
         if(fgets(ch,50,instream)==NULL)
            break;
         if(strcmp(ch, “\n” )==0)
            continue;
         else
            {
             content=atoi(ch);
             if(content<100)
                fprintf(outstream, “%s” ,ch);
            }
          }while(!feof(instream));
     }
   fclose(instream);
   fclose(outsteam);
 return 0;
}
```

(a) The function object using C code

```java
import java.io.*;
import java.lang.Integer;
import java.awt.*;
 Public class sort
{
 Public static void main(String[] args)
 throws IOException{

 File inputFile=new File(args[0]);
 File outputFile=new File(args[1]);
 FileReader data=new FileReader(inputFile);
 FileWriter result=new FileWriter(outputFile);

BufferedReader br=new BufferedReader(data);
String s;
 int i;

    while((s=br.readline())!=null){
         if(s.length()!=0){
            i=Integer.parseInt(s);
         if(i<100){
            result.write(s);
            s= “\n” ;
            result.write(s);
                }
            }
       }
    data.close();
    result.close();
  }
}
```

(b) The function object using Java code

Fig. 2. C and Java code for a function object that performs data filtering.

3.3. Association

Oasis allows users to associate a function object with an OSD object by saving the function object's ID and its parameters (e.g., encryption keys) in the OSD object's attributes. Such an association design gains several salient advantages. First, the association operation makes it possible to execute function objects during reading or writing OSD objects, making the data processing in the device completely transparent to the user. Second, this approach provides a simple and convenient way for users to flexibly apply different application functions to different kinds of files. For example, the user can apply an edge detection algorithm to an image file to acquire the edge feature of the image by associating the function object that represents the edge detection algorithm with the user object that represents the image file, while for a database file that contains millions of records, the user can apply an efficient database query to it by associating the function object that represents the database query with the user object that represents the database file. Third, associating function objects with different kinds of OSD objects can support different processing granularity. For example, associating a function object with the root object will affect the whole OSD logical unit, while associating a function object with a partition object will only affect all the sub-partitions and files in it. This flexibility allows a user to adopt the granularity that best fits the application.

Oasis also supports chaining multiple function objects. For example, to compress a file before encrypting it, a user can associate multiple function objects with a single user object.

Fig. 3 shows an example of the association between function objects and OSD objects in Oasis. File1 is mapped to user object1 and file2 is mapped to user object2, while the directory /dir2 is mapped to partition object1. Associating the edge detection operation with partition object1 will make all user objects mapped from image files under /dir2 (such as file3 and file4) be processed by the edge detection function using parameters4. Associating the sort operation with user object2 will make user object2 mapped from file2 be processed by the sort function using parameters3. For user object1, two kinds of operations, the compression and encryption that are identified by function object1's and function object2's ID respectively, have been associated with it. This makes user object1 be compressed first using parameters1 and then be encrypted using parameters2 in the write process, while the reverse processing steps will occur in the read process.

3.4. Security considerations

Executing a function object in an OSD can raise serious security risks, which are mainly caused by two aspects: bad code in function objects and illegal users who execute the function objects.

Oasis allows the function objects developed by both the OSD vendors and common users. The vendors have professional knowledge and tools to write and validate the code. For common users, they can also customize function objects according to their practical application needs. However, the vendors provide guidelines and templates for users to write the code, and also provide a set of code test method (such as software fault isolation [40] and proof-carrying code [41]) to guarantee that the code is completely safe.

Oasis also employs digital signature method to ensure the integrity of the code. The administrator controls what function object can run on the OSD and can allow a signed function object by installing the vendors or users' certificates. Of course, the administrator needs to review and test each function object thoroughly before signing it. If a user wants to run a customized function object, she needs to submit it to the administrator to test and obtain a digital signature before she can run it on the OSD.

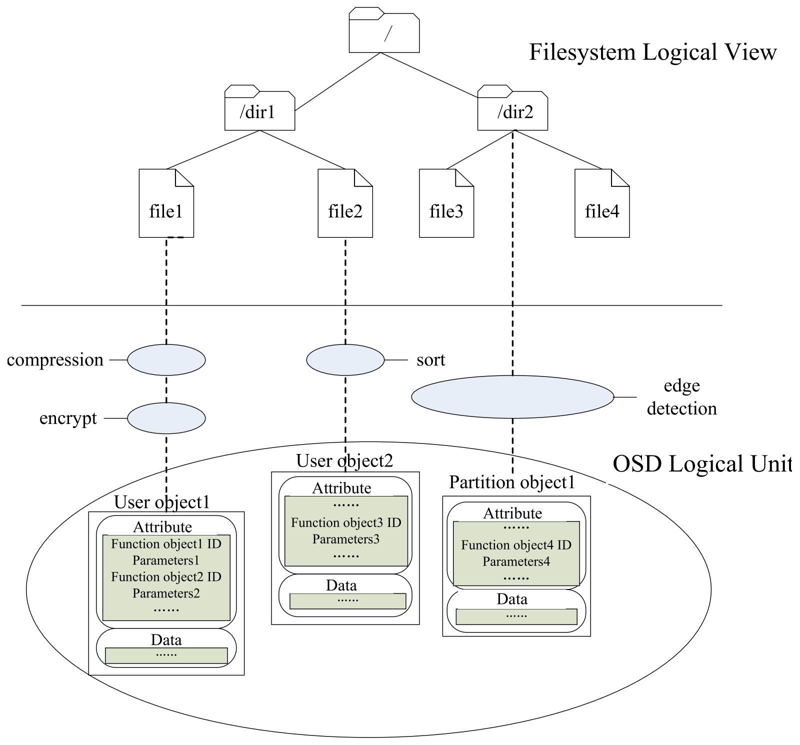


Fig. 3. Mapping from file to object and an example of the association between function objects and OSD objects in Oasis.

Table 1
Permissions bit mask format.

Bit	7	6	5	4	3	2	1	0
49	READ	WRITE	GET_ATTR	SET_ATTR	CREATE	REMOVE	OBJ_MGMT	APPEND
50	DEV_MGMT	GLOBAL	POL/SEC	M_OBJECT	QUERY	GBL_REM	**FUNC_EXE**	Reserved

In addition, Oasis can use the existing OSD security model to authorize the execution of the function object. For instance, a typical OSD security model relies on the capability that contains the Permissions Bit Mask fields of various operations. By simply adding a FUNC_EXE bit (see Table 1) to the Permissions Bit Mask field in the capability, Oasis provides access control for the execution of the function objects. A FUNC_EXE bit set to one allows the function object to be executed on a user object, while a FUNC_EXE bit set to zero prohibits the execution of the function object on a user object. A client wishing to access an OSD, requests such an extended capability from a metadata server and sends it to the OSD as the part of the command. The OSD can then use this capability to authorize the execution of the function object, thus efficiently preventing unauthorized users from intentionally destroying the system.

Even with the measures mentioned above, exceptions or even crashes during the execution of code inside OSD are inevitable. In this case, the client will re-send the READ/WRITE request to invoke the execution of function objects. If that still does not work, the system administrator will analyze the crash reason and upload a new function object to the OSD if possible.

3.5. Dynamic partition of computation

Oasis can partition the computation workload between the client and OSD according to the processing capability available in the client and OSD. Oasis monitors the CPU utilization and network workload in both client and OSD. It acquires these information from the system kernel or /proc file system every half seconds. Whenever a READ/WRITE request arrives at OSD, Oasis judges whether the task should be processed in OSD or client according to the CPU utilization and network workload information. If the workload is already very heavy in OSD and the network is not so congested, some of the computation workload will be transferred to client for processing.

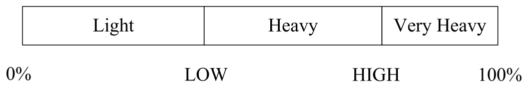


Fig. 4. CPU utilization partition.

For CPU utilization, we define two threshold: LOW and HIGH, as shown in Fig. 4. If the CPU utilization is lower than LOW, the CPU workload is not heavy. If the CPU utilization is between LOW and HIGH, the CPU workload is a little heavy. If the CPU utilization is higher than HIGH, the CPU workload is very heavy.

For network utilization, we first make statistics of the number of bytes per second sent from the OSD side, then divide by the maximum number calculated according to the system hardware information. If the value exceeds a predefined threshold (e.g., 0.8), we consider the network as congested.

As the CPU and network information we acquire may only record information half seconds ago (not exactly current status), so we do not take the information we acquire directly as CPU or network utilization, but instead, we use the following formula (1). CPU_{now} is the CPU utilization we directly acquire from the system. CPU_n is the CPU utilization we use to judge whether the computation task should be processed in OSD or client. CPU_{n-1} is the CPU utilization we computed at last time. This means that we compute the CPU utilization by considering both the CPU utilization we used the last time and the CPU information we currently acquire from system status information. p is a tunable weight factor that we can adjust to more accurately reflect the true CPU utilization.

$$CPU_n = p * CPU_{n-1} + (1 - p) * CPU_{now}. \tag{1}$$

Assumes the CPU utilization on the client side is CPU1 and the CPU utilization on the OSD side is CPU2. Algorithm 1 shows how to judge whether the computation should be done in OSD or client. Note that Oasis uses OSD Data-Out Buffer encapsulated in the OSD commands to transfer the client CPU information to OSD, and uses Data-In Buffer to transfer the computation task from OSD to client once the computation is judged to be done in client.

Algorithm 1 Adaptive Computation Partition algorithm. Function:Judge(CPU1,CPU2,LOW,HIGH)

```
Input: the CPU utilization on the client side, CPU1
Input: the CPU utilization on the OSD side, CPU2
Input: CPU threshold on the OSD side, LOW, HIGH
Output: The result for whether the computation task should be
    done in OSD or client
 1: if CPU2<LOW then
 2:   the computation task should be done in OSD
 3: else if CPU2>HIGH then
 4:   if CPU2 does not exceed HIGH too much && the network is
      congested then
 5:     the computation task should be done in OSD
 6:   else
 7:     the computation task should be sent to client for
        processing
 8:   end if
 9: else
10:   if CPU2>>CPU1 && network is not congested then
11:     the computation task should be sent to client for
        processing
12:   else
13:     the computation task should be done in OSD
14:   end if
15: end if
```

.

1. If CPU2 is lower than LOW, this means the workload on the OSD side is lightweight, in order to make full use of the computing resources in OSD, the computation task should be put on the OSD side for processing.
2. If CPU2 is higher than HIGH, this means the workload on the OSD side is very heavy. If CPU2 does not exceed HIGH too much (e.g., $(CPU2 - HIGH)/HIGH < 0.1$), OSD does not lose the processing capacity. If the network is congested at this time, we still put the computation task on the OSD side, otherwise, we send the computation task to client for processing. But if CPU2 far exceeds HIGH (e.g., $(CPU2 - HIGH)/HIGH > 0.1$), even if we put the workload in OSD, the OSD cannot process it properly. So we send the computation task to client.
3. If CPU2 is between LOW and HIGH, we compare CPU2 with CPU1. If CPU2 is lower than CPU1, i.e., the workload on the OSD side is lighter than the client side, we put the computation task in OSD for processing. If CPU2 is higher than CPU1, but they are very close in number (e.g., $(CPU2 - CPU1)/CPU1 < 0.1$), we still put the computation task on the OSD side. This is because sending the computation task to client can occupy network bandwidth. If CPU2 far exceeds CPU1 (e.g., $(CPU2 - CPU1)/CPU1 > 0.1$), we judge whether the task is processed in OSD or not according to the network congestion status. If the network is congested, we put the task on the OSD side, otherwise, we transfer the computation task to client for processing.

3.6. Implementation details

We prototyped Oasis on the Intel OSD reference implementation (REFv20) [42], which includes an initiator on the host side and a target on the OSD side (for one OSD). The initiator contains an OSD file system (OSDFS), an upper level OSD driver and an iSCSI device driver. It communicates with the targets on the OSDs through OSD commands. All the files and directories are stored as objects in the OSDs.

We implemented the function object in C and Java programming language. Both the C and Java code are compiled first before they are downloaded to the OSD. Upon receiving a piece of C or Java code, the OSD automatically converts them to function objects, and then assigns a function object ID for each function object. In accordance with the association approach outlined in Section 3.3, they will be scheduled during the read or write process. In our system, we apply a Java virtual machine in the Linux operating system platform. Once the function object is scheduled, the Java byte code will be interpreted to run. In Section 5.3, we will specifically explore the execution efficiency of these two kinds of code by running tests on a variety of applications to test Java's performance overhead when compared to C.

The Adaptive Computation Partition algorithm is also written as a function object. The *Function Scheduler* module (see Fig. 1) passes the system resource information (e.g., CPU utilization) to it and invokes its execution. It also executes in user space like any other function objects. However, its execution result will be returned to the *Function Scheduler* module, which then schedules other function objects to process the computation task in OSD or sends the computation task to client.

4. Comparison with other frameworks

This section compares and contrasts Oasis with the implementation of a few OSD-based active storage frameworks. Because each framework is built on different hardware and software platform, it is unfair to directly compare the performance of Oasis to these frameworks. The Oasis work has many similarities with the work such as RPC-based approach [36]. For instance, they both allow code to be downloaded from the client side and support both C and Java execution engines. However, there exist some essential differences, particularly with respect to the programme model and system design.

Execution mode: The RPC-based approach triggers a method (i.e., application function) to execute by explicitly specifying the method ID and passing the arguments to the execution code. It requires users to have a full knowledge of the methods and their IDs. The Oasis frees users from remembering the details on this. It associates a function object with an OSD object by saving the function object's ID and its parameters in the OSD object's attributes. This makes it possible to invoke the function object to execute during reading or writing the OSD object, thus making the data processing in the OSD device completely transparent to the user.

Code integrity and security: The RPC-based approach has ensured the security of code execution by employing the sandbox approach which limits the resource (e.g., a particular file directory and system library) that the code can access. In addition, the approach can also enable the multi-process execution simultaneously.

Oasis further strengthens the code by checking the integrity of the code using the digital signature. Only the function objects that have been signed by the administrator can be executed on the OSD. In addition, Oasis provides the access control for the function objects by extending the OSD capabilities.

Table 2
Characteristics of data analysis applications.

Name	Description	Input data	Ratio of data filtering
Database selection	Non-index select operation that applies to the entire dataset and returns the records that match a given search condition.	1.77 GB(33 million line records, each of which is a double.)	87.4%
Edge detection	This application employs Sobel edge detection algorithm [43] to perform convolution operation on entire images and extract the key features (i.e., edge) of them.	584.0 MB(10 000 images, each of which is a 8-bit map of 59.8 k)	96.7%
Blowfish decryption	This application employs the blowfish algorithm developed by Bruce Schneier [44] to decrypt an 8-byte record each time.	800 MB(100 million line records)	0

Functionality: The RPC-based approach has focused on the execution engine and code security. Oasis has provided more functionalities such as multi-granularity processing and load-balance management. The multi-granularity process property allows Oasis to process any set of OSD objects at a time, and the load-balance management property makes it easy to eliminate traffic when OSDs are heavily loaded. These functionalities make Oasis more practically used.

5. Evaluation

In this section, we will first evaluate the performance of Oasis through three kinds of widely used data analysis applications, i.e., Database Selection, Edge Detection and Blowfish Decryption respectively, then we evaluate how adaptive computation partition between OSD and client can further improve the Oasis performance. Lastly, we analyze the overhead in building an Oasis system from the aspects of system implementation and management.

5.1. Experimental setup

Our experiment test bed consisted of a host (or client) and 1, 2 or 4 OSDs. All of these nodes have the same hardware components, each with one Intel 604-pin EM64T Xeon 3.0 GHz processor, 512 MB PC2700 DDR-SDRAM physical memory and a 250 GB disk. The host and OSDs are connected via 1 Gbps Ethernet. All of these machines run Red Hat Linux 2.4.20. We use the low version operating system to emulate the limited software environment in OSDs where function objects execute.

5.2. Methodology and workload

We evaluate the performance of Oasis by running three applications shown in Table 2. We choose these applications because they are representative of the data analysis applications in the real world and are widely used in various fields. For example, Database Selection is one of the most important query operations in the database system that apply to the entire dataset and return only the records that match a given search condition. Edge Detection is an image processing algorithm that detects the edges or corners of "objects" in a scene (e.g., this application can detect the facial features of individuals in an image). Blowfish Decryption is an encryption algorithm that decrypts data in 8-byte blocks and is widely used in software such as SSH and in operating systems such as OpenBSD. We briefly describe these applications in the second column of Table 2. It should be noted that, we do not currently employ a real database in the OSD system. Instead, we have developed a filter applet that filters the records according to a certain degree of selectivity (e.g., applying a filter applet with a selectivity factor of ten to the dataset will return 1/10 of the total amount of data) to simulate the non-index operation that applies to the entire dataset. The third column shows the specified dataset used in each application. In the last column, we give how much percent of the input dataset would be filtered in the OSD-side. For example, Edge Detection shows the maximum amount of data filtering of 96.7%, while Blowfish Decryption does not filter any data.

In an OSD, all of these three data analysis algorithms are encapsulated into function objects and have their object IDs for reference. For the dataset, both the database records and encryption items are contained in a file that is striped into OSD objects across all the OSDs. All the images to be processed are also evenly distributed across all the OSDs and each image in the OSDs is taken as an OSD object. We begin our test by running several processes on the host at the same time, and each process is responsible for reading or writing the OSD objects in an OSD. The function objects will be invoked to execute if they are already associated with the OSD objects that are being read or written. They acquire the execution parameters (e.g., encryption keys and selection conditions) from the attributes of the OSD objects and are executed in the user space to avoid disturbing the system kernel.

5.3. Application performance

We evaluate the performance of Oasis by first analyzing the improvement on overall execution time, and then we look at the sensitivity analysis results, including the number of OSDs, the programming language of function objects, and the number of function objects that are associated with the same OSD object. We will mainly analyze two cases: Traditional Storage (TS) and Active Storage (AS). The former means that the data stored in an OSD should be shipped to the host to process, while the latter means that the data should be processed in the OSD using the function object. It should be noted that, in the following experiments, all the executed function objects are implemented in C language unless otherwise indicated (e.g., in Fig. 7).
(a) Performance improvement.

Fig. 5 shows the execution time breakdown for different applications using one host and one OSD. We simply divide the execution time into a process part and a transfer part. The process part indicates the execution time of the application function, including the overhead of copying data from kernel space to user space, while the transfer part measures the communication overhead on the interconnect network between host and OSD. As the figure shows, the AS scheme improves performance significantly on both Database Selection and Edge Detection applications, 83.8% and 70.4% respectively. One can see that these improvements can be attributed to the dramatic time reduction on the Transfer part, from 292.6 s to 3.9 s in Database Selection and from 198.0 s to 0.74 s in Edge Detection application. This is because most of the data has been filtered on the OSD-side, which results in less data transferred from the OSD to the host. While, the performance of the AS scheme and the TS scheme in Blowfish Decryption are almost the same, the reason is that Blowfish Decryption algorithm does not filter data on the OSD-side (see Table 2).

We then look at the scalability of Oasis when the number of OSDs is increased. Fig. 6 shows the performance of three applications in Oasis for the 1-OSD, 2-OSD and 4-OSD configurations, respectively. One can see that the performance of TS is scalable with the increase in the number of OSDs for all of the three applications. This is because the parallelism in the OSDs results in a great decrease in the transmission time over the interconnect network. We

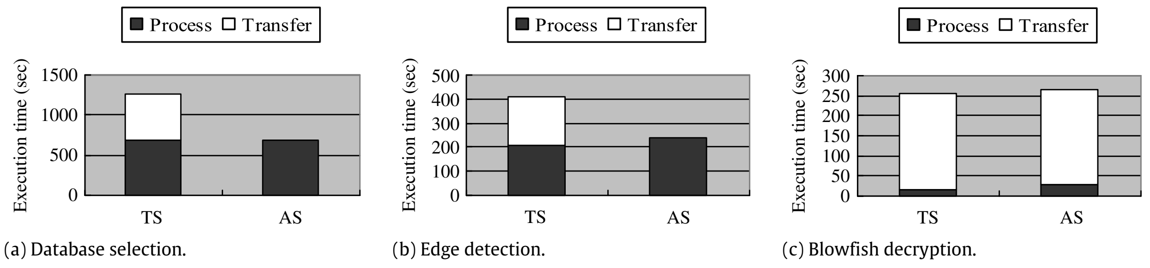


Fig. 5. Execution time breakdown for different applications with one OSD.

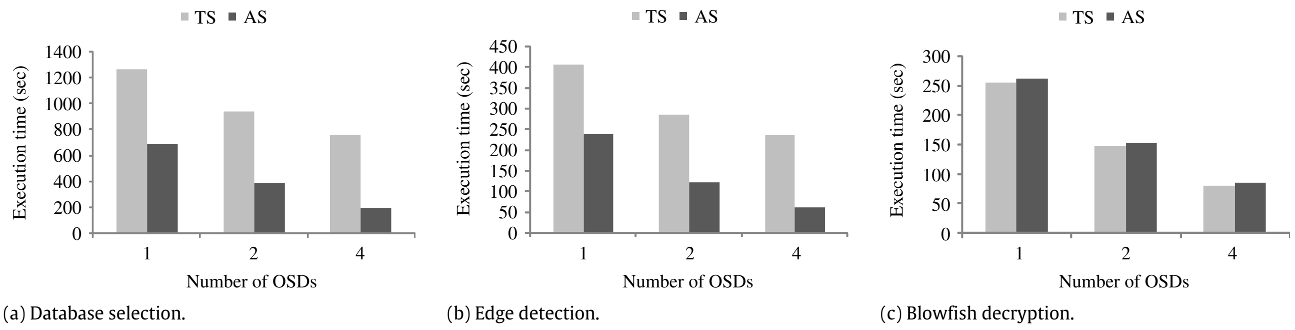


Fig. 6. Execution time for different applications with different number of OSDs.

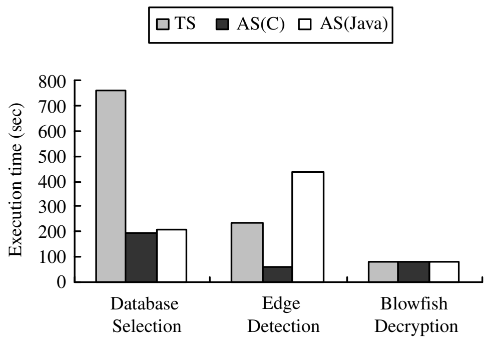


Fig. 7. Execution time for all three kinds of applications in the C and Java programming language. AS(C) means that the executed function object is implemented in C language, while AS(Java) means that the executed function object is implemented in Java language. In this experiment for all three applications, we use four OSDs for execution.

also observe that, in Fig. 6(a) and (b), the AS scheme outperforms the TS scheme significantly due to the reduction in data transfer, even with a single active storage node. We see that these improvements are consistent with the increase in the number of OSDs. For Blowfish Decryption application (see Fig. 6(c)), as there exists no data reduction in the data transfer, AS scheme achieves comparable performance with the TS scheme even though the number of OSDs increases.

(b) Impact of language of function object.

The C-powered function object may perform well in execution efficiency, but not well in portability when compared to a Java-powered function object. To analyze the impact of language of function object, we repeated the experiments with all three kinds of applications in three cases: processing data in host (TS), executing function objects written in C language in OSD (AS(C)) and executing function objects written in Java language in OSD (AS(Java)).

As illustrated in Fig. 7, AS(C) and AS(Java) achieve comparable performance for both the Database Selection and Blowfish Decryption applications, while for Edge Detection, AS(C) far outperforms AS(Java) by a factor of 6.12. It should be noted that TS in the Edge Detection implementation also outperforms AS(Java) by a factor of 0.84. The reason for this is that, since a large number of I/O operations are required for the Edge Detection algorithm to generate the output image, the algorithm implementation using the Java language is significantly slower than the implementation using the C language. And even such performance degradation with the Java implementation may compromise the benefits of data reduction in the Edge Detection application achieved by the active storage technology.

However, for the application such as Blowfish Decryption, the algorithm is basically composed of the ADD and XOR instruction (not I/O bound), so the algorithms using C language and using Java language will result in a comparable speed, implying that for non-I/O intensive applications, both the C and Java implementations of function objects can achieve comparable performance, while for I/O intensive applications, achieving a cross-platform implementation with the Java programming language means a potential performance bottleneck in the active storage system.

(c) Impact of multiple function objects.

The above evaluation focuses on applying one application function on user data each time. However, sometimes, users may want to perform multiple operations on user data at a time. For example, users may need to first decrypt a large piece of data and then select the data that they want. Oasis supports function composition by associating multiple function objects with a single OSD object using object attributes (see Fig. 3).

Fig. 8 shows the execution time on a hybrid application that stacks a Database Selection service on a Blowfish Decryption service with different number of OSDs. We evaluate the impact of multiple function objects by partitioning this hybrid application between the host and OSDs, namely, processing this hybrid application on host (TS), first decryption on OSD and then selection on host (AS(one)) and processing this hybrid application on OSDs (AS(two)). The results show that AS(one) does not improve the system performance, as a matter of fact, decreases slightly by 0.7%–3.9% when compared to TS. The reason is that, offloading the Blowfish Decryption application to the OSDs does not bring data reduction across I/O interconnect, but incurs a small overhead over the traditional object storage system. However, the performance

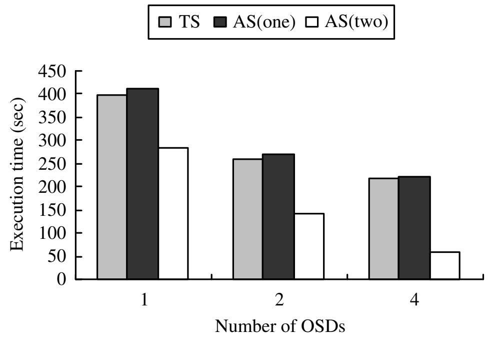


Fig. 8. Execution time on a hybrid application that stacks a Database Selection service on a Blowfish Decryption service with different number of OSDs. We analyze three cases: processing this hybrid application on host (TS), first decryption on OSD and then selection on host (AS(one)) and processing this hybrid application on OSDs (AS(two)).

of AS(two) significantly outperforms TS by a factor of 0.41 to 2.63, and also outperforms AS(one) by a factor of 0.45 to 2.66. This shows that offloading Database Selection to the OSDs can significantly improve the system performance. Again, this is because the Database Selection application reduces the data needing to transmit over the interconnect network by filtering data on the OSD side. This indicates that, for a hybrid application that is composed of multiply applications, only applications that can make data reduction across the I/O interconnect can really benefit system performance.

5.4. *Performance with adaptive computation partition*

The previous experiments show that active storage technology can improve the whole system performance by filtering the data on the OSD side. However, it is not efficient to process all the data inside the OSD especially when OSD is heavily loaded.

Fig. 9 shows the execution time of processing different sizes of data when an extra process is already running in the OSD and has occupied a certain ratio of CPU. We compare the performance of three cases: Traditional Storage (TS), Active Storage (AS), and Adaptive Active Storage (AAS). AAS employs the Adaptive Computation Partition algorithm that can judge whether the computation workload is suited to be processed in OSD or sent to the client for processing according to the status of workload.

In Fig. 9(a), AS performs better than TS when processing the same size of data. This is because AS can filter much of the data on the OSD side, thus greatly reducing the amount of data transferred over the network, resulting in the whole execution time smaller than TS. AAS has a comparable performance with AS. This is because the CPU resource in OSD is idle (i.e., no other process is running). The adaptive computation partition algorithm judges that the computation task can be processed in OSD. In this case, the whole processing flow in ASS is nearly the same as AS, so their performance are almost the same.

In Fig. 9(b), the tests run in a system where an extra load has already taken up 25% of CPU on the OSD side. In this case, TS still performs the worst, AS and AAS have a comparable performance. However, the execution time in both AS and AAS cases has increased a lot compared to those in Fig. 9(a). This change occurs because the extra load in OSD consumes a certain amount of CPU, thus resulting in insufficient CPU resource to process all the computation tasks, leading to an increase in processing time. However, AAS still processes a large number of the tasks in OSD as the workload in OSD is still in a lightweight level (only 25% of CPU is occupied). So the performance of AAS is close to the AS.

In Fig. 9(c), the tests run when half of the CPU is occupied by an extra task. We can see that the execution time has a significant increase in both AS and AAS when compared to the case in Fig. 9(a). Their performance even drops to the same level as TS. For AS, though filtering data in OSD can reduce the transfer time over the network, the lack of CPU resource in OSD can drag down the whole performance. For AAS, as the performance of AS is close to TS, the performance of AAS is comparable with both AS and TS regardless of processing data in OSD or client. The measurement shows that for AAS, up to 65.2% of the computation task is processed in OSD, and the rest is sent to client for processing.

In Fig. 9(d), the tests run when a majority of CPU is occupied by an extra task. The performance of TS is the best, AS is the worst, and AAS is in between them. This is because the lack of CPU resource has severely restricted the performance of AS. As TS processes data in client, the CPU resource shortage in OSD has nearly no impact on TS. For AAS, as the workload of OSD is very heavy, most of the data are processed in client, so its performance is between TS and AS. The statistic shows that only 10.9% of the computation task is processed in OSD in the AAS case.

From Fig. 9, we can see that AAS has a comparable performance with AS when OSD CPU has been occupied from 0% to 50%, and AAS outperforms AS when OSD CPU has been occupied up to 75%. We further aggravate the workload by initiating two processes in the client simultaneously as shown in Fig. 10. AAS outperforms AS and TS in nearly all the cases. This is due to the resource contention that has degraded the whole system performance in both of the TS and AS cases. For instance, two processes have to compete for the network bandwidth and the client CPU resource in the TS case, and compete for the OSD CPU resource in the AS case. In contrast, AAS can achieve the best performance by adaptively choosing where to process the computation task. For instance, AAS can alleviate the resource contention to the maximum extent by putting one computation task in the client and another in the OSD.

5.5. *Overhead analysis*

(a) Implementation overhead.

As depicted in Fig. 1, in an Oasis system, the *Association Check* module has to check whether there exists any function object associated with the OSD object that is being read or written by accessing the attributes of the OSD object during every read or write call even when no function object is associated with the OSD object. We evaluate this implementation overhead by comparing the completion time of reading and writing a file with different file sizes in the Oasis implementation when no function object is associated to the user object with the Intel OSD reference implementation under 1 Gbps interconnect network. As shown in Fig. 11, the implementation overhead of Oasis is minimal, between 1.2% to 5.9% for the read operation and 0.6% to 9.9% for the write operation, with the Intel OSD reference implementation as the baseline.

(b) Management overhead.

Oasis manages the function object by cleverly employing the object commands defined in the current T10 OSD standard [39]. Table 3 shows the completion time of various object commands for the management of function objects in Oasis under 100 Mbps interconnect network. For example, it takes 13.6 ms to create a function object with 1 kB size by using the CREATE AND WRITE command, while only 7.8 ms to delete this function object by using the REMOVE command. In summary, it incurs an overhead as little as 2.8 ms to 13.6 ms for managing function objects, implying that Oasis provides an effective and time-saving way to manage the function objects by using the existing object commands.

6. Deployment on the Kinetic object storage platform

This section first describes how Oasis can be integrated with the Kinetic object storage platform, then makes the performance analysis of the Kinetic platform with the Oasis software layer.

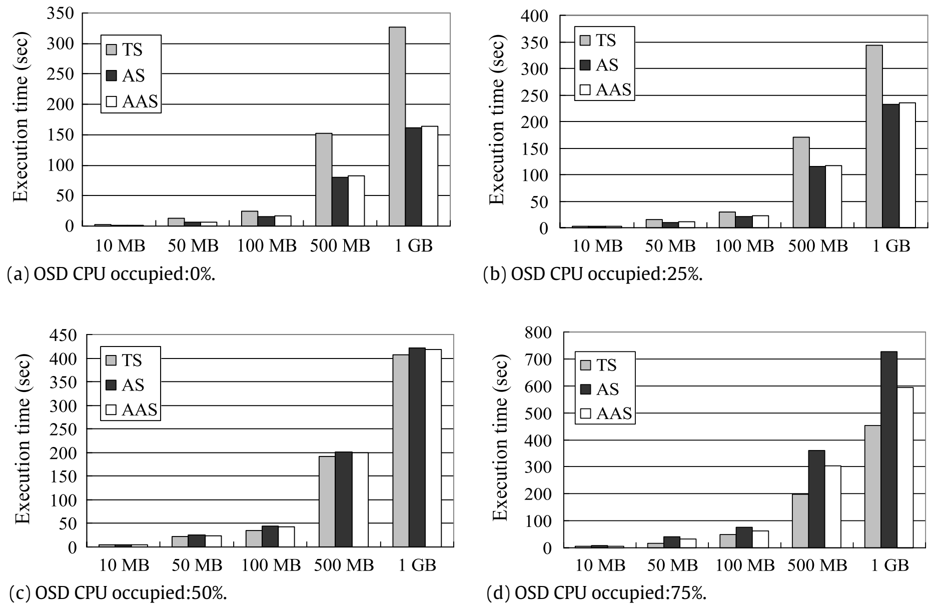


Fig. 9. Execution time of processing different sizes of data when an extra process is already running in the OSD and has occupied a certain ratio of CPU (0%–75%). The application used in this experiment is Database Selection. The size of the data produced after processing is 20% of the original data. To avoid memory becomes bottleneck, we increase the physical memory of both client and OSD to 1 GB in this experiment. All the tests run with one client process and one OSD. For the Adaptive Computation Partition algorithm in AAS, the value of LOW is 0.5, and the value of HIGH is 0.7.

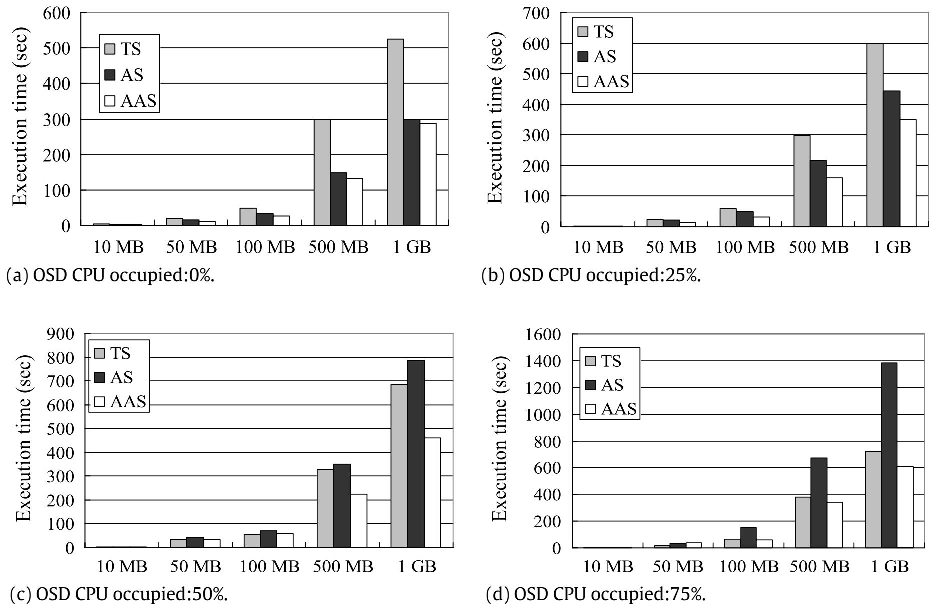


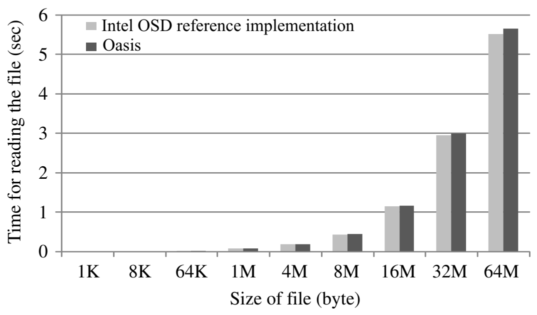
Fig. 10. Execution time of processing different sizes of data when an extra process is already running in the OSD and has occupied a certain ratio of CPU (0%–75%). The application used in this experiment is Database Selection. The size of the data produced after processing is 20% of the original data. To avoid memory becomes bottleneck, we increase the physical memory of both client and OSD to 1 GB in this experiment. All the tests run with two client processes and one OSD. For the Adaptive Computation Partition algorithm in AAS, the value of LOW is 0.5, and the value of HIGH is 0.7.

6.1. Integration with the Kinetic

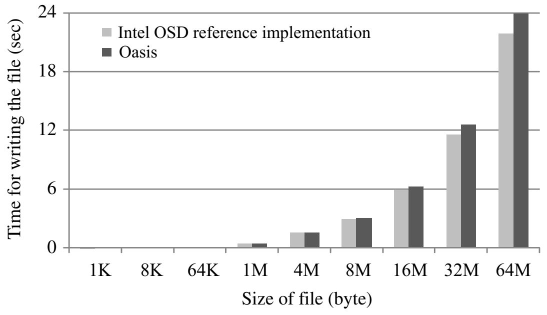
The Oasis object commands can be integrated with the Kinetic key/value API. The traditional CREATE, READ and REMOVE commands used to access the OSD objects or function objects correspond to the PUT, GET and DELETE commands in Kinetic respectively. For example, similar to the CREATE command, when we use PUT command to access the Kinetic device, we can specify the object ID in the key, and the data to write in the value field. The function execution method in Oasis can be also applied to Kinetic device. When we use PUT or GET API to access the OSD objects in Kinetic device, the function objects associated with the OSD objects will be scheduled to execute.

The security scheme on Oasis can be also applied to Kinetic device. For instance, the Kinetic system administrator can use signature technology to sign each function object that can be executed in Kinetic device. In addition, the Access Control List in Kinetic can authorize the legal users to execute the function object.

Fig. 12 shows the enforced Kinetic object storage platform that can process data intensive computation inside Kinetic OSD devices. The Oasis software layer interacts with the Kinetic client using the normal object commands, so it has no impact on the original Kinetic framework. As we have stated, the object command, function execution, and security scheme can be all integrated into the Kinetic framework which has been designed for the cloud storage platform. We believe the classical parallel computing



(a) Read overhead.



(b) Write overhead.

Fig. 11. Implementation overhead of Oasis over the Intel OSD reference implementation.

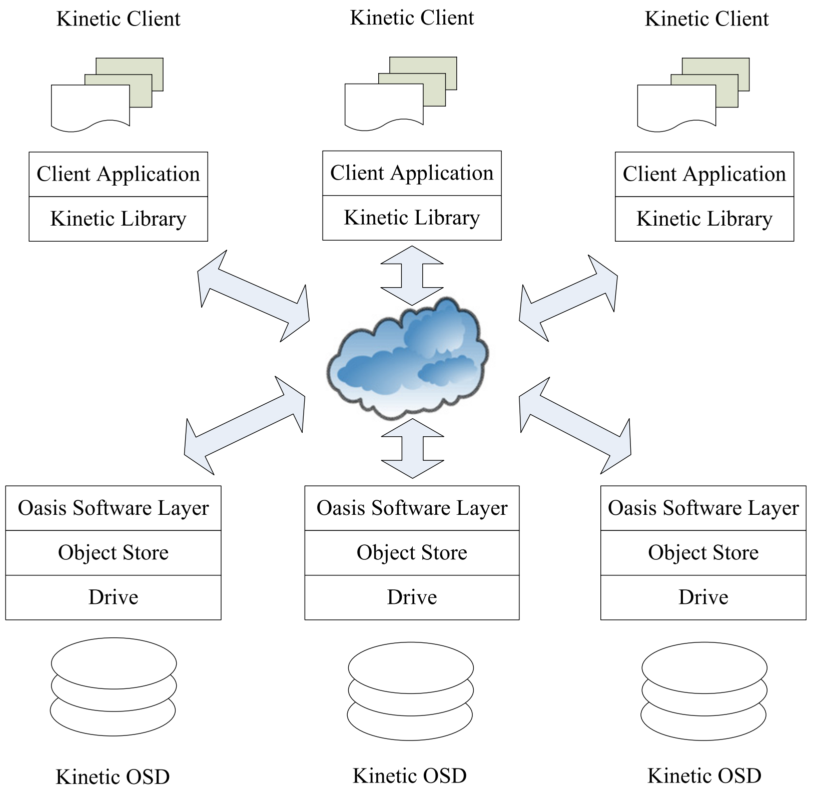


Fig. 12. Enforced Kinetic object storage platform that can process data intensive computation inside Kinetic OSD devices.

Table 3
Commands overhead for managing function object.

Number	Management description	Object commands	Completion time (ms)
1	Create a 1 kB function object	CREATE AND WRITE	13.6
2	Associate a function object	SET ATTRIBUTES	2.8
3	retrieve a 1 kB association information	GET ATTRIBUTES	12.1
4	List 512 bytes function objects	LIST	4.5
5	Delete a 1 kB function object	REMOVE	7.8

paradigm (e.g., Map/Reduce) can be applied to this platform. This will be our future work.

6.2. Performance analysis

As Kinetic cloud platform is composed of different types of OSD devices with different processing power, and different interconnects with different transfer bandwidth between the OSDs and the Kinetic clients, the performance measurement in Kinetic OSD platform is much more complicated than what we usually do in a local distributed file system where we assume all the OSDs have the same processing power and the transfer bandwidth between each OSD and the client is the same. However, we make a preliminary performance analysis, hoping to give a basic understanding of how Oasis can impact the performance of the Kinetic cloud platform.

To simplify the analysis, we only consider the one client case. Assume the size of the data to be processed in the whole system is D, the number of OSDs in Kinetic platform is m, the processing power (i.e., the size of the data processed per second) of these OSDs is $p_1, p_2, \ldots, p_m$, the data transfer bandwidth between these OSDs and the client is $T_1, T_2, \ldots, T_m$, and the processing power of the client is P. For the traditional case (without Oasis), the data in OSDs are first transferred to the client which then processes them locally. For the active storage case, the data are first processed in OSDs, then the process results are transferred to the client.

For the traditional case, the size of the data transferred from each OSD to the client is D/m. Assume the data in the client can be processed only after all the data have been transferred from the OSDs side. So the whole execution time depends on the smallest transfer bandwidth. The whole execution time is described as the following formula (2).

$$Time(traditional) = MAX\left\{\frac{D/m}{T_1}, \frac{D/m}{T_2}, \ldots, \frac{D/m}{T_m}\right\} + \frac{D}{P}. \tag{2}$$

For the active storage case, assume the size of the processing result generated in each OSD side is d. As the data processing and transfer are in parallel for all the OSDs, the whole execution time depends on the one that the combination of OSD processing and transfer is the slowest. It can be described as the following formula (3).

$$Time(active\ storage) = MAX\left\{\left(\frac{D/m}{p_1} + \frac{d}{T_1}\right), \left(\frac{D/m}{p_2} + \frac{d}{T_2}\right), \ldots, \left(\frac{D/m}{p_m} + \frac{d}{T_m}\right)\right\}. \tag{3}$$

So whether to use Oasis in Kinetic can be judged through comparing the performance numbers of these two cases.

7. Discussion

It is a trend that NVMe (Non-volatile Memory Extension) interface will become prevalent in the next decade. We adopt iSCSI in this paper for two reasons. On one hand, the SCSI interface is still widely used in toady's PC and cloud infrastructures. On the other hand, the iSCSI interface can be a choice for the ethernet-based Kinetic cloud platform. The implementation in this paper also provides a reference for NVMe interface, i.e., how data intensive computation can be offloaded to and executed on the underlying device through the utilization of an existing interface.

Another important issue in Oasis-based distributed storage system is data consistency. As we put function objects in each OSD and do not update them unless we have to remove them, the data consistency is not a problem for function object. For other OSD objects, we would like to keep three copies for each object across different OSDs and employ the eventually consistency technologies (e.g., read-your-writes consistency and session consistency) in Oasis-based systems.

8. Conclusions

Oasis, as an object-based active storage framework, plays a significant part in addressing the network bottleneck incurred by big data process and transfer. By exploring and exploiting the advantages of both the scalable object storage architecture and intelligent active storage technology, Oasis addressed the three critical challenges when designing active storage systems as follows: (1) supporting transparently and variable-granularity processing by using object attributes. (2) Ensuring the integrity of execution code via signature scheme and supporting capability-based access control by extending the object storage security model. (3) Supporting adaptive computation workload partition between host and OSD. In addition, Oasis design can be seamlessly integrated with the Seagate Kinetic object storage platform. Experiments using the extensive datasets demonstrate the performance and efficiency of Oasis.

Acknowledgments

We thank Wei Chen for her help with the experimental setup. This work was supported in part by the National Natural Science Foundation of China under Grant No. 61402189, Postdoctoral Foundation of China, and National Basic Research 973 Program of China under Grant No. 2011CB302301. Darrell Long and Yan Li were supported in part by the National Science Foundation under awards CCF-1219163 and CCF-1217648, by the Department of Energy under award DE-FC02-10ER26017/DE-SC0005417, and by the industrial supporters of the Center for Research in Storage Systems, a National Science Foundation I/UCRC.

References

[1] IDC iView, The digital universe decade-are you ready? May 2010.
[2] J. Gantz, D. Reinsel, Digital universe study: Extracting value from chaos, in: Int'l Data Corporation, IDC, June 2011.
[3] M. Armbrust, et al., A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58.
[4] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, J. Thelin, Orleans: Cloud computing for everyone, in: ACM Symp. Cloud Computing, SOCC, Cascais, Portugal, 2011, Article No. 16.
[5] S. Wu, F. Li, S. Mehrotra, B. Ooi, Query optimization for massively parallel data processing, in: ACM Symposium on Cloud Computing, SOCC, Cascais, Portugal, 2011, Article No. 12.
[6] A. Acharya, M. Auysal, J. Saltz, Active disks: Programming model, algorithms and evaluation, in: The 8th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, San Jose, CA, USA, 3–7 October 1998, pp. 81–91.
[7] E. Riedel, G. Gibson, C. Faloutsos, Active storage for large-scale data mining and multimedia, in: The 24th International Conference on Very Large Data Bases, VLDB, New York, NY, USA, 24–27 August 1998, pp. 62–73.
[8] D.J. Dewitt, P. Hawthorn, A performance evaluation of database machine architectures, in: The 7th International Conference on VLDB, Cannes, France, 9–11 September 1981, pp. 199–213.
[9] K. Keeton, D.A. Patterson, J.M. Hellerstein, The case for intelligent disks (IDISKs), SIGMOD Rec. 27 (3) (1998) 42–51.
[10] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick, Intelligent RAM (IRAM): Chips that remember and compute, in: IEEE International Solid-State Circuits Conference, San Francisco, CA, February, 1997, pp. 224–225.
[11] A. Fenn, J. Hughes, Seagate kinetic open storage platform, in: Data Storage Innovation Conference, 22–24 April 2014.
[12] Open-osd initiator, http://www.open-osd.org.
[13] M. Mesnier, G.R. Ganger, E. Riedel, Object-based storage, IEEE Commun. Mag. 41 (8) (2003) 84–91.
[14] Lustre: A scalable high-performance file system, White Paper, Clustre File System, Inc., 2002.
[15] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, B. Zhou, Scalable performance of the panasas parallel file system, in: FAST'08, San Jose, CA, USA, 26–29 February 2008, pp. 17–33.
[16] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, Ceph: A scalable, high-performance distributed file system, in: OSDI'06, Seattle, WA, USA, 6–8 November 2006, pp. 307–320.
[17] D.F. Nagle, G.R. Ganger, J. Butler, G. Goodson, C. Sabol, Network support for network-attached storage, in: Hot Interconnects 1999, Stanford University, Standford, CA, August, 1999.
[18] T. Clark, Designing Storage Area Networks: A Practical Reference for Implementing Fibre Channel and IP SANS, second ed., 2003.
[19] Y. Kang, J. Yang, E.L. Miller, Object-based SCM: An efficient interface for storage class memories, in: Massive Storage Systems and Technologies, Denver, Colorado, USA, 2011, pp. 1–12.
[20] Y. Lu, J. Shu, W. Zheng, Extending the lifetime of flash-based storage through reducing write amplification from file systems, in: The 11th USENIX Conference on File and Storage Technologies, San Jose, CA, 2013, pp. 257–270.
[21] X. Ma, A.N. Reddy, MVSS: an active storage architecture, IEEE Trans. Parallel Distrib. Syst. 14 (10) (2003) 993–1005.
[22] C.W. Smullen, S.R. Tarapore, S. Gurumurthi, P. Ranganathan, M. Uysal, Active storage revisited: The case for power and performance for unstructured data processing applications, in: The 5th Conference on Computing Frontiers, Ischia, Italy, May 5, 2008–May 7, 2008, pp. 293–303.
[23] S.V. Anastasiadis, R.G. Wickremesinghe, J.S. Chase, Lerna: An active storage framework for flexible data access and management, in: The 14th IEEE International Symposium on High Performance Distributed Computing, Research Triangle Park, NC, USA, 24–27 July 2005, pp. 176–187.
[24] Y. Zhang, D. Feng, An active storage system for high performance computing, in: The 22nd International Conference on Advanced Information Networking and Applications, Okinawa, Japan, 25–28 March 2008, pp. 644–651.
[25] S.W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.K. Liao, A. Choudhary, Enabling active storage on parallel I/O software stacks, in: The IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST, Incline Village, NV, USA, 3–7 May 2010, pp. 1–12.

[26] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: The USENIX Symposium on Operating System Design and Implementation, San Francisco, CA, USA, 6–8 December 2004, pp. 137–149.
[27] M. Zahariz, A. Konwinski, A.D. Joseph, R.H. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, in: The USENIX Symposium on Operating System Design and Implementation, 2008, pp. 29–42.
[28] C. Chen, Y. Chen, P.C. Roth, DOSAS: Mitigating the resource contention in active storage systems, in: IEEE International Conference on Cluster Computing, Beijing, China, 24–28 September 2012, pp. 164–172.
[29] C. Chen, Y. Chen, Dynamic active storage for high performance I/O, in: IEEE International Conference on Parallel Processing, Pittsburgh, PA, 10–13 September 2012, pp. 379–388.
[30] Y. Kang, Y. Kee, E.L. Miller, C. Park, Enabling cost-effective data processing with smart SSD, in: The 29th IEEE Symposium on Massive Storage Systems and Technologies, MSST 13, Long Beach, CA, USA, May 2013, pp. 1–12.
[31] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, S. Swanson, Willow: A user-programmable SSD, in: The USENIX Conference on Operating Systems Design and Implementation, 2014, pp. 67–80.
[32] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G.R. Ganger, E. Riedel, A. Ailamaki, Diamond: A storage architecture for early discard in interactive search, in: The 3rd USENIX Conference on File and Storage Technologies, FAST'04, San Francisco, CA, USA, 2004, pp. 73–86.
[33] J. Piernas, J. Nieplocha, E.J. Felix, Evaluation of active storage strategies for the lustre parallel file system, in: SC'07, Reno, NV, United states, November, 2007, pp. 1–10.
[34] E.J. Felix, K. Fox, K. Regimbal, J. Nieplocha, Active storage processing in a parallel file system, in: The 6th LCI International Conference on Linux Clusters: The HPC Revolution, 2006.
[35] A. Devulapalli, I.T. Murugandi, D. Xu, P. Wyckoff, Design of an intelligent object-based storage device. [Online]. Available: http://www.osc.edu/research/network_file/projects/object/papers/istor-tr.pdf.
[36] M. Runde, W.G. Stevens, P. Wortman, J.A. Chandy, An active storage framework for object storage devices, in: The 28th IEEE Conference on Massive Data Storage, San Diego, CA, USA, 16–20 April 2012, pp. 1–12.
[37] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, D.D.E. Long, Y. Kang, Z. Niu, Z. Tan, Design and evaluation of Oasis: An active storage framework based on T10 osd standard, in: The 27th IEEE Symposium on Mass Storage Systems and Technologies, MSST, Denver, Colorado, USA, May, 2011, pp. 1–12.
[38] K. Amiri, D. Petrou, G.R. Ganger, G.A. Gibson, Dynamic function placement for data-intensive cluster computing, in: The USENIX Annual Technical Conference, Berkeley, CA, USA, 2000, pp. 25–25.
[39] SCSI Object-Based Storage Device Commands -2 (OSD-2), T10 Technical Committee, INCITS Std. Project T10/1729-D, Revision 5, January 2009.
[40] R. Wahbe, S. Lucco, T. Anderson, S. Graham, Efficient software-based fault isolation, in: SOSP'93, Asheville, NC, USA, December 1993, pp. 203–216.
[41] G.C. Necula, P. Lee, Safe kernel extensions without run-time checking, in: OSDI'96, Seattle, WA, USA, 1996, pp. 229–243.
[42] Intel Corporation, Intel iscsi reference implementation. [Online]. Available: http://sourceforge.net/projects/intel-iscsi.
[43] Sobel edge detection algorithm. [Online]. Available: http://www.pages.drexel.edu/weg22/edge.html.
[44] Blowfish algorithm. [Online]. Available: http://www.schneier.com/code/bfsh-koc.zip.

Yulai Xie received the B.E. and Ph.D. degrees in computer science from Huazhong University of Science and Technology (HUST), China, in 2007 and 2013, respectively. He was a visiting scholar at the University of California, Santa Cruz in 2010 and a visiting scholar at the Chinese University of Hong Kong in 2015. He is now a postdoc researcher in Wuhan National lab for optoelectronics, China. His research interests mainly include digital provenance, network storage and computer architecture.

Dan Feng received her B.E, M.E. and Ph.D. degrees in Computer Science and Technology from Huazhong University of Science and Technology (HUST), China, in 1991, 1994 and 1997 respectively. She is a professor and director of Data Storage System Division, Wuhan National Lab for Optoelectronics. She is also dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, parallel file systems, disk array and solid state disk. She has over 100 publications in journals and international conferences, including FAST, USENIX ATC, ICDCS, HPDC, SC, ICS and IPDPS. Dr. Feng is a member of IEEE and a member of ACM.

Yan Li is a Ph.D. student working with Professor Darrell D. E. Long at the University of California, Santa Cruz. He is currently working on the performance, Quality of Service, and energy consumption of large-scale storage systems.

Darrell D. E. Long received his B.S. degree in Computer Science from San Diego State University, and his M.S. and Ph.D. from the University of California, San Diego. Dr. Darrell D.E. Long is Professor of Computer Science at the University of California, Santa Cruz. He holds the Kumar Malavalli Endowed Chair of Storage Systems Research and is Director of the Storage Systems Research Center. His current research interests in the storage systems area include high performance storage systems, archival storage systems and energy-efficient storage systems. His research also includes computer system reliability, video-on-demand, applied machine learning, mobile computing and cyber security. Dr. Long is Fellow of IEEE and Fellow of the American Association for the Advancement of Science (AAAS).