International Journal of Database Theory and Application
Vol.9, No.7 (2016), pp.121-134
http://dx.doi.org/10.14257/ijdta.2016.9.7.11

JAS: JVM-Based Active Storage Framework for Object-based Storage Systems

Xiangyu Li[1,4], Shuibing He[1,2,*], Xianbin Xu[1] and Yang Wang[3]

[1] School of computer, Wuhan University, Wuhan, Hubei 430072, China
[2] State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, Hunan 410073, China
Shenzhen Institute of Advanced Technology, China Academy of Science, Shenzhen 518055, China
[4] Wuhan Donghu University, Wuhan, Hubei 430212, China
**Corresponding author: heshuibing@whu.edu.cn*
xylee@whu.edu.cn, xbxu@whu.edu.cn, yang.wang1@siat.ac.cn

Abstract

We propose JAS, a JVM-based active storage framework for object-based storage systems. JAS programs the active storage functions of users as Java codes, and allows them to be executed on different OSD platforms (Operating systems and hardware) without recompiling. JAS offloads the active storage code from a client to the OSD by extending the standard OSD command set, and execute the Java code on the OSD on-demand by triggering them through extended client interfaces. We have implemented JAS under an object-based storage system. Experimental results show that the JVM-based active storage framework has been successfully set up, and this cross-platform design can largely improve system performance.

***Keywords**: Active Storage, Object-Based Storage, Java Virtual Machine*

1. Introduction

I/O access has become one of the major performance issues in modern computer systems. While processor speed has increased nearly 50% per year, the disk performance improvement is only 7% [1]. Despite this large performance gap, a lot of applications are becoming increasingly data intensive. For example, the *astro* program in astronomy, generates tens of gigabytes of data in one run [2]. As a result, these data intensive applications often move a large amount of data between the computing nodes and the storage devices, thus putting unprecedented pressure on both network and storage devices [3-6].

Storage devices or systems are having increasing powerful processors and plenty of memories nowadays, with the advancements of VLSI technology. For example, there are 4-8 core embedded processors and a few gigabytes of DRAM in some commercial disks [7]. Multi-core disks are likely to become more common place, largely due to Moore's law. Unfortunately, the considerable processing capabilities of the storage devices have not been fully utilized in most I/O systems.

Active storage technology has been proposed and is proved to be one of the most effective approaches to reduce the bandwidth requirements between storage and computing nodes [8-12]. By exploiting the under-utilized computing power and memory of storage devices to process data inside the storage devices, active storage can not only reduce the network traffic, but also provide aggregative processing intelligence when multiple devices are used in parallelism.

Due to the benefits of active storage, a variety of research institutions have made essential contributions to the active storage research field, such as Active Disk [9] and IDISK [13]. However, these earlier works are based on the narrow *block* storage interface (*e.g.*, IDE/SCSI), active storage may result in complicated management and high communication overhead, and hence offset the potential of active storage.

Object-based storage [14] has gained enormous popularity in storage area. Because of the benefits of cross-platform data sharing, policy-based security, direct access, and scalability, object-based storage systems have been developed by the industrial community [15-16]. To regulate the more and more sophisticated object-based technology, the object-based storage interface standard [17] (also referred as T10 OSD standard) was developed by the Storage Network Industry Association. With more expressive object interface, there is a potential for object-based storage devices (OSDs) to be more intelligent and effective. To further facilitate the object-based storage, there have been several efforts to integrate active storage into the object-based storage technology [12,18-21].

While effective to improve I/O system performance, most of current object-based active storage studies are designed for homogeneous I/O environments, that is, the clients and the OSDs usually run with similar operating systems or hardware platforms. However, in a practical object-based storage system, the clients and the OSDs may run on a heterogeneous environment, thus the active storage functions could not directly be executed on the OSDs. For example, clients and OSDs can run on different operating systems, such as Linux, Windows, and MAC OS X. Even with the same operating system, clients and OSDs may adopt different hardware platforms. A client may run on X86 processor, but an OSD may run on Intel XScale, ARM, or PowerPC processor. In this case, a client's active storage code cannot be directly executed without complicated recompiling, hence the benefits of active storage are lost.

In this paper, we propose JAS, a JVM-Based active storage framework for object-based storage systems. JAS programs the active storage functions of users as Java codes, and allows them to be executed on Java Virtual Machine (JVM) on different OSD platforms without recompiling. JAS offloads the active storage functions from clients to the OSDs by extending the standard OSD model, and execute the Java code on the OSDs on-demand by triggering them through extended client interfaces. In this way, the active storage functions can not only utilize the benefits of object-based storage system, but also can provide cross-platform execution.

To support JAS, we extend the standard OSD command set to enable four major interface: *code offloading*, *code association*, *code triggering*, and *code execution*. Furthermore, we implement JAS under an object-based storage system by modifying the client node and the OSD node. We also test the performance of JAS with two representative applications. Experimental results show that the JVM-based active storage framework has been successfully set up, and this cross-platform design can largely improve system performance.

The rest of this paper is organized as follows. In Section 2, we describe the background and the related work. Then we describe the design and implementation of JAS in Section 3. Section 4 gives the performance evaluation. Finally, we conclude the paper in Section 5.

2. Background and Related Work

2.1. Object-Based Storage

Object-based storage provides a new solution for large-scale I/O systems. Object is a logical data unit, it can includes any type of data, and can be created and deleted like a file. Compared to a traditional file, an object has two main differences. First, it has more attributes to describe the characteristics of the data. These attributes can facilitate the clients or storage nodes to more efficiently manage the data. Second, an object also can include several methods, which allow the storage devices to directly process the data without interacting with the clients.

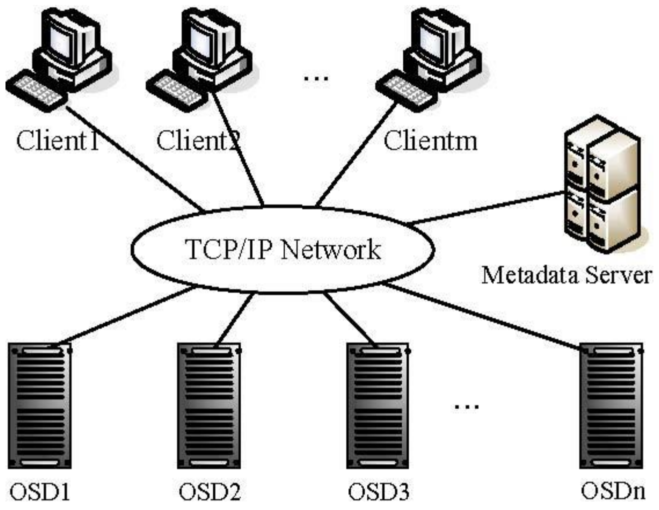

Figure 1. The Architecture of Object-Based Storage System

Object-based storage system (OBSS) [22] mainly consists three parts, as shown in Figure 1. The client provides standard interfaces for users to store data. The MDS manages the mapping information between logical objects and physical object-based storage devices (OSDs). The OSD is the real device to store the user objects. When a client accesses the data in an OSD, it first gets the data mapping information from the MDS, then interacts with the OSD(s) directly.

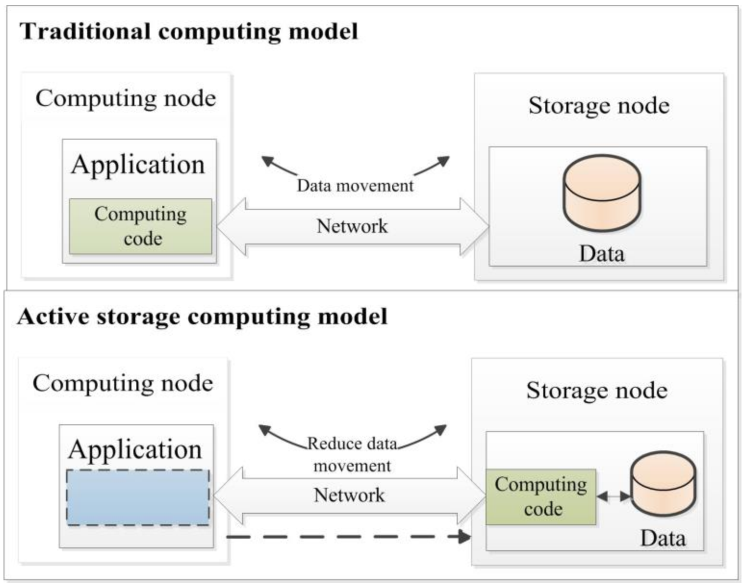

Figure 2. The Comparison of Traditional Computing Model and Active Storage Computing Model

To promote the development of object-based technology, the T10 OSD standard defines the OSD model and basic command set (*e.g.*, READ and WRITE command) [17]. There are four classes of objects: *user objects*, *root objects*, *partition objects*, and *collection objects*. Each object is identified by an object ID, and accessed by offset, length, and so on. In addition to these data access commands, the T10 OSD standard also defines attribute commands (*e.g.*, GET ATTRIBUTES and SET ATTRIBUTES) that are responsible for attributes retrieval and setting. Generally, each OSD relies an object-based file system to manage objects and attributes [16,22-24].

2.2. Active Storage

By exploiting the under-utilized hardware resource of storage nodes to process data without moving them to the computing nodes, as shown in Figure 2, active storage can largely improve the computer system performance. Active storage is first proposed to exploit the computing intelligence inside disk drives. These techniques are either designed for general applications [9-10,13], or some special fields, such as database [25-26]. However, these efforts are based on the block-level interface, thus incur significant overhead and hence offset the potential of active storage.

Due to the benefits of object storage, a lot of other researchers have gradually made efforts to integrate active storage into the object-based storage systems. Piernas *et al.* gave an active storage strategy implemented in Lustre parallel File System [19]. Huston *et al.* used an active storage architecture for interactive search of non-indexed data [27]. However, these systems do not comply with the T10 OSD standard. To promote the development of object-based storage technology, several works integrated the active storage technology into the object-based storage technology based on the T10 OSD standard [12,18,21,28-29].

Most of the above mentioned studies are especially based on a special OS platform or hardware architecture, thus the active storage code may not executed on cross-platform environment. MapReduce [30-31], a concept similar to active storage has also been employed in cluster computing field, MapReduce splits the computations and maps them to many computers processing the data locally, then the sub-results of the split computation are merged for form the global result of the problem.

2.3. Java Virtual Machine

The Java virtual machine is an abstract (virtual) computer defined by a specification. This specification omits implementation details that are not essential to ensure interoperability. For example, the memory layout of run-time data areas, the garbage-collection algorithm used, and any internal optimization of the Java virtual machine instructions (their translation into machine code). The main reason for this omission is to not unnecessarily constrain implementors. Any Java application can be run only inside some concrete implementation of the abstract specification of the Java virtual machine.

For the above reasons, in JAS, user's active storage function codes are written in Java. Java is designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the user for each separate platform. A Java virtual machine makes this possible because it is aware of the specific instruction lengths and other particularities of the platform.

3. Design and Implementation

In order to enable the JVM-based active storage, we need to consider the following four questions:

- How to offload the user's active storage code from the client to the OSD, without introducing significant changes to the existing OSD model?
- How to associate the active storage code with the user's data to support flexible and efficient data process?

- How to trigger the active storage code from the client, so that this code can be executed on demands?
- How to execute the active storage code on the JVM, without losing the efficiency of the current OSD software stack?

In this section, we answer these questions by discussing different modules of the JAS framework.

3.1. System Architecture

Figure 3 depicts the system overview of the proposed active storage framework. On the client node, the object-based system file system (OBSFS) is registered to VFS and provides a mounting point for user's data accesses. The OSD module is responsible for encapsulating user's file operations into the object-based operations, and the iSCSI Initiator sends and receives the OSD commands on the Ethernet network. On the OSD node, the iSCSI Target receives client's network commands, and call the OSD module to resolve the OSD commands. The OSDFS is responsible for storing objects and managing attributes. To support JAS, the OSD module on both client and OSD, adds new components in addition to handle the existing OSD commands. We describe the detailed design of these critical components in the following sections.

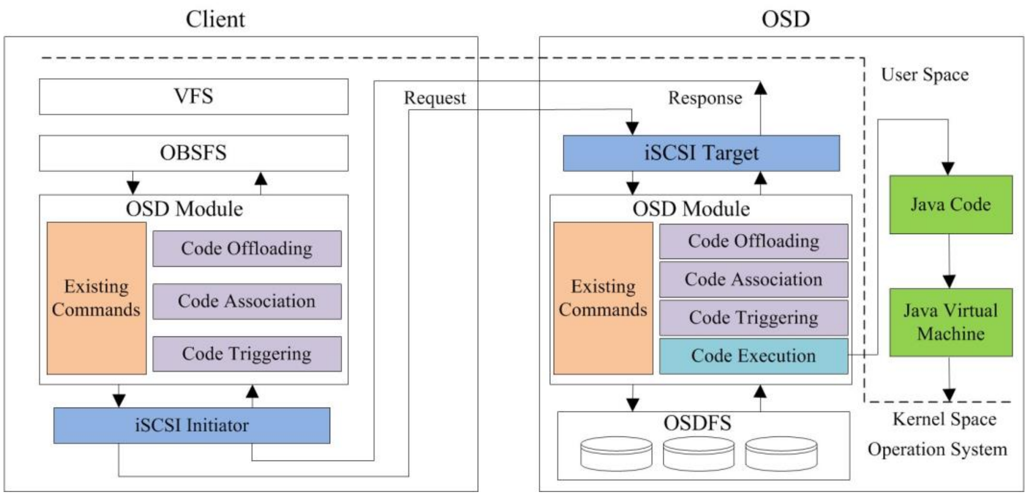

Figure 3. An Overview of the JVM-Based Active Framework

3.2. Active Storage Code Offloading

In our design, an active storage code of user is denoted by a piece of executable Java code, which is compiled on the client node. To run this code on a storage device, the active storage code needs to be delivered to the OSD in advance. A client provides a *code register* interface for users to offload their function codes on OSDs, and each target OSD returns a client with a *function ID* to identify which function should be executed later.

On the client, this interface can be achieved by using a new OSD command. To avoid significant modification of the existing OSD model, each client registers the active storage code by extending the current OSD_WRITE command, and the code itself are transferred to the OSDs as additional information in the Data-out Buffer of this extend OSD_WRITE command.

Once the OSD receives a *code register* command, it gets the active storage code from the Data-in Buffer and stores it on its physical storage media, such as hard disk, solid state disk, or DRAM. The code could be logically stored onto an OSDFS file system, a specific local file system, or a light-weighted database. For simplification, the OSD stores the active code as a regular user object in the OSDFS in our design, each with a unique 64-bit PARTITION_ID and USER_OBJECT_ID specified by the T10 OSD standard [17]. An OSD can hold multiple function codes from one client or multiple clients. To lookup the function code on the store media of the OSD in an efficient way, JAS maintains a code location table (CLT) to keep the code's location according to the *function ID*.

One possible case is that the active storage code may not be needed by the user any longer, it is necessary for JAS to provide a *code remove* interface for the user to delete this code on demands. To keep minimal change of the current OSD model, the client achieves this by extending the current OSD command. It issues an OSD REMOVE command to the target OSD, with an additional parameter function ID to specify the code needed to be deleted. For the OSD, it uses the *function ID* to search CLT once it receives a *code remove* command. After finding the specified location of the active code, the OSD removes the code on its store media and updates the entry in the CLT table.

3.3. Active Storage Code Association

To support the active storage on the OBSS, JAS must define what data an active storage code can process. Since data is accessed with the *object* interface, JAS associates each active store code with user data by specifying the *function ID* and the object ID. To provide a flexible execution for user functions, JAS provides two categories of associations between the active storage code and the user object.

The first one is *one-to-one* association, that is, a storage code can only process one user object in one OSD command. This mapping is straight forward and can be easily implemented. Previous work [18], [21] falls into this category. The other one is *one-to-many* association, in this case a storage code can process multiple objects' data in one command. Such a *one-to-many* mapping can largely reduce the OSD command transmission times between clients and OSDs. Since the iSCSI protocol in current OBSS implementation is a heavy-weight protocol [32], the *one-to-many* design can significantly improve the overall I/O system performance when the user needs to process small amount of data many times.

The current OSD standard does not support these associations. We use a new *code trigger* command, which we discuss in the next section, to record this information. For the *one-to-one* association, the active storage code is identified by the *function ID*, the object is specified by the object ID, and the data needed to be processed is specified by the offset and length parameters in the new command. For the *one-to-many* association, the active storage code is still specified by the *function ID*, but the involved objects are described by a collection object. Correspondingly, the data needed to be processed is specified by a list of <offset, length> pairs in the new OSD command.

3.4. Active Storage Code Triggering

An important issue of JAS is when to trigger the code execution? There are two chances for a client to trigger an OSD to execute the codes resident on them: *explicit triggering* and *implicit triggering*. By default, the framework allows the code to be executed on demands, which means the code will not run on the OSDs until the client issues an explicit request. During the execution of the application, the client will send a *code trigger* command to OSD if he wants to execute the code. Once the OSD receives this command, it will begin to invoke the corresponding code on the storage device. Such an explicit trigger mechanism can make the code execution without affliction to the execution of normal I/O tasks from clients. JAS also allows the code to run automatically based on certain system environments, such as a given time in a day or a specified status of OSD hardware resource, *etc*. This *implicit* trigger may work only when there are unused computing resources and I/O bandwidth on OSDs.

Another issue is how to trigger the code execution? JAS provides a new *code trigger* interface for this purpose. For the *explicit triggering*, a client will send a trigger command to the target OSD when it wants to run the code. Similarly, to avoid major changes of the current OSD command set, we extend the original OSD_READ and OSD_WRITE command to trigger code execution between a client and an OSD. The former command is suitable for a "read-process-write" active storage data process pattern, *i.e.*, the code will read data from the OSD first, then process the data, and finally write the processing result to the OSD. The latter command is designed for a "process-write" pattern, which means the code will process data from the client data buffer without fetching them from the storage device, and then write the result to the OSD. With these two extended commands, JAS can provide flexible data processes on the storage nodes. For the *implicit triggering*, each client also uses the two extended commands to register the conditions triggering the code execution on the OSD. The OSD uses a background thread to monitor the system status. Once the given condition is matched, the OSD will invoke the code execution.

Table 1. Read Command Format

Bit Byte	7	6	5	4	3	2	1	0
8	MSB	SERVICE ACTION(8885h)						
9								LSB
10	Reserved			DPO	FUA	ISOLATION		
11	Reserved		GET/SET		Reserved→ASC			
12	TIMESTAMPS CONTROL							
…	Reserved							
16	MSB	PARTION_ID						
23								LSB
24	MSB	USER_OBJECT_ID						
31								LSB
…	…							
…	…							
235	…							

As an example, Table 1 shows the modified command descriptor block (CDB) format of the OSD_READ command. The bit 3 to 0 of byte at offset 11, which are originally reserved, are used as an active storage control (ASC) field to support JAS. When the ASC=0000b, the READ command is the same as that in the current OSD standard. When the ASC is set to 0001b, the READ command becomes an active storage trigger command for "read-process-write" data processing pattern. Other values of ASC are reserved for future function extension. Table 2 lists the Data-out Buffer format of the extended READ command for a "read-process-write" pattern. It includes three new fields (F1-F4): F1 specifies the *function ID* of the active storage code, which will process the data specified by the parameters of CDB in Table 1; F2 specifies the parameters of the code; F3 defines which object is used to store the process result; F4 specifies the detailed location to store data in the result object.

Table 2. Data-out Buffer Format

Bit Byte	7	6	5	4	3	2	1	0
0	MSB	(F1)Function ID						
15								LSB
16	MSB	(F2)Function Parameter 1, if any						
31								LSB
32	MSB	(F2)Function Parameter 2, if any						
47								LSB
...	MSB	(F2)Function Parameter n, if any						
…								LSB
i	MSB	(F3)USER_OBJECT ID, if any						
i+15								LSB
j	MSB	(F4)Data offset, data legth, if any						
j+15								LSB
…								LSB

3.5. JVM-Based Code Execution on OSDs

Once an OSD receives the code trigger command from the client, it needs to prepare the run-time environment for this code, load it into the memory, process it, and even store the result on the OSD. Generally, to support the cross-platform execution, the active storage code and the JVM run in the user space of the operation system of the OSD. However, to maintain high efficiency, the current OSD command resolving module is usually resident in the kernel space. One concern is how to invoke the user space's code from the kernel space? To address this issue, we use the Linux API *call_usermodehelper()* to execute application's active storage codes from kernel space and deliver corresponding parameters to the active storage code.

Another concern is how to transfer result data between the OSD command module and the user process derived from the active storage code. Since the OSDFS is usually built on a general purpose local file system, which can also be easily accessed by a user process, we enable the data exchange between these two parts from different spaces by accessing a shared file. To improve the communication efficiency, the inter-process communication mechanism—pipe is used: the user writes the result to a pipe file, and the OSD command module gets this result from the pipe file with a read operation.

3.6. Implementation

We implement a prototype of JAS on the base of our previous work [24], which is derived from the Intel iSCSI implementation [33]. On the client side, the initiator realizes a file system (OBSFS) mounted under VFS. On the OSD side, the target is built on an OSDFS, which saves user files and directories as objects on OSD. We modify both the initiator and target to implement the above mentioned function modules to support the object-based active storage. Furthermore, to enable the cross-platform code execution, the Java run-time system is stalled on both the client and the OSD side.

4. Performance Evaluation

In this section, we evaluate the performance of JAS in a real object-based storage system. We use typical applications to show the benefits of JAS.

4.1. Experimental Setup

The test bed includes one client and one OSD. The MDS is also installed on the OSD. The client and the OSD have different hardware platforms, and they are connected through one gigabit Ethernet network. The configuration of the client and OSD node is shown in Table 3.

Table 3. The Client and OSD Configuration

	CPU	Memory	Disk	Network
Client HW	Xeon 3.0G	1GB	200GB/SATA	BCM5700
OSD HW	XScale 667M	512M	500GB/SATA	BCM5700
Switch	Cisco Catalyst 3750 GE switch			
Client OS	CentOS 6, Kernel version 3.0.0			
OSD OS	Redhat 9, Kernel version 2.4.20			

We use two typical applications, data compression and data selection, to evaluate JAS. The first application is responsible for compressing user's data with a configurable size. Data compression is a representative operation in file system and is widely used in a space-constrained storage environment. In our test, we use a user object to store the user's data, and adjust different sizes of data to control how much data should be compressed.

For comparison, we run two group of tests: first, we run the application under the original storage system without active storage (Original), then we run it again by enabling the active storage module (Active).

4.2. Data Compression

In the *Original* test, the client first fetches the given-length data from the remote OSD to the local memory through an OSD READ command, and then carries out the data compression operations. After that, the client writes the result back to the OSD. The execution time of the application consists of object read time through network, object process time on the client, and the writing back time to the OSD. In the *Active* test, the client first downloads the function code onto the OSD device, and then associates the code and triggers the data compression through a new OSD READ command. Finally, the OSD performs the data compression and stores the result onto itself. In this test, the execution time of the application consists of all the above mentioned parts.

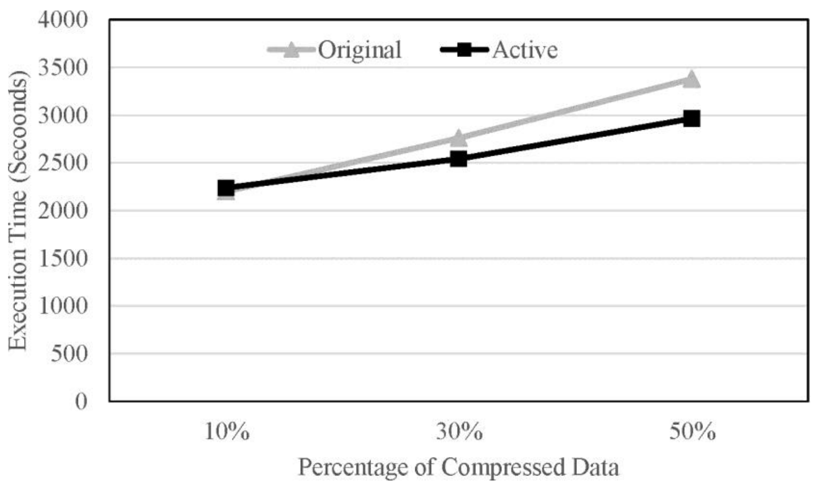

Figure 4. Performance Comparison Under the Input Data of 100MB

Figure 4 shows the application execution times for a small user object of 100MB. We test the performance of the application under 10%, 30%, and 50% of the object's data is compressed. From the figure, we can see that when the data amount is very small (10 % data is compressed), active storage shows comparable performance as the original storage architecture. Because the network and storage data movement is not very large, the reduced I/O time is not obvious. However, for large data processing, namely 30% and 50% of the data are compressed, the active storage can reduce the application's execution time by 8.5% and 14.1% respectively, compared to the original test. This result indicates that active storage can greatly improve the application's overall performance.

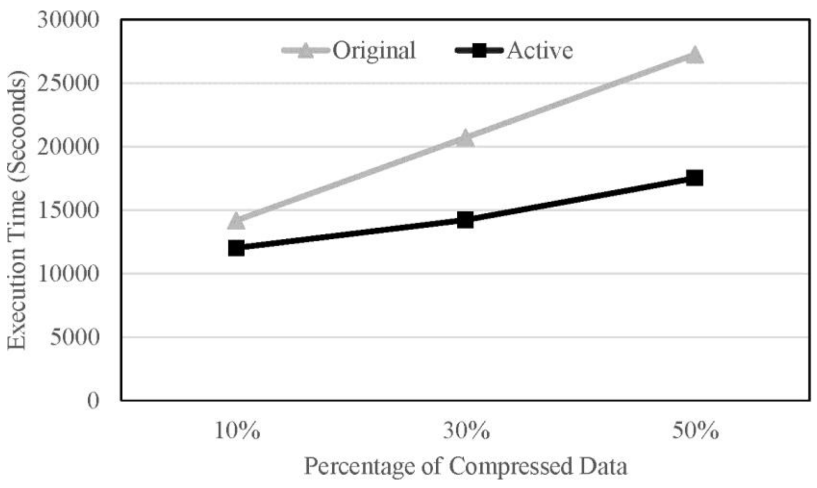

Figure 5. Performance Comparison Under the Input Data of 500MB

Figure 5 shows the application execution time with a large user object of 500MB. We can find that active storage can reduce the application's execution time by 17.8%, 45.5%, and 55.7%. These results show as the processed data amount increases, the benefits of active storage is more significant.

4.3. Data Selection

In the *Original* test, the client first fetches the given-length data from the remote OSD, and then carries out the data selection operations. The execution time of the application consists of object read time through network and data process time on the client. In the *Active* test, the application's execution time includes the client code offloading time, the data selection time on the OSD and the result returning time to the client. In our tests, the data set is a data sequence consisting of millions of data, each of which is 0-9, and the large-scale sequence is stored as a 1GB user object on the OSD.

Table 4 describes the application execution time under different data selection conditions. The percentage means how much data should be selected and returned to the client. From the table we can observe that the application execution time in the Active test under different conditions are decreased by 87.4%, 71.2% and 5.9%. With the change of the data selection conditions, more and more data need to be processed on the OSD, until the data in the result is nearly to the amount of the original data sequence. As a result, the difference between execution time in AS test and TS test becomes small. From the above discusses, it can be seen when the returned data is much smaller than the original data, the benefit of active storage is particularly evident. Of course, if more than one OSD are deployed in the system to parallel process, benefit of active storage will be more significant.

Table 4. Performance Comparison Under Data Selection Operation

Execution time(Seconds)	**Data Size(1GB)**		
	10%	40%	70%
Original	910	1309	1452
Active	120	378	1380

5. Conclusions

The great processing capacity and expressive object interface make it feasible to implement active storage for object-based storage system. In this paper, we propose JAS, a JVM-based active storage framework to enable cross-platform active code execution for object-based storage systems. JAS programs the active storage functions of users as Java codes, and allows them to be executed on different OSD platforms without complicated recompiling or modifications from users.JAS offloads the active storage codes from clients to OSDs by extending the standard OSD model, and executes the Java code on OSDs on-demands by triggering them through extended client interfaces. We have implemented JAS under an OSD-based device file system. Experimental results with representative applications show that the JAS supports cross-platform code execution, and can significantly improve system performance.

Acknowledgements

The authors are thankful to Junxing Yang of Wuhan University for his help toward this work. This research is supported in part by the China National Basic Research Program (973 Program, No.2015CB352400), NSFC under grant U1401258, No.61572377, No.61572370, and No.61373040, the Ph.D. Programs Foundation of Ministry of Education of China under Grant No.20120141110073, the Natural Science Foundation of Hubei Province of China under Grant No.2014CFB239, the Open Fund from HPCL under Grant No.201512-02, and the Open Fund from SKLSE under Grant No.2015-A-06.

References

[1] J. L. Hennessy and D. A. Patterson, “Computer architecture: a quantitative approach”, Morgan Kaufmann, **(2011)**.
[2] M. Kandemir, S. W. Son, and M. Karakoy, “Improving I/O performance of Applications through Compiler-DirectedCode Restructuring”, in Proceedings of the 6th USENIX Conference on File and Storage Technologies, **(2008)**, pp. 159–174.
[3] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A.

Snavely, T. Sterling, R. S. Williams, and K. Yellick, "Exascale computing study: Technology challenges in achieving exascale systems", Tech. Rep. TR-2008-13, DARPA, **(2008)**.

[4] S. He, X.-H. Sun, and B. Feng, "S4D-Cache: Smart Selective SSD Cache for Parallel I/O Systems", in Proceedings of the International Conference on Distributed Computing Systems, **(2014)**, pp. 514-523.

[5] S. He, X.-H. Sun, and A. Haider, "HAS: Heterogeneity-Aware Selective Data Layout Scheme for Parallel File Systems on Hybrid Servers", in Proceedings of 29th IEEE International Parallel and Distributed Processing Symposium, **(2015)**.

[6] S. He, X.-H. Sun, Y. Wang, A. Kougkas, and A. Haider, "A Heterogeneity-Aware Region-Level Data Layout Scheme for Hybrid Parallel File Systems", in Proceedings of the 44th International Conference on Parallel Processing, **(2015)**.

[7] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines", in Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13), **(2013)**.

[8] X. Ma, A. Reddy, I. Center, and C. San Jose, "Mvss: an active storage architecture", IEEE Transactions On Parallel and Distributed Systems, vol. 14, no. 10, **(2003)**¸ pp. 993-1005.

[9] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," ACM SIGPLAN Notices, vol. 33, no. 11, **(1998)**, pp. 81-91.

[10] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia", in Proceedings of the 24rd International Conference on Very Large Data Bases, **(1998)**, pp. 62–73.

[11] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu, "The panasas activescale storage cluster-delivering scalable high bandwidth storage", in Proceedings of the ACM/IEEE SC2004 Conference on Supercomputing, **(2004)**, pp. 53–62.

[12] S. He, X. Xu, and Y. Yang, "Oasa: An active storage architecture for object-based storage system", International Journal of Computational Intelligence Systems, vol. 5, no. 6, **(2012)**, pp. 1173-1183.

[13] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)", ACM SIGMOD Record, vol. 27, no. 3, **(1998)**, pp. 42-52.

[14] M. Mesnier, G. Ganger, and E. Riedel, "Object-based storage: pushing more functionality into storage", IEEE Potentials, vol. 24, no. 2, **(2005)**, pp. 31-34.

[15] P. Schwan, "Lustre: Building a file system for 1000-node clusters", in Proceedings of the 2003 Linux Symposium, **(2003)**, pp. 380-386.

[16] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system", in Proceedings of the 7th symposium on Operating Systems Design and Implement. USENIX Association Berkeley, CA, USA, **(2006)**, pp. 307-320.

[17] R. O. Weber, "Information technologyscsi object-based storage device commands-2 (osd-2), revision 5," Tech. Rep. Technical report, INCITS Technical Committee T10/1729-D, **(2009)**.

[18] T. M. John, A. T. Ramani, and J. A. Chandy, "Active storage using object-based devices", 2008 IEEE International Conference on Cluster Computing, **(2008)**, pp. 472-478.

[19] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system", in Proceedings of the 2007 ACM/IEEE conference on Supercomputing. ACM New York, NY, USA, **(2007)**, pp. 1-10.

[20] D. Nagle and B. Welch, "object-based cluster storage system", in Proceedings of the 23st IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies, **(2006)**.

[21] L. Qin and D. Feng, "Active storage framework for object-based storage device", in Proceedings of the IEEE 20th International Conference on Advanced Information Networking and Applications,**(2006)**, pp. 97-101.

[22] S. He and D. Feng, "Implementation and performance evaluation of an object-based storage device", in Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os(SNAPI'07), **(2007)**, pp. 129-136.

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

[24] S. He and D. Feng, "Design of an object-based storage device based on I/O processor", ACM SIGOPS Operating Systems Review, vol. 42, no. 6, **(2008)**, pp. 30-35.

[25] S. Y. W. Su and G. J. Lipovski, "Cassm: A cellular system for very large data bases", in Proceedings of the International Conference on Very Large Data Bases (VLDB), **(1975)**, pp. 456-472.

[26] E. A. Ozkarahan, S. A. Schuster, and K. C. Smith, "Rap: an associative processor for data base management", in Proceedings of the AFIPS Joint Computer Conferences. ACM New York, NY, USA, **(1975)**, pp. 379-387.

[27] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki, "Diamond: A storage architecture for early discard in interactive search", in Proceedings of the 3rd USENIX Conference on File and Storage Technologies. USENIX Association, **(2004)**, pp. 73-86.

[28] A. Devulapalli, I. Murugandi, D. Xu, and P. Wyckoff, "Design of an intelligent object-based storage device", **(2009)**.

[29] Y. Xie, K. Muniswamy-Reddy, D. Feng, D. Long, Y. Kang, Z. Niu, and Z. Tan, "Design and evaluation of oasis: An active storage framework based on t10 osd standard", in MSST. IEEE, **(2011)**, pp. 1-12.
[30] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", in Proceedings of the 6th symposium on Operating Systems Design and Implement, **(2004)**, pp. 138-150.
[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments", in Proceedings of the 8th symposium on Operating Systems Design and Implement, **(2008)**, pp. 29-42.
[32] B. K. Kancherla, G. M. Narayan, and K. Gopinath, "Performance evaluation of multiple tcp connections in iscsi", in Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies, **(2007)**, pp. 239-244.
[33] Intel Corporation, "Intel iscsi reference implementatioimplementation", [Online].Available: http://sourceforge.net/projects/intel-iscsi, **(2015)**.

Authors

Xiangyu Li, is a Ph.D. candidate of the Computer School, Wuhan University, Wuhan, China. He received a B.A. degree from Huazhong University of Science and Technology, China, in 2003 and a M.S. degree in computer science and technology from Wuhan University, China, in 2008. He is especially interested in file and storage systems, high performance computing, distributed system, and computer network.

Shuibing He, received the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology, China, in 2009. He is now an assistant professor at the Computer School of Wuhan University, China. His current research areas include parallel I/O systems, file and storage systems, high-performance computing and distributed computing.

Xianbin Xu, graduated from the department of system architecture in Huazhong University of science and technology and worked for teaching in Huazhong University of science and technology from 1977 to 1985. He got Ph.D. computer school of Wuhan University. He is now a professor at the Computer School of Wuhan University, China. His research interests focus on network storage, data grid and distributed system.

Yang Wang, received the BSc degree in applied mathematics from Ocean University of China (1989), and the MS and Ph.D degrees in computer science from Carleton University (2001) and University of Alberta, Canada (2008), respectively. He is currently with Shenzhen Institute of Advanced Technology, Chinese Academy of Science, as a professor. His research interests include cloud computing, big data analytics, and Java Virtual Machine on multicores.