UCONN LIBRARY

University of Connecticut
OpenCommons@UConn

Master's Theses University of Connecticut Graduate School

8-8-2012

Active Storage and SSD Caching in an Object Storage Environment

Michael T. Runde
University of Connecticut - Storrs, rundemt@gmail.com

Recommended Citation
Runde, Michael T., "Active Storage and SSD Caching in an Object Storage Environment" (2012). *Master's Theses*. 327.
https://opencommons.uconn.edu/gs_theses/327

This work is brought to you for free and open access by the University of Connecticut Graduate School at OpenCommons@UConn. It has been accepted for inclusion in Master's Theses by an authorized administrator of OpenCommons@UConn. For more information, please contact opencommons@uconn.edu.

Active Storage and SSD Caching in an Object Storage Environment

by

Michael Thomas Runde

B.S.E., University of Connecticut, 2010

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2012

APPROVAL PAGE

Master of Science Thesis

Active Storage and SSD Caching in an Object Storage Environment

Presented by
Michael Thomas Runde, B.S.E.

Major Advisor ______________________
John A. Chandy

Associate Advisor ______________________
Mohammad H. Tehranipoor

Associate Advisor ______________________
Zhijie J. Shi

University of Connecticut
2012

Acknowledgements

This thesis would not have been possible without Dr. John A. Chandy. I am grateful to him for challenging me to seriously consider a graduate degree when I initially thought it was beyond me. He has always been there patiently willing to do whatever he can to give me the opportunity to succeed if that meant answering emails late into the night in the rush to a deadline or spending time in the lab bringing an experienced set of eyes to an issue. I'd also like to thank Paul, Orko, and Cengiz at the SNSL for their willingness to always take a pause from their work to bounce ideas off with, provide another set of eyes on a troublesome error and for the help setting up the cluster on which I spent numerous weeks testing.

Finally I'd like to thank my parents, to whom this thesis is dedicated, who have always supported and encouraged me over the long years, twist and turns of my college career. They ensured I would be able to pursue any path I wished as long I worked hard to follow it. Without everyone's help this thesis would have never been completed.

Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Motivation 1

1.2 Object Storage Disks 1

1.3 Active Storage 3

1.3.1 Early Active Storage Work 4

1.3.2 Thesis Contribution 4

1.4 SSD Caching 5

1.4.1 SSD Technologies 6

1.4.2 Existing Caching Techniques 7

1.4.3 Thesis Contribution 8

1.5 Outline 8

2 Active Storage with Object Based Devices 9

2.1 Introduction 9

2.2 Related Work 10

2.3 Object Storage Devices 11

2.3.1 OSD Objects 13
2.3.2 OSD Access types 15
2.3.3 Security 15
2.4 Active Storage 17
2.5 Active Storage OSD Implementation 19
2.5.1 Programming Model 20
2.5.2 Execution Engines 27
2.5.3 Secure Implementation 29
2.6 Results 34
2.6.1 Performance 34
2.6.2 Impact of Data Transfer 35
2.6.3 Evaluation of Multiple OSDs 37
2.6.4 Overhead Analysis 39
2.6.5 Evaluation of Multiple Execution Engines 44
2.7 Summary 49

3 SSD Caching with Object Storage Devices 50
3.1 Introduction 50
3.2 Related Work 52
3.3 Architecture Overview 52
3.3.1 Request handling 53
3.4 SSD Cache Size Determination 55
3.5 Implementation 56
3.5.1 Supported OSD Features 56
3.5.2 OSD Format 56
3.5.3 OSD Create and Remove 57
3.5.4 OSD Read and Write 57
3.6 Results 58

3.6.1 Setup . . . 58
3.6.2 OSD Results . . . 59
3.6.3 Exofs Results . . . 63
3.7 Summary . . . 67

4 Conclusions 68
4.1 Future Work . . . 69
4.1.1 Active Storage . . . 69
4.1.2 SSD Caching . . . 70

Appendix

A Developer's Guide to Active Storage 72
A.1 Introduction . . . 72
A.2 Current state of the API . . . 72
A.2.1 Supported Libraries . . . 73
A.3 Full Function C Code Example . . . 73
A.4 Full Function Java Code Example . . . 77
A.5 Debugging . . . 81

Bibliography 82

List of Tables

Table

3.1 Linux Compilation Times 66

List of Figures

Figure

2.1 Comparison of traditional and OSD storage models [DHH+06] 12
2.2 OSD Security Model [DHH+06] 16
2.3 EXECUTE_FUNCTION Format 21
2.4 High Level View of Active Storage OSD 23
2.5 Example Active Storage Function Code 26
2.6 Local vs. Active Storage 36
2.7 Local vs. Active Storage Breakdown 36
2.8 Active Storage Wordcount Results 38
2.9 Active Storage Grep Results 40
2.10 Active Storage AES Encrypt Results 40
2.11 Active Storage FastRPC vs Direct 43
2.12 Active Storage Client vs Engine Times 45
2.13 Active Storage C vs Java Times 47
2.14 Active Storage C vs Java Read Times 47
2.15 Active Storage C vs Java Write Times 48

3.1 OSD SSD Caching Architecture 54
3.2 OSD Testing on SSD with 1KB-1024MB objects 60
3.3 OSD Testing on HDD with 1KB-1024MB objects 60

3.4 OSD Testing with 2MB cache with 1KB-1024MB objects 62
3.5 OSD Step Test with 2MB cache using 512KB blocks 62
3.6 IOzone results using Exofs 65

Abstract

The advancing performance and lowering costs required to implement additional processing power on system peripherals such as disk drives are increasingly allowing additional computing ability to be located directly on individual drives. Active Storage attempts to take advantage of this excess by moving some computationally intensive applications directly to the disk drives. This can remove the bottlenecks seen through interconnects between the drives and the CPU of an initiating system as well as remove the need for systems to handle these applications.

The contributions of this thesis are in two areas. The first is the development of a framework designed to enable active storage utilizing the iSCSI T10 OSD standard for eventual integration into this standard. The framework supports multiple programming languages with a focus on flexibility and security. This thesis then details and implements a novel solid state disk caching scheme utilizing the same iSCSI OSD standard which focuses on small file performance and latency improvements.

To my parents

Chapter 1

Introduction

1.1 Motivation

The increasing performance and decreasing cost of processors has enabled increased system intelligence at I/O peripherals. Disk drive manufactures have been using this trend to perform more complex processing and optimizations directly inside the storage devices. Such kind of optimizations have been available only at the disk controller level of the storage stack. The current trends in storage density, mechanics, and electronics are working towards eliminating the bottlenecks encountered while moving data off the media, and putting pressure on interconnects and host processors to move data more efficiently. We tackle this problem in two different ways: by moving computation to disk and improving the caching capabilities of Solid State Disks. These two topics are introduced in the following two sections and in more detail in Chapters 2 and 3.

1.2 Object Storage Disks

The interaction between operating system and storage system has been stagnant since its inception. This interaction between them has generally been at the block-/byte level where the OS has to specify the physical location on a storage system to store or read data by specifying specific blocks or physical addresses to the system in

order to interact with it. The increasing computational capability at the disk has led to the development of object-based storage devices whereby some of this filesystem functionality is moved to the disk [GVM00, GNA+98, MGR03, ANS02]. Developments in object-based storage systems and other parallel I/O systems where the data and control paths are separated have demonstrated an ability to scale aggregate throughput very well for large data transfers. In these systems, the metadata is placed on a distinct metadata server that is out of the data path. There are many parallel storage file systems [BZ01, NSM04, SH02, WBM+06, CLRT00] based on this idea of separating the metadata from the data. By separating the metadata, storage management functionalities are kept away from the real data access, thus giving the user direct access to data once the authorization to access the data is received. These file systems can achieve high throughput by striping the data across many storage nodes. An Object Storage Device or OSD is an abstraction of the low level interaction between the operating system and storage system enabling both programs and operating systems to specify an object to be accessed on the storage media which will then take over and complete the low level data access.

In high-performance computing applications where data throughput is typically more important than metadata latencies, this architecture works well. It however, does not take advantage of the full promise that object storage offers. There has been some recent work to exploit object storage in its application to parallel file systems, specifically PVFS [DDAW07, DDW+07, ADD+08]. The computation capability needed to enable object storage devices can also enable computation at the storage node in what has been called active disks or active storage. This active storage computation serves as a mechanism to enable parallel computation using distributed storage nodes [Wil92, AUS98, KPH98b, RFGN01, WCV02]. This thesis describes an architecture that enables active storage computations using object storage devices.

1.3 Active Storage

Moving portions of an application's processing so that it runs directly at the disk drives can dramatically reduce data traffic and take advantage of the parallel storage already present in large systems today. Situations where active storage have the greatest potential to speed up applications are those that are typically I/O bound. These include searches, sorts, and any similar data accesses where large amounts of data need to either be read or modified. For example, typically, a normal file system has to sequentially read the data it is searching, which will most likely be limited to the read speed of whatever drive or group of drives the data is stored on.

On an active storage enabled OSD, every drive containing part of the data that needs to be searched can be sent the code to perform the search. Instead of being limited by the sequential read speed of a single drive, each drive could search for the required data simultaneously allowing for search to only take as much time as necessary for the drive with the largest segment of data to search to complete due to the parallel use of the drives. Not only would this method increase the speed at which the search could be carried out, it would also significantly reduce the amount of data transferred from the storage system to the client performing the search as only the initial search command and results would have to be transferred instead of the entire data set. As the size of storage systems and data sets continues to increase the ability to limit the amount of data necessary to perform simple functions becomes more advantageous.

Parallel Active Storage therefore is best used on large data sets where little computation is necessary and disk throughput is usually the limiting factor. Other than searches, encryption/decryption presents itself as a case where offloading the entire read-encrypt/decrypt-write chain should work well. A final case that would be very useful for extremely large data sets would be to use Active Storage to calculate and check file hashes on a continuing basis to ensure the data held in datacenters is not

degrading. This could allow HPC compute nodes to continue their normal jobs as long as the AS functions were run in the background instead of having to use the HPC nodes to do the checking.

1.3.1 Early Active Storage Work

The advantages of moving computation to the disk has been demonstrated in early work in transaction processing, data mining and multimedia [RFGN00, RGF98, Rie99] by Erik Reidel. His system worked by specifying in the attributes of an object a piece of code to run when accessed through a read command. This framework is stream based in the sense that only data being accessed through a normal read command can be accessed by the remote function. It also therefore acts more as a filter to data access which is very limiting compared to the RPC model described in this thesis.

Acharya, et al. also applied active disk ideas to a set of similar applications, including database select, external sort, data cubes, and image processing, using an extended-firmware model for next-generation SCSI disks [AUS98]. This work was later expanded to large scale data processing using the concept of data filters in the DataCutter/Active Data Repository framework [CFSS99, BFSS00]. A group at Berkeley has independently estimated the benefit of Intelligent Disks for improving the performance of large SMP systems running scan, hash join, and sort operations in a database context [KPH98a].

1.3.2 Thesis Contribution

While active storage and object storage arose from the same OSD root, there has been little work in integrating both active storage and object storage particularly with respect to the OSD standard that has emerged from the T10 standards body. The Object-Based Storage Devices (OSD) specification [ANS08] has introduced a new set

of device-type specific commands into the SCSI standards family. The specification defines the OSD model and its required commands and command behavior. The lack of an active storage OSD platform led to the design of the active storage OSD framework described here.

The framework proposed in this thesis provides a standardized API that can express a rich set of functionality for both application and file system developers. The API defines a set of object access methods along with security and access control mechanisms built upon the existing OSD security protocols. The architecture allows for multiple virtual machines or execution engines to support multiple programming languages. In particular the mechanisms required to provide secure and safe execution of active storage functions. From the application layer, the programming model is similar to an asynchronous RPC. This model provides the most flexibility to the application and user.

1.4 SSD Caching

The second contribution of this thesis aims to improve the performance of OSDs by utilizing solid state drives (SSDs). SSDs built with NAND flash have great advantages over spinning disk based drives (HDDs) in sequential and especially random access throughput due to their lack of mechanical components. They are however hampered by two primary disadvantages of lower capacities and a greatly increased cost. While large RAID arrays can help increase sequential throughput from rotational media based drives, random access is much harder to speed up due to the slow random access times of the disks which dominate random throughput. The SSD's combination of attributes allows it to be a great candidate as a cache layer in a hybrid storage system. This system should combine the fast access times of the SSD and high throughput potential of a HDD RAID to enable SSD like access times and throughput while placing a majority of the data on the slower, less expensive HDDs.

Solid State Drives (SSDs), particularly NAND flash based drives have been quickly gaining in capacity, performance, and price competitiveness with typical platter based Hard Disk Drives (HDD) over the past five years (early 2007 through Mid 2012). In that time capacities have increased from approximately 16GB to 512GB, performance has increased from read/write speeds of 45/25MB/s to 520/390MB/s and price has decreased from $100/GB to $1.00/GB [Key08] [Vat12]. SSDs also have one other primary performance metric exceeding that of HDDs; their access times are approximately .1-.2ms while HDDs average 10-15ms, a 100x decrease. This allows the SSDs to have a 4Kbyte random read/write speed of up to 90/200MB/s while a faster than average HDD (10,000 RPM) can only manage speeds of .68/1.9MB/s [Shi]. This is a drastic increase in the factors that make up a competent storage device for the amount of time that has passed and make current SSDs a compelling alternative to the more common HDDs.

The problem with SSDs however continues to be the price. At the current average of $1.00/GB the drives are still much more expensive compared to large commodity drives such as 2TB 7200RPM drives which are available for prices as low as $120 [Gas12] giving a per gigabyte cost of approximately $.06/GB, which makes the SSD about 17x the cost of the platter based drive. If a multi-drive storage systems price was mostly dependent on drive costs, replacing all storage drives with SSDs would most likely be impossible due to budgetary constraints. SSDs speed and non-volatile nature do allow them to be a good candidate to act as a cache for a larger storage system such as an object based file system or OSD.

1.4.1 SSD Technologies

Although Solid State Drives have been around in some form for approximately 40 years they were quite rare until the introduction of relatively inexpensive NAND flash memory based SSDs around the year 2000. Early SSDs used various technologies such

as battery backed RAM, FRAM, and others. As of 2012, NAND flash based SSD dominate the market and come primarily in two types, either single-level-cell (SLC) or multi-level-cell (MLC) configurations. The difference between these two is simply the number of bits programmed into each cell with MLC referring to two bits per cell. NAND flash chips typically support either mode with the greatest difference being the write endurance which is typically approximately ten times higher for SLC. MLC however has almost completely replaced SLC due to its cost being about half and the development of high endurance MLC by one of the largest flash suppliers, Intel, who claims SLC like endurance.

Most consumer and enterprise drives share a common form factor and interface with 2.5 inch SATA hard disk drives. Inside there are slight differences between the two including on enterprise drives the use of high endurance flash and use of several capacitors to allow the drive to write any data still in buffers to be written in the case of a loss of power to increase reliability. The actual architecture of the drives is similar however with both containing typically 6 to 12 flash chips on the PCB surrounding a central controller chip which communicates from the I/O port to the flash chips. This controller sets up a RAID like array in order to extract parallelism from the multiple chips used and to increase fault tolerance by duplicating data allowing for data recovery in the case of the failure of a single chip.

1.4.2 Existing Caching Techniques

Current caching systems all try to provide most of the benefits of SSDs (fast access times and high transfer rates) while minimizing the additional cost by still storing most of the data on HDDs. There are several on the market currently targeting both consumers and enterprise environments. They all, however, work at the block level, intercepting requests and redirecting as they see fit. They cache commonly used blocks on the SSD while the rest are stored on the HDD. Some such as Intel's Smart

Response technology allow the user to decide between a write-back or write-through architecture while others such as Seagate's Momentus XT are only a write-through meaning the cache only improves performance on reads.

1.4.3 Thesis Contribution

In this thesis, we propose a different approach to SSD caching. The purpose of this caching scheme is to eliminate the vast majority of the access time necessary for HDD based drives by utilizing a SSD to store the first several MB of a file. When a read request arrives to read the entire file, it is split up between the two drives and the new requests are simultaneously sent to each physical drive. The SSD will return the data very quickly due to its fast access times and high transfer rates and this data can be written into the return buffer first. The HDD will then finish accessing the remaining data and begin returning it as well with the SSD having already hidden the high access times of the HDD. This cache size will be optimized to allow the SSD to just cover the access times of the HDD. This design is therefore optimized for entire file read or writes but should at worst return to HDD only speeds.

1.5 Outline

This thesis has four chapters and an appendix. This first chapter contains introductory material including background, features, and applications. The second focuses on Active Storage OSDs and the implementation and testing that resulted from the proposed design. The third chapter focuses on a new approach to SSD caching which was implemented and tested. The final fourth chapter finishes with closing thoughts and the challenges left to turn the current implementations into production quality. The appendix includes a guide for developers detailing the process to create active storage applications to be used with the system outlined in this thesis.

Chapter 2

Active Storage with Object Based Devices

2.1 Introduction

OSDs are primarily designed to be used as part of larger storage systems such as those used to store vast amount of data or those used by high performance computing clusters. These systems can consist of hundreds of individual drives and are usually in various configurations to maximize both speed and reliability. Even with these large arrays of drives, they are usually accessed at the block level similar to how a single drive is accessed leaving the storage system with little or no information about what exists on the drives since client machines keep the information regarding what blocks belong to specific files. The use of an OSD instead of a conventional file system can help to reduce costs and allow easier administration of the system including actions such as backup and upgrading the amount of storage through decreasing the amount of time and effort necessary to perform these tasks. Object based file systems represent one of the first attempts to fundamentally change the way users interact with the storage system. Block based storage has already been virtualized to an extent. For example, a multi disk RAID (Redundant Array of Inexpensive Discs) system appears as a single disk to the user, but the virtualized block numbers seen by the user may not line up with the actual blocks on the drives in use. An OSD just further expands on this virtualization to increase the amount of control the storage

system has to organize the data being stored to it to. This allows a simplification of file system commands due to absolving the file system from having to manage each block independently by moving the overhead of dealing with blocks from the operating system to the storage device. Commands on a typical file system such as a read can be complicated due to having to send a separate command to retrieve each block that is part of the file being retrieved from the storage media while an OSD only needs to be told what object number to retrieve in a single command.

This Active Storage implementation is one of the few to combine both Active Storage and Object Storage. This was done primarily because it allows these remote applications to be run closer to the disks which in other storage environments like a SAN would be impossible without falling to the block level. Active Storage applications are therefore able to interact with entire objects which are consistent between all users of the system greatly simplifying the access to data. Combining the ease which data as objects can be accessed and the open nature the applications in this implementation can modify that data lead to a more open and accessible infrastructure than other previous implementations. The openness however leads to possible security problems which is why security is of such high importance as well.

2.2 Related Work

There has been little work in the integration of OSD and active storage. Recent work has examined the use of active disk principles in the Lustre parallel file system [PNF07]. Their work also uses a “filter” in the object storage target stack that processes streams of data from defined active objects. The model is similar to the active disk streamlet model proposed by Acharya et al [AUS98]. While object-based, it is not compatible with the OSD standard. John et al. used an object oriented model that mapped OSD objects to a class and set of methods. The execution paradigm is also asynchronous RPC, but can only operate on a single object. More recently,

Xie et al. described an OSD active storage framework that defines special function objects that are linked to particular objects [XMRF+11]. Similar to the filter/stream model, the function objects are invoked automatically whenever the associated object is read or written. Again, the filter model is limited to a single or small set of objects and limits the types of applications as well. For example, functions can not create new objects or conditionally access other objects. The model implemented in this thesis provides a much richer API allowing functions to be much more full-featured. Security is provided because of the limited API (streams) and functions are assumed to be vendor-only. Earlier versions of this work allowed for policy specifications as well as integration with reconfigurable hardware [ZF08, QF06]. The closest to our work is iOSD from the Ohio Supercomputer Center [DMXW]. They also use an RPC model and allow functions full access to objects. However, they do not allow different virtual machine execution engines.

2.3 Object Storage Devices

An OSD effectively splits the file system into two parts (see Figure 2.1). The first part handles the metadata which can be thought of as data about data. This metadata contains information such as file names, permissions and what folders the object belongs to. This data can be stored separately from the actual objects data and its only relation to the object being stored is an identification number unique to every object being stored. The actual object being stored only contains this identification number and the data to store relies on this separate metadata to keep track of important information regarding the object. During a read or write the metadata can be accessed to determine the objects identification number which is then what is requested from the OSD. The OSD then handles the low level block/byte level interactions with the operating system specifying only the identification number of the appropriate object. This abstraction removes the operating system from having

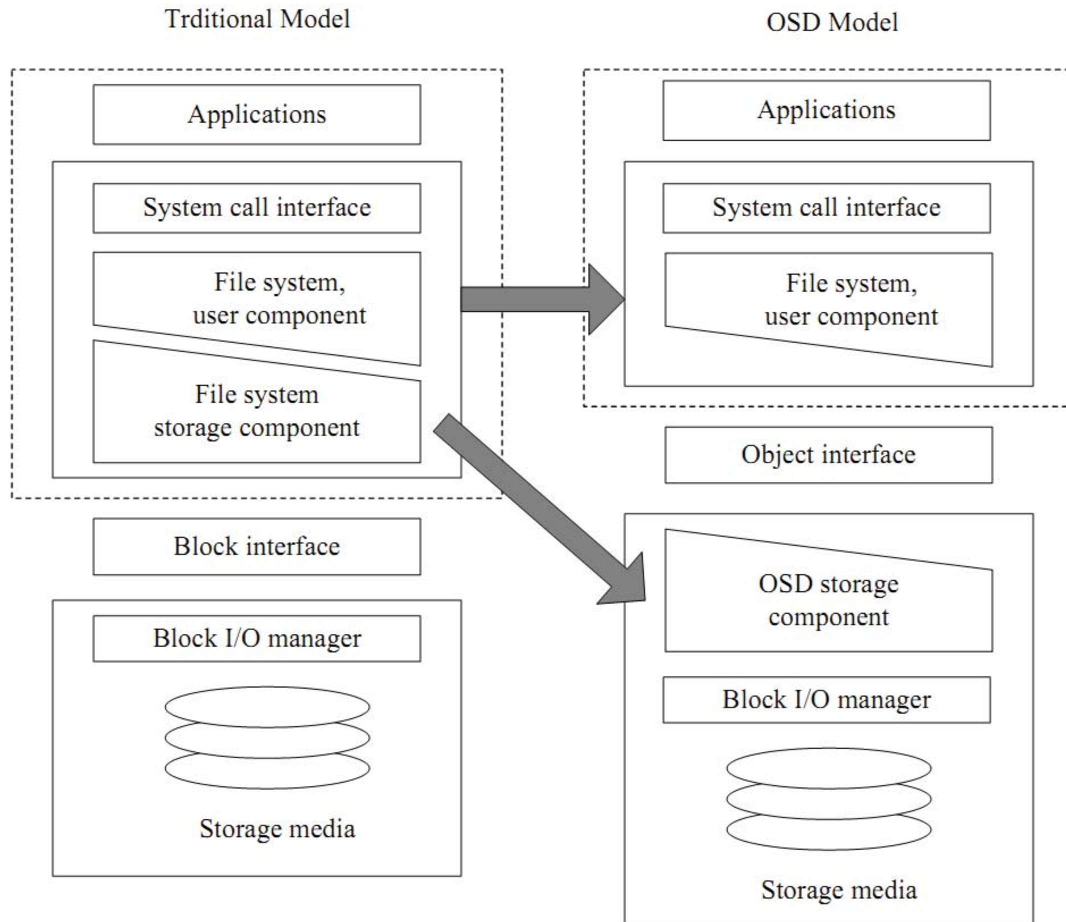

Figure 2.1: Comparison of traditional and OSD storage models [DHH+06]

a direct relationship at the block/byte level with the storage media and allows a more robust and multifaceted storage system to be implemented. OSDs so far are mostly relegated to test environments but some companies, such as Panasas [pan], EMC [emc], and Cleversafe [cle], are shipping object based storage solutions. Current storage solutions offered by these three companies are targeted towards cluster computing and data archiving. Panasas claims their Panasas ActiveStor object based system allows objects to be written in a massively parallel fashion, sets differing levels of performance, data integrity and security for individual objects, and permits faster reconstruction of objects in case of a drive failure [pan].

2.3.1 OSD Objects

This change in how the file system is viewed by both the operating system and programs should allow for much greater flexibility in how data is stored, accessed, and organized. This flexibility also comes with simplifications of the interface between a computer and its storage system through the OSD by implementing simple commands such as create, read, and write in reference to a specific object. This frees the operating system from having to determine what locations on a drive to access which can be a complex issue when dealing with operations such as changes in the size of files that are enclosed by other data around it which requiring the file to be split among many non contiguous blocks. An operating system utilizing an OSD only needs to use the simple commands and lets the OSD deal with low level administration of a storage medium. Because the operating system does not directly control the storage system at a low level an OSD storage system can be optimized as to how these commands are implemented according to the architecture of the underlying storage system possibly leading to advantages in speed compared to a typical storage system which can be made up of many discs but seen as a single disc by an operating system. This flexibility and simplification of an object based storage system from the operating

system's point of view does, however, lead to a more complex and expensive storage system. An OSD storage system has to be able to convert the simple commands from an operating system to control signals for the type of storage medium in use. It also must handle actions such as resizing of objects and changes to the storage capacity while using the storage medium as efficiently and transparently to a user as possible. Many of the advantages to using an OSD have to do with the decreased amount of support necessary to run a large high performance storage system.

In an OSD the stored objects each have a resulting ID number by which they are referenced in the future. This number does not indicate where on the storage system the object is to be found which allows the location of the object to be changed by the OSD whenever necessary and allows redundancy, backup, and increased performance to be more easily implemented without interacting with the client. Each ID number is also associated with metadata which is data about the stored object such as time of creation, user that created the object and many others. This metadata can be used to store basic information about objects in order to make searching and locating a specific object or group of objects possible without actually having to read the complete object. If an OSD can access information about the objects it is storing from either the metadata or from an attribute in the object itself it could possibly perform its own backup operations. Since an OSD would know information about the location and size of every object stored within it, it also has the possibility of easy backups without the necessity of every client handling its own backups. This is not typically possible in block accessed storage systems since the storage system is not informed of the underlying file structure and is only told what blocks to write or read and not even which blocks are in use and which are empty which forces the client responsible for the set of blocks to read then backup its own files since it is the only one with knowledge of what is stored.

2.3.2 OSD Access types

The use of an OSD system also necessitates compatible file system drivers which currently come in two variants. One variant allows the OSD to be seen and used as it was designed. This means the user can call the OSD specific commands to create, read, modify, etc. objects directly and allows for the greatest possible use of the advantages provided by the OSD architecture. The second variant provides an emulation layer of sorts which appears to the operating system and programs as a typical block accessed file system, but converts these commands into those which can be passed onto an OSD. With this emulation layer compatibility is retained with all current software not designed to work with OSDs. OSD compatible operating systems however are starting to become available due to the inclusion of OSD support in the Linux kernel starting from kernel version 2.6.30 in the form as an extension to the iSCSI protocol [Har]. SCSI which stands for Small Computer System Interface is a set of standards which define protocols, commands, and physical interfaces between computers and devices. The iSCSI protocol essentially allows SCSI protocols and commands to be sent over a typical Internet Protocol based network such as those used in almost all local area networks. Two hosts connected over the iSCSI protocol behave as if they are connected locally but can be located anywhere on the network. The iSCSI protocol was extended to support OSD commands in 2004 which provides an easily accessible way to develop OSD systems and software for use in networked environments utilizing remote storage.

2.3.3 Security

The security model used for OSDs is shown in Figure 2.2. The security protocol used has several goals, first it must prevent against attack on individual objects such as non-authorized access, second it must prevent against network attacks such as man-in-the-middle attacks, it must also defend against additional attacks such as denial

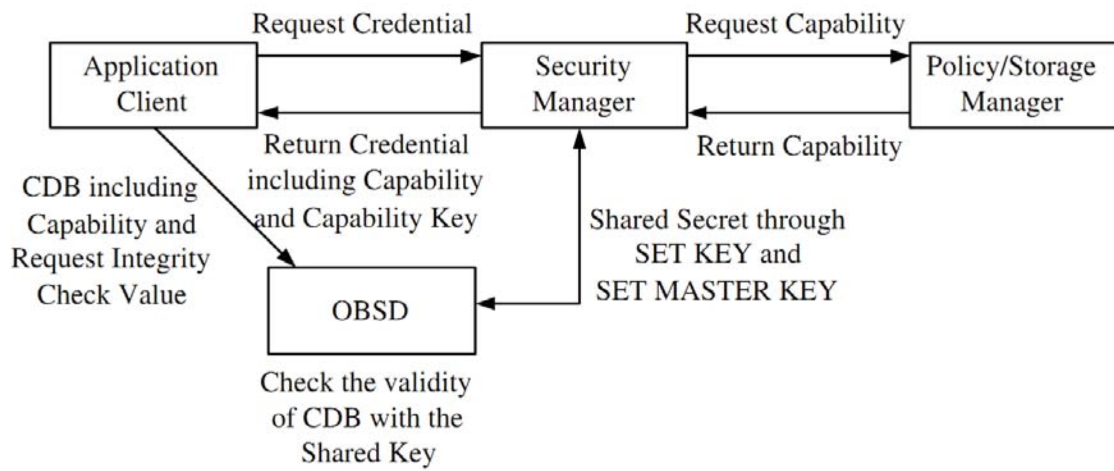

Figure 2.2: OSD Security Model [DHH+06]

of service and timing attacks [FNN+05]. The system contains three main entities: the client, a security manager, and the object based storage device. This system is capability-based meaning all requests to the object storage device include a capability which encodes the rights a holder has to an object.

The client first sends a request to the security manager requesting a capability for a desired operation on a specific object. The credential returned if approved includes the capability and a capability key. When the object storage device receives a request it uses a secret shared key it retrieves directly from the security manager to compare with the capability key to ensure the capability has not been tampered with.

There are actually two layers of security (access control and network) that can be applied in one of four configurations. The first is NOSEC which does not use any security. The second CAPKEY checks the validity of the capability sent along with every access command and provides access control security. The third is CMDRSP which ensures the integrity if the entire command and arguments instead of just the capability segment, this ensures a malicious client cannot modify or replay a command. The final is ALLDATA which provides a completely secure end-to-end verification of the request and secures all data being transferred.

2.4 Active Storage

In order for a computer to interact with its storage system it typically has to take a very active role. One way to streamline this process is to offload simple, storage intensive applications to an already existing storage system with direct access to the storage drives. A storage device with the ability to execute user defined code is known as an active storage device. Active storage incorporates very well with object based storage systems due to the fact that an object based file system knows about the objects stored inside of it and would have access to the metadata that describes the objects.

A storage system utilizing active storage would allow software code to be downloaded to and run on individual drives that are part of a storage system. This code could be used to accelerate parallel data functions including searching within objects, sorting and database operations. Since each disk would be able to run its own code on the set of data located at that drive the entire system would be capable of acting like a distributed computing system. This system represents an increase in the amount of authority over stored data given to the storage system. Not only would this system utilize an object based file system which removes the operating system from having block level control over the storage system, it would also allow the storage system itself to execute code that is another step away from a typical storage system.

The active storage system proposed here has the ability to execute any properly packaged C or Java application that conforms to the specifications and API presented here. The API includes commonly used commands such as read, write, and get/set attributes and will most likely be expanded to include all OSD commands in the future. Functions have access to many commonly used Linux libraries which have been explicitly enabled, many are left out however such as access to networking due to it likely constituting a security risk. Due to possible changes to the API over time attributes are set on both the functions and active storage enabled OSDs. This allows compatibility between a function written for a certain API version to be determined by the OSD which will most likely only support certain versions of the API.

Functions have the ability to access any OSD objects for which they have permission through normal OSD security measure. They also have essentially unlimited CPU and RAM resources while executing enabling the storage node to perform any necessary calculations as quickly as possible on the node. Although iSCSI has timeouts which limit the direct return of data at the end of a function's execution if it runs beyond them, support is given for basic long running functions and examples of how this can be done are explained in a following section.

2.5 Active Storage OSD Implementation

The active storage implementation is built on top of an open-source OSD stack provided by Panasas. They have developed an open source OSD initiator called **open-osd** that has been included in the Linux kernel as of the 2.6.30 release [Har]. In addition, Panasas has also taken over development of the Ohio Supercomputer Center **osc** OSD target [DDW+07] and released the code. Since this **open-osd/osc** stack is undergoing active development, is being supported commercially, and has been included in the Linux kernel, it is an ideal platform on which to build our active storage framework.

The implementation allows the specification of an active storage engine on the OSD node. The engine could be complete operating systems such as Linux, virtual machines such as Xen, application virtual machines such as Java, or interpreters such as Perl or Forth. The engine provides an API that at a minimum will allow methods to access storage objects and collections given sufficient capabilities. While our current API does not support these features, the API could also be extended to provide the ability to access networking functionality and other low-level device hardware such as cache, head control, and actuator control.

The current implementation has a C API engine based on a Linux OSD as well as a Java JVM engine. The engines are sandboxed so that active storage code will be restricted to specific operations allowed by the API. Ideally, the OSD would run a stripped down version of Linux that runs on the OSD that further limits the reach of active functions. Further development will include other active storage engines such as simple engines based on scripting languages such as Perl or Python.

With the ability to download code to a storage peripheral, there is certainly a concern that this code may perform unsafe operations on data. The engine sandboxing can limit any dangerous side effects of method execution by enforcing time limits on function execution and restricting resources touched by the method. In addition, the

OSD security mechanisms require that any command must provide capability keys that authenticate it in order to operate on an object. Our active storage access control methods can use the same OSD security mechanisms.

Though active storage method execution is essentially function shipping, the existing framework is not sufficient to support functional programming models such as MapReduce [DG08]. The primary deficiency is the ability to send key-value pairs to storage nodes and also repartition the data when necessary. Node-to-node communication could address part of this, but higher level constructs for functional model support of active objects is beyond the scope of this work.

2.5.1 Programming Model

The central programming model that applications use when using the active storage framework is a remote procedure call (RPC) model. In order to execute a precompiled function, it is first written to the OSD as an object, with its attributes having an added field that allows it to be specified by the type of function such as C, Java, etc. In order to invoke the function, an execute command is sent to the target along with a buffer which can contain any information the function would require, therefore acting as a way to bring any necessary arguments to the target side for execution. The command is also able to return data to the client which can include the final results of the function being run.

2.5.1.1 OSD Changes

In order to implement active storage remote execution, a few additions had to be made to the OSD specification. The main change is the addition of an `EXECUTE_FUNCTION` command that is used to trigger the execution of a function already existing as an object on the target OSD. The command format can be seen in Figure 2.3. The OSD command uses continuation descriptors to extend the send buffers to pass parameters

Bit / Byte	7	6	5	4	3	2	1	0
	CDB continuation descriptor header							
0	(MSB)	CDB CONTINUATION DESCRIPTOR TYPE (0003H)						
1								(LSB)
2	Reserved							
3	Reserved				PAD LENGTH (p-n)			
4	(MSB)	CDB CONTINUATION DESCRIPTOR LENGTH (n-7)						
7								(LSB)
CDB continuation descriptor type specific data								
8	(MSB)	CODE BUNDLE OBJECT_ID						
11								(LSB)
12	CODE BUNDLE API VERSION							
13	Reserved							
14	(MSB)	METHOD ID						
15								(LSB)
16	Method arguments							
n								
	CDB continuation descriptor alignment bytes							
n+1	Pad bytes (for eight-byte alignment)							
P								

Figure 2.3: EXECUTE_FUNCTION Format

necessary to execute a remote function - e.g. an encryption remote function that requires parameters that specify the source and destination objects as well as the key to use. Return buffers can be used as simple acknowledgments that the function executed correctly, the destination of an object where data had been written or contain the full output of a function as long as enough space was allocated before the call.

Several other smaller changes exist as well. These include an addition to the root information page on the OSD target containing information as to which virtual machines are supported. This allows a client to query the target and determine whether their executable function is compatible with the engines present on the target system. Another change is an addition to an object's attributes to specify the type of virtual machine that should execute it (C, Java, Python, etc) as well as information such as the active storage OSD API version it is compatible with, a range of APIs it is also compatible with and an implementation version which can be used to differentiate and identify differing versions of active storage functions. This attribute is set as part of a typical `set_attributes()` call from the client. If the virtual machine type is not specified or not supported by the system an error is returned to the client.

The EXECUTE command eventually is decoded on the target where the correct function object to be executed is retrieved from the OSD and hard linked into a temporary working directory where it will be executed. This is the same directory the engines are chrooted into. The type of engine to be called is then determined based upon the objects attributes and started if it has not been already. The engine then has the location of the executable object passed to it along with the arguments buffer sent with the execute command and the size of the return buffer expected. These are sent across a pipe which connects the iSCSI target with the engine currently in use. The engine then executes the function object which can call OSD commands to the target through the RPC interface and finally writes back the output data back through the pipe which is returned to the client. At a high level, the interconnected

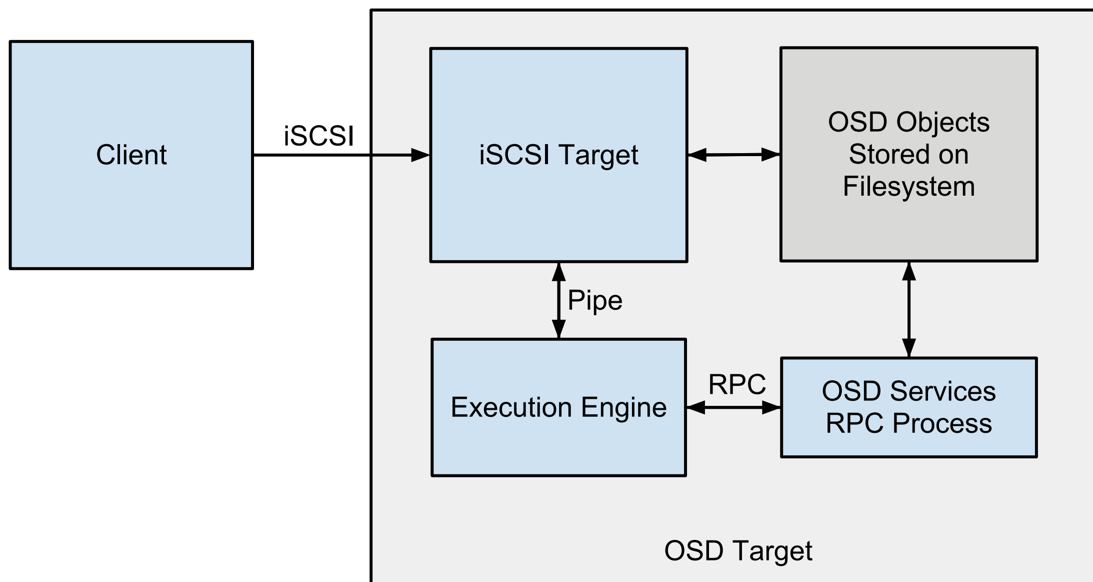

Figure 2.4: High Level View of Active Storage OSD

blocks of an active storage OSD look like the diagram in Figure 2.4. It shows the client on the left and the target on the right as well as some of the target's constituent parts, such as the separate executables in blue and the communication protocols they use.

Execution of an OSD active object function has two major complications with respect to high performance parallel computation - namely simultaneous execution and asynchronous RPC.

2.5.1.2 Simultaneous execution

Simultaneous execution is fairly tricky but allows for great speedups when information can be spread across multiple OSDs and processed concurrently. In the current implementation of the execute method, a timeout can be specified on the target. If this limit is reached the execute method called from the client returns without actually having completed and allows the function to continue its execution in the background until it completes. One way to allow multiple OSDs to execute simultaneously is to change the timeout to zero causing every execute method call from the client to cause the start of the selected function then return immediately. This would only be a short command timewise to tell the target to begin execution of the specified function. With essentially only a round trip time, a client could iterate over all the OSDs containing data the function needs to access and start execution in multiple OSDs almost concurrently. Our measurements show that the round trip is around one millisecond. The results utilizing multiple OSDs in this paper were generated using this method.

This OSD implementation is built on top of the iSCSI protocol which causes some problems primarily for long running active storage code. The first problem is that the protocol has a specified timeout of approximately 30 seconds. The second problem is that only the client can submit a iSCSI request meaning the target cannot send the

results of an function without first being requested to in some way. This limits the ability to do asynchronous RPC. Functions which execute synchronously and do not overtake the time limit are therefore the easiest to handle, but mechanisms still need to exist to allow both simultaneous execution and long running functions.

Long running functions, defined as those which take greater than the maximum allowable iSCSI timeout, have greater issues to overcome. Since the execute method would have returned there is no direct way to notify the client when the task is completed. One way to overcome this is to use one or more objects as a way to pass messages between the client and target. This could be implemented as simply creating or writing to a specific object upon completion of the function. This object could include information as to expected times to completion which could be updated at various times. The object could also possibly be used for sending control signals to the function as well such as creating a queue of objects for the function to execute on. However, this sort of communications requires polling, since the client and a running function can't directly communicate once started. Although polling can be resource intensive, if done with knowledge such as the expected time to completion it should prove to not be too hard to implement successfully.

2.5.1.3 Code Example

Figure 2.5 shows an slightly simplified example of an active storage function. The code implements a simple encryption function. The `osd_read()`, `osd_write()`, `obj_get_size()`, `osd_allocate()`, and `osd_free()` functions are part of the OSD API - i.e. OSD functions that can be called by the active storage function to access the OSD. The OSD API allows active storage functions to call the full set of OSD commands and is the only way for active storage functions to access OSD objects. Note, that the `encrypt()` call must be linked in with the downloaded active storage function since libraries such as libssl are not available on the OSD.

```
start(indata, outdata)
{
  int size;
  obj_get_size(indata.srcObj, size);

  inBuf = osd_allocate(size);
  osd_read(indata.srcObj, inBuf, size);

  encBuf = encrypt(inBuf, indata.key);

  osd_write(indata.destObj, encBuf);

  outdata = size;
  osd_free(inBuf);
  osd_free(encBuf);
  return 0;
}
```

Figure 2.5: Example Active Storage Function Code

2.5.2 Execution Engines

All the engines have at least some implementation in C which is the primary language used in this implementation. Each engine is compiled into a separate executable and contains a loop which runs indefinitely taking in jobs through the pipe interface to the target, executing them, then returning their output back through the pipe. They share a code base which contains the common code including setup such as initialization of their RPC interface and chroot setup. It also contains the C versions of all the OSD commands available in the active storage function API. When using fastRPC special `osd_allocate()` and `osd_free()` functions also exist to allow allocating data to the shared heap as well.

For now, each engine has parts written in C, although its possible to write them completely in different languages such as Java it would require rewriting some or all of the common code between the engines. In order to use other languages for now they must be able to interface with the common C code to use the fastRPC interface.

2.5.2.1 C Engine

The c-engine is the simplest of the two currently implemented engines since it requires no communication between different programming languages and should also be the fastest. Once an execute request is received by c-engine, which includes any arguments which will be sent to the function and an expected output size, the object is readied to be executed. In this case, the object is in the form of a shared library. So, first `dlopen` is called on it which returns a handle if correctly opened. The main function in every C executable should be named `start`, so `dlsym` is called to return the address of the function using the dynamic library handle and is mapped to a function prototype with the same signature as that used in the `start` function. The `start()` function is simply called as if it is a local function with its associated arguments, including a pointer to the arguments which is used as in data and to an output buffer

already allocated to the maximum expected return size. Once finished, the output buffer is simply returned by copying it over the pipe back to the main target to be returned to the client. Finally, `dlclose()` is called which closes the handle and allows for a function to be loaded on the next call to `dlopen()`.

2.5.2.2 Java Engine

The Java-engine is the other currently implemented engine. It allows for the execution of Java archives (JAR) and provides for the same API calls as the c-engine which include all those which pass through the RPC framework. It consists of a combination of Java and C code and its primary executable is written in C. The Java-engine's `main` is simply used to first initialize its side of the RPC connection, and then begin a loop which waits for data on the pipe which like c-engine consists of arguments to be passed to the function that will be called. This loop then begins executing the Java program by setting up a JVM which utilizes the Java Native Interface (JNI). The JNI relies on a set of function headers and implementation as well as a Java class containing Java versions of the same headers to call the C code from the Java program. It is through these that the OSD commands are accessed from Java functions and once past the translation layer utilize the same code as those through c-engine. Once the JavaVM is started, JNI commands are used to call a `main()` function which is part of one of two intermediate Java classes that help execute the Java remote function and the function's arguments are passed in as part of that call to its main.

The first Java class is the javaExecuter which is where the main function is defined which was called from Java-engine. It begins by using the other Java class called javaClassLoader to read in the appropriate .jar archive so it can be executed like any Java class. Finally javaExecuter creates a new instance of the active storage function that was just read in, finds the main method which will be named 'run' in active

storage Java applications. The executer then invokes the new instance of the active storage function that was requested to be run and passes in the arguments that had been transferred from the client through Java-engine. These arguments also provide for a return path for data back to java-engine and eventually the client.

The JARs themselves only contain two files. The first is the compiled form of the Java class that will be executed as a .class file. When it's compiled it requires access to the JNI header class however. The second can be generated while creating the .jar and consists of a manifest file which contains a line identifying it as a jar compiled for running on an OSD. This manifest file is checked when executed.

2.5.2.3 Execution Engine Support

Since the engines are separate pieces of code for security reasons, they need a way to access the OSD objects once they are committed to disk. The fastest way would be to use some of the same code as the iSCSI OSD server and open the directory where the objects are stored and access the objects directly from the engines. This, however, causes problems by acting in opposition to both types of security that were to be implemented. This method was however tested just to gauge the speed of the RPC OSD servers that were eventually used as a comparison to the fastest possible connection. These RPC servers are actually based off of much of the same code as the main iSCSI target, but instead of providing an iSCSI interface they provide one only through RPC allowing the engines to be segregated from the rest of the machine.

2.5.3 Secure Implementation

In order to limit the potential damage of any harmful code we utilize two methods which added together should provide a fairly significant barrier against attempts to affect more than just the OSD objects. The two methods are chroot sandboxing and multiprocess implementations.

2.5.3.1 `chroot` Sandboxing

We begin by sandboxing our execution engines individually using the operating system `chroot` command. This command is called during the initialization of the engine and limits the engine's view of the surrounding file system to only a specific directory we create and populate with only a limited selection of libraries and initial configuration files. This allows control over both what functions are available and the directories that are accessible from inside the active storage functions being run within one of the engines.

By limiting and controlling the libraries available to the downloaded code it is also possible to further decrease the potential damage by modifying those libraries. Although providing a large number of libraries allows additional functionality, it also can expose the ability for code to run functions that could be dangerous to the system that reside alongside a library containing other important or useful functions. To solve this it would be possible to create active storage specific versions of common libraries that remove any functionality deemed harmful. Libraries could also be created to provide additional functionality such as inter-OSD target communications which could allow active storage functions to communicate to other targets or even active storage functions running on those targets.

2.5.3.2 Multiprocess Implementation

Sandboxing can prevent active storage functions from causing damage to objects outside the sandbox. However, there is a potential for other unauthorized activity by active storage functions. For example, consider an active storage function that is linked directly in the same process to libraries that provide OSD services such as create object, read object, write object, etc. These libraries will have code that check for capabilities that enforce OSD security mechanisms. However, if the active storage function is in the same process space as the OSD services library, the function could

theoretically sidestep these checks and access objects that it may or may not have access to. The solution to this is to implement the OSD services library in a separate process from the engine executing the active storage function. That requires an RPC mechanism to communicate between the engine and the OSD services library. This split allows the engine to remain in its sandboxed environment but still be able to access the stored objects, though indirectly.

Remote Procedure Calls (RPC) are a type of inter-process communication (IPC) and are used here to allow a separation between the execution engines and the OSD targets for security purposes. As opposed to other IPCs such as pipes or shared memory areas which are usually fairly simple, RPCs essentially allow functions to be called on the remote process. The specified function calls to be remotely executed have to be packaged up with their parameters in order to be sent to a waiting process typically over switched network protocols such as TCP or UDP. RPC is typically used for communications over IP networks but can also be used locally on a machine between two processes that have the ability to execute those functions. Output data is then returned to the process that started the RPC. RPC is a very robust framework and is used as a basis for the Linux Network File System (NFS).

One of the advantages of RPC that led to its use here is the ability to communicate out of a sandboxed environment such as the one that is used here. The RPC implementation will be used to allow aforementioned sandboxed active storage functions to retain their ability to interact with the OSD such as through reads, writes, etc.

At the bottom and supported by several individual active storage functions is the code to be executed. In the case of a C program it is a shared library and with Java, it is a Java archive (JAR). These active storage functions are then executed from within execution engines that exist on an active storage enabled OSDs. It is also through these engines that OSD commands sent from the running downloaded code are able

to make their way across an RPC interface and out of the sandboxed environment before being executed at the RPC target. The RPC API currently includes the most important OSD commands, though not all, including read, write, get attributes, etc. The RPC API is the only way to access the OSD objects from the active storage functions.

RPCs have two ends, one is built into each engine and the other is a separate process that interfaces directly to the OSD and is effectively an RPC version of the iSCSI target reusing much of the same code. This system allows the engines running their untrusted code to exist in their sandboxed environment while still being able to access the objects stored on the OSD. The RPC target is however only able to handle one request at a time and sends back its result when the OSD command is finished.

Our original implementation of RPC used the standard SunRPC/ONC library [Sri95]. SunRPC is a relatively heavy RPC implementation designed to allow distributed systems communication across multiple heterogeneous computers. As a result, it supports object serialization/deserialization and machine-independent data transfers. For the purposes of our requirements, these were features that were beyond our needs. Since all RPC calls are to the local machine, there is no need for machine-independent data conversions and data transfers do not need to be performed through socket calls but can be done through other local IPC mechanisms such as shared memory.

FastRPC The FastRPC [Hea06] system is designed to provide a fast intranode RPC mechanism using shared memory as well as a secure way to call functions in a piece of code that was either not trusted or had the possibility of crashing. One example of which is an image decoder that could contain exploits and could be run separately from the main program to limit possible unwanted activity including access to private data. FastRPC uses two simpler and faster methods of communication between the two processes and only works in one direction (master calls functions on the slave) which for active storage purposes is from the engine in its sandbox to the

OSD services process.

The first communication type it uses is a pipe that is used to send the function prototype to the slave. This includes a number to reference which function is requested and any data necessary to run that function such as its arguments. This pipe is also used to send back any return arguments. Because the heaps are not synchronized between the processes, variables passed by their pointer would not normally work and these limits would make it impossible to call the OSD functions. Pass by reference is required in order to allow for large buffers. This is accomplished by using a shared heap that is mapped to both processes (engine and OSD services) with the same starting address and size. This heap can have large data structures allocated on it which enables the functions to pass pointers instead of having to copy the structures or buffers as an argument. This is used in all the OSD commands such as read and write to move their possibly large buffers between the sandboxed engines and the OSD services process.

FastRPC Security The strength of RPCs is that only methods defined and implemented on the remote side can be run since no actual code is passed over the connection, just the arguments and eventually the return data. The RPC API is limited to those used to access the OSD and is the only means provided to the active storage functions to access data outside of their environment ensuring along with the chroot that private data such as system files cannot be accessed. This provides an additional layer of protection here since it means the code that will actually run on the RPC target can be considered trusted, limiting the untrusted code execution to the sandboxed engine.

FastRPC also has similar security implications as the original RPC interface, although with slightly different mechanisms due to being designed to work only on a single machine. FastRPC also allows only the predefined functions to be called from within the engine and also will only be running trusted code on the FastRPC target

which implements these functions. This RPC sends similar information compared to the Sun version though its differences are mostly in how. Here the engine is only sending a number which is used to reference which function is to be executed on the remote side along with the arguments that are part of the function being called instead of having to pack up any large data structures before being sent off.

Unlike the original RPC, the engine and FastRPC interface have a shared memory heap on which they can allocate memory and can pass pointers as the arguments instead of having to copy large amounts of memory over the pipe interface between the two. It would, therefore, be easy to write program code into the shared heap as it would be for any buffer and attempting to have it executed. It would, however, require finding a way to have the OSD services process somehow begin executing that code.

2.6 Results

Active storage enabled OSDs have a chance to increase performance for data intensive applications in several ways. The first is the ability to do data intensive operations on the storage node where the data is stored instead of having to transfer it to the client for computation, also reducing network congestion. The second is that when using a system with multiple OSDs, one can split the data equally among the OSDs, then simultaneously call an active storage function on all OSDs. This allows for potential parallelism by reducing the time needed to do certain data intensive tasks by a factor of the number of nodes that can be utilized.

2.6.1 Performance

Since OSDs are a network storage system, it is a safe assumption that a storage node or set of nodes could contain all the data of interest to someone and in these tests we make that assumption. The multi-node results were taken from experiments

performed on a 16-node cluster where each node contains two 1.8 GHz dual core CPUs, 2 GB of memory, and a 80GB 7200 RPM SATA drive used for testing. The standard Linux kernel (2.6.38) was used with buffer caching enabled. The nodes are all connected through a dedicated gigabit network switch.

2.6.2 Impact of Data Transfer

This first test shows the results for an AES encryption, then decryption of a 128MB file. Both local and active storage based tests worked on a 4MB block size and would read, encrypt or decrypt, then write the result to an object which holds the output. Times were recorded from the client and include any overheads associated with either the iSCSI protocol and/or active storage functions and are in milliseconds. These overheads will be quantified in a following section however. The object to be acted upon already existed on the OSD and was generated with a random file generator. The numbers are the averages of 10 runs with variances between runs being less than 250 ms.

Figure 2.6 shows how much active storage can decrease the total times for even a very CPU intensive operation such as encryption. Here the active storage versions take approximately 2/3 the time that the local version takes. Figure 2.7 shows a closer look at why the AS version is faster by looking at only read, write, and encrypt/decrypt times. These times are from the same test as the previous graph.

Even without any of the overheads shown here, we can see the total times are almost exactly the same as the previous graph. The time needed to do the actual AES operations is fairly static between local and AS versions with local being faster for encryption but AS faster for decryption. The read and write times are very different though. The ability to work on data locally (in the case of the AS times) removes the need to shuffle the data across the network and is able to almost completely remove the data transfer time, leaving only the AES operation as the main contributing time.

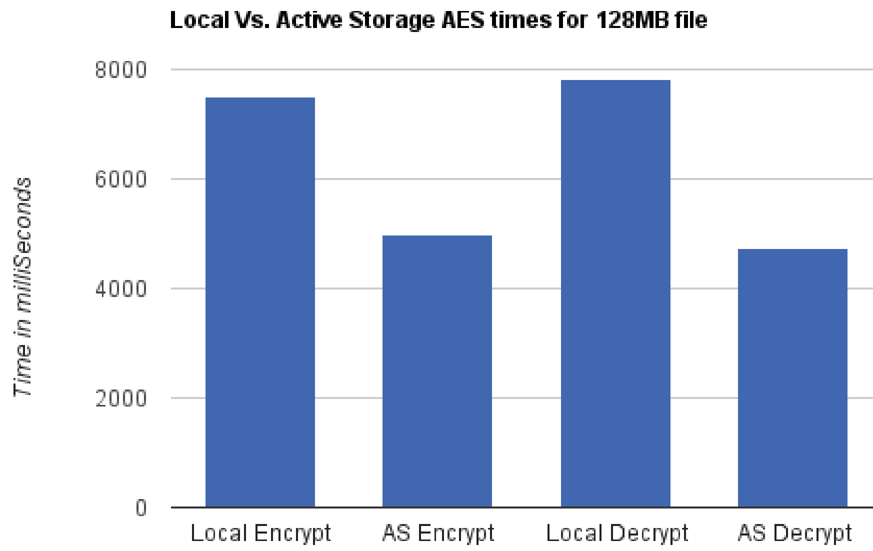

Figure 2.6: Local vs. Active Storage

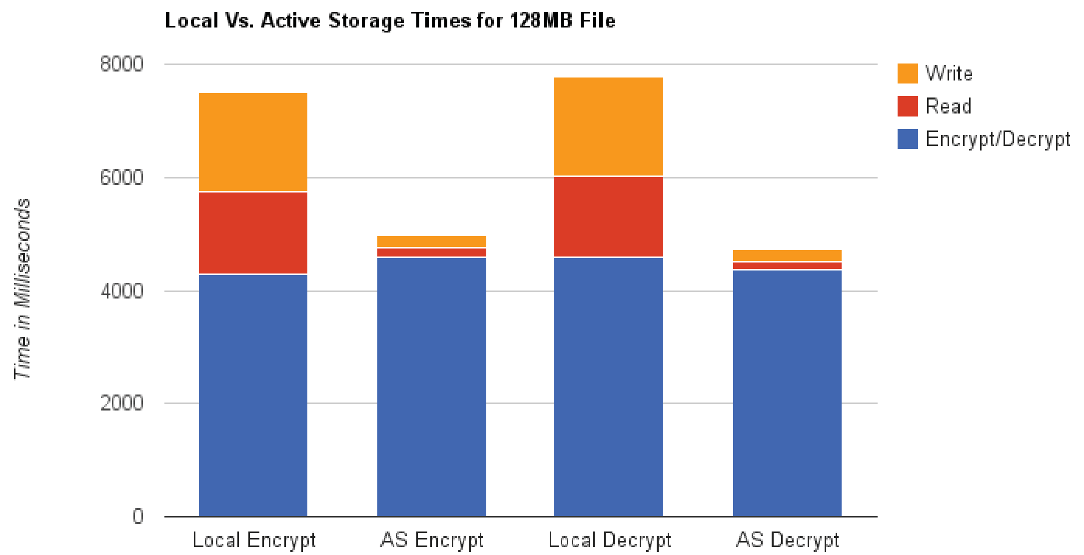

Figure 2.7: Local vs. Active Storage Breakdown

2.6.3 Evaluation of Multiple OSDs

One of the biggest areas impacted by active storage is when multiple OSDs can be used simultaneously to quickly handle large data intensive applications. Several active storage functions were run on the cluster utilizing up to 8 nodes as AS enabled OSDs. These included a function to simply count the number of occurrences of a value, a simple version of grep and AES encryption. They were run on 1, 2, 4, and 8 nodes and used files of sizes 128, 156, 512, and 1024 MB. These files were generated with a random file generator so the pattern matching and simple grep program were able to locate data instead of searching a zeroed out file. Five runs of each test were completed at each combination of the number of nodes and file size. The average time was taken across all five runs and the speedup compared to a single Active OSD was then calculated where a speedup of 2 would indicate a halving of time. The standard deviation was also calculated as a percentage of the total execution time in order to compare the variation among runs whose run times can vary drastically.

The first is a function we call wordcount, which simply reads in any size buffer and looks for appearances of a specific 8-bit character, in this case a space. It is the simplest program used in our testing and represents an application that is only read intensive with very little output other than a single number.

Figure 2.8 shows the speedup in multiples compared to the first run for the different file sizes. The lines for the various file sizes overlap each other as they had very similar speedups. The scaling of speedup with the increasing number of OSDs is nearly perfect with little deviation. The average standard deviation percent of total time here is only 3.51% which is consistent with the near perfect speedup seen.

The second function is a version of grep. This simply takes in a parameter from the client of the word they are looking for along with an object for its input and output. Instead of text files with line breaks, we use randomly generated files for these large file sizes. As a result, the grep function reads through the input object

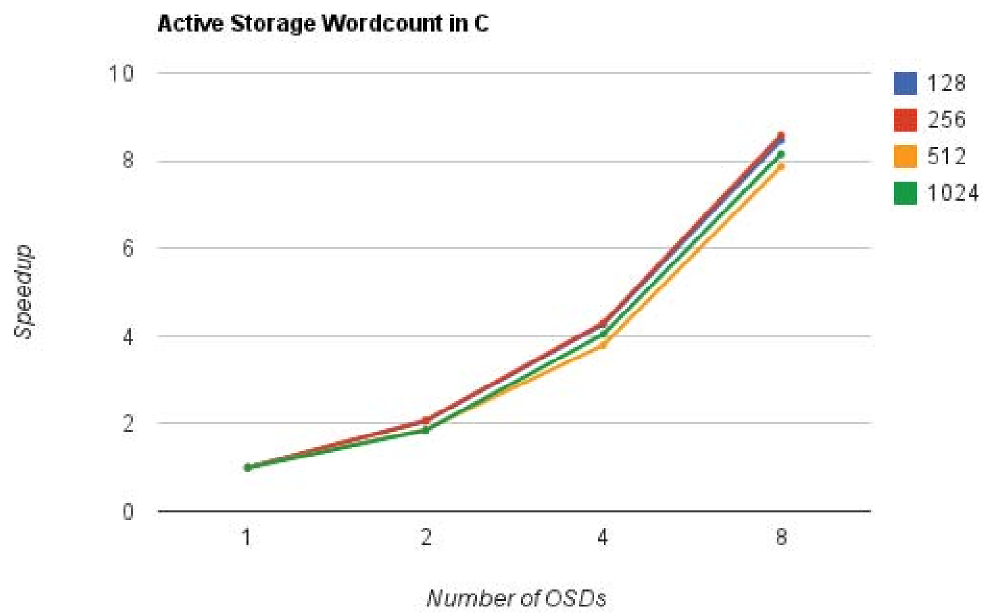

Figure 2.8: Active Storage Wordcount Results

128 characters at a time which is about the length of a normal line of text. If it finds a match it writes the output line to the output object. In this case the word "the" was used as the search word and showed up numerous times in every random file.

Figure 2.9 shows a similarly concentrated grouping, with all file sizes scaling fairly close to optimally. The average standard deviation percentage here is very close to the previous test at 3.76%.

The last function provides AES encryption or decryption. It takes in arguments of an encryption key, input object and output object. It represents a program that has reads and writes along with a CPU intensive task. The results shown in Figure 2.10 are only from encryption though decryption shows near identical results as seen in Figure 2.7.

This more write and CPU-intensive program shows slightly superlinear with sizes of 1024MB. This is most likely due to the total read + write size of 2048MB which will not fit into the disk cache of a single OSD which has 2048MB RAM. Previous tests only handled up to 1024MB reads at maximum which should have been cacheable. Tests with 2 or more OSDs used at most 1024MB reads + writes which should fit in the cache. The average standard deviation percentage here was slightly higher than the other two at 5.15%.

2.6.4 Overhead Analysis

A system such as this one with multiple layers and communication methods being used concurrently has many possible significant sources of overheads. A single EXECUTE_FUNCTION OSD call from the client must be encoded and sent over the iSCSI link to the main OSD target. It is then passed to an engine for execution over a pipe. The object to be executed has to be either loaded as a shared library in the case of a C function or read in as a class if it is a Java function. While executing, the functions must also use RPC in order to call OSD commands instead of calling them

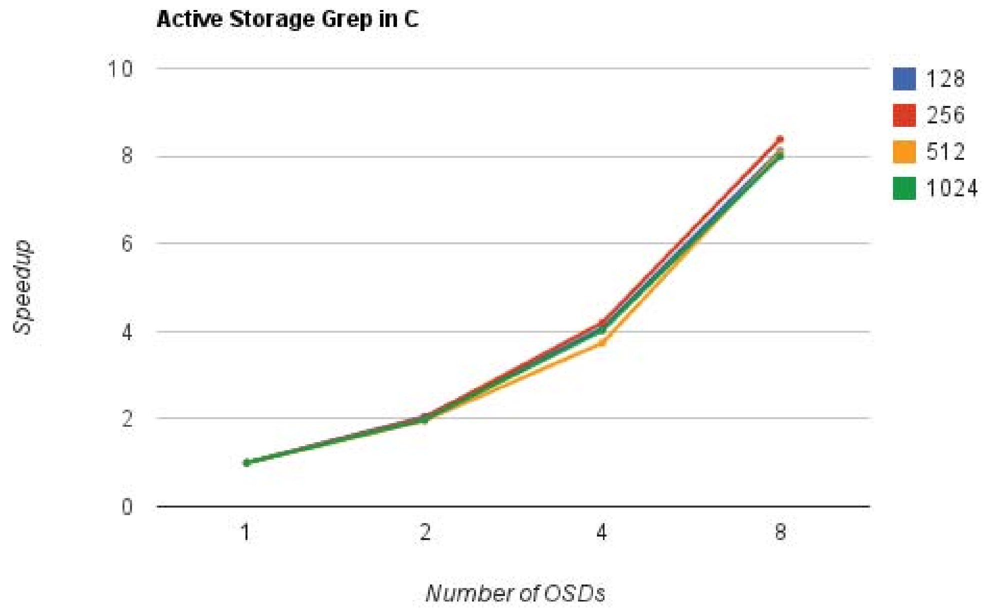

Figure 2.9: Active Storage Grep Results

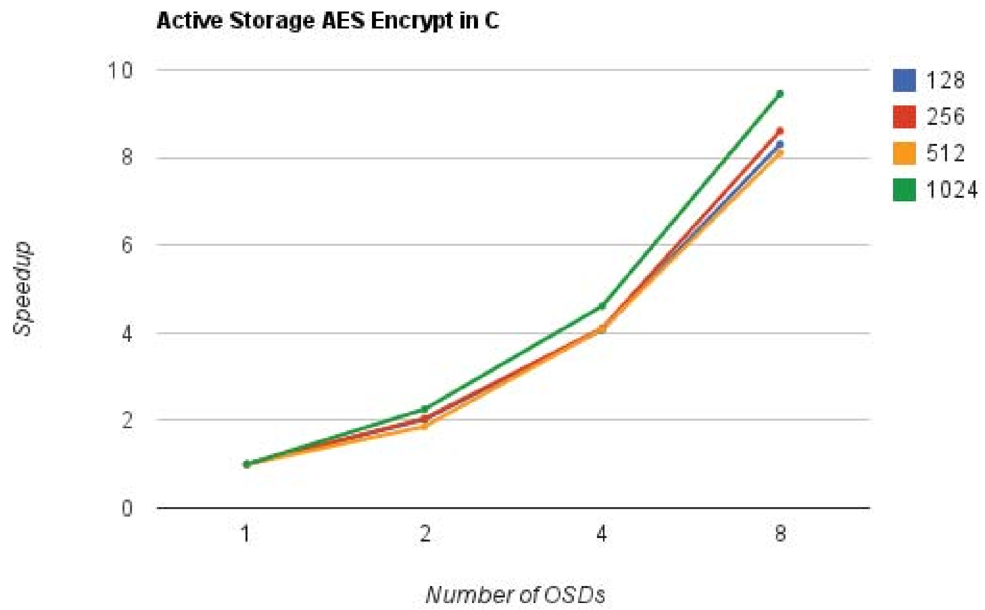

Figure 2.10: Active Storage AES Encrypt Results

directly. Finally when the functions return, their return data is sent back over the pipe out of the engine back to the iSCSI target, then over iSCSI back to the client.

These overheads exist as two kinds. The first is fairly independent from the function being called and includes iSCSI latency and the time needed to communicate from the iSCSI target to the engines and back. This overhead is actually quite small however. A test was run using a "no-op" function which is executed on a remote target. This function only takes in a typical input (PID and OID) which is only 128 bytes, and returns 96 bytes of hardcoded information without making any RPC calls or doing any arithmetic operations. This, therefore, is a measure of these latencies and averaged over ten runs was only 415μs with a maximum of 552μs.

The second depends on the number of OSD commands that are called from the function and include all communication over the RPC interface and for Java the additional interface between Java and C. These add up as multiple OSD commands are called and have the greatest possibility of slowing the executing functions down, especially when many small accesses are used.

2.6.4.1 Impact of Sandboxing

If security was not of importance, a RPC system would not be needed to allow OSD commands to pass from the function being executed to the OSD system. This allows the engine to have the OSD code compiled into itself which gives the executing functions direct access to OSD commands without having to use any communication system. For testing, this direct communication system was implemented and represents the fastest possible way to interact with the OSD and its objects. This direct method, therefore, does not perform `chroot()`, does not exist in a sandbox or use RPC.

Figure 2.11 contains the read and write times that were collected as part of an AES encryption function in C. These are the total read and write times collected

from inside the AES function that was being executed. The file size was 64MB which resulted in 16 - 4MB reads and 16 - 4MB writes and these were averaged out over ten runs. The times reflect that taken for all 16 reads or writes not for a single read or write.. One function was run using fastRPC through an engine that was sandboxed while the other did not use RPC, was not sandboxed, and communicated directly to the OSD files. We can see that reads take approximately the same amount of time with only a .4ms slowdown or 1.9% for RPC. Writes have a larger difference of 8.9ms or 12.9%. The speed of the AES encryption which was not shown in the graph remained unaffected, the direct function actually averaged a few milliseconds slower than the RPC encrypt, but over 8̃00ms the difference of 14ms should be due to run-to-run variations that are typically greater than this difference.

2.6.4.2 Impact of RPC

The current communication method between the engines and the OSD objects is through fastRPC due to its speed in comparison to other tested RPC frameworks such as SunRPC which was initially used in this system. Figure 2.11 can also be viewed as showing the slowdowns caused by using RPC over direct calls. SunRPC was also tested in the same group, but is not included in a graph due to its large increases in both read and write times. SunRPC reads took approximately 169 times longer while writes took 26 times longer. This is due to SunRPC being designed for communication not just between local processes but between those processes across networks.

FastRPC benefits greatly from it's being designed only for local communications. This allows its speeds to be very similar to those achievable without any interprocess communication. Although the small increases in time due to fastRPC compared with direct calls are near those of run to run variations, they always show at least some increase in time compared to direct OSD calls.

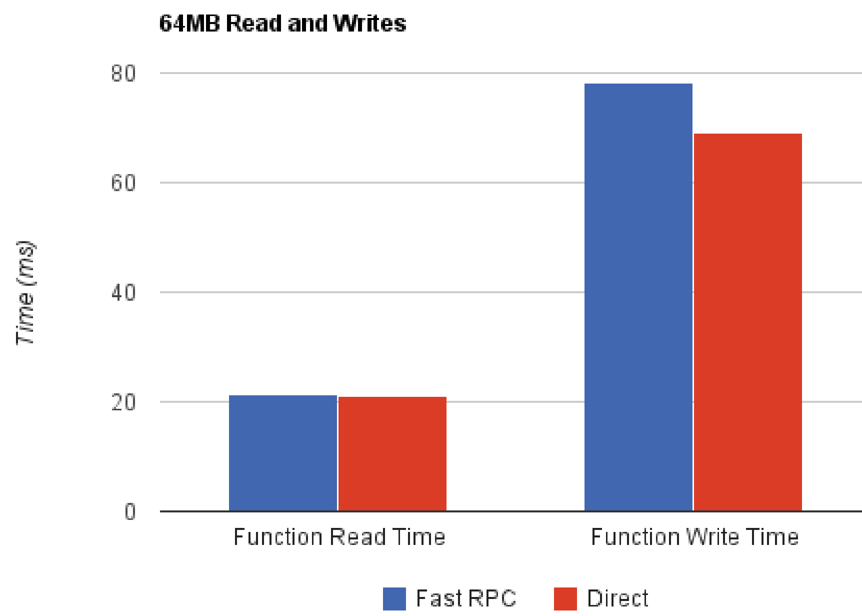

Figure 2.11: Active Storage FastRPC vs Direct

2.6.4.3 iSCSI and Asynchronous Timing

When testing asynchronous active storage functions, total times cannot be determined from the client due to the iSCSI target returning the execute function method immediately. The timing must therefore be done from within the engines and covers the time necessary to load the function and execute it. The data for the results section utilizing multiple active storage OSDs was obtained this way by timing the call to the shared library that contains the function to execute. This, however, does not include either the iSCSI latency or that from the communication between the iSCSI target and the engine being used. This section will quantify the additional time that was not accounted for in those tests.

Figure 2.12 shows the results for a 128MB AES encryption then decryption averaged over ten runs. The execute command was used synchronously so the client times include the time required to execute as well as the additional overheads such as the iSCSI transfer times. The engine execute time only covers the time to execute the active storage function and is measured the same as the tests covering multiple OSDs asynchronously. The difference between the two times is very small with the client registering an extra 1.6ms for the encrypt and 1.7 ms for the decrypt. These represent a very small overhead unless very small objects are to be acted upon. It is due to this small value that this overhead was not mentioned by value in the previous section which focused on scaling across multiple OSDs.

2.6.5 Evaluation of Multiple Execution Engines

Currently two engines are implemented enabling both C and Java code to be executed. The C engine is the simplest of the two due to it being able to use common dynamic library loading, unloading, and execution commands. The Java engine however must load the Java function into the Java virtual machine and also pass all OSD commands through the Java to C interface before passing over the same

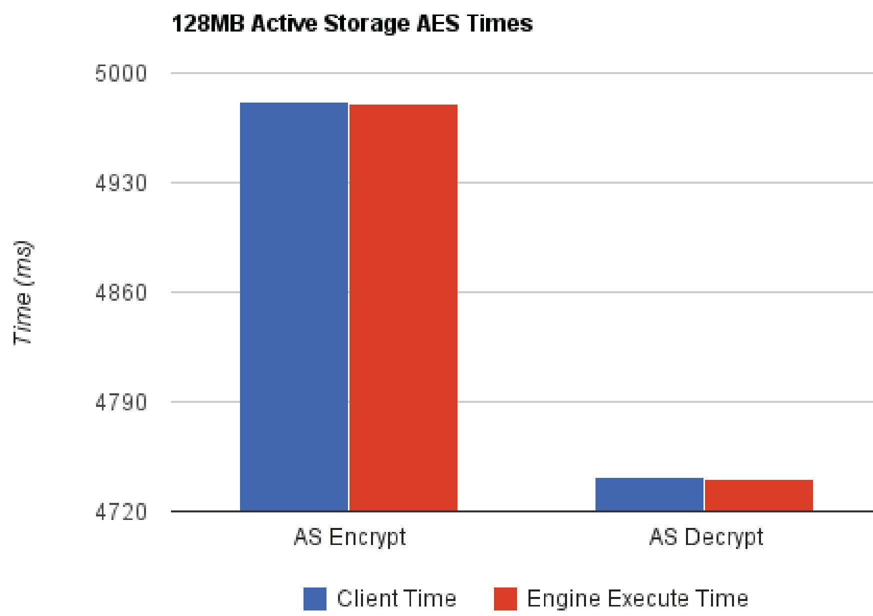

Figure 2.12: Active Storage Client vs Engine Times

RPC interface also used by C functions. This should cause a decrease in performance in addition to the typical performance difference between similar Java and C code. The code used between languages is very similar in all cases, only switching out any necessary function calls and using data types relative to each.

Figure 2.13 shows a comparison between the total execution times of both grep and wordcount run in both languages. The times were taken from the client with neither performing any writes. With only a 32MB file size the C version is much faster in both cases while the Java version takes 3-4 times as long. This difference is most likely due to slower execution, and Java to C interface issues. In order to remove any purely language related execution time issues and focus on the Java to C interface, a new function was created to simply copy an input object to a destination object using as little code as possible. It performs the copy in 4MB chunks and includes timers for the read and writes.

The file copy was run with a 128MB file which would require 32 reads and 32 writes. Figure 2.14 shows read times while Figure 2.15 shows the write times. Times were taken from two locations, the first from inside the function being executed which are named either function read or function write times. The second set was taken from the engine which recorded the times to make the RPC calls for either the reads or writes. The times were then combined to get the totals for each set of reads and writes.

The engine read and write times are fairly consistent between languages. This is expected since the RPC calls to the OSD Services RPC process are the same for both Java and C. However, the timing for the function calls shows a difference. C shows almost no overhead as expected in calling the RPC OSD functions. An overhead however can be seen when comparing the function times for both Java read and writes and is approximately 200ms for each. This number does not include any JavaVM setup times since it is derived by timing only read and write calls inside the

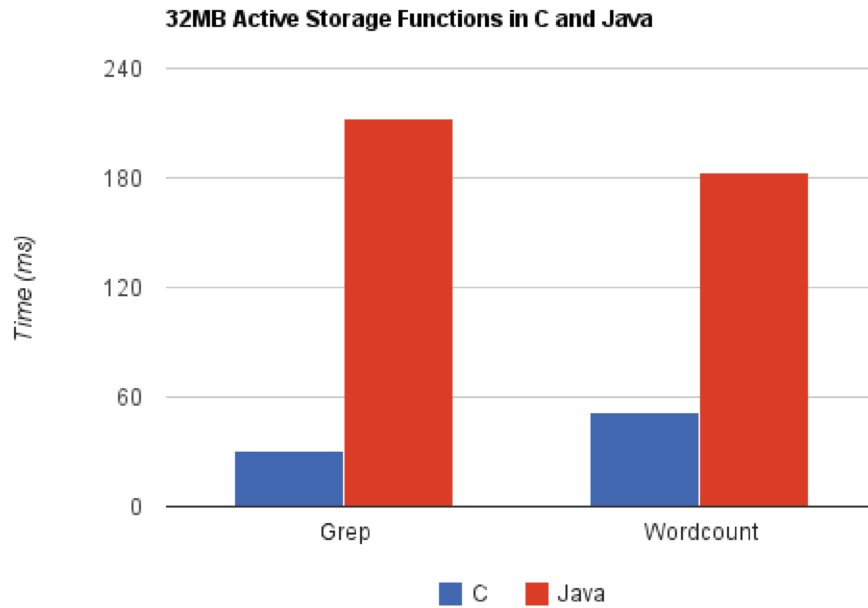

Figure 2.13: Active Storage C vs Java Times

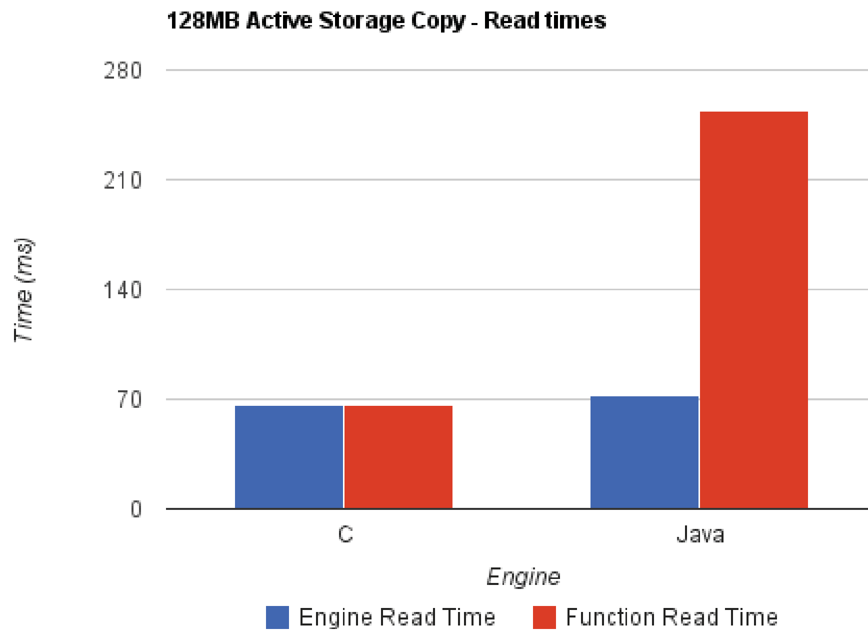

Figure 2.14: Active Storage C vs Java Read Times

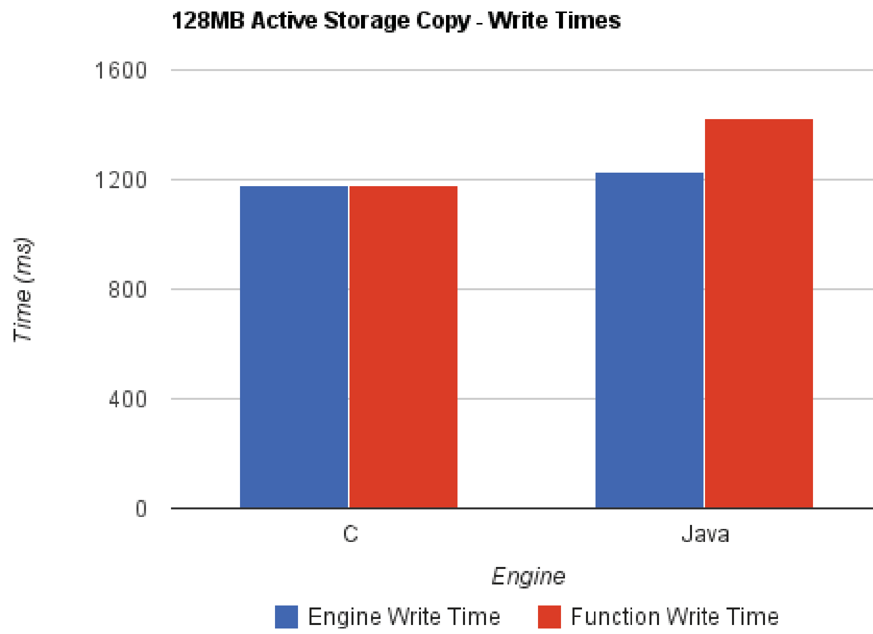

Figure 2.15: Active Storage C vs Java Write Times

Java active storage functions. This time is therefore attributable to the time needed to cross the Java to C interface. It includes the JNI interface as well as memory allocations, copies, and frees that are necessary to allow the Java functions to utilize fastRPC due to them not being able to perform these functions directly to the shared heap. Methods to allow Java to directly call the fastRPC routines without making the Java to C transition are being investigated to reduce this penalty.

2.7 Summary

This chapter explores the design of an active storage framework utilizing OSDs and demonstrates the ability to remotely execute functions allowing for performance increases compared to local execution. It utilizes an execution model based on a remote procedure call model whereby functions are downloaded to an OSD for execution. The design allows for multiple execution engines which each support different programming languages with high sandboxed security and relatively low overhead. Performance results show performance gains utilizing a single OSD with additional scalability using additional OSDs. The results also show low overhead due to the sandbox excluding the additional overhead due to the JNI interface usage.

Chapter 3

SSD Caching with Object Storage Devices

3.1 Introduction

RAID arrays can help increase sequential transfer rates for group of HDDs to over that of a single SSD but they cannot decrease the access times below the physical limits of a drive, especially when all drives are mostly filled and random access patterns are prevalent. This leaves open an area for optimization where new hardware and/or software can be used to increase performance. The method proposed here is to use an SSD as a cache to decrease the access times to data that primarily exists on a RAID array in order to decrease the apparent access times to files from the 10-15ms of HDDs to close to the .1ms of a SDD. This could provide a big decrease in access times on either a local machine or on networked storage.

One way to do this is to optimize for entire file accesses and place the first few MB on the SSD while the rest remain on the RAID array. Small files will therefore fit completely on the SSD and receive a significant increase in throughput. When a request comes in to read or write a larger file this request it is split up and the first few MB are accessed from the SSD while a request simultaneously goes out to the HDD accessing the rest of the file. The first few MB of the file should begin returning from the SSD very quickly (.1ms) and should transfer enough data to cover the access time of the HDD. Once the HDD has moved it's head to the currently requested file

(assuming little or no fragmentation), further access requests will have little to no access time penalties.

This structure, however, is fairly hard to implement in a way that would interact with the Linux kernel and allow for redirecting of file read/write requests from a single drive to multiple drives. One way around this is to use an object based file system using an Object Storage Device (OSD). An OSD instead of being a block level device deals with objects at the lowest levels in the operating system. In an OSD all requests use an object ID including read and writes which also specify what part of the object to access using a simple offset (in bytes) and length to read or write. This allows requests to easily be split on the target between two physical disks while appearing as a single OSD, one which uses the SSD and the other which uses the HDD or an HDD RAID array.

This system is designed to greatly reduce the penalty of random access to many small files over a large percentage of a drive. It does this by saving small objects completely on a low latency device while also allowing large objects to be spread between this drive and a larger high latency drive. This split helps to hide the large drives latency by utilizing this time to begin the transfer from the small low latency drive while the larger drive is seeking. It is also light-weight due to the simplicity of the algorithm that splits the incoming read or write access requests being much less than those used on more complicated systems.

Other systems tend to hold only a subset of blocks which must be determined through searching databases holding the listing of blocks available in the cache due to dynamic mapping. Typical write back caches, once full, must also move blocks to the HDD which causes additional overhead where multiple reads and writes can be created from a single initial write due to thrashing between the SSD and HDD.

3.2 Related Work

There are many SSD caching methods proposed in various academic papers[KRM08] [SPBW10] [YNS+08]. The first of these three [KRM08] proposes flash plus small DRAM system primarily as a replacement for DRAM that is typically embedded on HDDs. The second, [SPBW10], introduces Griffin, which uses a HDD running a log-structured file-system as a write cache for a SSD in order to extend the SSD lifetime and reduce I/O latency. The final one, [YNS+08], proposes using a high speed Ferro-electric RAM which has a high speed random read access to speed up a NAND flash based SSD.

One of the more interesting is titled "SieveStore: A Highly Selective, Ensemble-level Disk Cache for Cost-Performance". [PT10] It was designed to capture the top 1% of the most popular blocks of a storage system and place them into the SSD cache. Its architecture was based on storage traces they conducted which showed 1% of blocks accessed each day make up 14-53% of accesses, 99% of blocks are accessed less then 10 times each day, and the distribution of popular blocks varied across the storage nodes. From this they decided that it would be more efficient to use one central SSD instead of one at each node, and instead of caching recently used data they use sieving to determine the most popular blocks over a specified time period and cache them. In the end they were able to use sieving to improve the hit rate 35% over unsieved caching, and reduced the number of writes to the SSD by a factor of 100 compared to the unsieved caching. They also were able to use a single SSD to satisfy the bandwidth requirements for the entire system (5-10TB).

3.3 Architecture Overview

The overall architecture is shown in Figure 3.1 and is conceptually very simple with the OSD target appearing to the OSD client as a single OSD target. When the

client sends a request the target receives it and determines whether it fits within the SSD's range and/or the HDD's range. The request is then forwarded to one and/or both drives asynchronously which allows one or both drives to access the read/write buffer simultaneously. There are two kinds of OSD commands that are important to differentiate between in regards to caching. The first are metadata requests which have no interaction with the underlying data and are thus unchanged by adding caching. An example is a get_attributes request which returns data such as the file size and other metadata. This call is satisfied by returning data stored in a database so does not need to be modified to handle caching. The second type are requests such as format, create, write, and read which interact with the stored objects. These all need to be modified to handle two separate drives.

3.3.1 Request handling

As requests are received by the target their type is determined and they are then dispatched to specific functions that handle the low level execution of the specific command. OSD commands supported by this caching implementation have additional code which determines whether further interaction will be required with the HDD and/or the SSD. An example is an object write. If the offset + size is less than the current SSD cache size then only a single write request is sent to the SSD. However, if, for example, the write size is larger than the SSD cache size and the object is currently empty, two write requests will be dispatched asynchronously. The first request is to the SSD with a write size the same size as the SSD cache size, and a second to the HDD containing the remaining data. Once both writes are complete the completed write request information is returned to the client and if any errors occurred writing to either drive a write error would be returned.

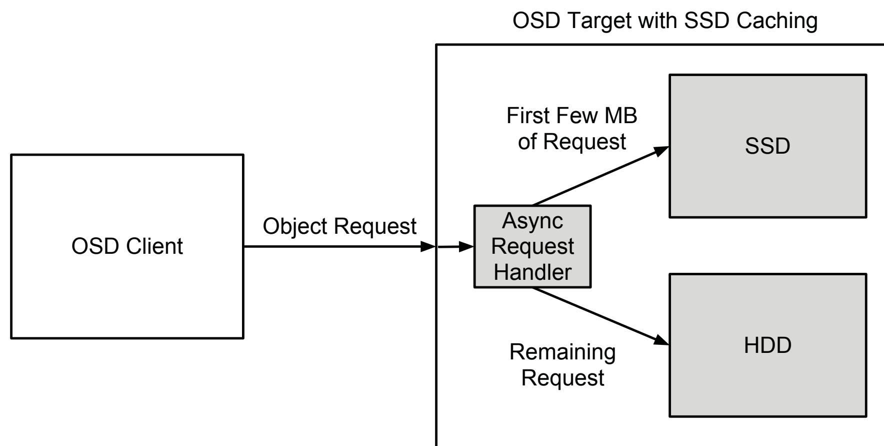

Figure 3.1: OSD SSD Caching Architecture

3.4 SSD Cache Size Determination

One of the largest driving factors in building this SSD caching implementation is determining the ideal cache size for the SSD - i.e. the ratio between HDD and SDD sizes, and also the threshold for the size of files/objects to put on the SSD. One factor is the distribution of file sizes in a file system and another is the difference in access times between a SSD and HDD. In 2007 a paper was published detailing the metadata from over 10,000 filesystems [ABDL07]. The results are fairly surprising and show a large percentage of files are actually very small. The data shows that files of 1MB and below make up approximately 98% of the number of files on the systems. These same 98% however only take up about 20% of the actual space on the drives which would mean an 500GB HDD could be easily supplemented by a 125GB SSD and provide SSD access times and data rates to 98% of the files. So many files have a small size that a target of storing 80% of all files on the SSD could be met by only caching 32KB files and smaller on the SSD which would result in storing about 5% of all used data on the SSD which gives a ratio of 20:1 allowing the same 500GB HDD to be supplemented by a 26MB SSD.

Another factor in determining the size is the difference in access times and throughputs of the SSD and HDD. In benchmarks the HDD used in these tests typically averages a throughput of 65MB/s read and write and an access time of 15ms while the SSD manages a throughput of 150MB/s and 275MB/s for writes and reads respectively and an access time of only .2ms. Even using the slower SSD write speed due to it's negligible access time the SSD can transfer 2.25MB before the HDD even begins returning data. Between these two factors a file cache size of 2MB was decided upon for testing.

3.5 Implementation

This implementation is completely relegated to the OSD target requiring no change to an OSD client in order to benefit from any and all advantages this system has. It is built on top of the OSD target initially created by the Ohio Supercomputing Center which has since had its development taken over by Panasas [Har]. This is also the same target which was modified to enable active storage and both can be run concurrently. Although OSDs support many commands, only a subset were implemented to enable testing. In order to use applications not designed with objects in mind, Exofs was used which allows Linux interact with an OSD using the API from a normal Linux file system.

3.5.1 Supported OSD Features

Due to the design of the OSD Target every OSD function that interacts by reading or writing to an object has to be modified to handle the SSD caching system. For this implementation only commonly used OSD commands were modified which include `osd_format`, `osd_create_partition`, `osd_create_object`, `osd_remove_object`, `osd_write`, and `osd_read`. These few commands in addition to the unaffected metadata commands allow for all of the typical filesystem commands to be used as expressed by this target supporting the EXOFS filesystem.

3.5.2 OSD Format

A format is usually one of the first functions called on an OSD, it removes any objects and their associated metadata from the OSD and creates a partition of a requested size for new objects to be created in. In this implementation a format removes the database file which holds the metadata along with all objects stored in the root directory. It then creates a new database and rebuilds a directories to hold the objects. For SSD caching the deletion and recreation of the location for the

objects is duplicated so the format affects both locations where objects are stored equally.

3.5.3 OSD Create and Remove

Before an object is written to it must be created then when deleted it is removed. The create is done by simply writing a blank file on the filesystem the OSD runs on top of, and for SSD caching is just duplicated to the SSD root directory. This creates two files for each create request however which could possibly slow down large numbers of object creations. Removal of objects is done very similarly with the file removal request being duplicated to the SSD as well.

3.5.4 OSD Read and Write

OSD reads and writes required the greatest amount of coding, requiring the 1-2 lines necessary to do a simple Linux read/write to be replaced by approximately 25. First a function was created with takes in the full location of the object on the main HDD and modifies it to the path of the object on the SSD. Calculations are then done which take in the request size and offset and compare it to the size of the SSD cache. These determine whether the SSD and/or HDD are going to require being accessed. They also correct for the file offsets and length of access which will possibly be used on both drives as well as offsets for the buffers used.

The calculation results in one or two readied file requests which can handle any combination of file access offsets and lengths, including any boundary conditions which will not initiate a request to a drive unless necessary. One or two asynchronous requests are then issued and the read/write will wait for each request to finish before returning which completes the read or write. When an error is encountered with either drive the appropriate error condition is set and returned from the OSD as if that error had taken place without caching enabled allowing the client to decide how

to proceed.

3.6 Results

Testing focused on two aspects primarily. The first was to prove the correct functionality of the caching implementation and to determine the maximum speeds possible. The second utilizes the exofs storage system to allow the OSD to be accessed from Linux as any other storage system which allows for standard benchmarks to be run such as IOZone.

This testing was complicated due to a combination of two factors. The first is due to the Linux storage system cache which acts to store recently used files in free memory. This causes very high read and write speeds compared to a typical disk and even in excess of a SSD when only writing small amounts of data. The problem was dealt with by either disabling the caching or mitigating its effects and will be explained in the section for each test. The second factor was limitations in the use of exofs which caused the decreased speeds seen in the benchmarks. These were caused by limitations in a gigabit Ethernet network and overheads associated with the storage system.

3.6.1 Setup

The main system contains a quad core Intel i7-2600 processor and 8GB of RAM with a 1TB main OS drive along with two extra drives dedicated for testing which are a 500GB 7200RPM and a 128GB Crucial M4 SSD and uses Windows 7 as the host operating system. In order to provide the greatest control and flexibility for testing virtual machines were used on top of this system using VMware 8 for primary development and testing. The VMs are stored on the main OS drive while each testing drive contained a 20GB pre-allocated virtual disk file which are only accessed by the virtual machine during testing. The OSD caching enabled target runs the Ubuntu

11.04 operating system and is the only VM which has access to the testing drives. When using multiple VMs the virtual network used is limited to 1Gbit/s and with overheads is slightly less, this however does allow very low network latencies. The initial OSD testing results however are run directly on a dedicated Linux installation of Ubuntu 11.04 on the above mentioned system without the VMs. This was done to decrease any overheads and to enable a comparison with results obtained from the VMs.

3.6.2 OSD Results

For testing as a pure OSD a more theoretical approach was taken. This was due to the need to create custom benchmarks designed for an OSD as well as to determine the maximum throughput limits for the HDD, SSD, and their combination in the caching system. To enable this both the OSD client and target were run on the dedicated linux installation. In order to further eliminate bottlenecks due to an Ethernet connection which would limit speeds to 100MB/s which is half the speed of the 2000MB/s and greater throughput that the SSD is capable of the single Linux installation contained both the client and server which utilized the loop-back network interface. To disable the Linux storage system cache the O_DIRECT flag was added to all the file open calls in the OSD target code which causes all reads and writes to proceed directly to disk. For these tests a 2MB SSD cache was used and the 1MB test was run twice to compare the variation between the same run. From file sizes of 64KB to 1024MB multiple runs are timed and averaged to ensure at least 1024MB have been written and read for consistency. This means that for the 64KB data points 16,384 writes are actually performed and the total transfer speed over this time is calculated. File sizes below 64KB use the 16,384 write limit as well due to the excessive time taken to perform them.

The first two figures 3.2 and 3.3 do not utilize the cache and show the maximum

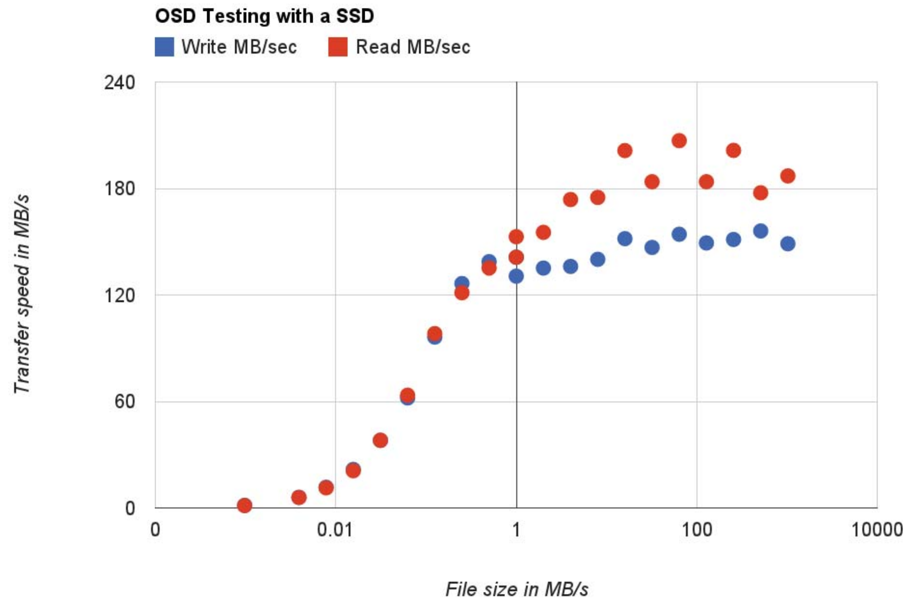

Figure 3.2: OSD Testing on SSD with 1KB-1024MB objects

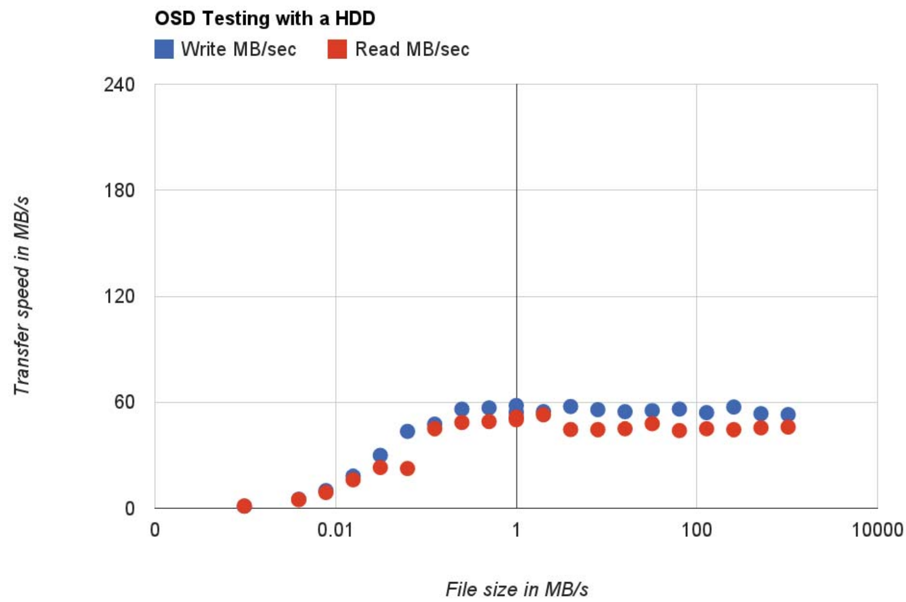

Figure 3.3: OSD Testing on HDD with 1KB-1024MB objects

(SSD) and minimum (HDD) speeds that the caching system is expected to achieve. Both figures have the same Y axis to make the advantage the SSD holds in speed evident which varies between 2.5x for writes and 3.5x for reads easily comparable to the HDD. Even with the extra overhead associated with the OSD protocol the lack of network limitations allows the maximum read and write speeds to hit 207MB/s and 156MB/s respectively. The HDD however is only able to reach up to 60MB/s transfer speeds. The most unexpected occurrence on the graphs are the extremely low throughputs that both drives have below 64KB. At this size and as the files decrease in size by half the throughput effectively is cut in half each time as well. From this data it is hard to determine whether transfers of this size and under being primarily limited by the overheads of this OSD implementation or the physical drives themselves.

Figure 3.4 shows the same test as the SSD and HDD only tests with the cache enabled sized at 2MB. Here the speed curve follows that of the SSD only test up to the cache size of 2MB. From that point on, the speed exponentially decreases until the dominating speed is that of the HDD at the transfer size of 16MB where only 1/8 of the transfer is enhanced at this point. Although large files do not show large speedups, files between 128KB and 4MB show a 2-3x improvement in throughput. Files below 128KB showed no difference in speeds between the SSD and HDD and this trend continues when using the cache due to OSD overheads.

To further ensure the cache was working correctly a test was run which tested read and write speeds across a 4MB object in steps of 512KB and the results are shown in Figure 3.5 and this also utilizes the same 2MB sized cache. The first four read/write pairs are serviced only by the SSD and therefore at the high speed it is capable of. Once the offset into the 4MB object moves out of the cache to only be serviced by the HDD the throughput drops significantly to approximately 1/3rd the speed of the SSD due to its slower transfer speeds. The slightly slower final 3 pairs is

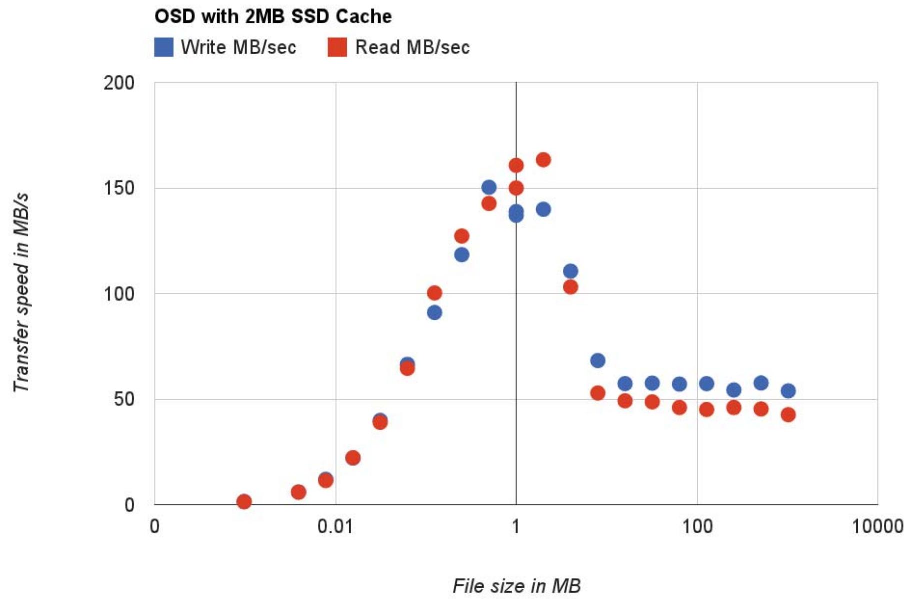

Figure 3.4: OSD Testing with 2MB cache with 1KB-1024MB objects

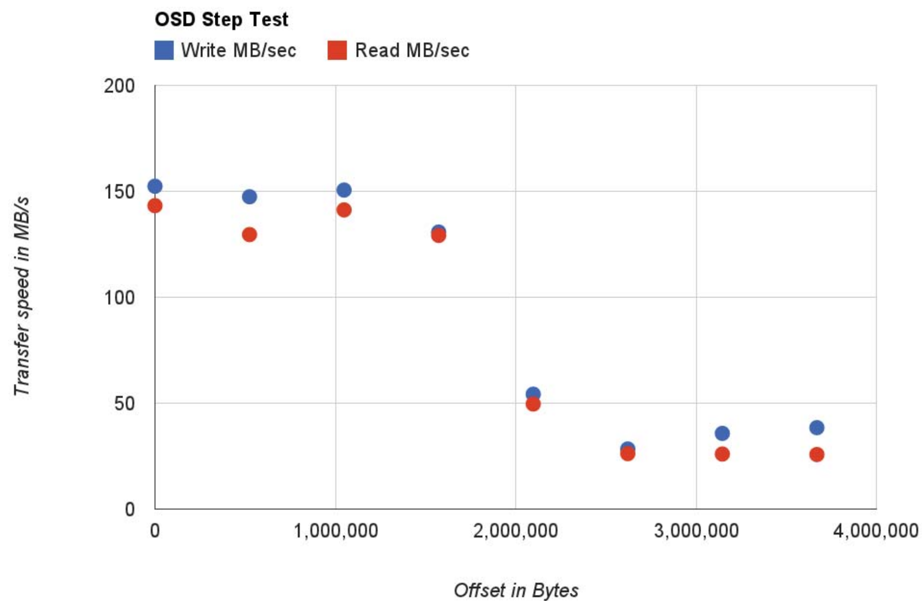

Figure 3.5: OSD Step Test with 2MB cache using 512KB blocks

most likely due to the write taking place at an offset from the beginning of the object instead of with an offset of 0 as the pair at offset of 2MB has.

3.6.3 Exofs Results

Although OSDs support many forward looking commands and abilities they require applications to be written with file I/O specifically designed for them. This limits support to use the networked based OSDs with legacy application and even mount them directly to Linux based operating systems. In order to provide this backwards compatibility Exofs (Extended Object File System) was written by an IBM researcher and is currently maintained and developed by Panasas who also maintains the open source OSD client and target code [Har]. Exofs has been included with the mainline Linux kernel since version 2.6.30 which makes it widely available and fairly simple to implement. Exofs connects to a remote OSD target, in this case one supporting SSD caching and presents itself to Linux as any other storage system enabling any application to take advantage of a network based OSD storage system.

Exofs testing was done in two ways intended to show more real world based results beginning with the standard benchmarking tool IOzone and concluding with the time taken to do a Linux kernel compile. The test setup was fairly different than the early OSD based tests due to the use of two VMs being used. Both run Linux with one acting as the target which uses the OSD caching code and saves objects to either the 20GB partition on the SSD or HDD which are only used by the OSD system (the VM files are stored on the main system HDD). The other VM is the target which runs Exofs, it connects to the target through the virtual network connection setup by VMWare and is where IOzone and the kernel compile are initiated.

In order to mitigate the effects of the Linux filesystem cache which affects both sides of exofs (accesses on the client to exofs, then from the OSD to the HDD or SSD) both VMs had their memory limited to 512MB which severely reduced the unused

memory which was free for caching. IOzone was run with 512MB files which due to the circular writes then reads, written information would be flushed from the cache before being read again. IOzone was also run with an option that remounts Exofs between every test which clears the cache on the client. The simple O_DIRECT flag cannot be used with Exofs since the client does not support it and due to the non 512 byte aligned accesses exofs sends to the target it cannot be used there either.

3.6.3.1 IOzone Results

The need to use large files in IOzone and the measures necessary to eliminate the default Linux cache severly reduce the benefits of using the SSD based cache due to the very small percentage (.5%) of data being accessed from the faster SSD. Without these throughputs for small files below 64MB typically register around 5GB/s. Five runs of IOzone were averaged to create the results shown in Figure 3.6. Even with a small percentage of the data being stored on the faster SSD the caching method provides a slight speedup of between 1.5% and4.5% which is always slightly faster than the HDD. This shows that any overheads due to the implementation are negligible. The overheads from using Exofs on a gigabit network are very obvious here. On large files a pure OSD was able to achieve near 200MB/s which is over twice that seen here. The speedups from using purely an SSD are also small for the contiguous reads and writes which likely means the storage subsystem (HDD or SSD) is not the limiting factor here. Random reads and writes at least show the expected speedup due to the SSD which is where it has the greatest advantage over a platter based drive typically.

3.6.3.2 Linux Kernel Compile Results

Compiling the Linux kernel version 3.4.1 was used here to try to show an example of an operation that could benefit by storing many smaller files on the faster SSD.

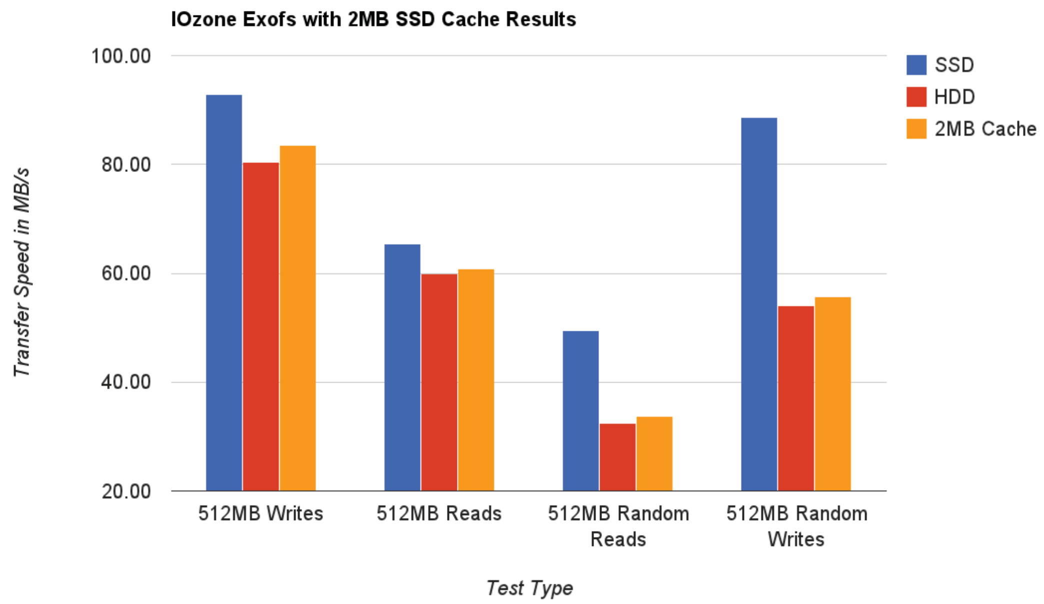

Figure 3.6: IOzone results using Exofs

The base kernel code is 461.4MB and contains almost 48,000 files and all of these are greater than the 2MB cache size. After compilation this grows to 5.4GB and just over 70,000 files, of these only 266 are greater than 2MB but they take up 1.5GB. A script was use which would time a compilation then clean the compiled files out before running another timed compilation which saved the data from five runs for each target type which is then averaged.

Exofs OSD Target Type	Compile Time (s)	CPU Time (%)
SSD	4915	81.6
HDD	5272	77.6
2MB Cache	5227	79.0

Table 3.1: Linux Compilation Times

Table 3.1 shows that the SSD has a slight advantage over the HDD and the cache having a very slight but beneficial effect. With the majority of the files being less than 2MB and saved onto the SSD the reason for the cache being as slow as the HDD is not completely known. The time was expected to be almost exactly that of the SSD. It, however, proved consistent over the five runs each of these is averaged from with the difference in times between runs typically below 60 seconds from the average. One can also see the CPU utilization percentages during the compile which line up with the times assuming any time not used for the compile was used waiting for or executing the filesystem requests.

There are two possible explanations for why the cache did not significantly help decrease the time, the first being related to file distribution after the compile during which includes five files over 50MB of which four are approximately 200MB. These large files combined could potentially be one of the limiting factors due to so little of them fitting in the SSD, leaving a large amount of data that was accessed solely on the HDD. Due to the high CPU times it would appear as through the compilation is CPU limited which combined with the long waits for the largest data files is a possible

reason why the times between the HDD and 2MB cache system are so similar.

3.7 Summary

This chapter established an architecture, its implementation, and the testing of a SSD (Solid State Disk) caching system built around the advantages of an Object Oriented File System. Current storage systems contain large numbers of files that are small in size and take up a proportionally small percentage of total drive space. These small files also provide a weakness to typical platter based disks which do not exist with SSDs but due to their high cost/GB cannot be used to replace the spinning disks yet. The caching architecture that is implemented here attempts to apply the low latency, high throughput of an SSD with the large size and low cost/GB of spinning disks using an object based storage system. The results show all objects at or below a specified size are successfully accessed with full SSD speeds while larger objects of which only a small proportion is accessed from the SSD have minimum throughputs consistent with only using the platter based drive. Testing using a translation layer which allows Linux programs to use the object oriented storage system without modification showed only minimum throughput increases compared to platter based drives due to limitations in testing the translation and network layers.

Chapter 4

Conclusions

This thesis focuses on the design and implementation of two system which take advantage of Object based Storage Devices (OSD) which is an iSCSI standard. The OSD's use of objects at the lowest level instead of blocks allow the targets access to information they normally would not have. This object level information at the target allows it to have additional functionality integrated which would typically be either very difficult or impossible. Two types of additional functionality that were explored here are Active Storage and SSD Caching.

This Active Storage framework and implementation allows for great flexibility in the remote execution of functions while still focusing on security laying the groundwork for a completed implementation to be added to the iSCSI OSD standard. The API combined with the security model developed here provide a good compromise between performance and security by utilizing sandboxing and efficient means of communicating outside of it. Multiple programming languages are supported by this implementation with others to follow as the system is fully developed. Results show performance improvements utilizing Active Storage and nearly linear scaling when using multiple Active Storage based OSDs and easily parallelizable applications and data.

The OSD based Solid State Disk (SSD) caching proposed and implemented here

also benefits from the object based nature of OSDs. It was designed to combine the low latency, high throughput but expensive SSDs with the high latency, lower throughput but cheaper platter based Hard Disk Drives (HDD). It focuses on speeding up accesses to small objects and hiding the access times due to the mechanical nature of HDDs for larger objects. This was done is a much simpler way than previous SSD caching systems and the results show that when using an OSD client the system performs exactly as expected, drastically speeding up small object accesses while leaving larger objects speeds no worse than on a system without caching.

4.1 Future Work

Both the Active Storage framework and SSD caching system were successfully implemented for testing purposes but there remain features to be completed and optimizations to explore before either can be put into practice. The completion of the Active Storage framework is the most pressing since upon its completion the code can be provided for others to build off of.

4.1.1 Active Storage

While Active Storage functions are currently capable of calling select OSD commands, most have not yet been implemented due a majority of the commands not being necessary for the basic functions used for testing. Asynchronous calls were used for testing but no system is in place currently to allow the mixing of them with synchronous calls or to inform the system of what mode the function being executed should utilize.

4.1.1.1 Complete OSD Services API

The API currently only has a minimal set of OSD commands implemented which were required for testing that was presented in this thesis. These include support

for reads, writes, accessing attributes, retrieving the size of an object, allocating and freeing memory on the shared heap. The commands most in need of being added include object create and remove, setting attributes. After that commands that interact with collections should be added to allow functions to create, list, and remove collections which should allow simplification of functions that interact with large numbers of objects. The design of the API, execution engines, and FastRPC allow new commands to easily be added with a minimal amount of additional code.

4.1.1.2 Real Asynchronous Calls

Asynchronous function calls used for testing were the result of a compile time option for the target which caused the iSCSI target to return immediately after starting the Active Storage function. For async calls to be easily serviceable there are a few features that could be implemented. The first is an additional argument added to `execute_function()` which would allow the client to initially set whether the function should execute async and be allowed to run in the background immediately or if the target should wait for the iSCSI timeout before returning an error if the function runs too long. If a possibility exists that the function will run too long, the application should be written so it outputs all its data to an object instead of relying on the return buffer which will no longer be able to return to the client due to iSCSI not allowing a target to initiate communication with a client.

4.1.2 SSD Caching

The SSD caching method proposed, implemented, and tested in this thesis works well when utilizing applications designed for object storage but when utilizing Exofs to allow compatibility with the Linux file I/O API limits were introduced due to the limitations of gigabit Ethernet. A possible way around this problem is to implement the caching scheme directly into the Linux kernel. The caching scheme was also not

compared to others due to the performance limitations of Exofs and a lack of available established SSD caching schemes to test.

4.1.2.1 Kernel Implementation

In order to bypass the gigabit Ethernet limitations the possibility exists to bypass using an OSD system and implement the caching system directly into the Linux kernel. Instead of intercepting reads and writes on the OSD target the Linux file I/O accesses would need to be redirected. This will allow the HDD and SDD to exist on the same system bypassing the speed limits and allowing any Linux application to use the caching system.

4.1.2.2 Comparison to Current methods

There were not any SSD comparison tests done in this thesis primarily due to the performance limitations seen with Exofs which would most likely have been slower than other caches. With a completed kernel implementation this would no longer be a problem and proper comparison tests should be possible. Candidates for comparison testing include those mentioned in Section 3.2.

Appendix A

Developer's Guide to Active Storage

A.1 Introduction

This appendix attempts to provide a resource for developers interested in writing active storage applications using the Active Storage framework discussed in this thesis. It includes a full list of OSD functions supported by the API and the libraries accessible from inside the sandbox. It also contains full code examples in both C and Java of an active storage function.

A.2 Current state of the API

The API for C programs is named c-osd.h and is one of the includes in the example C program. These functions each have several arguments which are explained in the API file so will be omitted here. It currently contains the following functions:

- `start()` – start function required for all AS functions, instead of main
- `obj_getattr_val()` – returns an objects specified attribute
- `obj_get_size()` – returns size of specified object
- `obj_read()` – reads specified amount at an offset in an object
- `obj_write()` – writes specified amount at an offset in an object

- `osd_allocate()` – allocated memory of the specified size on the shared heap
- `osd_free()` – frees specified pointer to shared heap

These are the only currently implemented functions and were all that was necessary to complete all the Active Storage testing in this thesis. Adding additional functions to the API is however fairly easy and plans are in place to support all OSD functions through the API at a later date beginning with the most commonly used functions.

A.2.1 Supported Libraries

The current list of libraries that are copied to the chroot jail for C are:

- libsqlite3.so
- libm.so
- libc.so
- libpthread.so
- ld-linux.so
- libnss_files.so

A.3 Full Function C Code Example

The following is a full working Active Storage function which copies one object to another in 4MB chunks. The basic includes are used along with the Active Storage specific `osd-defs` and `c-osd`. The first thing to notice is the `copy_params` struct, it contains the expected format of the indata buffer, in this case one source and one destination partition ID, object ID pair. The start function is essentially the main in this program and will be the first function called, if no start with the correct format exists the program will not be loaded.

The bytes_read pointer will be set using the obj_get_size() function to the size in bytes of the source object. The copy_buff pointer will shortly be allocated to either the smallest between the source object and the 4MB maximum using the osd_allocate() function which has to be used for all buffers used with API functions. After that a loop begins that just copies from the source to destination objects until finished.

Once the loop is finished the void * outdata variable from the start() function is cast so the size of the source object in bytes can be copied into it. The used_outlen variable is then set to 8 bytes since outdata is 64bits. Finally the API specific osd_free is used to clear both pieces of memory that had been allocated on the shared heap.

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdint.h>
//object definitions
#include <../osd-util/osd-defs.h>
//Active Storage C API
#include "../../osd-target/activeosd-engines/c-osd.h"
//enum just makes math easier to look at
enum {
        K = 1024,
        M = 1024 * K,
        G = 1024 * M,
};

//structure for input aes parameters
struct copy_params {
        uint64_t source_pid;
        uint64_t source_oid;
        uint64_t dest_pid;
```

```
        uint64_t dest_oid;
};
//start is the main function called from the engine
int start(void *indata,                  //input buffer
          size_t inlen,                  //input buffer size in bytes
          void *outdata,                 //output buffer
          size_t outlen,                 //output buffer max size in bytes
          size_t *used_outlen,           //bytes of output buffer used
          uint8_t *sense)                //used to return errors from
                                         //OSD functions to client
{
        uint64_t olen;
        uint64_t *bytes_read = osd_allocate(sizeof(uint64_t));
        unsigned char *copy_buff;
        struct copy_params *copy_params = indata;

        int ret;
        //if input struct is wrong size return error
        if (inlen != sizeof(struct copy_params)){
                osd_free(bytes_read);
                return -1;
                }
        //get the size of the input object
        ret = obj_get_size(copy_params->source_pid,
                copy_params->source_oid, &olen, sense);
        //allocate size of input up to 4MB
        copy_buff = osd_allocate(olen<4*M?olen:4*M);

        //the for loop uses up to 4M increments
        uint64_t len_rem = olen;         //init length remaining
                                         //to copy to size of input
        uint64_t curr_copy_offset = 0;   //start read offset at 0
```

```
int copy_len = 0;          //init write length to 0
unsigned i;                //used to keep track of loop number
for(i=0; len_rem > 0; i++)
{
        //set amount we need to copy this round
                //up to 4MB at a time
        copy_len = len_rem<4*M?len_rem:4*M;
        //set amount offset into object
        curr_copy_offset = 4*M*i;

        //time to read a 4M chunk from the object
        ret = obj_read(copy_params->source_pid,
                copy_params->source_oid, copy_buff,
                copy_len, curr_copy_offset,
                bytes_read, sense);
        //use to write read buffer to the destination
        ret = obj_write(copy_params->dest_pid,
                copy_params->dest_oid, copy_buff,
                copy_len, curr_copy_offset,
                NULL, sense);
        //determine how much is left to copy
        len_rem = len_rem>4*M?len_rem-4*M:0;
}

//return size of input object
*(uint64_t *)outdata = olen;
//set bytes used by outdata buffer
*used_outlen = 8;
//free memory
osd_free(copy_buff);
osd_free(bytes_read);
return 0;
```

```
}
```

A.4 Full Function Java Code Example

The Java version of the object copy code is fairly similar to the C code with the exception of having to handle the difference in byte order between Java and C. This is due to using the JNI Java to C conversion which is necessary to call the C functions that run the OSD. After initialization and the class definition the first swap function us used to convert the byte order of the input to what Java uses and is very simple to use.

In Java the name of the main function is `run()` and it carries a very similar format to the `start()` used in C. Next some input variables are declared then a try catch block is used to read the input data through the `DataInputStream`. This replaces the struct used in C and contains the same data. Next the `olenp` variable is passed to `OSDjava.obj_get_size()` which is the first API call and will set `olenp` to the size of the specified object.

The buffer `buff` is set to the smallest size between the source object and the max block size of 4MB, notice here that the allocate used here does not have a specific OSD version, the Java Engine code automatically handles converting the buffer to use the shared heap when necessary. The for loop is exactly like in C, copying up to 4MB at a time until the entire object is copied to the destination. The `obj_read()` and `obj_write()` even have the same order of arguments.

At the end the size that was copied is calculated in MB then has the order swapped so it can be sent out with the `outdata` buffer and the output size is set to 4 bytes, the same as the 32bit integer that is being returned.

```
import java.io.*;


public class CopyApp
```

```
{
    public CopyApp()
    {}
        //swaps the byte order for up to a 64 byte long input
    private long swap(long l)
    {
        long r = ((((l >>  0) & 0xff) << 56) |
                   (((l >>  8) & 0xff) << 48) |
                   (((l >>  16) & 0xff) << 40) |
                   (((l >>  24) & 0xff) << 32) |
                   (((l >>  32) & 0xff) << 24) |
                   (((l >>  40) & 0xff) << 16) |
                   (((l >>  48) & 0xff) << 8) |
                   (((l >>  56) & 0xff) << 0);
                return r;
    }
    //In java instead of start we have run
    public int run(      ByteArrayInputStream in, //input buffer
                byte[] outdata, //output buffer
                int outlen,      //max output buffer length
                long[] used_outlen,      //used output buffer length
                byte[] sense)   //used for returning errors to client
    {
        //Init variables
        long pid, oid, out_pid, out_oid, olen;
        try {
                //setup input buffer
            DataInputStream dis = new DataInputStream(in);
            pid = swap(dis.readLong()); //read and swap pid order
            oid = swap(dis.readLong()); //read and swap oid order
            out_pid = swap(dis.readLong());//read and swap output pid
            out_oid = swap(dis.readLong());//read and swap output oid
```

```
}
catch (Exception e) {   //if any reads fail exit
    System.err.println(e);
    return -1;
}
//olenp will be set to size of input object
long[] olenp = new long[1];
//Use OSDjava api to get the size of input object
int ret = OSDjava.obj_get_size(pid, oid, olenp, sense);
if (ret!=0) return ret; //if fail return error
//set max copy block size
long block_size = 4*1024*1024;

//use normal long instead of pointer
olen = olenp[0]<block_size?olenp[0]:block_size;
//init copy buffer up to size olen
byte[] buff = new byte[new Long(olen).intValue()];

//init variables for copy loop
long[] sizep = new long[1];
long len_rem = olenp[0];
long curr_read_offset = 0;
long read_len = 0;
int i = 0;        //used to keep track of loop number
long curr_write_offset = 0;

for(i=0; len_rem > 0; i++)
{
        //set amount we need to read this round
        read_len = len_rem<block_size?len_rem:block_size;
        curr_read_offset = block_size*i;
        //use API to call a read
```

```
                ret = OSDjava.obj_read(pid, oid,
                        buff, read_len,
                        curr_read_offset,
                        sizep, sense);
                if (ret != 0)
                        return -3;
                //write buffer to destination object
                ret = OSDjava.obj_write(out_pid, out_oid,
                        buff, read_len,
                        curr_read_offset,
                        sizep, sense);
                if (ret!=0)
                        return -4;
                //calculate length left to copy
                len_rem = len_rem>block_size?len_rem-block_size:0;
        }
        //convert olenp in bytes to int in MB
        int input_size_mb = (int)olenp[0]/(1024*1024);
        //reverse byte order to output buffer
        outdata[0] = (byte)(input_size_mb & 0xff);
        outdata[1] = (byte)((input_size_mb >> 8) & 0xff);
        outdata[2] = (byte)((input_size_mb >> 16) & 0xff);
        outdata[3] = (byte)((input_size_mb >> 24) & 0xff);
        //set used buffer size
        used_outlen[0] = 4;
    return 0;
    }
}
```

A.5 Debugging

When debugging C functions gdb can be used, but since the functions are being executed asynchronously it can be hard to find a breakpoint that will get you into the AS function. It is however possible to use a `sleep()` for several seconds in order to discover its process ID and attach to it.

Debug messages can also be returned to the command line from within both languages. In C using a printf outputting to stderr in this form: `fprintf(stderr, "Debug Message")` allows the message to make it to the terminal where the target was initiated, messages to stdout are lost. In Java a similar message is sent also to an error output and will appear on the terminal window the target was started from: `System.err.println("Debug Message")`.

Bibliography

[ABDL07] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. **Trans. Storage**, 3(3), October 2007.

[ADD+08] Nawab Ali, Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, and P. Sadayappan. An OSD-based approach to managing directory operations in parallel file systems. In **Proceedings of the IEEE International Conference on Cluster Computing**, pages 175–184, 2008.

[ANS02] ANSI. **Information Technology - SCSI Object Based Storage Device Commands (OSD)**, March 2002.

[ANS08] ANSI. **Information Technology - SCSI Object Based Storage Device Commands -2 (OSD-2)**, January 2008.

[AUS98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. 1998.

[BFSS00] Michael Beynon, Renato Ferreira, Alan Sussman, and Joel Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. 2000.

[BZ01] Peter J. Braam and Rumi Zahir. Lustre technical project summary. Technical report, Cluster File Systems, Inc., Mountain View, CA, July 2001.

[CFSS99] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. 1999.

[cle] Why object storage. http://www.cleversafe.com/overview/why-object-storage.

[CLRT00] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In **Proceedings of the Annual Linux Showcase and Conference**, pages 317–327, October 2000.

[DDAW07] Ananth Devulapalli, Dennis Dalessandro, Nawab Ali, and Pete Wyckoff. Attribute storage design for object-based storage devices. pages 263–268, 2007.

[DDW+07] Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, Nawab Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices. In **Proceedings of Supercomputing**, pages 263–268, 2007.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. **Communications of the ACM**, 51(1):107–113, January 2008.

[DHH+06] David Du, Dingshan He, Changjin Hong, Jaehoon Jeong, Vishal Kher, and Yongdae Kim. Experiences in building an object-based storage system based on the OSD T-10 standard. Technical Report DTC 2006/13, Digital Technology Center, University of Minnesota, Minneapolis, MN, 2006.

[DMXW] Anath Devulpalli, Iyyappa T. Murugandi, Da Xu, and Pete Wyckoff. Design of an intelligent object-based storage device. Technical report, Ohio Supercomputing Center.

[emc] EMC Centera. http://www.emc.com/products/detail/hardware/centera.htm.

[FNN+05] Michael Factor, David Nagle, Dalit Naor, Erik Riedel, and Julian Satran. The OSD security protocol. In **Proceedings of the Third IEEE International Security in Storage Workshop**, SISW '05, pages 29–39, Washington, DC, USA, 2005. IEEE Computer Society.

[Gas12] Geoff Gasior. The post-flood decline of hard drive prices is slowing. http://techreport.com/articles.x/23185, 2012.

[GNA+98] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Harding, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. October 1998.

[GVM00] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. **Communications of the ACM**, 43(11):37–45, November 2000.

[Har] Boaz Harrosh. http://www.open-osd.org.

[Hea06] Michael Hearn. Security-oriented fast local RPC. M.S. thesis, Dept. of Computer Science, University of Durham, 2006.

[Key08] Gary Key. 64gb on the desktop: Samsung and ocz go mainstream. http://www.anandtech.com/show/2527, May 2008.

[KPH98a] K. Keeton, D. A. Patterson, and J. M. Hellerstein. The intelligent disk (IDISK): A revolutionary approach to database computing infrastructure. Technical report, University of California at Berkeley, May 1998. White Paper.

[KPH98b] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). **SIGMOD Record**, 27(3):42–52, September 1998.

[KRM08] Taeho Kgil, David Roberts, and Trevor Mudge. Improving nand flash based disk caches. In **Proceedings of the 35th Annual International Symposium on Computer Architecture**, ISCA '08, pages 327–338, Washington, DC, USA, 2008. IEEE Computer Society.

[MGR03] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based storage. **IEEE Communications Magazine**, 41(8), August 2003.

[NSM04] David Nagle, Denis Serenyi, and Abbie Matthews. The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage. In **Proceedings of Supercomputing**, pages 53–62, November 2004.

[pan] Object raid, performance, reliability and availability. http://www.panasas.com/products/panfs/object-raid.

[PNF07] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In **Proceedings of Supercomputing**, November 2007.

[PT10] Timothy Pritchett and Mithuna Thottethodi. Sievestore: a highly-selective, ensemble-level disk cache for cost-performance. In **Proceedings of the 37th annual international symposium on Computer architecture**, ISCA '10, pages 163–174, New York, NY, USA, 2010. ACM.

[QF06] Lingjun Qin and Dan Feng. Active storage framework for object-based storage device. In **Proceedings of International Conference on Advanced Information Networking and Applications**, apr 2006.

[RFGN00] Erik Riedel, Christos Faloutsos, Gregory R. Ganger, and David F. Nagle. Data mining on an OLTP system (nearly) for free. May 2000.

[RFGN01] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. 34(6):68–74, June 2001.

[RGF98] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. August 1998.

[Rie99] Erik Riedel. **Active Disks - Remote Execution for Network-Attached Storage**. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 1999. Tech. Report no. CMU-CS-99-177.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. pages 231–244, January 2002.

[Shi] Anand Lal Shimpi. http://www.anandtech.com/show/4256/the-ocz-vertex-3-review-120gb/6.

[SPBW10] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In **Proceedings of the 8th USENIX conference on File and storage technologies**, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[Sri95] Raj Srinivasan. RPC: Remote procedure call protocol specification version 2. Technical Report Network Working Group RFC 1831, Sun Microsystems, August 1995.

[Vat12] Kristian Vatto. Plextor m5s 256gb review. http://www.anandtech.com/show/6090/plextor-m5s-256gb-review, 2012.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlons Maltzahn. Ceph: A scalable, high-performance distributed file system. pages 307–320, November 2006.

[WCV02] Rajiv Wickremesinghe, Jeffrey S. Chase, and Jeffrey S. Vitter. Distributed computing with load-managed active storage. July 2002.

[Wil92] John Wilkes. DataMesh research project, phase 1. In **Proceedings of USENIX File Systems Workshop**, pages 63–69, May 1992.

[XMRF+11] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Darrell D. E. Long, Yangwook Kang, Zhongying Niu, and Zhipeng Tan. Design and evaluation of oasis: An active storage framework based on t10 osd standard. In **msst**, 2011.

[YNS+08] Jin Hyuk Yoon, Eyee Hyun Nam, Yoon Jae Seong, Hongseok Kim, Bryan Kim, Sang Lyul Min, and Yookun Cho. Chameleon: A high performance flash/fram hybrid solid state disk architecture. **IEEE Comput. Archit. Lett.**, 7(1):17–20, January 2008.

[ZF08] Yu Zhang and Dan Feng. An active storage system for high performance computing. In **Proceedings of International Conference on Advanced Information Networking and Applications**, pages 644–651, 2008.