An Active Storage Framework for Object Storage Devices

Michael T. Runde Wesley G. Stevens Paul A. Wortman John A. Chandy
University of Connecticut
Storrs, CT
{*mtr03003,wgs10001,paul.wortman,chandy*}*@engr.uconn.edu*

***Abstract*—In this paper, we present the design and implementation of an active storage framework for object storage devices. The framework is based on the use of virtual machines/execution engines to execute function code downloaded from client applications. We investigate the issues involved in supporting multiple execution engines. Allowing user-downloadable code fragments introduces potential safety and security considerations, and we study the effect of these considerations on these engines. In particular, we look at various remote procedure execution mechanisms and the efficiency and safety of these mechanisms. Finally, we present performance results of the active storage framework on a variety of applications.**

I. Introduction

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 disk controller level of the storage stack. Another factor to consider is the current trends in storage density, mechanics, and electronics, which are eliminating the bottleneck encountered while moving data off the media, and putting pressure on interconnects and host processors to move data more efficiently. This computational capability at the disk has led to the development of object-based storage whereby some of the filesystem functionality is moved to the disk [4], [14], [15], [19]. 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 [7], [8], [20], [27], [29] based on the 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.

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. For example, object storage has the capability to specify attributes that can provide some metadata functionality. There has been some recent work to exploit object storage in its application to parallel file systems, specifically PVFS [3], [11], [12]. 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 [2], [18], [25], [30], [31]. 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. The advantages of moving computation to the disk has been demonstrated in early work in transaction processing, data mining and multimedia [24], [26], [23]. 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 [2]. This work was later expanded to large scale data processing using the concept of data filters in the DataCutter/Active Data Repository framework [6], [9]. Similarly, 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 [17]. While active storage and object storage arose from the same NASD 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 [5] 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 in this paper. From the application layer, the programming model is similar to an asynchronous RPC. This model provides the most flexibility to the application and user.

The framework 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, we are interested in the mechanisms required to provide secure and safe execution of active storage functions.

In the remainder of the paper, we describe the implementation and performance results of the framework for a variety of applications.

II. 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 [1]. In addition, Panasas has also taken over development of the Ohio Supercomputer Center *osc* OSD target [12] 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 [10]. 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.

Byte \ Bit	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 1. EXECUTE_FUNCTION Format

A. 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.

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 1. The OSD command uses continuation descriptors to extend the send buffers to pass parameters 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.

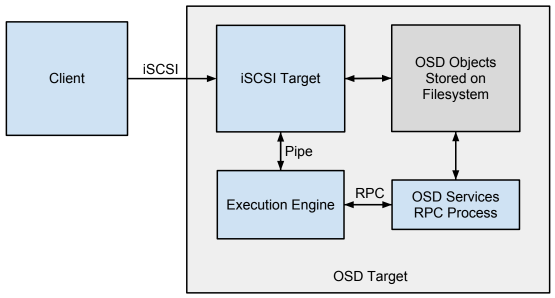

Figure 2. High Level View of Active Storage OSD

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 blocks of an active storage OSD look like the diagram in Figure 2. 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.

2) Expressing parallelism: Execution of an OSD active object function has two major complications with respect to high performance parallel computation - namely simultaneous execution and long running functions.

Simultaneous execution allows for functions to be executed on multiple OSDs simultaneously. Implementating simultaneous execution can be 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 to return immediately. In terms of time, this would only be a short command 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 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.

OSD 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 for the client to do asynchronous RPC - i.e. sending a request and not waiting for the return. 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 OSD execute method would have returned due to the timeout, 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. When the function returns due to the timeout, it could include information such as expected times to completion which could be updated at various times. It could also possibly be used for sending control signals to the function as well - for example, creating a queue of objects for the function to execute on. However, using the object to indicate completion requires polling, since the client and a running function can not 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.

```
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 3. Example Active Storage Function Code

3) Code Example: Figure 3 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.

B. Execution Engines

All the engines have at least some implementation in C which is the 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.

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) 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 is 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.

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.

C. Secure Implementation

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

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) 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 which are usually fairly simple such as pipes or shared memory areas RPCs essentially allow functions to be called to 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 [28]. 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 [16] 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 which 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: This limitation 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 cant 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.

III. 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.

A. 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

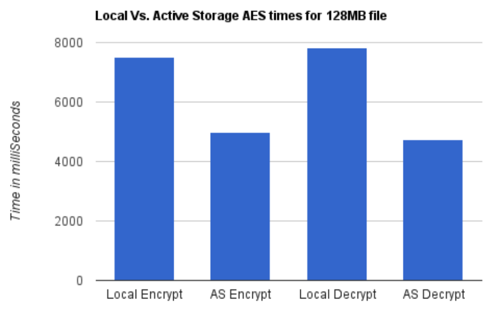

Figure 4. Local vs. Active Storage

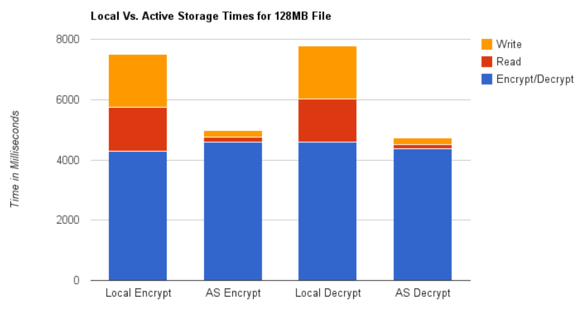

Figure 5. Local vs. Active Storage Breakdown

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. They nodes are all connected through a dedicated gigabit network switch.

B. *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 4 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 5 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.

C. *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 heavy 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 6 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 relative standard deviation of the speedup is only 3.51% which is consistent with the near perfect speedup seen.

The second function is a version of grep. This function simply takes from the client the string that is being searched for along with object ids for its input and output. Instead of text files with line breaks, we use randomly generated files for large file sizes. As a result, the grep function reads

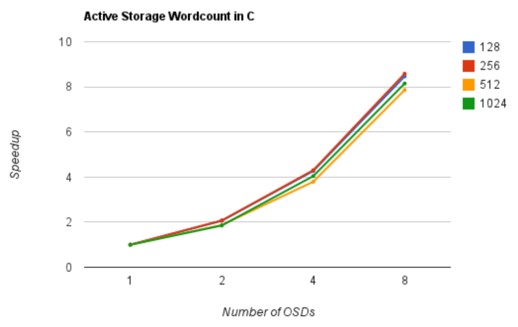

Figure 6. Active Storage Wordcount Results

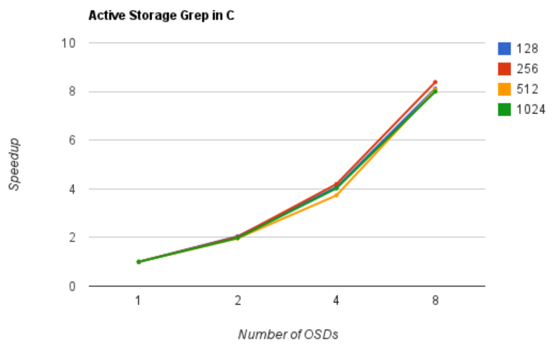

Figure 7. Active Storage Grep Results

through the input object 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 7 shows a similarly concentrated grouping, with all file sizes scaling fairly close to optimally. The relative standard deviation, again, is very small and 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 8 are only from encryption though decryption shows near identical results as seen in Figure 5.

This more write and CPU-intensive program shows slightly superlinear with sizes of 1024MB. This is most likely due to the total read and write size of 2048MB which will not fit into the disk cache of a single OSD which has 2048MB RAM. With smaller sizes, the reads and writes are cacheable. Also, tests with 2 or more OSDs used at most 1024MB reads and writes which will fit in the cache. The relative standard deviation here was slightly higher than the other two at 5.15%.

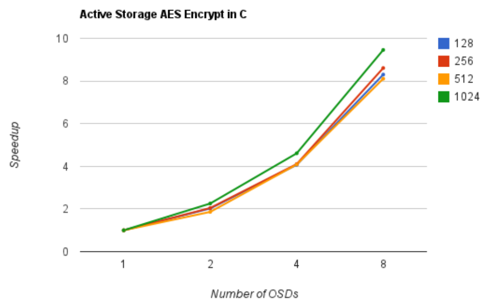

Figure 8. Active Storage AES Encrypt Results

D. 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 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 hard-coded 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 415us with a maximum of 552us.

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.

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

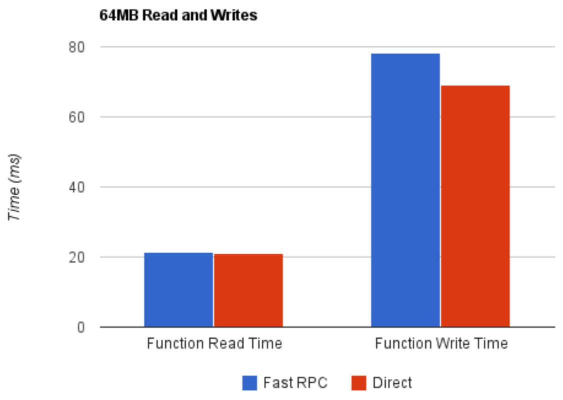

Figure 9. Active Storage FastRPC vs Direct

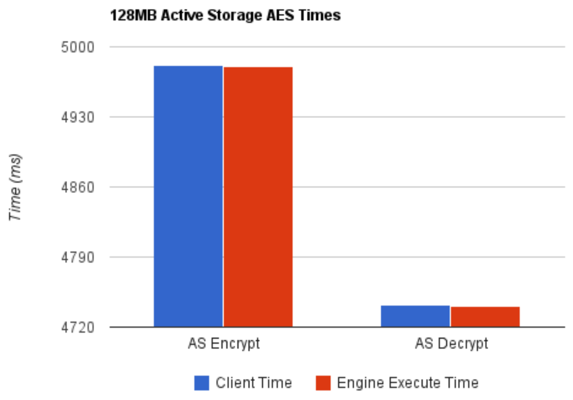

Figure 10. Active Storage Client vs Engine Times

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 9 shows 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 $\tilde{8}$00ms the difference of 14ms should be due to run- to-run variations that are typically greater than this difference.

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 9 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 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.

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 10 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.

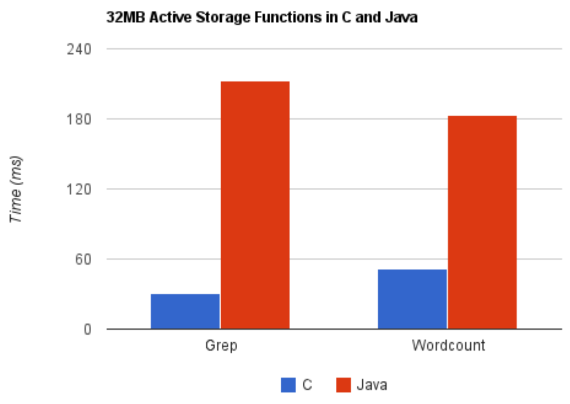

Figure 11. Active Storage C vs Java Times

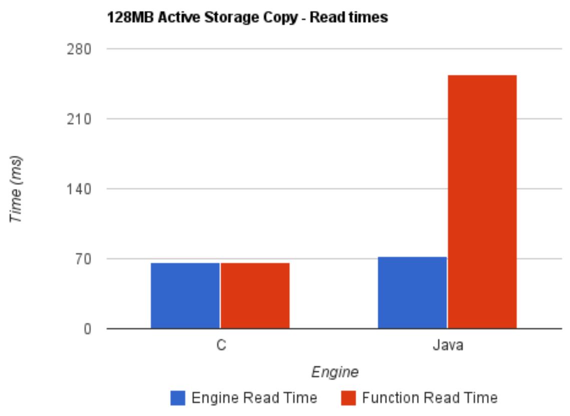

Figure 12. Active Storage C vs Java Read Times

E. 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 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 11 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 a 32MB file size the C version is much faster for both functions with speedups 3-4 times the Java version. This difference is due to slower Java execution as well as Java to C interface inefficiencies. 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 4MB reads and 32 4MB writes. Figure 12 shows read times while Figure 13 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 reads 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 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. We are investigating methods to allow Java to directly call the fastRPC routines without making the Java to C transition.

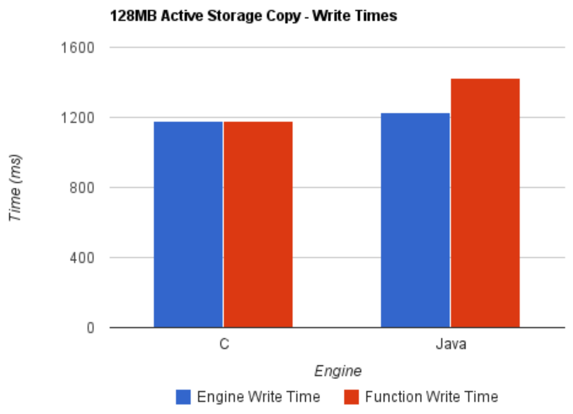

Figure 13. Active Storage C vs Java Write Times

IV. 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 [21]. 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 [2]. 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 [32]. Similar to the filter/stream model, the function objects are invoked automatically whenever the associated object is read or written. Security is provided because of the limited streams/filter API and functions are assumed to be vendor-only. Earlier versions of this work allowed for policy specifications as well as integration with reconfigurable hardware [22], [33]. 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. Our model provides a much richer API allowing functions to be much more full-featured. The closest to our work is iOSD from the Ohio Supercomputer Center [13]. They also use an RPC model and allow functions full access to objects. However, they do not allow different virtual machine execution engines.

V. Conclusions

We have presented a design of an active storage system using OSDs. The execution model is based on the use of remote procedure calls whereby functions are downloaded to an OSD for execution. The design allows for multiple execution engines with high sandboxed security and relatively low overhead. Performance results show scalability across multiple OSDs.

Acknowledgment

This work was supported in part by a National Science Foundation High End Computing University Research Activity grant (award number CCF-0937879). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

References

[1] http://www.open-osd.org.

[2] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[3] N. Ali, A. Devulapalli, D. Dalessandro, P. 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*, 2008, pp. 175–184.

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

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

[6] M. Beynon, R. Ferreira, A. Sussman, and J. Saltz, "DataCutter: Middleware for filtering very large scientific datasets on archival storage systems," in *Proceedings of the IEEE/NASA Goddard Symposium on Mass Storage Systems and Technologies*, 2000.

[7] P. J. Braam and R. Zahir, "Lustre technical project summary," Cluster File Systems, Inc., Mountain View, CA, Technical Report, Jul. 2001.

[8] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proceedings of the Annual Linux Showcase and Conference*, Oct. 2000, pp. 317–327.

[9] C. Chang, R. Ferreira, A. Sussman, and J. Saltz, "Infrastructure for building parallel database systems for multidimensional data," in *Proceedings of the International Parallel Processing Symposium*, 1999.

[10] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communnications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[11] A. Devulapalli, D. Dalessandro, N. Ali, and P. Wyckoff, "Attribute storage design for object-based storage devices," in *Proceedings of the IEEE/NASA Goddard Symposium on Mass Storage Systems and Technologies*, 2007, pp. 263–268.

[12] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, "Integrating parallel file systems with object-based storage devices," in *Proceedings of Supercomputing*, 2007, pp. 263–268.

[13] A. Devulapalli, I. T. Murugandi, D. Xu, and P. Wyckoff, "Design of an intelligent object-based storage device," Ohio Supercomputer Center, Tech. Rep. [Online]. Available: http://www.osc.edu/research/network_file/projects/object/papers/istor-tr.pdf

[14] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Harding, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[15] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Communications of the ACM*, vol. 43, no. 11, pp. 37–45, Nov. 2000.

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

[17] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "The intelligent disk (IDISK): A revolutionary approach to database computing infrastructure," University of California at Berkeley, Tech. Rep., May 1998, white Paper.

[18] ——, "A case for intelligent disks (IDISKs)," *SIGMOD Record*, vol. 27, no. 3, pp. 42–52, Sep. 1998.

[19] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, no. 8, Aug. 2003.

[20] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage," in *Proceedings of Supercomputing*, Nov. 2004, pp. 53–62.

[21] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *Proceedings of Supercomputing*, Nov. 2007.

[22] L. Qin and D. Feng, "Active storage framework for object-based storage device," in *Proceedings of International Conference on Advanced Information Networking and Applications*, apr 2006.

[23] E. Riedel, "Active disks - remote execution for network-attached storage," Ph.D. dissertation, Electrical and Computer Engineering, Carnegie Mellon University, 1999, tech. Report no. CMU-CS-99-177.

[24] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle, "Data mining on an OLTP system (nearly) for free," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 2000.

[25] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *IEEE Computer*, vol. 34, no. 6, pp. 68–74, Jun. 2001.

[26] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Aug. 1998.

[27] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of USENIX Conference on File and Storage Technologies*, Jan. 2002, pp. 231–244.

[28] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2," Sun Microsystems, Tech. Rep. Network Working Group RFC 1831, Aug. 1995.

[29] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, Nov. 2006, pp. 307–320.

[30] R. Wickremesinghe, J. S. Chase, and J. S. Vitter, "Distributed computing with load-managed active storage," in *Proceedings of the IEEE Symposium on High Performance Distributed Computing*, Jul. 2002.

[31] J. Wilkes, "DataMesh research project, phase 1," in *Proceedings of USENIX File Systems Workshop*, May 1992, pp. 63–69.

[32] Y. Xie, K.-K. Muniswamy-Reddy, D. Feng, D. D. E. Long, Y. Kang, Z. Niu, and Z. Tan, "Design and evaluation of oasis: An active storage framework based on t10 osd standard," in *msst*, 2011.

[33] Y. Zhang and D. Feng, "An active storage system for high performance computing," in *Proceedings of International Conference on Advanced Information Networking and Applications*, 2008, pp. 644–651.