

Design of an Intelligent Object-based Storage device

Ananth Devulapalli*, Iyyappa T. Murugandi†, Da Xu†, Pete Wyckoff‡

* Ohio Supercomputer Center
ananth@osc.edu

† The Ohio State University
{murugandi.1, xu.188}@buckeyemail.ohio-state.edu

‡ Network Appliance
pw@padd.com

Abstract—Intelligent storage systems were an active area of research in later half of last decade. The idea was to improve the throughput of data intensive applications from database and image processing domains by offloading computation onto the active storage elements and hopefully reducing unnecessary data traffic between data sources and compute nodes.

With the advent of Object-based Storage Devices (OSD) we feel that the time is appropriate to have a relook at notion of intelligent storage. OSDs provide a higher level object abstraction that makes it easier to describe the computation to the devices. OSDs provide the right platform to ask several questions in the context of intelligent storage devices.

This is a working document capturing our experience in designing an intelligent OSD (iOSD) and using it to build scalable systems. In this preliminary document we describe the design goals for an intelligent OSD. We explain the implementation details of iOSD and challenges faced in integrating the offload engine into OSD ecosystem. We describe how offloaded computation is expressed, executed and managed in an storage system made up of iOSDs. Finally we quantify the performance and overhead of iOSD.

I. INTRODUCTION

The idea of intelligent storage was quiet popular during later half of last decade. Projects like active storage [1], active disks [2] and intelligent disks [3] made a convincing case of adding more processing power closer to disk for faster computations of data intensive workloads. These workloads spend a significant portion of their execution time in fetching the data from the remote storage to local machine for computations. And in many cases the computation per byte of the data fetched is not that much. A key observation made by active storage [1] is that the aggregate bandwidth and aggregate processing power of the disks is far more than the total bisection bandwidth of the interconnect. Offloading computations for such data intensive workloads would take advantage of the aggregate disk bandwidth and idle processing power of the disk drives. Further the disk drives themselves are getting equipped with powerful ASICs as VLSI technology is improving [4].

One important property of the disks is that their storage capacity is increasing at a much faster pace than the disk bandwidth. Similarly the VLSI technology through it has hit physical power and frequency limits, is still making integration of more functionality for the same price point as in past. Thus we have a happy coincidence of two powerful technological

trends: increasing storage capacity and increasing system integration capabilities, when we coupling the them together in a storage system.

Simultaneously, with the with these developments in hardware, we are also seeing maturity in disk interface technology. Today most disks export logical block address (LBA) interface to their users. However, management of filesystems storing petabytes of data on a 512-byte blocks becomes complex. The size of the metadata itself for these filesystems is in the order of gigabytes. In order to overcome such a bottleneck, today several distributed filesystems like PVFS [5], PanFS [6], Google filesystem [7], and Lustre [8] build a global filesystem on top of local filesystems like ext3 [9]. The local filesystem is responsible for managing the data on a single or a small set of disks. This raises a question: can we migrate some of the complexity of these local filesystems to the hard disks themselves, so that we can build scalable filesystems. The reason for this is to run a local filesystem we need server grade resources. However with some of the complexity of data management shifted to the disk, the whole package is compact, power-efficient and makes an ideal building block.

A start has been made in that direction with the development of Object-based storage disks (OSD). Unlike traditional block-based devices, OSDs export an object interface. Accessing data in these disks is done by specifying an unique object identifier, a logical offset into the object and length of the access. This access style is quiet similar to accessing files in a local filesystem. However OSDs offer far more richer interaction. Each object can have large number of attributes, some of which are user-settable. This enables rich semantics like transparent backup, compression etc. can now be communicated to the disk and disk can take it on behalf of the user. OSD specification is currently developed by T10 standards committee of SNIA [10].

Given this background, OSDs are an ideal platform for the realization of intelligent storage on which computation can be offloaded. OSDs are already intelligent given the amount of work they do which is currently done be local filesystems like on-disk data management. However, the ability of executing user specified computation is lacking the current OSD specification [10]. Expression of computation is much more compact on a OSD since the block metadata need not be communicated to the disk. Further, the amount of processing

power required to implement OSD interface makes it straightforward to extend it to execute user specified computation. That is why intelligent storage is one of the key agenda items of OSD version 3.

In this paper we describe our design of an intelligent OSD (iOSD). In Section II we go over the current state of the art in intelligent storage. Then we give a background of OSD in Section III. We describe the design goals for an intelligent OSD in Section IV. In Section V we explain our implementation of intelligent OSD, and software infrastructure we built. We do quantitative evaluation of the performance of our implementation in Section VI. Finally we layout our future goals in the concluding section.

II. RELATED WORK

The idea of intelligent storage was resurrected during later half of 1990s with projects like Active Storage [1], Active Disks [2] and iDisks [3]. The target applications for these intelligent disk architectures were data intensive applications like image processing and data mining applications in decision support systems for ever growing databases of warehouses and retail businesses. These works themselves were derivative of the work in 1980's relating to database machines [11]. CMU's NASD [12] project is the foundation of the Object-based Storage Device (OSD) [10]. Even though the latest OSD specification lacks capacity for computation offload, OSDs provide an excellent path for realization of intelligent storage.

The idea of deploying intelligent peripherals for building highly responsive systems has been studied in field of networks and parallel programming models. Active networks [13] proposed injection of code modules on the routers and other network components, allowing on demand customization of the network infrastructure. The network services could now be represented using these code modules, thereby decoupling them from the actual hardware on which they run. This enhances the reliability and availability of the network infrastructure, making it more robust to changes in network loads. Active messages [14] proposed a parallel programming model for massive multi-processor architectures, with the specific goal of overlapping computation with communication. It depends on intelligent peripherals and network aware processing elements, which can recognize the handler instruction in the active message, invoke the handler and include the message in a running computation. An example of active peripheral is programmable NICs which have been programmed for reducing network overhead, by offloading computation to the NICs and reducing unnecessary copies in the critical path of data transfer [15]. Advanced interconnects like InfiniBand [16] allow remote direct memory access and remote atomic operations simplifying programming in distribute memory systems like shared-nothing clusters.

Coming back to active storage architecture, several hardware designs have been proposed. IBM's Intelligent Bricks [17] and HP's Federated Array of Bricks [18] are self-contained and self managed disk systems built using low cost COTS drives. These systems are targetted at the problems

faced by data centers: low power and cooling requirements, building high density storage systems, high reliability and reduced management costs by self healing and built in fault tolerance. An important point about these systems is that they are designed around hardware resources which are near PC or server grade. This point was also made in storage bricks [19], about future intelligent bricks systems. A recent study [20] investigated the architecture design space of multi-level active storage systems, from disk level to disk-array-controller level while making a case for intelligent storage from power consumption perspective. Wise drives [4] also supports the idea that disk drives are at present well provisioned for sophisticated disk management operations, and their hardware resources could be further augmented for more richer interactions with user applications.

Intelligent disks not only improve application performance, but can also improve the design of the middlewares like the filesystems. Semantically smart drives [21] and type-safe disks [22] exploit the information that is mined from the filesystem data structure operations on the disk to improve the overall system performance by techniques like prefetching, efficient data layout and better caching support. These ideas can be incorporated in an OSD since it takes care of data layout, leaving file systems with higher levels of management. However, this issue is complementary to main thrust of this paper which is enabling infrastructure for offloading computation.

Our work is closely related to Diamond [23] which has built an intelligent storage system on OSD. Diamond targets read-only data-intensive applications like data mining, unlike our system which is both read and write capable. In Diamond's case offloaded computation uses a library of pre-compiled filters residing on the storage server. Multi-view storage system (MVSS) [24] offloads computation by migrating the application level code to the storage servers. MVSS achieves this by exploiting unused address space by creating virtual disks in that address space and trapping any calls made to those locations. The advantage of this approach is that applications need not be rewritten, however it implicitly assumes that storage servers are binary compatible with the applications, with is not necessarily true. Unlike Diamond and MVSS, Script-RPC [25], intends to extend the functionality of NFS servers by enabling execution of user scripts at the site of the servers. Unlike Script-RPC, our focus is not just extensibility of OSD, but to allow full-fledged computation. However, we do believe that scripts are more portable way of offloading computation. The concept of offloaded computation has been generalized in the context of cluster computing by MapReduce [26]. Its key contribution is the realization of a fault-tolerant infrastructure for embarrassingly parallel read mostly data-intensive workloads on shared-nothing clusters. In MapReduce, the system automatically splits the computations and maps them to nodes with data so that computations can be executed close to the data, then the sub-solutions of the split computation are reduced or merged into the global solution of the problem.

III. TECHNICAL BACKGROUND

The Object-based Storage Devices (OSD) is a new disk interface technology that is being standardized by SNIA T10 technical committee [10]. The OSD traces its origin to NASD [12] project at CMU. An OSD places increasing responsibility for data layout and management on the devices themselves. Unlike traditional block-based disks that represent storage as a linear array of bytes and export a logical block address (LBA) interface, OSDs export an object interface. Enabling a high-level object interface that allows OSDs to take over low-level data layout management operations from the host operating system. Another promising feature of OSDs is user-defined attributes which gives users powerful semantic control over the management of the data.

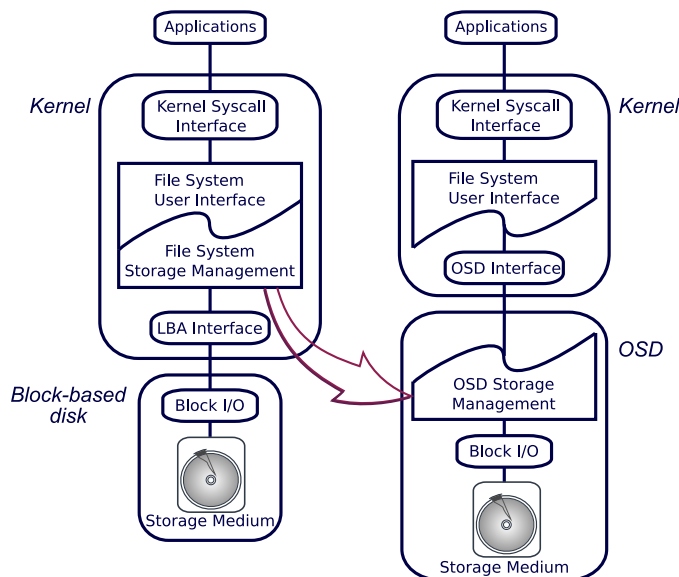


Fig. 1. Comparison of Block-based and Object-based Storage Models. The Data Storage Management unit has moved from the kernel to the OSD.

At present intelligent storage is neither standardized nor available as a commodity. However, the OSD is considered a promising platform for realizing intelligent storage because expression of computation is much more compact and easier when done over objects compared to blocks. In the case of block based device, since the data layout is determined by the filesystem, the relevant metadata also need to be transferred for executing computation against the data blocks, which is a non-trivial task.

IV. DESIGN GOALS

Our design philosophy with respect to offloaded computation is that it should be naturally expressible, conveniently offloadable, offering intuitive interaction with a running job and easily manageable at the device. We discuss these goals in more detail below.

A. Interaction

A primary decision which underpins the subsequent steps is interaction semantics. Given the nature of offloaded com-

putation, one may either expose interactive semantics or treat the offloaded computation as a batch job. Under interactive session, a user might be able to stop, introspect the current state and resume the computation just like in a debugger. This may prove useful as far as programmer productivity is concerned, however it will complicate the design of the target, by putting undue demands on it. Since OSD commands anyway follow request-response paradigm, one can simplify the interaction using a batch mode type operation, where computation is offloaded, with periodic status checks and termination capabilities, quite similar to batch queueing systems found in clusters [27].

Another dimension of interaction semantics is the nature of interaction: synchronous versus asynchronous. Since OSD commands rely on kernel resident iSCSI stack, it exports split-phase asynchronous semantics allowing multiple outstanding requests from a single client. A similar interface could be exported for offloaded computation, where the task of offloading computation could be split into multiple steps: like task submission, task execution etc. That allows a client to overlap computation with offloaded computation. This might be useful for long running tasks, however for short tasks synchronous semantics are sufficient, in fact desirable as there is no additional overhead.

Offloaded computation is expected to generate status information about its current state. A related question, is once the task is over where is the state stored? One way of solving this state capture problem is to associate a special *OSD userobject* with the offloaded computation and pipe all the status information to it. Final state of the computation could also be stored there. This also allows the user application to periodically check the status using normal OSD commands like `GETATTR`, `READ`, `REMOVE` to verify and administer the job. The conflicts between sandbox and application could be resolved by only allowing read capabilities while the job is running and allowing modification capabilities once it is over.

B. Computation

There are several ways in which offloaded computation can be expressed. One of the straight forwards ways of offloading computation is *operation compounding*. At present most of the OSD commands [10], are targetted to a particular object. Therefore multiple round trips are required if different operations are to be performed on different objects, or even a single object. One way to reduce the round trips is to compound multiple operations in a single command, analogous to NFS compound operations [28]. However, this approach does not offer decision based computation, and does not fully exploit the advantage of operating close to the objects. A more powerful and semantically rich way of expressing computation is to expose the object structure and operate directly on the objects. This will require addition of new functionalities like content based queries on both object data and attributes. Convenient syntactic sugar can be added to make it more amenable to object-oriented programming paradigm. The offloaded computation should have full fledged computation

primitives like branches, loops, functions, modules like any ordinary program.

A related issue to the type of computation is granularity at which computation is offloaded. In case of Diamond [23], computation was at the granularity of filters used for image analysis. The offloaded computation is more coarser grained, where a offloaded program selected a filter depending on the output of another filter. We think that finer-grained computation operating on the exposed object structures is more desirable. We would also like to restrict the amount of computation to light weight with less computation per byte type operations. The second choice is whether to restrict the computation to read only type workloads [23], [3], [1] or to have read-write ability. Many data intensive applications like data-mining, semi or un-structured text processing exhibit read-only type behavior. However, ability to modify data will reduce network round trips for read-modify-write cycles.

Offloaded computation can have several formats like plain text scripts [25], byte code [29], compiled or cross-compiled object code [23], [15], [30]. While the binary format is more compact, secure and has less overhead, the interpreted scripts are more productive, portable and more amenable to quality and safety analysis. The system constraints will determine the appropriate format of the offloaded computation. Our idea is to articulate a generic framework without bias towards any particular format.

C. Consistency and coherence

An important constraint on the offloaded computations which modify the data on the OSDs is consistency. The effects of the offloaded computation should be equivalent to changes made by a stream of OSD commands directed to the OSD. Since conservation of system resources is important, offloaded computation would have to share time and space with normal OSD requests. This means that the changes affected by the offloaded computations must be coherent and correct and should be not different than changes affected by OSD commands. The modifications by offloaded computations must be at least the same granularity as the OSD commands, and must respect the isolation and atomicity constraints as the OSD commands.

D. System architecture

Given an opportunity for a clean slate design of the system, how should offloaded computation be managed and run? An ideal system would create a system or kernel layer and application layer. The system layer would be responsible for resource management providing a common platform for different types of commands to be run. The application layer would concern itself with execution of normal OSD commands or offloaded computation. Optimal utilization of system resources implies that multiple tasks should be time and space multiplexed. This requires isolation, minimizing side-effects, appropriate context-switching with fairness and progress as goals.

E. Management

Since offloaded computation is going to be run concurrently with OSD commands, its effects must be isolated from the rest of the system. One way of achieving this is to run the computations in a sandbox. Isolation of offloaded computation simplifies management. For eg. enforcement of resource requirements like time-for-computation, memory requirements etc. could be managed per computation task and enforced per sandbox. Any abnormal behaviour like runaway tasks could be immediately identified and terminated, if that fails, the sandbox itself could be killed.

An important component of system design is resource management. Having a systems layer servicing all resource requests from application layers, allows it a global perspective of current system needs and enables better allocation of resources like cpu-time, memory, I/O and network bandwidth. A related design decision is priority assignment for different classes of tasks: normal OSD commands v/s offloaded computation. Unlike previously, where the commands could be classified under different classes [31] from which resource allocation followed, a single offloaded computation may execute all the commands. So, simple command based classification will not be optimal.

The system must conserve resources for optimal efficiency. This requires dynamic provisioning of the resources depending on load. The number of sandboxes, the amount of cpu available for the tasks etc. must be determined based on the current load.

Every OSD command carries with it an embedded *capability*, identifying the credentials of the user executing the command. Unlike the OSD commands, an offloaded computation may affect multiple objects. This raises new questions like: should there be a single capability for the whole computation, should each access be tested for capabilities, how long should capability be enforced?

F. Hardware resources

Since OSD is an interface technology, it could be implemented at levels ranging from a single disk to large disk-controller level. The hardware systems on which OSD would be implemented constitute a diverse eco-system. At one end we have embedded processors on disk with limited resources. However even COTS disks have firmwares so complex that they already have around 400,000 lines of code [32]. This implies that the embedded processor, is reasonably powerful and sophisticated to handle complex tasks with onboard cache of around 2-4 MB. At the other end of the spectrum are heavily provisioned disk-array controllers with memory cache of 4-8 GB [33] with server-grade processors. Future architectures of intelligent storage bricks are designed around the fact that these bricks will have PC or server grade hardware resources [17], [19]. Due to such diversity of compute resources, care must be taken in budgeting for hardware resources and design must submit to those constraints.

V. INTELLIGENT OSD TARGET

The design goals discussed in Section IV articulate the design space for the implementation of an intelligent OSD. The key component of focus is the computation execution environment. We have implemented our system by building upon the software implementation of OSD done in our previous work [34], [35], [36], [37], [38]. In the following sections, we will give a brief background of our software OSD infrastructure and explain the intelligent OSD target we have implemented.

A. Overview of OSD Target

Our OSD target is implemented as a user-level multi-process multi-threaded application. The OSD initiator library is implemented as a user-level library that is linked to the client application codes so that they can communicate with an OSD target. The OSD target presents itself as a SCSI target, and most of the initiator connection management is taken care of by the Linux kernel's iSCSI stack.

The OSD command set [10] is an extension of iSCSI commands and clients interact with an OSD target using these commands. Some examples of the commands are `CREATE`, `WRITE`, `READ`, `SETATTR`. These commands are dispatched to the target in a self-describing Command Descriptor Block (CDB), which the target parses and executes. Figure 2 gives an overview of different components of the OSD target which we describe in the following sections.

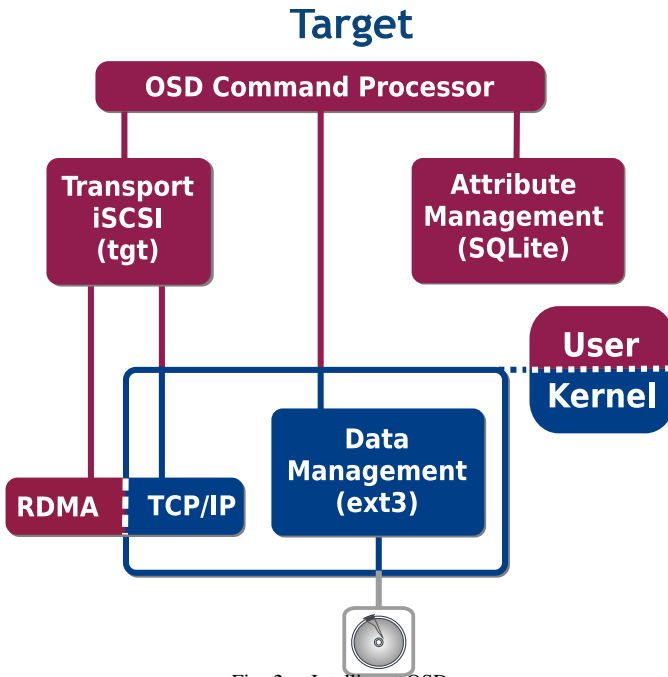


Fig. 2. Intelligent OSD

When a command is first received by the target, it is dispatched to the iSCSI layer for initial header parsing and sanity checks. This iSCSI layer is a user space implementation [39] since there is no iSCSI target in the mainline Linux kernel. The iSCSI layer is generic enough to handle different interconnect technologies from TCP/IP to advanced interconnects like

InfiniBand [16]. The iSCSI layer is responsible for connection management, buffer management and other client interfacing tasks.

Next, the OSD command along with its payload is dispatched to the OSD command processing layer. This is where the CDB is parsed, validated and executed. The command descriptor layer implements the business logic of an OSD target. At present our OSD target implements all mandatory OSD commands: object manipulation, input/output, and attribute manipulation commands. Several optional commands related to multi-object commands like `QUERY`, `SET MEMBER ATTRIBUTES` are also implemented. We have further extended the target with atomic operations [38] for special workloads.

Our userspace target relies on kernel based filesystem (EXT3 [9]) for data layout and management of data on the physical drive. The data of each object is treated as a file on the device and managed using standard filesystem interface. An interesting aspect of OSD is per object attributes. Since there can be several attributes, colocating attributes with objects complicates the attribute management. We collect all attributes of all the objects in a separate database. Unlike our previous implementation [34] we use Berkeley DB4 [40] for our current implementation.

B. User interface

At a high level, we chose batch interface for offloaded computation. As discussed in Section IV-A, batch mode type interface is more natural for offloaded computation. However, the users still have a control over the tasks by being able to determine the current state and having the ability to terminate the task. Keeping the different trade offs in mind we enabled both synchronous and asynchronous interface. Synchronous interface is suitable for short tasks while asynchronous interface is suitable for long running jobs.

We have created a new command OSD EXEC for the offloaded computation and fashioned it after OSD WRITE. The affect of this command is restricted to a zone, determined by the arguments supplied by the user. The zone restricts the set of objects that the offloaded computation can access during its lifetime. The zone can be all the objects in the OSD, all the object within a partition, or a collection or just a single user object. Next, a user must also supply capability indicating its privileges to execute the command. At present we have a coarse granular security arrangement granting user the privileges to execute commands on all the objects within a zone. Lastly they must supply the code to be executed as a payload of the command.

Every offloaded computation that is successfully accepted by the target for execution gets a surrogate object which is used for its state capture. This object's identifier is also used as a token by the users to determine the status of the jobs and final clean up. When the user submits the command it can optionally ask for this token by using the optional get attributes component of the OSD command. All other get/set attributes are ignored under the current implementation.

C. Offloaded computation format

Among the several choices for the offloaded computation as discussed in Section IV-B, we opted to expose the full object interface to the user rather than operation compounding [28]. This is more powerful and allows a simpler and intuitive interface to the users. Secondly, we have allowed both read and write operations. This prevents any unnecessary round trips during read-modify-write operations.

```
import osd

p = osd.partition(65536, 0)
if not p.exists():
    p.create()
p.get() # get reference to the partition

u = osd.userobject(65536, 65536)
if not u.exists():
    u.create()
u.get() # get reference to the user object

# set some attributes
attrs = [(10, 3, "attribute"), (11, 3, 0xdeadbeef)]
u.setattr(attrs)

# get the attributes
attrs = [(10, 3), (11, 3)]
print u.getattr(attrs)

# write to the user object
u.data.write("Hello World\n")

# read from user object
for l in u.data:
    print l

u.put()
u.remove()

p.put()
p.remove()
```

Listing 1. Example of offloaded computation

Users write their programs in Python [41] language. We chose an interpreted language since it is relatively more portable and productive compared to compiled binaries. Extensibility and embeddability features of Python with existing C-based target infrastructure was one of the reasons for its selection over other interpreted languages. Further Python code can be compiled into byte-code, compressed and encrypted if security is important. Another feature of Python is rich ecosystem of modules, which makes diverse workloads possible. By selecting Python we traded off performance for compactness and rich library of tools and modules. The intuitive object-oriented paradigm of Python is well-suited for operating on the object model of an OSD.

Listing 1 shows an example of offloaded computation. In that example, a partition is created, then an userobject is created within that partition, some attributes are set, data is written and finally the object is removed.

The object abstraction is still work in progress. The future additions include, searches on attribute value, rich comparison

of attribute values, query by object content and other functionalities. The object manipulation is analogous to operating on a file.

D. Computation execution environment

The computation execution environment (or sandbox) is responsible for actual execution of the offloaded computation. As discussed in Section IV-E, sandbox must provide isolation to the executing tasks, ensure correct execution of the code, maintain coherence and consistency of the state, and provide management interface to control the executing tasks. To ensure both synchronous and asynchronous semantics, the sandbox is designed with asynchronous architecture from the user all the way to the sandbox, and builds synchronous semantics on top of asynchronous architecture.

To ensure isolation, the sandbox is implemented as a separate process. Each sandbox is made up of two threads: management and compute thread. Management thread is responsible for managing the compute thread, constructing compute tasks, getting task status and other management tasks. The compute thread is the one which executes the task. Besides these, two threads, a proxy thread is created for each sandbox. This proxy thread is a part of the main OSD application thread which is also responsible for executing normal OSD commands like CREATE, REMOVE. The job of a proxy thread is to execute OSD commands, especially those involving attribute manipulation, on behalf of the sandbox.

When the sandbox is started, it initializes and embeds a Python [42] interpreter, which the compute thread takes over. After initial book-keeping, each compute task is fed to this python interpreter which executes the code. As shown in Listing 1, every script begins with `import osd` line which loads the OSD module. This module exposes the OSD API to the applications, in other words the module extends the Python interpreter for OSD tasks. Every API call made by the script is trapped by the OSD module, which translates the call into appropriate OSD command and dispatches it to the proxy thread. The proxy thread executes the task, returns the status and results to back to the sandbox, which demarshals the response into a Python structure and sends it back to the interpreter, completing the request-reply loop.

Python poses some peculiar challenges with respect to state management. A initial design idea of spawning a separate thread for a new computing task was considered. However, python interpreter initialization is computationally expensive, so its cost must be amortized over many compute tasks. Secondly, python interpreter allows only one thread to execute at any one time. This creates two problems: lack of isolation, that is one thread can see what other thread is doing and overwrite its state, secondly performance, since only one thread is executed at any given time, python's interpreter conflicts with operating system scheduler. This means that having multiple threads each with their own interpreters, under a single address space is not possible. This constraint, however was not a hinderance for us as we were anyway planning to isolate execution of computational tasks. We solve multi-

tasking problem, by having multiple sandboxes. Another point is that a running task leaves some state in terms of imported modules, non destroyed objects etc. Since the interpreter is going to be used by many offloaded tasks, this state must be cleaned prior to running a new task, else it can compromise security.

E. Management

For optimal use of resources, a number of commands should be run concurrently. We use a master-worker model, where the master thread creates a task and submits them to the work-queue. The worker threads in turn pull the tasks from the work queue and execute the tasks. The result of the task is communicated back to the master thread, which is responsible for sending the response back to the initiator. The master is responsible for the iSCSI layer (see Figure 2), receives the incoming requests, does iSCSI header processing, translates the command to OSD task which it submits to the worker queue. The workers are responsible for the OSD business logic and once command is executed they send the status back to the master, which in turn sends the status back to the initiators.

There are two classes of tasks: the normal OSD commands and the special offloaded computation. For the normal OSD commands, the worker thread is responsible for parsing the command, executing it and finally sending the response back to the master thread. For the special commands, the worker thread just submits the task to the appropriate sandbox proxy. All further communication of the status is handled by the proxy thread. For example, once the task is completed, the proxy thread sends the status back to the master thread. The reason for treating the two tasks separately follows from the tension between resource conservation and implementation of synchronous semantics for offloaded computation. For synchronous commands, the initiator expects the response *after* the completion of the computation. However, executing the computation is the responsibility of the sandbox. If a worker thread is made responsible for the delivery of the status, then it must either wait for the task, thereby wasting the resource, or be prepared for notification by the sandbox, which complicates the design. Instead making the sandbox’s proxy responsible for status delivery, is a cleaner design, since proxy will always be notified by the sandbox once the task is completed.

Since resource conservation is important, different actors *viz*, the number of worker threads and the number of sandboxes are dynamically adjusted by the resource manager. The metric used for determining the current load is the number of pending requests. As the request pressure increases, the resource manager starts more workers and sandboxes as required. There is an upper threshold for the number of workers and sandboxes, (8 and 4 respectively under the current implementation). The key objective of load management is graceful degradation under heavy load [43], [44], while remaining agile during normal load.

VI. EXPERIMENTS

All of our experiments are run on a 71-node Linux cluster. Each node has dual AMD Opteron 250 processors, 2 GB of RAM and an 80 GB SATA disk. The cluster runs the Linux operating system, version 2.6.30. The onboard Tigon 3 Gigabit Ethernet NIC is used for communication, with a single SMC 8648T 48-port switch.

A. Python overhead

The main goal of this experiment is to quantify the overhead of python for a subset of most frequently used operations. We chose CREATE, REMOVE, SETATTR, GETATTR commands for this experiment as they are representative of most frequently used operations. The create command measures time to create an object, remove measures time to remove an object, setattr measures time to set 6 attributes of different types, and getattr times how long it takes to fetch 6 attributes. For all the commands times were measured over 1000 iterations.

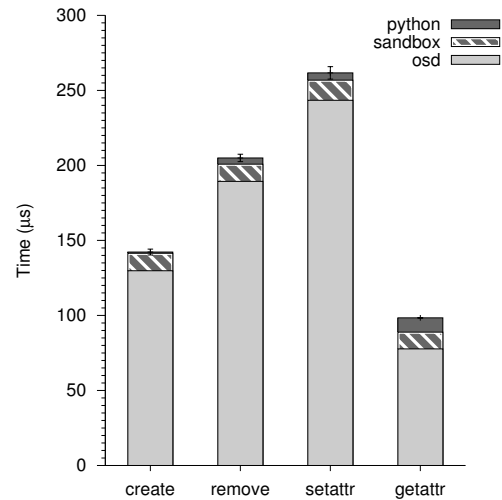


Fig. 3. Share of execution time taken at different layers.

When a command is invoked in python, it is trapped by the OSD module which demarshals the request and converts it into C format. Then a call is made to *sandbox* library which dispatches the command to the sandbox’s proxy where the command is executed, with the results traveling the reverse way up the stack. In this experiment, the time taken at each of the three layers: *osd*, *sandbox*, *python* is measured.

Figure 3 shows the time for the four commands. The region within a single bar shows the amount of time spent at that layer. The errorbars show the 95% confidence interval of the cumulative time measured at the *python* layer. The remove command takes more time than create as it has to scan all the tables to remove any traces of the object. The setattr command takes the maximum among the four, as attribute insertion is serialized and is directly proportional to the number of attributes being set. The getattr is the fastest, as it involves mostly reads.

TABLE I
PERCENTAGE OF TIME SPENT IN EACH LAYER

	OSD (%)	Sandbox (%)	Python (%)	Latency (μ s)
create	91.26	8.16	0.58	142.26
remove	92.39	5.58	2.02	204.98
setattr	93.01	5.12	1.87	261.69
getattr	78.96	11.33	9.70	98.41

Table I shows that the python overhead varies from as low as 0.83μ s in case of create to as high as 9.6μ s in case of getattr. The getattr command incurs more overhead since it involves Python-to-C and C-to-Python conversions of attributes, which involves copies. The setattr only incurs Python-to-C conversion as no attributes have to be returned. The sandbox layer accounts for around 11–14 μ s, this is the price paid for isolating the computation execution environment from other OSD commands.

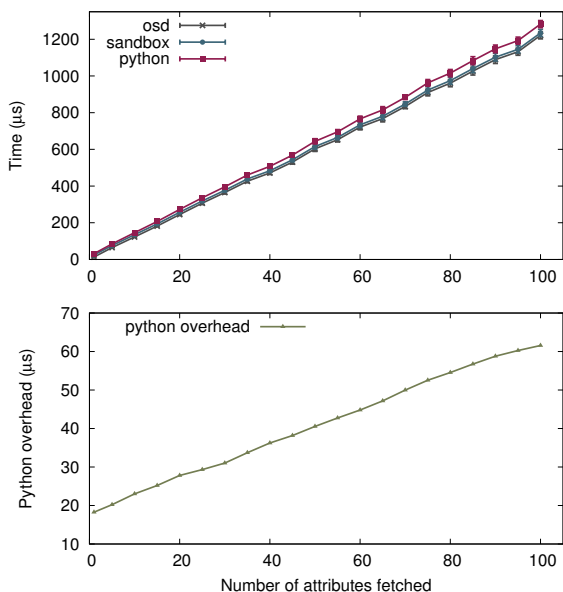


Fig. 4. Absolute time at different layers (top) and getattr’s python overhead (bottom) as a function of number of attributes.

We further analyzed getattr overhead as a function of number of attributes fetched. In this experiment, the attribute value is a string of 26 characters. For each timing datapoint N attributes of 1000 objects were fetched, giving 1000 samples, from which the mean and standard deviation were obtained and plotted. The overheads at each layer were measured and python overhead was tabulated. Figure 4 shows the absolute times and overhead at the python layer as a function of number of attributes fetched. The python overhead increases as a function of number of values that have to be marshalled and demarshalled, due to the number of copies involved in Python-to-C and C-to-Python conversions.

The overhead of Python is very low for the commands

which involve minimal Python-to-C translation and increases for the commands like getattr which require marshalling and demarshalling of the arguments. Python’s overhead for command processing is acceptable tradeoff for flexibility and portability the language provides to user applications.

B. Python I/O overhead

A major operation in offloaded computation is I/O, hence it is important to quantify Python’s I/O overhead. Python uses file stream API exported by the underlying system for I/O. We define I/O overhead as the time consumed by the Python interpreter in doing extra work besides I/O. We measure this overhead for both read and write. For read, a file is read repeatedly from its start into a buffer and the read times were measured. For write, data in a buffer is written repeatedly to the start of a file and the write times were measured. The reason we read and write to the same offset of the file is to remove any disk effects and get pure compute time overhead of the python interpreter.

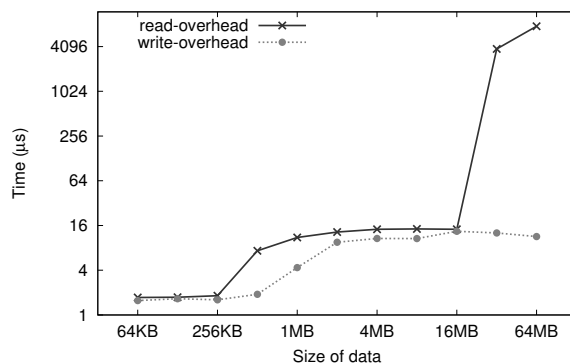


Fig. 5. Python’s I/O overhead as a function of size of data. Note X and Y axes are log₂ scale.

Figure 6 plots Python’s I/O overhead against different sizes of data buffer. The write overhead is reasonably stable for increasing buffer sizes. The reason for this is that the write buffer is being reused in different iterations and both source and destination memory locations are cached. The first major increase in write overhead is at 1 MB. L2 cache size is 1 MB, and size of both source and destination buffers exceeds cache size.

The plot for read is more interesting, there are two bumps, one at 0.5 MB and another big one at 32 MB. The first bump is due to cache size limitation. Unlike writes where at 0.5 MB both source and destination buffers fit in the cache, in case of the reads, the cache is polluted due to buffer allocation and deallocation operations as explained next.

There are two reasons for the second bump. The first is the way the python interpreter translates a file read operation. For every read, the interpreter allocates a new buffer and reads the file into it, and after read operation the buffer is deallocated. This means the buffer is not reused between two consecutive read operations. The second reason is the underlying memory manager malloc. For small buffers malloc uses heap to

service the allocation requests, however for large buffers it uses `mmap`. It does this for fragmentation and memory utilization reasons. These `mmap` operations and python interpreter’s inability to reuse read buffer across multiple iterations makes read operation expensive. First, the read into a freshly memory mapped region is expensive due to page faults to bring the page into the memory and secondly due to the fact that kernel must zero out the memory mapped pages before handing them to the user. This means that first reads into memory mapped region are twice as expensive as subsequent reads, and since Python is going to free those pages after the read, this cost is not amortized over multiple iterations. Secondly, when python frees these memory mapped pages, it uses `munmap` system call which itself is expensive. Moreover the cost of `munmap` is directly proportional to the size of the memory being unmapped. This is the cost that we see at the second bump at 32 MB and subsequent sizes.

There are two ways of tackling this buffer reuse problem in read. One is to restrict buffer sizes to less than 32 MB for reads. A second solution that we are working on is to export an API similar to unsupported `readinto` API of file read, where a user supplies the buffer into which data is read.

C. Python computation overhead

The aim of choosing Python as a runtime system for offloaded computation is diversity of applications that Python can interface. However, this makes defining the Python overhead different. One important class of applications is data mining, more specifically processing of semi-structured and unstructured documents. An example of that is processing network file system logs generated at Ohio Supercomputer center. These logs help the system administrators to keep a record of the historical performance of the servers, diagnose any performance problems and use the anomalies to forecast any system related problems. These logs are usually in plain text generated by a packet sniffer [45]. Each line of the log is usually between 20—1024 characters long and gives details about the RPC header information like command opcode, arguments, status etc. In this experiment we use a 15 minute log from an NFS server which has 503937 lines and is ≈ 98 MB in size.

At each line we check if it is a RPC request, then the `userid` and the command opcode are parsed out. Rather than tokenizing, testing and parsing are done using regular expressions. This information is used for generating summary statistics: the number of commands per unique user, and number of requests of a particular opcode type. Hash tables are used to gather these statistics. During a given run of the experiment, N lines are processed and the processing time is recorded. We use number of lines as control variable since log processing is line oriented. We implemented this code in python and C. We Python’s `re` module and `pcre` [46] for C implementation. The C code is run by running `exec` command against its executable from sandbox, which ensure that both Python and C are run under similar environments.

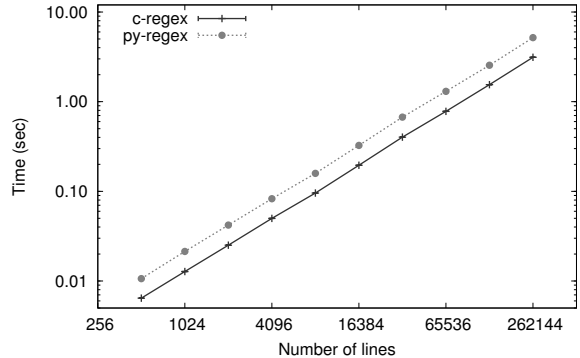


Fig. 6. Compute time in seconds versus number of lines processed. Note Y-axis is on \log_{10} and X-axis is on \log_2 scale.

D. When to offload?

One important question for the applications is whether offloading will benefit them or not. If this decision could be automated by the system, it can ease application development. There exist many applications with deterministic data sizes. For example, generation of summary statistics, computing image histograms, scanning a log file etc. where the size of data known beforehand. For these applications, it is possible to estimate execution times and make the decision at runtime. As discussed in Section IV-F, OSD interface could be implemented on wide variety of hardware and the dynamic characteristics of the system can change. Hence this decision needs to be made on the fly based on latest system parameters. The problem is to understand what parameters are important.

Let $T_{fac}(x)$ be the time to fetch-and-compute as a function of data size: “ x ”. There can be three scenarios depending on the possibility of overlapping computation with communication: if there is no overlap, execution time is the sum of network and compute times ($T_{net}(x) + T_{cpu}(x)$), otherwise if the network time dominates then ($T_{net}(x) + T_{cpu}(k)$) or else if compute time dominates ($T_{net}(k) + T_{cpu}(x)$). Here “ x ” is the size of data and “ k ” is the size of a chunk, that is used to break the data and pipeline data fetches. T_{fac} would be the minimum of the three choices.

$$T_{fac}(x) = \min \begin{cases} T_{net}(x) + T_{cpu}(x) \\ T_{net}(k) + T_{cpu}(x) \\ T_{net}(x) + T_{cpu}(k) \end{cases} \quad (1)$$

$$T_{off}(x) = T_{code} + T_{sb}(x) + T_{result} \quad (2)$$

Equation 2 describes offloaded computation time, T_{off} , which is the sum of T_{code} , the time to offload the code, T_{sb} , the time to compute code in sandbox and T_{result} , the time to fetch result back. Instead of sending the result of the computation as a response, only the status of the computation is sent. A second call is required to retrieve the results. This is done in order to free the resources during the computation, that would otherwise have to be reserved for retrieving the result. Further, in many cases the size of result is unknown, so it is better to separate result retrieval into a separate request.

$$T_{net}(x) = L_{net} + B \times x \quad (3)$$

$$T_{cpu}(x) = C_o + C \times x \quad (4)$$

$$T_{sb}(x) = \alpha \times T_{cpu}(x) \quad (5)$$

$$T_{code} = L_{net} + B \times (\text{code size}) \quad (6)$$

$$T_{result} = L_{net} + B \times (\text{result size}) \quad (7)$$

Equations 3—7 show first-order models of the individual terms. T_{net} is the network time, where L_{net} is the latency and B is the inverse of the bandwidth, that is the network processing time per byte. T_{cpu} is the computation time on local machine, where C is the computation time per byte and C_o is the overhead. T_{sb} can be expressed in terms of T_{cpu} , with α as sandbox penalty. Usually sandbox is slower than local machine, hence $\alpha > 1$. T_{code} and T_{result} are functions of code size and result size respectively.

Based on above equations it is better to offload when:

$$T_{off} < T_{fac} \quad (8)$$

When the size of the data x is large, and the size of the response is relatively small, we can make following approximations: $T_{net}(x) \approx B \times x$, $T_{off}(x) \approx T_{sb}(x)$ and $T_{cpu}(x) \approx C \times x$. These are valid since the question of offload arises when the size of the data is large. Using these approximations for the terms in equation 8, we can decide whether to offload or not according to the following rules:

Scenario	Criteria	Offload
No overlap	$\alpha < 1 + \frac{B}{C}$	Yes
T_{net} dominates	$\alpha < \frac{B}{C}$	Yes
T_{cpu} dominates	—	No

T_{cpu} dominates T_{net} when $\frac{B}{C} < 1$. The above result is justified since if T_{cpu} dominates, and the local machine is powerful than OSD, it is better to fetch-and-compute rather than offload. If the computation is uniform across data, C and α can be estimated by a executing the computation on a small data size. C would be approximately the slope of the curve and α the ratio of sandbox time to local machine time. If that is not possible, applications can provide hints about C . B can be computed based on past network performance. To summarize, the key parameters are $\{\alpha, B, C\}$, and their mutual relationships determines whether to offload or not.

E. Resource consumption profile

One of the important questions is how much compute resources are available for offloaded computation. We believe that the primary mission of OSD is to execute normal OSD commands like CREATE, REMOVE, SET ATTRIBUTES, READ, WRITE etc. In case there are computation resources available the offloaded computation should be run. This raises the question about the source consumption profile of the OSD when it is busy doing normal OSD.

In this experiment we try to answer that question. A single OSD target is chosen and two initiators continuously stream requests to the target to ensure that target is continuously busy. The request stream is derived from the NFS traces [45] by converting the NFS RPCs calls into corresponding OSD commands. Care was taken to maintain causality between the commands and we ensure that no command fails, and every command is executed successfully at the target. Since the target machine is a dual-cpu node, we confine all the processes and threads of OSD application to a single CPU by setting affinity for all schedulable entities. While this trace was executing the cpu resource usage of different entities was sampled periodically at every 1000 commands.

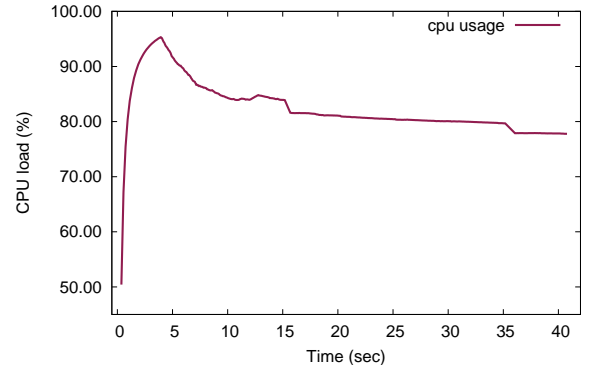


Fig. 7. CPU usage of OSD target as function of time.

Fig. 7 shows the overall resource consumption of the complete OSD target application as a function of time. Each time instant is a timestamp when the sample was collected. On the Y-axis we show the relative CPU load of the OSD target. As we see after the initial peak load of 90% resource consumption was continuously falls and stabilizes at around 80%. This means that even when the OSD target is continuously busy, there is still around 20% CPU resources still available which could be used by offloaded computation.

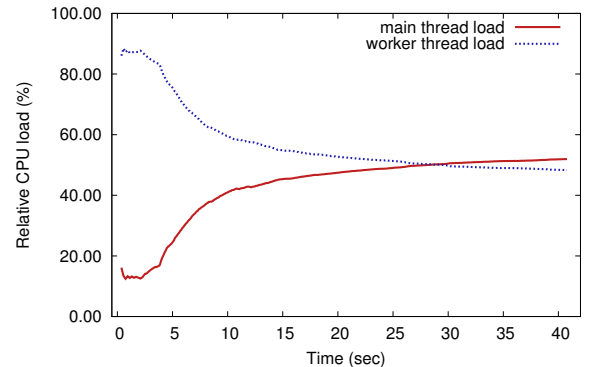


Fig. 8. Relative CPU usage of main and worker thread as a function of time.

Now that we have a global resource consumption profile, we wanted to know the breakup of the resource consumption by different tasks. There are two main tasks in the OSD target:

communication and OSD command processing. As discussed in Section V-A communication is handled at the iSCSI layer. We implement iSCSI layer as a separate thread, called the main thread. The rest of the OSD processing is split among several worker threads. In this experiment we work with a single worker thread as our initial tests showed that gives the best performance.

Fig. 8 shows the relative performance of the main thread and the worker thread against the time. Initially the worker thread dominates the processing since initial part of the trace has many operation with heavy OSD processing load. However as the time progresses the main thread starts dominating. Rather than the shape of the curves the main lesson from this experiment is the equal amount of resources must be dedicated to communication and OSD processing. This information will be helpful when future OSD devices will be implemented on multi-core processing elements.

VII. FUTURE WORK

This work is still in progress. The following list gives a brief overview of the things we have planned for the immediate future:

- Thorough evaluation of different resource allocation strategies for OSD.
- Testing of OSD on different speeds to quantify performance envelop and determine what can and cannot be offloaded.
- Quantification of benefits of OSDs on representative data intensive applications.
- Development of a scalable, fault-tolerant infrastructure for auto-parallelization of data parallel applications.

REFERENCES

- [1] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *VLDB '98: Proceedings of the 24th International Conference on Very Large Data Bases*, 1998, pp. 62–73.
- [2] A. Acharya, M. Uysal, and J. Saltz, "Active disks: programming model, algorithms and evaluation," *SIGPLAN Not.*, vol. 33, no. 11, pp. 81–91, 1998.
- [3] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "A case for intelligent disks (idisks)," *SIGMOD Rec.*, vol. 27, no. 3, pp. 42–52, 1998.
- [4] G. F. Hughes, "Computers: wise drives," *IEEE Spectrum*, vol. 39, no. 8, pp. 37–41, 2002.
- [5] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proc. of the 4th Ann. Linux Showcase and Conf.*, 2000, pp. 317–327.
- [6] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage," in *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, Pittsburgh, PA, Nov. 2004.
- [7] S. Ghemawat, H. Gobiuff, and S.-T. Leung, "The Google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2003, pp. 29–43.
- [8] Cluster File Systems, Inc., "Lustre: a scalable high-performance file system," Cluster File Systems, Tech. Rep., Nov. 2002, <http://www.lustre.org/docs/whitepaper.pdf>.
- [9] S. Tweedie, "Ext3, journaling filesystem," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, Jul. 2000.
- [10] R. O. Weber, "Information technology—SCSI object-based storage device commands -2 (OSD-2), revision 3," INCITS Technical Committee T10/1729-D, Tech. Rep., Jan. 2008.
- [11] D. J. DeWitt and P. B. Hawthorn, "A performance evaluation of data base machine architectures (invited paper)," in *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, 1981, pp. 199–214.
- [12] G. A. Gibson and R. V. Meter, "Network attached storage architecture," *Communications of the ACM*, vol. 43, no. 11, pp. 37–45, 2000.
- [13] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 5, pp. 81–94, 2007.
- [14] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer, "Active messages: a mechanism for integrated communication and computation," in *ISCA '91: Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.
- [15] P. Shivam, P. Wyckoff, and D. Panda, "EMP: zero-copy os-bypass NIC-driven gigabit ethernet message passing," in *Supercomputing '01*, 2001, pp. 57–57.
- [16] *InfiniBand Architecture Specification*, InfiniBand Trade Association, Oct. 2004. [Online]. Available: <http://www.infinibandta.org/specs/>
- [17] W. W. Wilcke, R. B. Garner, C. Fleiner, R. F. Freitas *et al.*, "IBM intelligent bricks project – Petabytes and beyond," *IBM journal of research and development*, vol. 50, no. 2/3, pp. 181–197, 2006.
- [18] Y. Saito, S. Frölund, A. Veitch, A. Merchant, and S. Spence, "FAB: building distributed enterprise disk arrays from commodity components," in *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 48–58.
- [19] J. Gray, "Storage bricks have arrived," in *Invited talk FAST '02*, 2002, pp. 1–18.
- [20] C. W. Smullen, S. R. Tarapore, S. Gurumurthi, P. Ranganathan, and M. Uysal, "Active storage revisited: the case for power and performance benefits for unstructured data processing applications," in *CF '08: Proceedings of the 5th conference on Computing frontiers*, 2008, pp. 293–304.
- [21] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Semantically-smart disk systems," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, 2003, pp. 73–88.
- [22] G. Sivathanu, S. Sundararaman, and E. Zadok, "Type-safe disks," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 15–28.
- [23] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki, "Diamond: A storage architecture for early discard in interactive search," in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004, pp. 73–86.
- [24] X. Ma and A. L. N. Reddy, "Implementation and evaluation of an active storage system prototype," in *In Workshop on Novel Uses of System Area Networks*, 2002.
- [25] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Evolving RPC for active storage," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 264–276, 2002.
- [26] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI '04: Proceedings of the 5th symposium on Operating systems design and implementation*, 2004, pp. 138–150.
- [27] A. Inc, "OpenPBS." [Online]. Available: <http://www.pbsgridworks.com/>
- [28] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network file system (NFS) version 4 protocol," IETF RFC 3530, Tech. Rep., Apr. 2003. [Online]. Available: <http://www.ietf.org/rfc/rfc3530.txt>
- [29] S. M. S. Inc, "Java applets." [Online]. Available: <http://java.sun.com/applets/>
- [30] O. M. Group, "CORBA component model, v4.0." [Online]. Available: <http://www.omg.org/technology/documents/formal/components.htm>
- [31] K. K. Osowski, T. Ruwart, and D. J. Lilja, "Communicating quality of service requirements to an object-based storage device," in *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2005.
- [32] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Iron file systems," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 206–220, 2005.
- [33] H. Inc., "Adaptable Modular Storage 2100." [Online]. Available: <http://www.hds.com/assets/pdf/hitachi-ams-2100.pdf>
- [34] A. Devulapalli, D. Dalessandro, P. Wyckoff, and N. Ali, "Attribute storage design for object-based storage devices," in *Proceedings of 24th*

- IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2007, pp. 263–268.
- [35] D. Dalessandro, A. Devulapalli, and P. Wyckoff, “iSER storage target for object-based storage devices,” in *International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2007, pp. 107–113.
 - [36] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, “Integrating parallel file systems with object-based storage devices,” in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–10.
 - [37] D. Dalessandro, A. Devulapalli, and P. Wyckoff, “Non-contiguous i/o support for object-based storage,” in *International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, 2008.
 - [38] A. Devulapalli, D. Dalessandro, and P. Wyckoff, “Data structure consistency using atomic operations in storage devices,” in *International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008.
 - [39] T. Fujita and M. Christie, “tgt: framework for storage target drivers,” in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, Jul. 2006.
 - [40] M. A. Olson, K. Bostic, and M. Seltzer, “Berkeley DB,” in *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, 1999, pp. 43–43.
 - [41] G. V. Rossum *et al.*, “Python programming language.” [Online]. Available: <http://www.python.org/>
 - [42] G. V. Rossum, “Extending and embedding the python interpreter.” [Online]. Available: <http://docs.python.org/extending/>
 - [43] M. Welsh, D. Culler, and E. Brewer, “SEDA: an architecture for well-conditioned, scalable internet services,” in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 230–243.
 - [44] V. S. Pai, P. Druschel, and W. Zwaenepoel, “Flash: an efficient and portable web server,” in *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, 1999, pp. 15–15.
 - [45] D. Ellard *et al.*, “nfsdump.” [Online]. Available: <http://www.eecs.harvard.edu/sos/software/index.html>
 - [46] P. Hazel, “Perl Compatible Regular Expressions (PCRE).” [Online]. Available: <http://www.pcre.org/>