H-0320 (HAI1408-001) August 4, 2014
Computer Science

IBM Research Report

Storlet Engine: Performing Computations in Cloud Storage

Simona Rabinovici-Cohen, Ealan Henis,
John Marberg, Kenneth Nagin
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden – Austin – Beijing – Cambridge – Dublin - Haifa – India – Melbourne - T.J. Watson – Tokyo - Zurich

Storlet Engine: Performing Computations in Cloud Storage

Simona Rabinovici-Cohen, Ealan Henis, John Marberg, Kenneth Nagin
IBM Research – Haifa
Haifa 31905, Israel
{simona,ealan,marberg,nagin}@il.ibm.com

ABSTRACT

The emergence of the cloud storage as infrastructure for serving large amounts of data over the WAN suggests new storage/compute paradigms. We propose expanding the cloud storage from only storing data to directly producing value from the data by executing computational modules - storlets - close to where the data is stored. This paper describes the Storlet Engine, an engine to support computations in secure sandboxes within the cloud storage. We describe its architecture and characteristics as well as the programming model for storlets. A rules mechanism allows implicit storlet activation via predefined conditionals complementing the explicit storlet activation. The security model supports storlets multi-tenancy and various types of sandboxes that constrain the resources available for the storlet execution. We also provide a performance study of the Storlet Engine prototype for OpenStack Swift cloud storage.

1. INTRODUCTION

Cloud object storage is emerging as an infrastructure suitable for serving large amounts of data expeditiously. Storage resources provisioned from converged infrastructure and shared resource pools present a cost-effective alternative to the traditional storage systems. The cloud provides new levels of scalability, elasticity and availability, and enables simple access to data from any location and any device.

Unlike traditional storage systems that have special purpose hardware optimized for I/O throughput with low compute power, the cloud object stores are built from commoditized off-the-shelf hardware with powerful CPUs to cheaply serve large data sets accessed from anywhere over the WAN. We propose to leverage the storage hardware processing capabilities to execute computations. We define *Storlets* as dynamically uploadable computational modules running in a sandbox within the storage, and propose them for the cloud object store. While with traditional storage systems, analyzing data requires moving the data objects from the storage system to the compute system, we propose to upload data-intensive storlets to the storage system and execute the processing close to the data residency.

For example in the pathology department, a tissue block is cut off into thin slices and mounted on glass slides. The thin slices may then be stained with different stains, e.g. HER2, PR, ER, and images are taken for the original and the stained slices. These images may be very large up to 200K over 200K pixels and can consume about 5-10 GB storage. Various computations and analytics need to be performed on those large images such as aligning images taken from the same tissue block, extracting a region of interest from an image and transforming it to a standardized format for a mobile device, detecting the cells in an image via computer vision algorithms, finding similar images by analyzing their features. Moving these large pathology images from the storage system to the compute system for processing is expensive, time consuming and sometimes even impractical. We created instead, Image Alignment Storlet, ROI Extraction Storlet, Cell Detection Storlet, and Features Similarity Storlet respectively. These storlets are uploaded to the storage and executed close to the data. Only, the result of the computations, which is generally much smaller than the original images, is returned to the pathology workflow that initiated these computations. Moreover, when the computation modules are updated e.g., due to new computer vision algorithms, they are dynamically uploaded to the storage, sparing the need to move large data again.

Briefly, the benefits of storlets are:

- **Reduce bandwidth** – reduce the number of bytes transfered over the WAN. Instead of moving the data over the WAN to the computational module, we move the computational module to the storage close to the data. Generally, the computational module has much smaller byte size than the data it works on.
- **Enhance security** – reduce exposure of sensitive data. Instead of moving data that has Personally Identifiable Information (PII) outside its residence, perform the computation in the storage and thereby lower the exposure of PII. This provides additional guard to the data and enables security enforcement

at the storage level.

- **Save costs** – consolidate generic functions that can be used by many applications while saving infrastructure at the client side. Instead of multiple applications writing similar code, storlets can consolidate and centralize generic logic with extensive or complex processing, and all applications can call these storlets.
- **Support compliance** – monitor and document the changes to the objects and improve provenance tracking. Instead of external tracking of the objects transformations over time, storlets can do that internally where the changes happen.

To enable the use of storlets, the storage system need to be augmented with a *Storlet Engine* that provides the storage cloud with capability to run storlets in a sandbox that insulates the storlet from the rest of the system and other storlets. The Storlet Engine provides a powerful extension mechanism to the cloud storage without changing its code; thus making the storage flexible, customizable and extensible. It expands the storage system's capability from only storing data to directly producing value from the data. At a high level one can say that storlets in cloud storage are analogous to stored procedures in a database.

The concept of storlets was first introduced in our paper [1] where storlets were used to offload data-intensive computations to the Preservation DataStores (PDS) [2][3]. It was afterwards investigated in the VISION Cloud project [4]. Our main contribution is the definition of the Storlet Engine, an engine to support computations in a sandbox within the cloud storage. We describe the architecture of the Storlet Engine in the cloud storage, its characteristics as well as the programming model for storlets. We describe a rules mechanism that allows implicit storlet activation via predefined conditionals, thereby enabling automatic conditional activation of storlets. The security model of the Storlet Engine supports storlets multi-tenancy and various types of sandboxes that constrain the resources available for the storlet execution including access to filesystem, network connections, memory and cpu consumption. We also provide a performance study of the Storlet Engine prototype for OpenStack Swift.

The Storlet Engine was developed as part of PDS Cloud [5] that provides storage infrastructure for European Union ENSURE [6][7] and ForgetIT [8][9] projects. We prototyped a Storlet Engine for the OpenStack Object Storage (code-named Swift) that is an open source software for creating redundant, fault-tolerant, eventually consistent object storage. It is a distributed scalable system, and uses clusters of commoditized hardware to store petabytes of accessible data. The Storlet Engine prototype is used to process archived data sets in medical, financial, personal and organizational use cases. Various data sets and storlets were examined for the ENSURE and ForgetIT projects including deidentification storlet to deidentify sensitive medical records, image transformation storlet to transform images to sustainable formats, fixity storlet for flexible integrity checks of objects, and various image analysis storlets to study pathology images.

The rest of this paper is organized as follows. Section 2 surveys related work. In Section 3 we provide the Storlet Engine architecture and the programming model for storlets. The rules mechanism for implicit storlet activation is introduced in Section 4. Section 5 discusses the security model to support multi-tenancy and various types of sandboxes. In Section 6 we study the performace of the Storlet Engine. We conclude in Section 7 with a summary and future plans.

2. RELATED WORK

Storage Clouds (public or private) are recent models for storage provisioning. They are highly distributed (employ multiple storage nodes), scalable and virtualized environments that offer availability, accessibility and sharing of data at low cost. OpenStack is an emerging open source software for building cloud computing platform for public and private clouds, with a huge amount of interest and rapidly growing community. The OpenStack software includes the Swift object storage [10] on which we implemented our Storlet Engine prototype.

Efficient execution of data-intensive applications was investgated for many years e.g. [11]. As the data objects become larger and distant, there is an increasing focus on performing computations close to the data.

Stored procedures and active databases are at a high level analogous to storlets in object storage. However, object storage is meant for large blobs and provides eventual data consistency while databases are meant for tabular data and provides strong data consistency. Consequently, the Storlet Engine need to confront with different challenges and solve them with distinct approaches.

Ceph [12] is a free software storage platform designed to present object, block, and file storage from a single distributed computer cluster. Ceph's main goals are to be completely distributed without a single point of failure, scalable to the exabyte level, and freely-available. The data is replicated, making it fault tolerant. Ceph software runs on commodity hardware. The system is designed to be both self-healing and self-managing and strives to reduce both administrator and budget overhead. Ceph has the ability to add class methods which enables computations in the storage. However, these class methods cannot run in a sandbox for now. In contrast, our Storlet Engine runs code within a sandbox, which provides better security.

OpenStack Savanna [13] attempts to enable users to

easily provision and manage Hadoop clusters on OpenStack by specifying several parameters like Hadoop version, cluster topology, nodes hardware details. Savanna quickly deploys the cluster and provides means to scale already provisioned cluster by adding/removing worker nodes on demand. A similar technology is Amazon Elastic MapReduce (EMR) service that exists for several years and provides elastic Hadoop cluster on Amazon cloud. In both technologies, the data is still transferred from the object storage to the compute nodes, unlike our Storlet Engine that performs the compute in the proxy and object nodes of the cloud object storage.

ZeroVM [14] (bought by RackSpace) provides a virtual server system (called a hypervisor), specifically for cloud use. It is an open source lightweight virtualization platform based on the Chromium Native Client (NaCl) project. Each request to a ZeroVM App (zapp) results in the spawning of an independent instance, isolated and secure, spinning up in under 5 milliseconds, and destroyed after finishing the request. ZeroVM allows to bring the hypervisor to your data, with a fully executable size under 75KB, allowing embedding directly within the storage system. ZeroVM has been integrated into OpenStack Swift in a project code named Zwift. It claims to be fast, secure and disposable but the use of NaCl requires customized versions of the GNU toolchain, specifically gcc and binutils. Consequently, existing code need to be recompiled for the NaCl toolchain which is sometimes non-trivial. In contrast, in our Storlet Engine recompilation is not required, and existing code can be executed unaltered.

Offloading data maintenance functions from the application to the storage system is an ongoing trend. For example, Muniswamy-Reddy et al. [15] introduce protocols for the cloud that collect and track the provenance of data objects. In another example, You et al. [16] present PRESIDIO, a scalable archival storage system that efficiently stores diverse data by providing a framework that detects similarity and selects from different space-reduction efficient storage methods. In both cases, some specific computations are performed in the storage system, but the mechanism for that is not generic as in the Storlet Engine that enables dynamically uploading and executing new functions wrapped in storlets.

The Storage Resource Broker (SRB) developed by the San Diego Supercomputing Center (SDSC) is a data grid technology middleware built on top of file systems, archives, real-time data sources, and storage systems. The SRB code was extended into a new system called iRODS [17], Intelligent Rule-Oriented Data management System. The iRODS was integrated with Amazon S3 to provide policy-based data management via flexible policies and a rule engine. iRODS creates a data grid with computational micro-services to manage the various policies in the grid. It includes a uniform name space for referencing the data, a catalog for managing information about the data, and mechanisms for interfacing with the preferred access method. iRODS and SRB run above the storage level, whereas our Storlet Engine resides within the object storage layer.

Docker [18] is an open source project to pack, ship and run any application in a lightweight container. The same container that a developer builds and tests on a laptop can run at scale, in production, on VMs, bare metal, OpenStack clusters, public clouds etc. Docker containers can encapsulate any payload, and will run consistently on and between virtually any server. Common use cases for Docker include: automating the packaging and deployment of applications, creation of lightweight, private PAAS environments, automated testing and continuous integration/deployment, deploying and scaling web apps, databases and backend services. The Docker mechanism can assist our Storlet Engine by providing quick and lightweight sandboxes.

3. STORLET ENGINE ARCHITECTURE

The Storlet Engine provides the cloud storage with a capability to use storlets that run in a sandbox close to the data. Figure 1 illustrates the Storlet Engine architecture that is generic with respect to the cloud object storage system. It includes two main services:

- **Extension Service** – connects to the object storage and evaluates whether a storlet should be triggered and in which storage node.
- **Execution Service** – deploys and executes storlets in a secure manner.

The extension service is tied to the storage system at intercept points and identifies requests for storlets by examining the information fields in the request and by evaluating predefined rules. The extension service then performs a RESTful call to one of the execution services that executes the storlet in a sandbox according to the client request and its credentials. The only component that will change when implementing the Storlet Engine in one cloud storage or another is the intercept handler of the extension service. The intercept handler needs to adapt the hooks into the object storage and may need for that either the source code or an open interface to internally connect to cloud storage. We prototyped the Storlet Engine for OpenStack Swift Object Storage.

The Storlet Engine supports multiple execution services and multiple programming languages for storlets. It supports execution of Java storlets in an execution service based on IBM Websphere Liberty Profile product that is a web container with small footprint and startup time. We also support native execution service for storlets written in Python or other languages that are wrapped in a Docker container [18], which is open

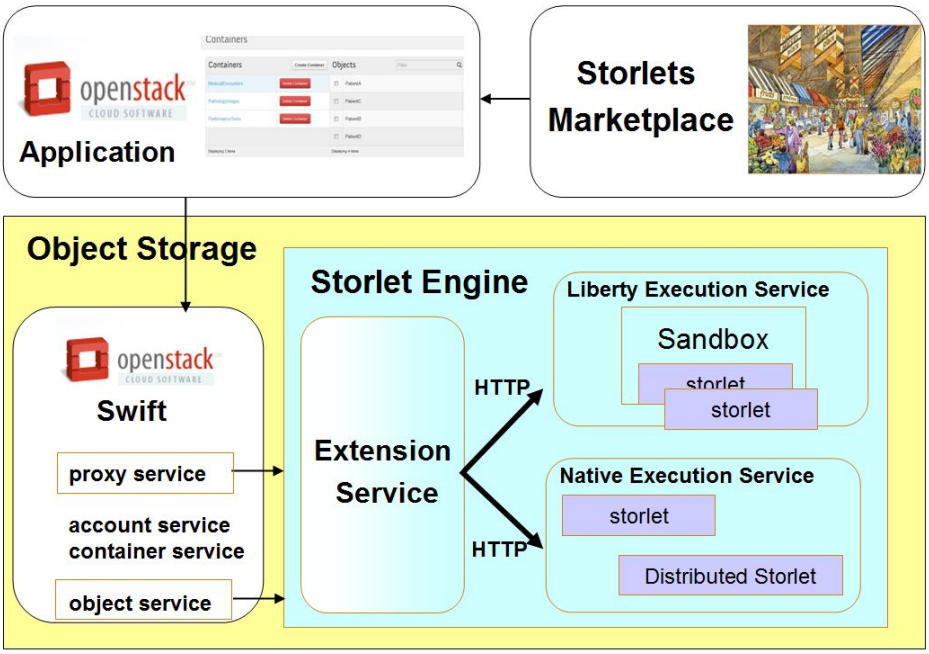

Figure 1: Storlet Engine Architecture

source software to pack, ship and run an application as a lightweight Linux container. A storlet may call other storlets; even if the other storlet is in a different execution service.

The Storlet Engine can reside in the storage interface node (e.g. proxy node in Swift), and in the storage local node (e.g. object node in Swift). The Swift proxy service is responsible for the external interface. It is generally more CPU and network intensive; thus, the proxy nodes that host those proxy services have generally powerful CPUs and large network bandwidth. The Swift object service holds the actual data objects. The object services are generally more disk and I/O intensive; thus, the storage nodes that host those object services have generally large disks and good I/O throughput.

We implemented the ability to run storlets either in the interface proxy servers or in the local object servers to take advantages of each node underutilized resources. The performance study in section 6 shows that it's preferable to run data-intensive storlets in the local object node. Yet, for storlets that are compute intensive or access several objects in different object servers, it may be preferable to execute them in the interface proxy node.

A Storlets Maketplace can be used as a repository of storlets from different vendors. An application on top of the storage can mashup and use different storlets from the marketplace for creating its functionality. For example, an HMO can create in the Marketplace a deidentification storlet that complies with HIPAA. A PACS provider can create in the Marketplace an imaging analytics storlet. Then, another application can use the two storlets from different vendors to analyze deidentified medical images and correlate them with deidentified medical records.

Storlets can be triggered either explicitly by adding specific information fields in the HTTP requests to the cloud storage. Alternatively, storlets can be triggered implicitly by rules as described in section 4.

The Storlet Engine supports two modes of operations. In the *synchronous mode* the storlet runs within the HTTP request/response that initiated its execution, namely the HTTP request ends after the storlet ends its execution. In the *asynchronous mode* the HTTP request that initiated the storlet execution ends as soon as the system registered the request in a queue. The storlet runs in the background and may accumulate results in an output object that also includes information about its completion status. Later on, the storlet initiator may access the output object and retrieve the results of the computation. Currently, we support the synchronous mode only.

3.1 Storlet Development

A storlet is a data object in the object store that includes executable code. The Java storlet code is based on the standard Java servlet technology; thus the storlet developer may use the extensive existing development tools for servlets. The storlet includes three aspects:

- **Lifecycle management** – includes operations such as storlet deployment, storlet configuration and initialization, processes and threads manage-

ment, storlet execution, inter-storlet communication. Lifecycle management is provided by the Storlet Engine.

- **Business logic** – the actual functionality of the storlet. This is provided by the client or application that creates the storlet.
- **Services invocation** – calls performed by the storlet to external functionality provided by the Storlet Engine, e.g. calls to access objects in cloud storage, create index, integrity check. Some of the services are basic primitives that the storlet cannot implement by itself while other services provide "ease-of-use" functionality that could be implemented by the storlet as well.

A storlet may originate from several sources: written by a system administrator; written by a user; or bought as part of a third-party package or downloaded from some site. Further, storlet execution can be initiated at different privilege levels: by an administrator; by a privileged user; or by a regular user.

The Storlet Engine must guard against malicious intruders abusing the storlets, and legitimate users overreaching their privileges in the storlets and accessing data that is not permitted to them. Based on the source of the storlet, the initiator, and the storlet functionality, a certain level of trust should be associated with the storlet. We provide a mechanism for sandboxing of storlets that is further described in section 5.

3.2 Services Storlets

The Storlet Engine provides special services storlets that can be used either by external clients or called by other running storlets. One such service storlet is the Distributed Storlet. The Distributed Storlet is a compound service storlet that executes multiple storlets in a pre-defined flow. It is a compound storlet in the sense that it calls other storlets and consumes their results. It is a service storlet as it is provided by the Storlet Engine itself, and not by external developers. This storlet is intended for analytics processes where distributed data-intensive processing on multiple objects is required.

The input parameters to the Distributed Storlet include the *resources* data objects to process, the *split* storlets to activate on the resources and the *merge* storlets that combine the results. The Distributed Storlet activates multiple split storlets in parallel on each data object at its residence. Then, it waits for all split storlets' results and calls the merge storlet (if exists) to summarize the results. The merge storlet may be absent if no merge of the splits results is needed e.g., when the split storlets just add or change content in the object store. In the end, the distributed storlet will send as output the result of the merge storlet. In case there is no merge storlet, the distributed storlet will return back a success/failure message to indicate the split storlets results.

Conceptually, the Distributed Storlet has similarity to MapReduce programs that transform lists of input data elements into lists of output data elements [19]. However, the distributed storlet is specialized for distributed processing within the cloud storage nodes, sparing the data transfer to the compute nodes.

Another service storlet is the Sequence Storlet which is a compound service storlet that executes multiple storlets one after the other. The output of one storlet is the input to the next storlet in the sequence. The output of the final storlet is returned to the user. The Sequence Storlet is provided by the Storlet Engine as all services storlets.

Those services storlets are written in Python and run in the native execution service within a WSGI server container.

4. RULES MECHANISM

The goal of the rules mechanism is to allow implicit storlet activation via predefined conditionals, thereby enabling automatic conditional activation of storlets. The storlets implicitly invoked by rules are in addition to the explicit storlet activation. Normally explicit storlet activation, if requested, takes priority and overrides implicit storlet activation. However, rules are also used to enforce access control related actions by marking the request with a specially marked flag. For such requests passing through the rules mechanism is mandatory. For example, mandatory de-identification storlet on sensitive data is enforced for requests having limited access credentials.

The rules mechanism is implemented via a rules handler that implements the rules logic (per stored rules), in combination with the user and system parameters. The rules handler is part of the extension service in each one of the Swift nodes that includes the Storlet Engine.

Rules consistency is an important issue. When using rules and rules logic, logical inconsistencies may be easily inadvertently introduced. For example, a user may define a (first) rule for a named container, stating that every pdf formatted data object is to be automatically transformed (via storlet) into a jpg formatted data object. Another (second) rule may specify, for the same named container, that the inverse format transformation should be applied. It is beneficial to have an automated consistency checker within the rules handler; however, in the current implementation we assume that the consistency of rules (and lack of conflicts or repetitions) is done by the rules author. Automatic rules consistency verification is left for future work.

4.1 Approach

We use a predefined rules set and logic to infer at

runtime which storlets should be implicitly invoked on which objects. Since in our approach explicit storlet activation overrides implicit activation, the REST request is first analyzed to determine if an explicit storlet activation is present, and only if it does not, the rules handler mechanism is called (with the exception of a restricted-access marked request, as mentioned above).

Since many tenants may be using the same storage system, the rules of one tenant should not affect other tenants operations. Hence, by design, the rules are kept as a per tenant editable object in the storage cloud. For a specific tenant, the rules object has specified access control to prevent unauthorized rules definition or modification. Typically, administrator access rights are required for rules editing, although these editing rights may be delegated to other users of that same tenant.

Typical inputs of the rules handler include request parameters, system metadata (e.g., content-type), user metadata (e.g., role, tenant), and the stored rules set. The output of the rules handler includes a full specification of the objects, parameters and REST request headers required for invoking a storlet. On the basis of this output a specific explicit REST request for storlet invocation is issued by the extension service, and handled by the existing explicit storlet dispatch mechanism.

Separate rules may be defined per HTTP command (PUT/GET/POST etc.), which provides greater flexibility for the rules handling mechanism.

The rules form an ordered list, in the sense that when the rules conditionals are tested for a match against request parameters, the first rule that matches the request inputs is activated. The following rules in the list are ignored (even if some would have been considered a match too). This design implements a prioritized list of rules, and the prioritization simplifies both rules definition and activation, as well as performance (following a match, the remaining rules need not be checked).

Examples of executing storlets on the basis of predefined rules and various input parameters include: a) deidentification storlet (depending on the role attribute of the user in the request), b) content transformation for the entire container (depending on the content type of the object provided in the request).

5. SECURITY MODEL

The Storlet Engine is an extension of the storage system. As such, it needs to adhere to the following security requirements.

- Storlets should execute in a protected environment where users (clients) do not over-reach their privileges. Specifically, access to data (e.g., Swift objects) from within a storlet should be allowed according to the user's permissions.
- In a multi-tenant storage environment, full isolation among tenants is a fundamental concept. This must be extended into the storlet's environment. Storlet requests coming from a given tenant should not be exposed to any aspect of requests coming from a different tenant (including data, state, etc.).
- Storlets originate from different sources. The code is not always fully verifiable for safety and consistency. The storlet engine must guard against malicious storlets, as well as intruding requests wanting to abuse the storlets.
- Multiple storlets serving multiple clients co-exist in the storage system. The storlet environment should support scalability. Storlets must not consume excessive resources (cpu, memory, etc.) that might degrade the performance of the system to an unacceptable level.

5.1 Sandboxes

When a storlet is deployed, it is placed in a "sandbox", that controls and limits the capabilities and actions of the code. A sandbox typically can block undesirable actions, such as direct access to cloud storage, or communication outside the local machine. It can also restrict access to the local filesystem and to local ports.

Currently two types of sandboxes are available, associating different levels of trust with different storlets:

- **Admin Sandbox:** the storlet is fully trusted, has no restrictions, and can perform all actions.
- **User Sandbox:** the storlet is less trusted, and therefore restricted from performing certain actions, such as writing to the filesystem except in designated work areas, reading from areas of the filesystem that are irrelevant to the storlet, making arbitrary network connections, or issuing sensitive system calls.

5.2 Implementing Multi-Tenancy and Sandboxes

We implement a lightweight sandbox mechanism, leveraging capabilities of the underlying operating system. It does not rely on unique properties of a given storlet execution environment, and can potentially be applied in multiple different storlet execution environments in the same system. The only assumption is that the operating system is of the Linux type.

In Liberty Profile, each web server executes in a separate single Linux process. Each running process has an effective UID (Linux user identifier) and GID (Linux group identifier), as well as a real UID and GID. Consequently, all the storlets deployed in a given web server have the same UID and GID.

In our implementation, a web server provides a single type of sandbox for all storlets that are deployed in that server. In addition, we restrict each server to be engaged on behalf of a single tenant. A given storlet can be deployed independently in multiple servers, thus separate instances of the same storlet can be made available in

each sandbox for each tenant.

Each tenant/sandbox pair is associated "permanently" with a unique UID and GID. In particular, different sandbox types of the same tenant have different UIDs/GIDs. All servers run as non-privileged (non-root) processes.

Each tenant/sandbox pair is also associated with a unique port number – on which the server listens for storlet requests.

A mapping from tenant/sandbox to the UID, GID and port number is made available to the relevant components of the Storlet Engine, such as the dispatch handler. The mapping can be pre-configured or dynamic, depending on the implementation. The UIDs/GIDs are leveraged to support filesystem permissions, firewall policies, and tenant isolation.

An effect of the unique UID/GID approach is that a well behaved server and its storlets can be protected from some types of intrusion. More importantly, a storlet in one server cannot cause damage to another server, since in Linux a non-privileged process cannot assume another UID/GID.

5.3 Access Control

Each storlet request is performed on behalf of a specific userid in the storage cloud. If the invocation is on the data path of the storage system (e.g. get or put of an object), the userid is the one issuing the original data request. If the request is not on the data path (e.g. an internal request, possibly event driven), an internal userid can be used.

Access control concerns authentication and authorization to perform certain actions. In the context of the Storlet Engine, there are two aspects of permission:

- *Permission to run the storlet* on a given request on behalf of a given userid. This can be approached in several ways. It may be user/role based. Data objects specified in the request may be associated with a set of rules determining what users and/or storlets are allowed to operate on the object. In fact, when the storlet itself is maintained as an object, the rules of the storlet object can be used.
- *Permission to access an object* in the storage system (get, put, etc.). A storlet should be permitted to access only the objects that the client originating the request is allowed to access. Access rights to objects are enforced by the security mechanisms of the storage system. On access to an object, a storlet needs to present some credentials, usually obtained as a parameter. For example, a token or userid/password in Swift.

Permission to run a storlet must be enforced before the storlet is invoked. Once the storlet starts handling a request, it is assumed that the storlet is authorized for the request. A typical storlet is performed on the data path of the client's original request to the storage system. The storlet is invoked after the storage system has already authenticated and authorized the client's request.

5.4 Scalability

Having multiple servers on each storage node can have significant impact on the performance. Conceptually, the number of servers would grow linearly with the number of tenants.

Some resources consumed by servers can be configured at the process level. However, the number of tenants is unbounded, and thus also the number of different servers. To support scalability, it becomes necessary to limit the maximum number of servers that are concurrently active at any point in time.

There are two approaches to this: Terminate running servers in favor of new servers, or keep a fixed pool of servers that can dynamically change their associated tenant/sandbox.

6. PERFORMANCE STUDY

6.1 Test Goals

The performance study attempts to answer the following questions:

1. *What performance benefits can be derived by wrapping a function as a storlet?* To answer this question we compare the performance of alternative storlet wrappings against their performance when equivalent functions are run outside of the storage system. The comparison are taken while a fixed workload is running concurrently in the background to simulate the environment in which storlets will run. We also examine the performance with no workload running in the background.
2. *How storlets affect system performance?* To answer this question we change our view point and examine the workload's performance when running concurrently with storlets or equivalent functions running outside of the storage system.
3. *What is the performance implications of running storlets on the storage system's interface nodes as oppose to running them on the local storage nodes?* The interface nodes are the public face of storage system and handle all incoming API requests. In Swift interface nodes are known as proxy servers. Local nodes are where the data resides. In Swift local nodes are known as object storage nodes. Our initial storlet implementation ran only on the interface node. We questioned whether it was necessary to distribute the storlets to run on the local nodes since it was thought that the primary performance benefit of storlets was derived from saving network bandwidth outside of the storage

system and that relative short distance between the local nodes and the interface node would not greatly affect performance. To answer this question we instrumented the storlet API to allow the user to specify whether a storlet is run on the interface node or local node.

4. *What host resources are most affected by storlet?* To answer this question we examine host memory utilization, load, cpu utilization and swap utilization.
5. *Do the performance issues (described above) change when the storage system is a private cloud accessed over a high speed internal network or a public cloud accessed over the WAN?* To answer this question we created a Swift test bed with a 10Gb network to simulate a private cloud and add delays to the incoming packets on the client host running the storlets or equivalent function to simulate WAN access.

The rest of the paper provides details about the test methodology, set up, and the test results.

6.2 Test Methodology

This section describes the testing methodology and terminology. *Treatments* are alternative storlet wrappings or equivalent functions. The study compares the performance of the different treatments under different conditions. The test conditions are described below:

- Vary the number of concurrent threads running the treatments
- Vary object size accessed by the treatments. Two classes of object size are used and listed below:
  1. *KB1-100* Objects with content length ranging from 1-100 Kilobytes are chosen randomly and are assigned to a thread running a treatment. The number of concurrent threads for *KB1-100* ranges from 0 to 200 threads.
  2. *MB5-100by5* Objects with content length ranging from 5-100 Megabytes in units of 5 Megabytes, i.e. 5MB,10MB,...100MB, are chosen randomly and assigned to a thread running a treatment. The number of concurrent threads for *MB5-100by5* ranges from 0 to 20 threads.
- Vary latency from host running the treatments to the storage system interface node. The native latency is less than 1ms and we emulate a WAN latency by adding 100ms delay to incoming packets using the netem Linux tool.

A fixed workload is run concurrently to each sampling of a treatment and the workload's performance is evaluated. The workload used is 100 concurrent threads of REST get requests for objects matching the content type 1MB5-100by5 described above.

Figure 2 illustrates the storlet performance test bed.

Treatments and workloads are run on separate hosts. The treatment engine is limited to a 1Gb network card while the workload engine drives its I/O through a 10Gb network card. Their I/O is directed to an interface node, which services the REST requests on its 10Gb network card. A test orchestrator running outside of the test bed uses Java RMI to orchestrate the test. It sends orders to the treatment and workload engines to launch threads of REST requests to the interface node. It also uses RMI to monitor the host resource utilization of the treatment engine and interface node.

Performance measurements are evaluated based on following dimensions: response time, throughput, bandwidth, failures, and resource usage. Resource usage includes memory, cpu, and swap usage.

6.3 Fixity Test Set Results Overview

This section compares the performance of alternative fixity storlet wrappings with its counterpart equivalent application. The fixity test set includes treatments that calculate a data object's digest, using MD5 and SHA256, and returns the results to the client. We chose fixity as the subject of the performance test since it allowed us to easily compare an eqivalent function running inside and outside the storage system while varying the object size and number of concurrent threads. The different treatments are described below:

- *fixityStorletAtInterfaceNode* : FixityStorlet run on interface node
- *fixityStorletAtLocalNode* : FixityStorlet run on local node
- *fixityAppWithStorletInfrastructure* : The fixity application runs on the treatment engine where it calculates the data object's digest. Even though the fixity application is not a storlet it is run while the storlet infrastructure is installed in the storage system. [1]
- *fixityAppWithOutStorletInfrastructure* : The fixity application runs on treatment engine, but the storage system is in its native form without the storlet infrastructure. This treatment is included so we can evaluate the additional overhead associated with the storlet infrastructure and compare it with the native storage system performance.

6.4 Fixity Test Results for Megabyte Size Objects

This section compares the performance results of the fixity test set working with megabyte size objects.

Figures 3,4 and table 1 illustrate the relationships between the different fixity treatments' response times. We observe that the fixityStorletAtLocalNode's response time is consistently better than the two fixityApp treatments' response time. While the fixityStorletAtInterfaceNode response time is the worst. When we compare fixityAppWithStorletInfrastructure against the fixityAppWithoutStorletInfrastructure treatment

[1]The fixity application is a Java class. This is an example of how the test tool can be easily extended by a user.

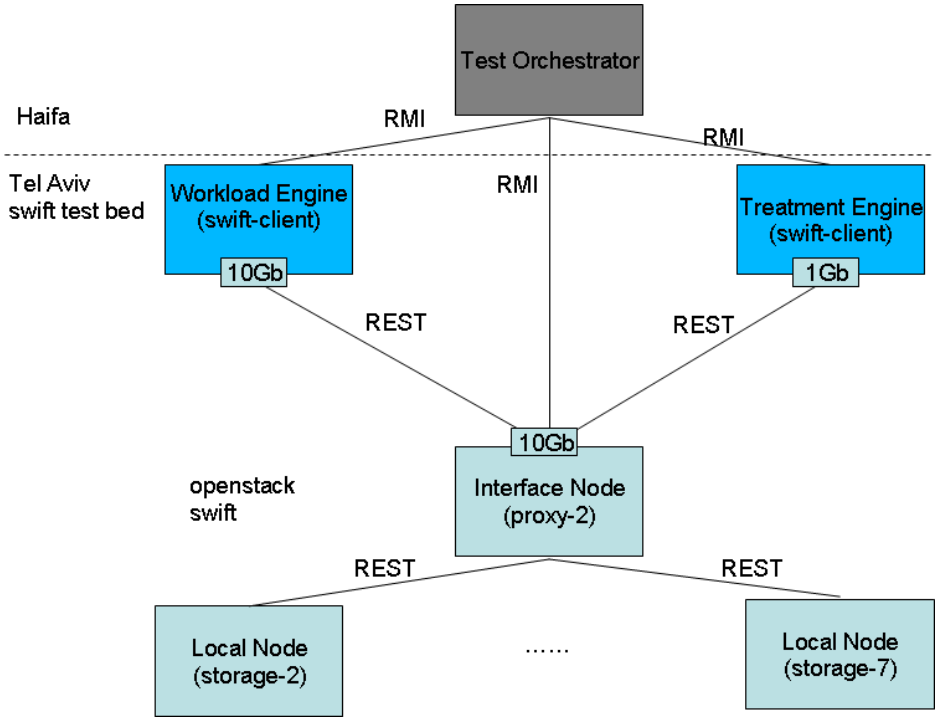

Figure 2: Storlet Performance Testbed

we also observe that there is overhead associated with the storlet infrastructure but that it is not significant.

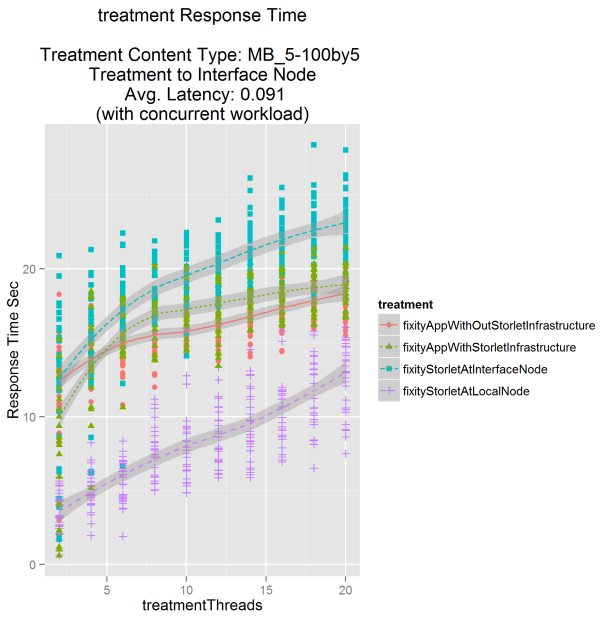

Figure 3: Fixity treatments for MB size objects response time with less than one millisecond latency between treatment host and interface node.

Figures 5,6 and table 2 shows that workload response time follows the same pattern; the workload's response time is best when it overlaps with the fixityStorletAtLocalNode treatment, followed by the fixityApp treatments, and worse when it overlaps with the FixityStorletAtLocalNode. However, the differences between the treatments is not as significant as what was observed when comparing the treatment response time. Since the workload is composed of megabyte get operations the figure also shows the degradation associated with

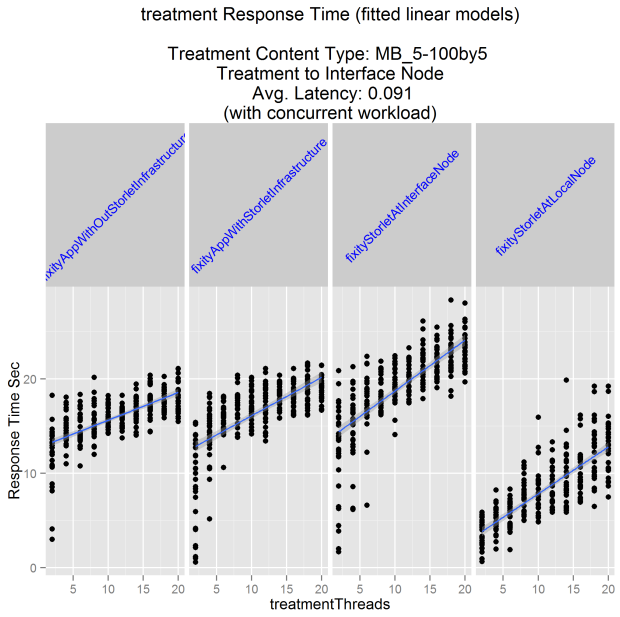

Figure 4: Fixity treatments for MB size objects response time linear models with less than one millisecond latency between treatment host and interface node.

	Intercept	Slope	R.squared	std-error
fixityApp WithOutStorletInfrastructure	13	0.29	0.479	0.0177
fixityStorlet AtInterfaceNode	13	0.54	0.557	0.029
fixityStorlet AtLocalNode	2.9	0.5	0.609	0.023
fixityApp WithStorletInfrastructure	12	0.41	0.443	0.026

Table 1: Fixity treatments for MB size objects response time linear model terms with less than one millisecond latency between treatment host and interface node.

the storlet infrastructure for get operations but it is not very significant. Notice that we captured the workload's response time when no treatment threads were concurrently running. In this case the average response time for the gets on MB object without the storlet infrastructure was 15.8 ms and with the storlet infrastructure 16.4 ms. The difference, 1.4 ms, means that there was a 3.8% degradation.

Figure 7 shows that the overall system throughput follows the same pattern, namely throughput is enhanced by leveraging storlets that run on the local nodes but worsens when the same storlet is run on the interface node.

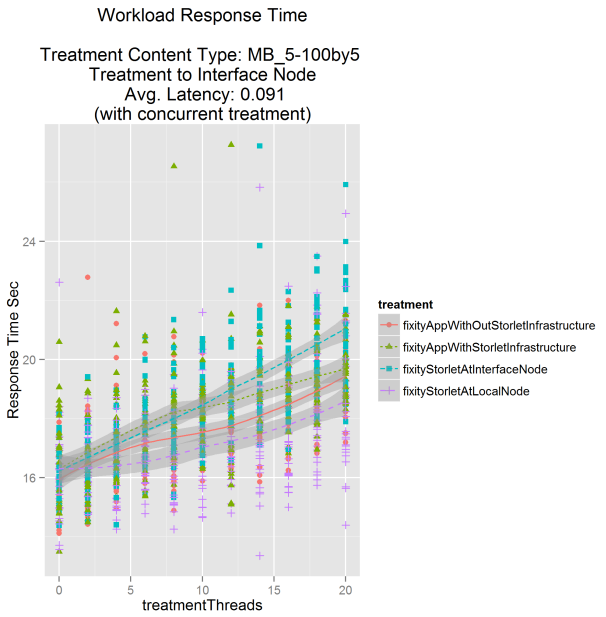

Figure 5: Workload response time for fixity treatments for MB size objects response time with less than one millisecond latency between treatment host and interface node.

	Intercept	Slope	R.squared	std-error
fixityAppWithOutStorletInfrastructure	16.1	0.156	0.397	0.0106
fixityStorletAtInterfaceNode	16.2	0.239	0.554	0.0118
fixityStorletAtLocalNode	16	0.115	0.164	0.0143
fixityAppWithStorletInfrastructure	16.6	0.162	0.328	0.0128

Table 2: Workload for MB size objects response time linear model terms for MB size objects with less than one millisecond latency between treatment host and interface node.

When we run the same test set but add 100 millisecond delay to incoming packets to treatment test engine and include the same megabyte workload but with no added delay, the initial response relationship is still 1) fixityStorletAtLocalNode, 2) fixityAppWithoutStorlet-

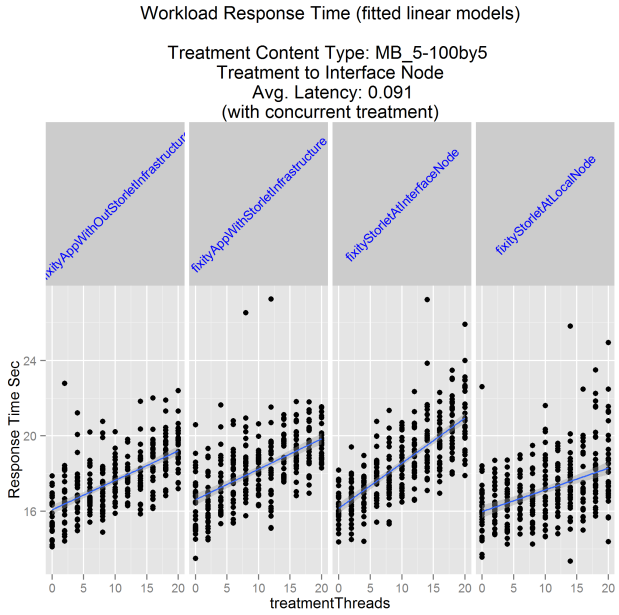

Figure 6: Workload response time linear models for fixity treatments for MB size objects response time with less than one millisecond latency between treatment host and interface node.

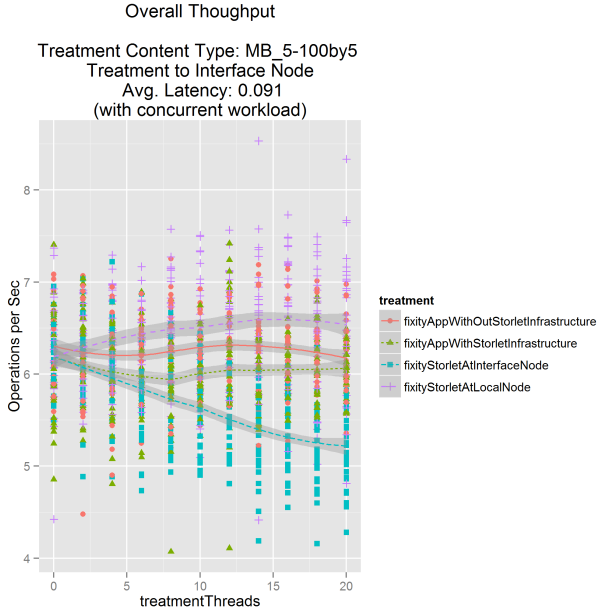

Figure 7: Overall system throughput for fixity treatments for MB size objects response time with less than one millisecond latency between treatment host and interface node.

Infrastructure, 3) fixityAppWithStorletInfrastructure, and 4) FixityStorletAtInterfaceNode, but as the number of threads per treatment increases FixityApps' response time increases at a faster rate than the FixityStorletAtInterfaceNode's response time until it passes the storlet thus rendering even FixityStorletAtInterfaceN-

ode treatment a more desirable alternative than calculating fixity outside of the storage system. We also observe as the latency between the treatment test engine to the interface increases that variability between fixityAppWithStorletInfrastructure samples increase, but storlet treatments remain more stable.

When we run the same test with no workload and with no additional delay then there is little difference between fixityStorletAtLocalNode and FixityStorletAtInterfaceNode.

6.5 Fixity Test Results for Kilobyte Size Objects

This section compares the performance results of the fixity test set working with kilobyte size objects.

The benefits of using storlets for kilobyte objects does not become apparent until delay is added to incoming packets to the treatment host. The response time for all four treatment are similar when no delays are added. However, when 100ms delay is added to the treatment host's interface to the interface node the fixityStorletWithStorletInfrastructure's response time is the worst. There is no significant difference between the two storlet treatments, presumably because the FixityStorletAtInterfaceNode does not require much memory to calculate the data objects digests. Like the tests results on megabyte size we also observe as the latency between the treatment test engine to the interface node increases the variability between fixityAppWithStorletInfrastructure samples increase, but storlet treatments remain more stable.

6.6 Fixity Test Conclusions

1. *What performance benefits can be derived by wrapping a function as a storlet?* The test demonstrates that storlets that run on the local node and process megabyte size objects perform better than the equivalent functions running outside of the storage system. However, storlets that run on the local node and process kilobyte size objects only perform better than equivalent functions running outside of the storage system when delay is added to incoming packets to the client host.
2. *How storlets affect system performance?* The test demonstrates workloads that overlap storlets that run on the local node and process megabyte size objects perform better than those that overlap equivalent functions running outside of the storage system. For kilobyte size objects there is no significant difference to the workload. The storlet infrastructure does degrade performance for non-storlet related operations but it is not very significant.
3. *What is the performance implications of running storlets on the storage system's interface nodes as oppose to running them on the their local storage nodes?* storlets that run on the interface node and process megabyte size data perform worse than storlets that run on the local node justifying adding the complexity to the storlet infrastructure to support the distribution of storlet processing onto the local nodes. Using the interface node to run storlets is only recommended when the storlet does not require much data handle as is the case of the DistributedStorlet(see Section 3).
4. *What host resources are most affected by storlet?* There is a correlation between memory utilization and load at the interface node and the resulting performance. However, it is not significant enough to explain the performance difference between running storlets on the local nodes or the interface node. Rather, bandwidth constraints between the local and interface node are a better explanation. This argument is further strengthened by the fact that when no workload is run in the background and there is no network bottleneck storlet performance on the local and interface nodes converge.
5. *Do the performance issues (described above) change when the storage system is a private cloud accessed over a high speed internal network or a public cloud accessed over the WAN?* The performance improvements attributed to storlets that run on the local nodes is amplified as the latency increases between the client host and storage system's interface node.

7. CONCLUSIONS AND FUTURE WORK

The paper presented Storlet Engine, an environment supporting computations within cloud storage. The idea is to extend the traditional role of object storage as a repository for data, by exploiting the computing resources of storage nodes to run computation modules – storlets – on the data close to where it resides.

We described the architecture and key features of the system. A rules mechanism allows implicit storlet activation via predefined conditions, thereby enabling automatic activation of storlets during data access. A security model supports multi-tenancy and various types of sandboxes to control the execution of storlets. We conducted a performance study, evaluating the Storlet Engine within an OpenStack Swift prototype.

There are multiple opportunities for further work and research. Our future plans include implementing additional execution environments in the Storlet Engine, offering diverse capabilities and computation models. Exploring the merit of a storlet marketplace, analogous to app marketplace, is a another direction, and in particular identifying major use cases and applications. Finally, for this evolving methodology, a standardization effort is a long term goal.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement № 270000 and under grant agreement № 600826.

REFERENCES

[1] M. Factor, D. Naor, S. Rabinovici-Cohen, L. Ramati, P. Reshef, J. Satran, and D. Giaretta. Preservation DataStores: Architecture for preservation aware storage. In *MSST 2007: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 3–15, San Diego, CA, September 2007.

[2] S. Rabinovici-Cohen, M. Factor, D. Naor, L. Ramati, P. Reshef, S. Ronen, J. Satran, and D. Giaretta. Preservation DataStores: New storage paradigm for preservation environments. *IBM Journal of Research and Development, Special Issue on Storage Technologies and Systems*, 52(4/5): 389–399, July/September 2008.

[3] M. Factor, E. Henis, D. Naor, S. Rabinovici-Cohen, P. Reshef, S. Ronen, G. Michetti, and M. Guercio. Authenticity and provenance in long term digital preservation: Modeling and implementation in preservation aware storage. In *TaPP 2009: Proceedings of the USENIX First Workshop on the Theory and Practice of Provenance (TaPP)*, San Francisco, USA, February 2009.

[4] E.K. Kolodner, S. Tal, D. Kyriazis, D. Naor, M. Allalouf, L. Bonelli, P. Brand, A. Eckert, Elmroth E, S.V. Gogouvitis, Harnik D, F. Hernandez, M.C. Jaeger, E.B.Lakew, J.M. Lopez, M. Lorenz, A. Messina, A. Shulman-Peleg, R. Talyansky, A. Voulodimos, and Y. Wolfsthal. A cloud environment for data-intensive storage services. In *CloudCom 2011: Proceedings of the IEEE Third International Conference on Cloud Computing Technology and Science*, pages 357–366, Athens, Greece, November 2011.

[5] S. Rabinovici-Cohen, J. Marberg, K. Nagin, and D. Pease. PDS Cloud: Long term digital preservation in the cloud. In *IC2E 2013: Proceedings of the IEEE International Conference on Cloud Engineering*, San Francisco, CA, March 2013.

[6] ENSURE: Enabling kNowledge Sustainability, Usability and Recovery for Economic value, EU FP7 project. URL http://ensure-fp7.eu.

[7] M. Braud, O. Edelstein, J. Rauch, S. Rabinovici-Cohen, J. Marberg, K. Nagin, D. Voets, I. Sanya, F. Randers, A. Droppert, and M. Klecha. Ensure: Long term digital preservation of health care, clinical trial and financial data. In *iPRES 2013: Proceedings of 10th International Conference on Preservation of Digital Objects*, Lisbon, Portugal, 2013.

[8] ForgetIT: Concise Preservation by combining Managed Forgetting and Contextualized Remembering, EU FP7 project. URL http://www.forgetit-project.eu/en/start/.

[9] N. Kanhabua, C. Niedere'e, and W. Siberski. Towards concise preservation by managed forgetting: Research issues and case study. In *iPRES 2013: Proceedings of 10th International Conference on Preservation of Digital Objects*, Lisbon, Portugal, 2013.

[10] OpenStack Swift Object Storage. URL https://wiki.openstack.org/wiki/Swift.

[11] S. Rabinovici-Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. *J. Comput. Syst. Sci.*, 47(1):2–44, 1993.

[12] 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 *OSDI 2006: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.

[13] OpenStack Savanna. URL https://wiki.openstack.org/wiki/Savanna.

[14] Zerovm. URL http://zerovm.org/.

[15] K. Muniswamy-Reddy, P. Macko, and M.I. Seltzer. Provenance for the cloud. In *FAST 2010: Proceedings of the 8th USENIX Conference on File and Storage Technologies*, pages 197–210, 2010.

[16] L. You, K.T. Pollack, D.D.E. Long, and K. Gopinath. Presidio: A framework for efficient archival data storage. *ACM Transactions on Storage*, 7(2), 2011.

[17] A. Rajasekar, R. Moore, C. Hou, C.A. Lee, R. Marciano, A. Torcy, M. Wan, W. Schroeder, S. Chen, L. Gilbert, P. Tooby, and B. Zhu. *iRODS Primer: Integrated Rule-Oriented Data System*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers, 2010.

[18] Docker. URL https://www.docker.io/.

[19] A. Rajaraman and J.D. Ullman. *Mining of Massive Datasets*. Lecture Notes for Stanford CS345A Web Mining. 2011.