uStorage - A Storage Architecture to Provide Block-Level Storage Through Object-Based Storage

Felipe Oliveira Gutierrez[1(✉)], Vinicius Cardoso Garcia[1], Jose Fernando S. Cardoso[1], Thiago Jamir[1], Josino R. Neto[1,2], Rodrigo Assad[3], and Marcos Barreto[4]

[1] Universidade Federal de Pernambuco (UFPE) - CIn, Recife, PE, Brazil
{fog,vcg,jfsc}@cin.ufpe.br, tjamir@gmail.com

[2] Instituto Federal de Pernambuco (IFPE), Palmares, PE, Brazil
josino.neto@palmares.ifpe.edu.br

[3] Universidade Federal Rural de Pernambuco (UFRPE), Recife, PE, Brazil
assad@deinfo.ufrpe.br

[4] Universidade Federal da Bahia (UFBA), Salvador, BA, Brazil
marcosb@ufba.br

Abstract. Block-level Storage is widely used to support heavy workloads. It can be directly accessed by the operating system, but it faces some durability issues, hardware limitations and performance degradation in geographically distributed systems. Object-based Storage Device (OSD) is a data storage concept widely used to support *write-once-read-many* (WORM) systems. Because OSD contains data, metadata and an unique identifier, it becomes very powerful and customizable. OSDs are ideal for solving the increasing problems of data growth and resilience requirements while mitigating costs. This paper describes a scalable storage architecture that uses *OSD* from a distributed P2P Cloud Storage system and delivers a Block-level Storage layer to the user. This architecture combines the advantages of the replication, reliability, and scalability of a OSD on commodity hardware with the simplicity of raw block for data-intensive workload. We retrieve data from the OSD in a set of blocks called *buckets*, allowing read-ahead operations to improve the performance of the raw block layer. Through this architecture we show the possibility of using OSD on the back end and deliver a storage layer based on raw blocks with better performance to the end user. We evaluated the proposed architecture based on the cache behavior to understand non-functional properties. Experiments were performed with different cache sizes. High throughput performance was measured for heavy workloads at the two storage layers.

Keywords: Software architecture · Cloud computing · Storage as a service · Object-based Storage · Block-level Storage

© IFIP International Federation for Information Processing 2017
Published by Springer International Publishing AG 2017. All Rights Reserved
F. De Paoli et al. (Eds.): ESOCC 2017, LNCS 10465, pp. 213–228, 2017.
DOI: 10.1007/978-3-319-67262-5_16

1 Introduction

The concept for Block-level Storage is almost universally employed for all types of storage [10,17]. It splits files into evenly-sized blocks of data, each with its own address, but with no additional information (metadata) to provide more context about the block. This data storage concept can be directly accessed by the operating systems as a mounted drive volume and it delivers a significant performance improvement compared to others data storage concepts. Beyond a hundred of terabytes, however, it may run into durability issues, hardware limitations, or management overhead. Moreover, performance degrades on geographically distributed systems.

Object-based Storage (OSD) emerged after promising to be more powerful and customizable [6,13]. This data storage concept does not split files into raw blocks of data, but into entire clumps of data stored in objects that contain data, metadata, and an unique identifier. There is no limit on the type or amount of metadata, so objects can include anything (e.g., security classification). OSD is mainly used to solve problems related to data growth due to its scalability properties. Nevertheless, OSD typically does not offer good throughput for the end user.

In this work we attempt to combine the benefits of both data storage concepts. We propose a storage architecture that uses Block-level Storage on its first layer and OSD on its second layer. We created a set of blocks on the first layer and called them buckets. The buckets have metadata that are used to upload and retrieve them from the OSD to a very small and fast device for the Block-level Storage (cache). This storage architecture we call *uStorage*. The OSD layer we call *CSP* (Cloud Storage Platform) and it consists of a low-cost distribute system to store files using P2P networks.

There are several other systems that deliver a Block-level Storage to the end user, but store data in a different architectures. *Ceph* uses POSIX [24], *Panasas* uses a cluster with RAID [14], *IBM Storwize* uses a cluster [9] and *Nexenta* uses a Cloud Storage Service based on Objects [1]. It is very acceptable to use Block-level Storage on the user interface and couple with another storage architecture, as long the back-end Storage architecture addresses non-functional requirements to the Block-level Storage (e.g., reliability, responsiveness, availability). However, it is also crucial to have a good strategy on the first storage layer to handle heavy workloads.

This paper presents *uStorage*, an architecture that can handle heavy workloads on the Block-level Storage and also be coupled to an OSD. We evaluated the architecture based on the cache non-functional properties, using the same methodology adopted by others [3,7]. The buckets that are not used on the cache can be removed, and those that are requested have to be restored from the OSD. The challenge in this architecture is to optimize the bucket and the cache size to avoid performance degradation. This paper presents the uStorage architecture on Sect. 2 and its implementation details on Sect. 3. The evaluation is presented on Sect. 4 followed by some related works on Sect. 5. The final Sect. 6 describes the main conclusions and proposed future work.

2 uStorage Architecture

The *uStorage* architecture delivers raw block interface to the users through the iSCSI components presented on the Fig. 1. All the components on the *CSP* [4,5] module are used as a OSD storage. The *CSP* is a module of the *uStorage* that is responsible for saving all buckets on the *DataPeers* with its metadata on the *Metadata Storage*. The *CSP* also provides high availability for the buckets because there are at least two copies of them in different *DataPeers*. Each *DataPeer* is a commodity hardware where the buckets are stored. The horizontal scalability property is achieved by adding more *DataPeers* on the *Data Storage*.

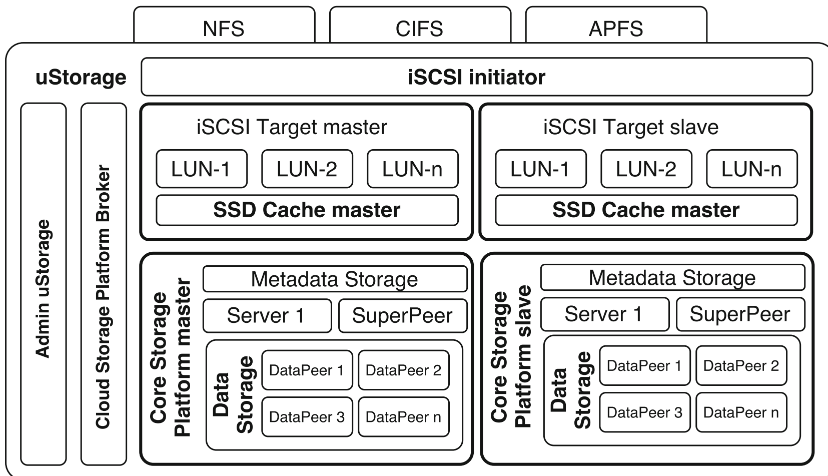

Fig. 1. *uStorage* architecture components: *CSP* (Cloud Storage Platform) contains a *Metadata Storage*, *Server* and *SuperPeer* to manage all buckets on the *DataPeers*. The iSCSI Target cache is located in a SSD drive. The admin layer uses the broker layer to manage the whole *uStorage* platform.

Block-level Storage devices are usually designed to be the first storage layer because it can be directly accessed by the operating system as a mounted drive volume. Meanwhile, OSD cannot do so without significant performance degradation. The OSD storage does not split files up into raw blocks of data. Instead, entire clumps of data are stored in an object that contains the data, metadata, and an unique identifier. There is no limit on the type or amount of metadata, which makes OSD powerful and customizable. In the enterprise data center, OSD is used for these same types of storage needs, where the data needs to be highly available and durable [4,5]. This type of system has certain characteristics that would be impossible to achieve if only one of these types of storage architectures were used. *uStorage* achieves through this architecture the following properties:

– *unlimited* storage space as much the *CSP* module can offer. This is achieved through the *DataPeers* that can be added dynamically.
– It does not use the user's computer as a cache stage or replication of files as *CSP* uses, but still guarantee the scalability and reliability of the *CSP*.
– It reaches an acceptable performance compared to other storage system.
– It provides all these features using a coupled architecture of raw blocks and OSD.

2.1 uStorage Components

uStorage delivers a virtual disk with large capacity using block-level virtualization. This virtual disk delivered to the user is on the master mode. The slave mode achieves the reliability feature of this storage architecture. The architecture uses an iSCSI Target component as cache with the *CSP* as back end to store objects. The users connect to the *uStorage* through the iSCSI initiator or by any interfaces connected to it (e.g., NFS, CIFS, APFS). The original iSCSI systems have the data centralized in a server called Target [18]. However, the iSCSI Target at the *uStorage* is a cache with high performance and small storage capacity. The data arrives in the iSCSI Target cache and as soon the metadata has been created, the data is flushed to the *DataPeers* on the *CSP*, according to the iSCSI Target cache algorithm.

The iSCSI protocol is used between the two iSCSI components and to connect the iSCSI initiator with the user. This protocol transports SCSI messages over TCP/IP [23]. Other SCSI protocols include SCSI Serial [22] and *FCP* (Fibre Channel Protocol) [15]. A major advantage of iSCSI over *FCP* is that it can run over standard *off-the-shelf* network components, such as Ethernet. Moreover, iSCSI can exploit existing IP-based protocols such as *IPSec* for security and *SLP* (Service Location Protocol) for discovery.

The files copied to the iSCSI initiator are sharded in pieces in the iSCSI Target cache (buckets). The bucket size is 2 MB and it has a set of raw blocks. Once the buckets are request by the iSCSI initiator, the iSCSI Target cache has the role of restore them from the *CSP* if they are not in the cache. If the buckets are less used by the iSCSI initiator, the iSCSI Target cache keep them only in the *CSP*. Only the frequently used buckets remain in the cache. The others will stay in the *CSP* to be requested on demand. These buckets are replicated in different *DataPeers* with the minimum quorum of two. Although the *uStorage* architecture does not use RAID as other storages architectures to replicate data [1,14], the *DataPeers* contain replicated buckets.

The cache on the iSCSI Target is a possible spot of failure that has to be always available to the iSCSI initiator. *uStorage* provides a master and slave architecture for the iSCSI Target cache on a second server. The slave server has an iSCSI Target configured with the same IP address of the master server. Its network interface is enabled only when the master goes down. All the buckets are replicated from the master *CSP* to the slave *CSP* through a process running on the *Server*. All the buckets on the master are independent from the slave server. The buckets frequently used on the master iSCSI Target cache are also updated on the slave iSCSI Target cache.

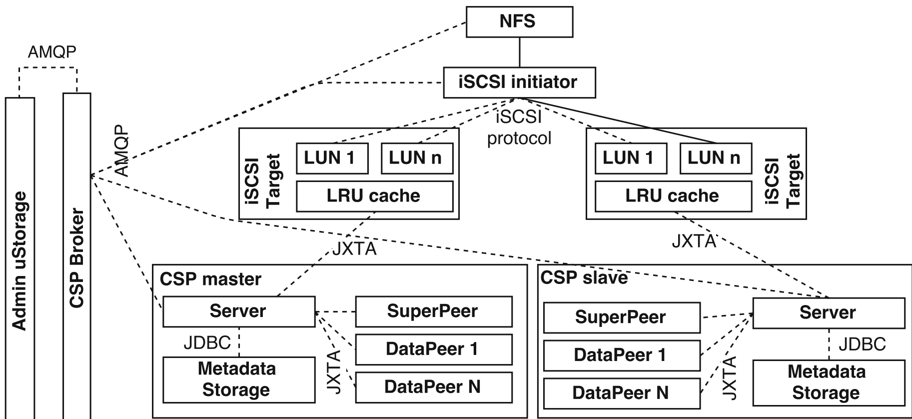

Fig. 2. Physical or deployment architectural view of the *uStorage* according to *architectural model 4+1* [11]. The users can mount the NFS drive or direct connect to the iSCSI initiator. The iSCSI Target manages all buckets on the cache and it uses an improved JXTA protocol to transfer them to the *CSP*. The user can mount the iSCSI drive and manage to add or remove *DataPeers* on the Admin through *AMQP* messages.

The iSCSI Target capacity is limited by the filesystem (i.e., ext4, xfs, NTFS) because it is virtualized. The bucket size (2 MB) allows read ahead on the files and the iSCSI Target cache spares restore operations on the *CSP*. The size of the buckets is a trade-off approach because, if they are too small it is necessary to do a lot of restore operations. Nevertheless, if they are too big the write operation lacks performance on the iSCSI protocol. Figure 2 presents the physical architecture model of the iSCSI Target connected to the *CSP*. The iSCSI Target can contain multiple instances of *LUNs* (Logic Unit Number). These instances can share the same cache space. It is better to have several iSCSI virtual disks of 8 TB capacity connected to the NFS instead of having just one very large iSCSI virtual disk of 128 TB connected. Because the iSCSI Target virtualise its size based on the *CSP* module, it can deliver a very large disk to the iSCSI initiator.

2.2 CSP (Cloud Storage Platform)

The distribute storage module *CSP* (Cloud Storage Platform) [4,5] consists of a low-cost architecture to store files as objects using P2P networks. The files are split and stored in pieces of predefined size (buckets) and recorded on several computers (network nodes) connected to the P2P network. The bucket replication is done by a running algorithm on the *Server* component. The *CSP* module can be used to store large amounts of data (greater than 1 PBytes). Figure 2 presents the communication among all components.

Superpeer is responsible to manage the *Server* and the *DataPeers* connected on the *CSP* through a P2P network. It works as a proxy that has a list of services, managing their availability. The *Server* manages the services used

on the P2P network: authentication, bucket management, peers availability, *DataPeer* and bucket lookup. *Metadata Storage* stores the buckets metadata, that we use a relational database. The *Data Storage* is composed by *DataPeers* to store buckets. Buckets are files and its name start with letters that correspond to the *DataPeer* and directory that they located. The algorithm that chooses the location to store each bucket is on the *Server* component. The communication among the *CSP* components is done by a improved *JXTA* protocol, an open source P2P protocol specification created by Sun Microsystems [8].

CSP enables *unlimited* storage space reachable through large levels of horizontal scalability by simply adding new *DataPeers*. The availability feature is done by ensuring the level of bucket replication greater than one. The buckets are built on the iSCSI Target as a set of raw blocks. The *CSP* can sharded them, however this operation is not efficient when they need to be restored. The bucket represents the OSD objects because it is a file with a set of attributes that define various aspects of itself on the metadata (i.e., size, host and filesystem locality, deduplication version). This simplifies the task of the storage architecture and increases its flexibility by distributing the management of the data with the data itself.

After saving the buckets on the *CSP*, they can be removed from the iSCSI Target Cache when they are not being frequently accessed, according to the cache algorithm. The access control of these buckets is done by a read and write cache algorithm, which may have its size configured on the iSCSI Target. The cache size is usually 10 GB less the size of the SSD drive, where the iSCSI Target cache is configured (safety approach in case of the overloaded cache). The cache accelerates the read and write access to the iSCSI Target like some RAID controllers use cache with the same propose.

2.3 Providing Block-Level Storage Through Object-Based Storage

The *uStorage* architecture uses raw block (iSCSI) on the first layer and OSD on the second layer (*CSP*). The capacity of the iSCSI Target cache is virtualized to the OSD layer, that has much more scalability provision through the *DataPeers*. This strategy makes possible to have more space on the iSCSI initiator virtual disk than it is available on the iSCSI Target. Through the raw blocks on the first layer, the I/O operations are faster than OSD [6,13]. Moreover, to achieve scalability the OSD architecture is more recommended [12]. As a result, we deliver a fast interface to the user through the iSCSI initiator and scale horizontally the OSD.

The iSCSI Target cache is the main component that improves the performance of the whole storage architecture. It is possible to create different *LUN's* on this component and connect them on the same iSCSI initiator. So we can have different users sharing the same iSCSI Target cache. The *Server* is responsible to replicate all the buckets on the iSCSI Target master to the slave. All the buckets state are saved on the *Metadata Storage*, so it is possible to know which buckets were on the iSCSI Target cache master fail over.

3 uStorage Architecture Implementation

Given the *uStorage* architecture described before, the implementation of the iSCSI Target cache has the following requirements:

R.1 which clean policy must be used on the iSCSI Target cache?
R.2 when the clean policy must be executed?
R.3 how many buckets must be removed on the policy execution?
R.4 which buckets must be removed from the iSCSI Target cache?

Most of the cache policies use *LRU* (Least Recently Used) algorithm [3,7]. Moreover, it makes more sense to leave the frequently used buckets on the cache and remove the buckets not used (R.1). The policy must be executed on the same pace that the buckets are accessed in a real scenario. So, if any file (on the raw block layer) is accessed, the policy will execute and the cache is going to have the same buckets that it had before (R.2). For this calibration we set the policy execution schedule between five and seven seconds, based on the evaluation demonstrated at Sect. 4.

It is necessary to make room for the new buckets by removing an amount of buckets that can be written between each policy execution or when the cache is almost full (90%). When the stored buckets amount exceeds 90% of the cache size, the buckets less accessed and already recorded on the *CSP* can be removed from the iSCSI Target cache. When these buckets are required to read or write, they are restored from the *CSP*. In another view, 10% of the cache free space must be enough to have a high write throughput to store new buckets and remove the ones already stored on the *CSP* (R.3), as we demonstrated on Sect. 4.

The *Least Recently Used* on the iSCSI Target cache queue must be removed. This module also knows which buckets have higher probability to contain *inodes*[1]. The metadata of the bucket has two status: *INDEX* or *DATA*. If the status is set to *INDEX*, even if it is not recently used, the bucket will not be removed. The *INDEX* buckets are set during the disk format. All other buckets after the disk formatting are set as *DATA* status. They are eligible to be removed by the policy even if they have *inodes* (R.4), because they are not crucial to open the disk and then can be found through the *inodes chain* [16].

3.1 iSCSI Target Cache Algorithm

Figure 3 shows the sequence diagram of the iSCSI Target cache algorithm, with write and read operations, according to the *architectural model 4+1* [11]. The process can be split into six components. The iSCSI components are the initiator and the Target cache modules of the *uStorage*. It receives *read* and *write* operations from the initiator and the communication is through the iSCSI protocol [18].

[1] An *inode* is the primary structure used in many UNIX file systems. It contains file attributes such as access time, size, and group and user information.

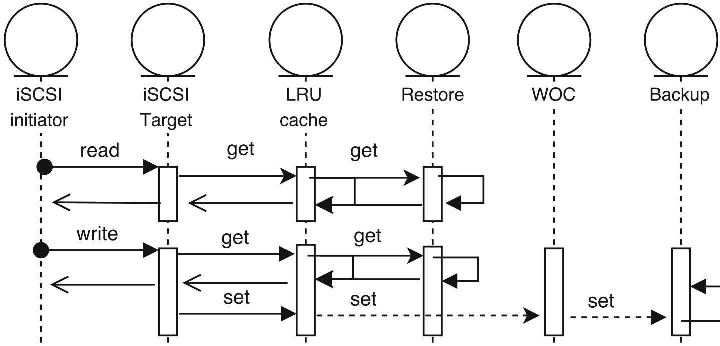

Fig. 3. Sequence diagram of the *uStorage* architecture according to *architectural model 4+1* [11]. The iSCSI's components read and write all buckets from the *LRU cache*. The *get* messages are synchronous and do not use the *WOC* component. When a bucket has to be send to the *Backup* component it is an asynchronous message and it uses the *WOC* component to make sure it is not being edited while it is being send to the *CSP*.

The *LRU cache* component is the *Least Recently Used* algorithm implemented to increase the performance on the iSCSI Target cache. The cache is allocated on a SSD hard drive with 100 GB size, which I/O responses are 250 MB/s, while the HDD are 70 MB/s. If the iSCSI Target receives a *read* operation and the bucket is in the cache, it is not necessary to restore it from the *CSP*, otherwise the process goes to the *Restore* component in a synchronous way to get the bucket. The *write* operation get the bucket from the *LRU Cache* if it is present, otherwise get it from *CSP* through the *Restore* component in a synchronous process. While this operation is writing on the *LRU Cache*, it also set a time on the *WOC* component (*policy write-on-close*) to save this bucket on the *CSP* in an asynchronous process. We decided to allocate the *Metadata Storage* component on the SSD drive to improve the cache performance.

The *WOC* component guarantees the backup process (saving buckets into the *CSP*) more efficient in an asynchronous process. This is based on the *policy write-on-close* for filesystem [20]. This strategy ensures that the bucket will only be sent to the *CSP* when it is no longer receiving bytes and also after 10 s without write access (also configurable). If the bucket contains several *inodes*, it will be changed a lot in less then 10 s and the cache algorithm will not send it to the *CSP*. If the bucket contains only data and no *inodes*, it will be written and not accessed afterwards. After 10 s with out receiving *write* I/O operations it will be sent to the *CSP* in an asynchronous process.

3.2 FSM (Finite-State Machine) for Bucket Management

Figure 4 depicts the usage control flow of buckets within the uStorage architecture. These states are persisted on the *Metadata Storage* component, as was presented at the Fig. 2. A given bucket can be in one of four states:

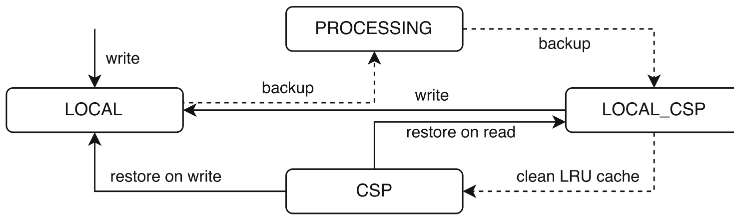

Fig. 4. Buckets metadata Finite-State Machine applied to bucket management within the *uStorage* architecture. The dotted line are asynchronous operations and the continuous lines are synchronous operations. The buckets can have four states (LOCAL, LOCAL_CSP, CSP, PROCESSING).

S.1 *LOCAL*: the bucket is only on the iSCSI Target cache and *CSP* has an old version.
S.2 *LOCAL_CSP*: the bucket has the same version on the iSCSI Target cache and *CSP*.
S.3 *CPS*: the bucket is only on *CSP* and not on the iSCSI Target cache.
S.4 *PROCESS*: the bucket is on backup process to the *CSP* and it is locked to edition.

Five operations can be performed over the buckets:

O.1 *write*: this operation always modify the buckets state to *LOCAL* and the bucket version on *CSP* became old.
O.2 *restore on write*: this operation modifies the bucket state to *LOCAL*, since it is also a *write* operation. It happens when the bucket is only on *CSP*. The bucket needs to be restored before its content is modified. If the restore operation fail the file can be corrupted.
O.3 *restore on read*: this operation doesn't modify the bucket content, however, it happens when the bucket is only on the *CSP*. The bucket state is modified to *LOCAL_CSP*.
O.4 *backup*: this operation is asynchronous (dotted line), so it does not decrease the *uStorage* architecture performance. It happens in two steps. First the buckets are changed to *PROCESS* state and when the backup is successful, its state is changed to *LOCAL_CSP*. These two steps ensure the same version on the iSCSI Target cache and *CSP*.
O.5 *clean LRU cache*: this operation is also asynchronous and it is executed by the *LRU* algorithm. The less accessed buckets and the buckets with the *LOCAL_CSP* state can be deleted from iSCSI Target cache and their state are changed to *CSP*.

3.3 Architecture Calibrations

Since the *uStorage* writing operation should be performed immediately when it is starting, a batch was created to periodically reserve space on the cache. The

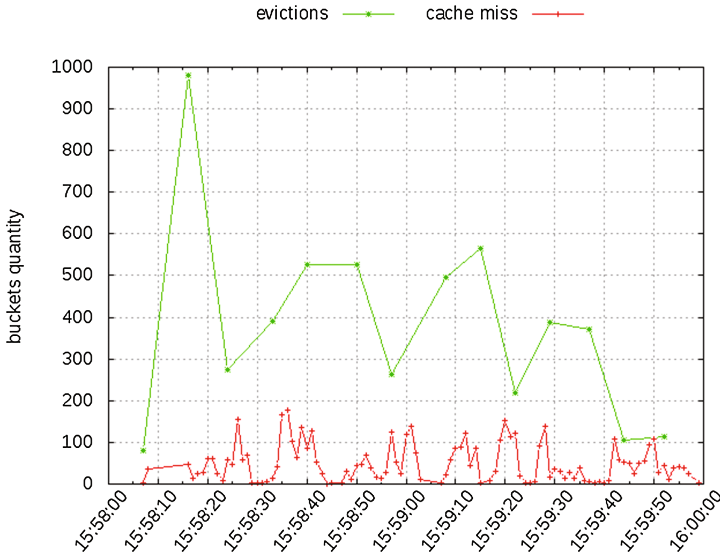

Fig. 5. *uStorage* iSCSI Target cache configured to 20 GB and set of files with 20 GB in total. The cache has the same size of the set of files we are storing. All *eviction* (bucket remove) operations are mostly due to create *inodes*. We have few *Cache Miss* (restore) operations and the *eviction* operations don't need to work often.

cache algorithm will not remove buckets until it reaches 90% of its usage, so we guarantee that 90 GB of buckets are in use. We judge to always keep 10% of the cache space free to receive new buckets and if the *CSP* is off. This space will always be available if there is any recording buckets on the iSCSI Target [19,20]. It is hard recommended to always have a reserve free space on the iSCSI Target cache to write buckets and avoid performance degradation. This space was estimated according to calibrations and tests with the system explained on the Figs. 5 and 6.

The restore operation had to be implemented to lock the bucket for any possible write. Otherwise it is a big chance to lose the file we are accessing or even lose the whole filesystem. This situation was avoid by making sure that there is always one thread restoring the bucket requested by the iSCSI Target cache. After the restore operation is completed, read and write threads operations can be done concurrently, because the came from the iSCSI protocol. In our scenarios, the restore operations took less than 0.5 s. Considering we are using a 2 MB bucket size, the read-ahead operation gains performance just in this restore process.

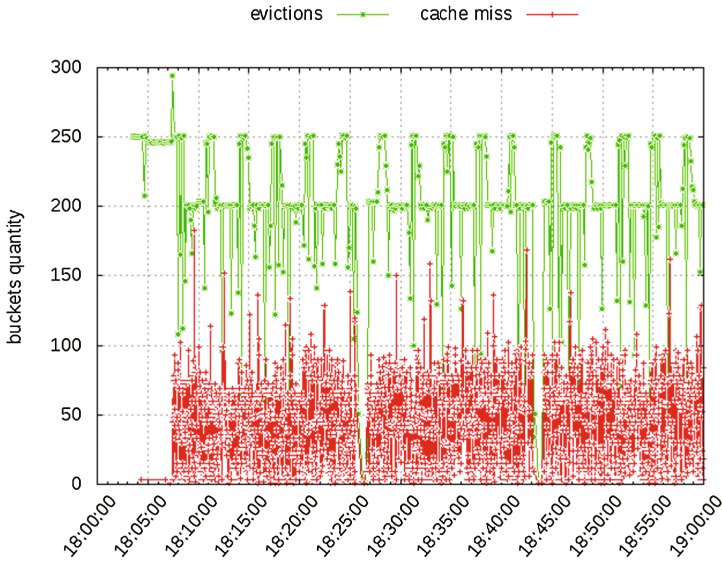

Fig. 6. *uStorage* iSCSI Target cache configured to 5 GB and set of files with 20 GB in total. The cache is four times fewer than the files we are storing. The algorithm is always making space on the cache with the *eviction* (bucket remove) operation and all the *Cache Miss* (restore) operations never overlaps the quantity of *evictions*.

4 uStorage Architecture Evaluation

We used three methodologies to evaluate the *uStorage* architecture. The purpose of each methodology was to simulate the max performance that the iSCSI Target cache can deliver to the user. The first methodology we evaluate the cache using the same concept used by *Amazon ElastiCache* to measure and take the best performance of the iSCSI Target cache [3]. The main factor of an effective cache strategy is to enable systems to have good scalability. *Amazon ElastiCache* has two major operations that are available on its cache. The *eviction* operation has the same semantics of the bucket remove operation of the iSCSI Target cache. Its objective is make place for new buckets that are arriving in the cache. The *Cache Miss* operation has the same semantic of the bucket restore operation of the iSCSI Target cache. When the bucket is not found in the iSCSI Target cache it is necessary to restore it.

A large number of *eviction* operations (bucket remove) can be a sign that the space on the cache is overloaded. If a *Cache Miss* (restore) operation is stable there is nothing to worry. However, if the combination of a large number of *Cache Miss* with a large number of *eviction* operation is happening, it is a sign that the cache is failing due a lack of memory. Figure 5 presents a 20 GB cache size and how many buckets have been processed by the *eviction* (remove) operation and by the *Cache Miss* (restore) operation. Figure 6 presents the same metrics but with a 5 GB cache. Both evaluation we used a file of 20 GB size. Through

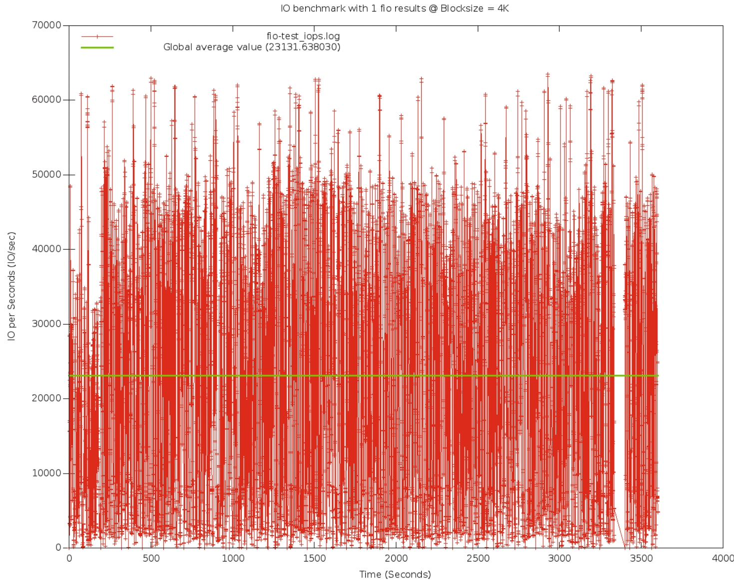

Fig. 7. Benchmark FIO of *uStorage* with 20 GB file and cache of 5 GB. Avg: 23131 IOPs.

these two metrics we can see the good behavior of the cache because the number of *Cache Miss* never exceed the number of the *eviction*. Even in a very small cache of 5 GB size and with files four times its size (20 GB size) the *eviction* and *Cache Miss* operations are more concentrated.

The second evaluation we used *FIO* benchmark [2] to test how many IOPs the iSCSI Target cache could reach. *FIO* is a popular tool to measure IOPs on Linux storage servers. We configured the cache for 5 GB size and the *FIO* benchmark to work with a set of 20 GB files. Through this configuration we could achieve 75% of *Cache Miss* (restore) and *eviction* (bucket remove) operations on the cache. Figure 7 shows that the *uStorage* reaches 63 K IOPs on its peak and an average of 23 K IOPs.

The third evaluation on the iSCSI Target cache we used the methodology proposed by [21]. The *uStorage* was set to work with different number of *LUNs* in each measure and different size of buckets as well. We variated the bucket size from 1 KB to 1 MB and took the average of the results. Then we took 10 metrics for each *LUN* quantity that we configured for the same iSCSI Target cache (1 until 4). Table 1 shows the results for writes on the same iSCSI Target cache using the *dd* command pointed to */dev/random* with a 10 GB file. The write process was in parallel as we add more *LUNs*, and we calculated the average of all results. *uStorage* could reach more IOPs comparing to a *SATA* disk, also when we add more *LUNs* on its cache to do parallel writing. The number of the threads on the *uStorage* didn't grow on the same pace of the number of *LUNs*. The same behavior can be said for the memory and cpu usage.

Table 1. Parallel write comparison on only one *SATA* hard disk and different number of *LUNs* on the same iSCSI Target cache.

SATA	IOPs	*LUNs*	Target cache	IOPs	Threads	Mem GB	CPU%
10 m 30 s	62.6	**1**	11 m 21 s	168.5	642	2.3	26.1
23 m 30 s	59.1	**2**	17 m 26 s	207.2	881	3.4	46.5
39 m 46 s	54.2	**3**	33 m 45 s	203.6	1127	3.6	59.4
58 m 10 s	81.3	**4**	41 m 35 s	217.2	1254	4.5	69.4

5 Related Work

This section describes some related works to the *uStorage* architecture, which therefore involve cloud storage and the closest possible to the *SAN* architectures.

NexentaStor [1] is a storage system with access levels to I/Os for files (NFS) and blocks (iSCSI). Its architecture provides techniques for spreading I/O workload over multiple domains (local and remote), while at the same time increasing operational mobility and data redundancy. File and block level I/O access are addressed. *NexentaStor* uses metadata for blocks, the domains are physical, and it has characteristics of horizontal scaling. It has a method for resolving a single server bottleneck performing one or more of the following operations: splitting a filesystem into two or more parts; extending a filesystem residing on a given storage server with its new filesystem part in a certain specified I/O domain; migrating or replicating one or more of those parts into separate I/O domains; merging some or all of the filesystem parts to create a single combined filesystem, and then redirecting the filesystem clients to use the resulting filesystem spanning multiple I/O domains. *NexentaStor* uses File-based Storage and Block-level Storage while uStorage architecture uses also OSD to reach scalability on the second storage layer.

IBM Storwize [9] has an architecture to create, read and write compressed data for utilization with a block mode access storage. The compressed data are packed into plurality of compressed units and stored in a storage *LU* (Logical Unit). One or more corresponding compressed units may be read or updated with no need of restoring the entire storage *LU* while maintaining de-fragmented *LU* structure. *IBM Storwize* specially works with cluster. The blocks are compressed and decompressed from nodes, due to save time for many restore operations and space on the entire cluster. The main goal of the *IBM Storwize* is to use less restore operations as possible, by compressing the data in blocks. The *uStorage* architecture does have similar goal and it is achieved by grouping a set of blocks in buckets, that we are configured to 2 MB size. The larger the bucket is, less restore operations are need. We tested the compress and decompress operations with set of buckets, but it waste a lot of cpu process and compete with the iSCSI PDU process.

Ceph [24] maximizes the separation between data and metadata management by replacing allocation tables with a pseudo-random data distribution function

(*CRUSH*) designed for heterogeneous and dynamic clusters of unreliable OSDs. Its device has intelligence of distributing data replication, failure detection and recovery to semi-autonomous OSDs running on a specialized local object filesystem. The design goals of *Ceph* are a POSIX filesystem (as much as close) that is scalable, reliable, and has very good performance. However, probably the most fundamental core assumption in the design of *Ceph* is that large-scale storage systems are dynamic and there are guaranteed to be failures. Therefore, the storage hardware is added and removed and the workloads on the system are changing. It is presumed there will be hardware failures and the filesystem needs to adaptable and resilient. *Ceph* uses its own filesystem based on OSD, while *uStorage* architecture uses Linux filesystem to deliver raw blocks on the primary storage layer. As result both architectures can provide a reliable storage layer to save contents with heterogeneous and dynamic cluster, but they use different ways to achieve it.

Panasas [14] is a company that builds object storage systems and took over the Extended Object FS (exofs) project, previously called *osdfs* (OSD file system). The exofs is a traditional Linux filesystem built on an object storage system with the origin of the ext2 filesystem. The *Panasas Storage Cluster* architecture is a Block-level Storage interface to OSD. This filesystem is partitioned between clients and manager, and uses RAID to strip data across OSDs. The *Panasas ActiveScale Storage Cluster* core is a decoupling of the datapath (read, write) from the control path (metadata). This separation provides a method for allowing clients direct and parallel access to the storage devices, providing high bandwidth to individual clients and to workstation clusters. It also distributes the system metadata allowing shared file access without a central bottleneck. Metadata is managed in a metadata server, a computing node separate from the OSDs, but residing on the same physical network. While *Panasas* uses RAID to make the data reliable, *uStorage* uses the *DataPeers* to store buckets.

6 Conclusion

This paper presented a storage architecture that uses Object-based Storage Device (OSD) on the back-end and delivers a Block-level Storage interface to the user. The motivation to use these two data storage concepts is because raw blocks handle heavy workloads and the OSD can easily scale horizontally with great reliability. The cache algorithm was evaluated by three methodologies. First we analyzed the health of the cache based on *Amazon ElastiCache* parameters [3]. Second we used the *FIO* benchmark to see how much IOPs the iSCSI Target cache achieves when it is configured in a very small size. Third we instantiated several *LUNs* on the same iSCSI Target cache and took metrics with different buckets size. When multiple users are connected to the *uStorage* the architecture uses less resources than multiple iSCSI Target on the same architecture.

Some considerations of future work on this architecture can be spread to its components. The iSCSI Target cache algorithm can be improved using *black-box* model for storage system [25]. The *Metadata Storage* can use a different

database to improve its performance, but it is still necessary to guarantee the ACID properties.

Acknowledgment. This work was supported by Ustore(http://www.usto.re) and partially supported by the National Institute of Science and Technology for Software Engineering (INES 2.0(http://www.ines.org.br)) grants 465614/2014-0, funded by CNPq(http://www.cnpq.br) grants 573964/2008-4 and FACEPE(http://www.facepe.br) grants APQ-1037-1.03/08.

References

1. Aizman, A.: Location independent scalable file and block storage. Google Patents (2012). https://www.google.com/patents/US20120011176, uS Patent App. 12/874,978
2. Axboe, J.: Fio-flexible i/o tester synthetic benchmark (2005). https://github.com/axboe/fio. Accessed 13 June 2015
3. Chiu, D., Agrawal, G.: Evaluating caching and storage options on the amazon web services cloud. In: 2010 11th IEEE/ACM International Conference on Grid Computing (GRID), pp. 17–24. IEEE (2010)
4. Duarte, M.P., Assad, R.E., Ferraz, F.S., Ferreira, L.P., de Lemos Meira, S.R.: An availability algorithm for backup systems using secure p2p platform. In: 2010 Fifth International Conference on Software Engineering Advances (ICSEA), pp. 477–481. IEEE (2010)
5. Durão, F., Assad, R., Fonseca, A., Fernando, J., Garcia, V., Trinta, F.: USTO.RE: a private cloud storage software system. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE 2013. LNCS, vol. 7977, pp. 452–466. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39200-9_38
6. Factor, M., Meth, K., Naor, D., Rodeh, O., Satran, J.: Object storage: the future building block for storage systems. In: 2005 IEEE International Symposium on Mass Storage Systems and Technology, pp. 119–123, June 2005
7. Fitzpatrick, B.: Distributed caching with memcached. Linux J. **2004**(124), 5 (2004)
8. Gong, L.: Jxta: a network programming environment. IEEE Internet Comput. **5**(3), 88–95 (2001)
9. Kedem, N., Amit, Y., Amit, N.: Method and system for compression of data for block mode access storage, 9 September 2008. https://www.google.com/patents/US7424482, uS Patent 7,424,482
10. Khattar, R.K., Murphy, M.S., Tarella, G.J., Nystrom, K.E.: Introduction to Storage Area Network, SAN. IBM Corporation, International Technical Support Organization (1999)
11. Kruchten, P.B.: The 4+1 view model of architecture. IEEE Softw. **12**(6), 42–50 (1995)
12. Martin, B.E., Pedersen, C.H., Bedford-Roberts, J.: An object-based taxonomy for distributed computing systems. Computer **24**(8), 17–27 (1991). doi:10.1109/2.84873
13. Mesnier, M., Ganger, G.R., Riedel, E.: Object-based storage. IEEE Commun. Mag. **41**(8), 84–90 (2003)
14. Nagle, D., Serenyi, D., Matthews, A.: The panasas activescale storage cluster: delivering scalable high bandwidth storage. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, SC 2004, p. 53. IEEE Computer Society, Washington (2004). 10.1109/SC.2004.57

15. Neto, A.J., da Fonseca, N.L.: Um estudo comparativo do desempenho dos protocolos iscsi e fibre channel. IEEE Latin Am. Trans. **5**(3), 151–157 (2007)
16. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. (TOCS) **10**(1), 26–52 (1992)
17. Ruemmler, C., Wilkes, J.: An introduction to disk drive modeling. Computer **27**(3), 17–28 (1994)
18. Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M., Zeidner, E.: Ietf rfc 3720: internet small computer systems interface (iscsi), April 2004. http://www.ietf.org/rfc/rfc3720.txt
19. Gnanasundaram, S., Shrivastava, A. (eds.): Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments. EBL-Schweitzer, Wiley, Hoboken (2012). https://books.google.com.br/books?id=PU7gkW9ArxIC
20. Silberschatz, A., Galvin, P.B., Gagne, G.: Sistemas Operacionais com Java. Elsevier, Rio de Janeiro (2004)
21. Performance Test Specification: Solid state storage performance test specification enterprise. Citeseer (2013)
22. Steinberg, D., Birk, Y.: An empirical analysis of the ieee-1394 serial bus protocol. IEEE Micro **20**(1), 58–65 (2000)
23. Troppens, U., Erkens, R., Mueller-Friedt, W., Wolafka, R., Haustein, N.: Storage networks explained: basics and application of fibre channel SAN, NAS, iSCSI, infiniband and FCoE. John Wiley & Sons (2011)
24. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. pp. 307–320, OSDI 2006, USENIX Association, Berkeley, CA, USA (2006). http://dl.acm.org/citation.cfm?id=1298455.1298485
25. Yin, L., Uttamchandani, S., Katz, R.: An empirical exploration of black-box performance models for storage systems. In: 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2006, pp. 433–440. IEEE (2006)