SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

July 2019, Vol. 62 072101:1–072101:14
https://doi.org/10.1007/s11432-017-9295-4

LCCFS: a lightweight distributed file system for cloud computing without journaling and metadata services

Li WANG[1*], Jingling XUE[2], Xiangke LIAO[1], Yunchuan WEN[3] & Min CHEN[3]

[1]*School of Computer Science and Technology, National University of Defense Technology, Changsha* 410073*, China;*
[2]*School of Computer Science and Engineering, University of New South Wales, Sydney* 2052*, Australia;*
[3]*Kylin Corporation, Changsha* 410073*, China*

Received 15 May 2017/Revised 27 October 2017/Accepted 23 November 2017/Published online 4 April 2019

Abstract The major usage of a file system integrated with a cloud computing platform is to provide the storage for VM (virtual machine) instances. Distributed file systems, especially those implemented on top of object storage have many potential advantages over traditional local file systems for VM instance storage. In this paper, we make an investigation in the requirements imposed on a file system in cloud computing scenario, and claim that the implementation of a file system for VM instance storage could be reasonably simplified. We demonstrate that on top of an object storage with simple object-granularity transaction support, a lightweight distributed file system, which requires neither journaling nor dedicated metadata services, can be developed for cloud computing. We have implemented such a distributed file system, called LCCFS (lightweight cloud computing file system), based on the RADOS (reliable autonomic distributed object storage) object storage. Our experimental results show that for the main workloads in cloud computing, LCCFS achieves almost the same or slightly higher performance than CephFS (ceph filesystem), another published distributed file system based on RADOS. Compared to CephFS, LCCFS has only one tenth of its LOCs (lines of code). This theoretical simplicity makes it easy to implement LCCFS correctly and stably by avoiding the sheer design and implementation complexity behind CephFS, thereby making LCCFS a promising candidate in the cloud computing production environment.

Keywords cloud computing, distributed file system, journaling, metadata service, garbage collection

Citation Wang L, Xue J L, Liao X K, et al. LCCFS: a lightweight distributed file system for cloud computing without journaling and metadata services. Sci China Inf Sci, 2019, 62(7): 072101, https://doi.org/10.1007/s11432-017-9295-4

1 Introduction

Cloud computing is widely considered as the next dominant technology in IT (information technology) industry. In cloud computing, the VM (virtual machine) instance storage serves as a crucial component to offer the storage for running VM instances. The object storage (also known as object-based storage [1]) architecture is an emerging architecture that promises improved manageability, scalability, reliability, and performance. Distributed file systems implemented on top of object storage demonstrate many potential advantages for VM instance storage, including scalable throughput, flexible dynamic resizing, high reliability and shared storage for VM live migration. However, the traditional object storage backed distributed file systems, such as Lustre [2], Panasas [3], do little to exploit storage device intelligence,

* Corresponding author (email: liwang@nudt.edu.cn)

© Science China Press and Springer-Verlag GmbH Germany, part of Springer Nature 2019

limiting the scalability of the storage cluster, especially when it consists of cheap PC servers, by placing a large burden on clients or metadata servers.

RADOS (reliable autonomic distributed object storage) [4] is a scalable and reliable object service that provides petabyte-scale storage for many thousands of nodes. RADOS leverages the intelligence presented in individual storage nodes to achieve high data reliability, and it introduces a globally replicated cluster map to avoid the need for object lookup presented in conventional object storage architectures. These features make RADOS a very promising backend for implementing distributed file systems for cloud computing.

CephFS [5] is a distributed file system implemented on top of RADOS. CephFS is designed as a general-purpose distributed file system, by targeting especially metadata-intensive and rapidly-varying workloads. The combined demand for metadata performance and dynamic load balancing poses a significant challenge on conventional metadata storage and journaling approaches. For example, the large number of small metadata updates will cause a very poor performance on conventional journal based architecture. To address these issues, CephFS presents a sophisticated distributed metadata management architecture [6], by introducing many distributed optimization algorithms in the MDS cluster, such as collaborative distributed caching, directory fragments, dynamic adaptive subtree partition/copy/migration, dynamic MDS cluster expansion, failure detection and failure recovery. In addition, distributed locking and message mechanism are used within the MDS cluster to synchronize the state and distributed large journals for failure recovery. This design has resulted in a complex cluster implementation with more than 80000 LOCs in C++. While theoretically attractive, its sheer complexity has affected its maintainability, stability and availability, preventing it from being deployed in production environments. According to the statistics [1)], there are 157 CephFS bugs reported in 2014, and in January 2015, thirteen new CephFS bugs were reported. Besides the bugs, there are other notable issues, for example, as a long-existing issue, performance thrashing is still observed on the latest version of the code even with only one MDS [7], and it got worse with more MDSs according to our experience.

In this paper, we make an investigation in the usage of distributed file system in the cloud computing platforms, and make some observations. These observations motivate us to develop a distributed file system dedicated for cloud computing with reasonably simplified implementation, guaranteed correctness and comparable performance.

In this paper, we make the following contributions:

- We analyze the requirements imposed on the file system in cloud computing platforms and demonstrate that the file system implementation could be reasonably simplified.
- We introduce a lightweight distributed file system, LCCFS, by storing both the metadata and data in an object storage. LCCFS needs neither a metadata server nor a journal, making it easy to be implemented correctly and stably, thereby making LCCFS a promising candidate in cloud computing production environments.
- Our evaluation shows that for file data read/write operations, which are the main workloads in cloud computing, LCCFS achieves almost the same or slightly higher performance than CephFS. For file metadata operations, LCCFS also delivers acceptable performance.

The rest of the paper is organized as follows. For background information, Section 2 describes how a distributed file system is used in the cloud computing scenarios. Section 3 motivates our work. Section 4 introduces the framework of LCCFS. Section 5 discusses the design of LCCFS. Section 6 describes the garbage collection process of LCCFS. Section 7 evaluates our work. Section 8 discusses related work. In Section 9, we summarize and conclude the paper.

2 Background

Interest and investment in both public and private cloud computing have grown rapidly due to their ability to reduce costs and speed of application deployment. Enterprises no longer need to maintain

1) Ceph fs bug tracker. http://tracker.ceph.com/projects/cephfs.

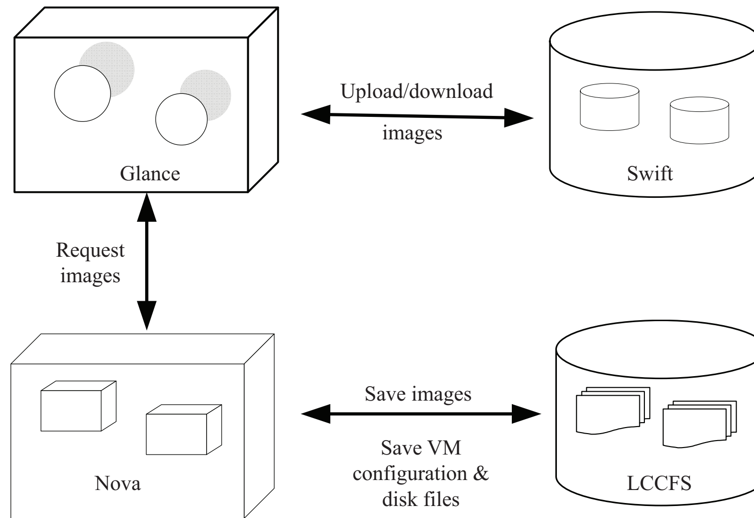

Figure 1 The integration of LCCFS with OpenStack.

computing resources that are used periodically but are often idle, and staff time can be devoted to application development rather than system maintenance. OpenStack is one of the largest and most active open source cloud computing platforms; others include CloudStack, Eucalyptus, and OpenNebula. While introduced with respect to OpenStack, LCCFS is applicable to other platforms as well.

OpenStack provides an IaaS (Infrastructure-as-a-Service) solution through a variety of complemental services. Among which, Nova manages the lifecycle of virtual machine instances in an OpenStack environment, including spawning, scheduling, and decommissioning of virtual machines on demand. Glance stores and retrieves virtual machine disk images. Swift stores and retrieves arbitrary unstructured data objects via a RESTful (representational state transfer), HTTP based API. A distributed file system, such as CephFS or LCCFS, is integrated with OpenStack to provide the shared storage for virtual machine instances, as shown in Figure 1. When spawning virtual machines, if images based by the virtual machines are not found in the local instance store directory, normally, /var/lib/nova/instances, Nova will request the images from Glance, Glance will retrieve the requested images from Swift and then return them to Nova. Nova will save the images into the instance directory. Besides the images, all the files needed by the running virtual machine instances are also stored in the instance directory, including the configuration file and disk difference files. Mounting the instance directory with LCCFS has many advantages compared with traditional disk based file systems, such as EXT4 (fourth extended filesystem) and XFS. The obvious ones include scalable throughput, flexible dynamic resizing and high reliability, and it enables virtual machine live migration. When mounting the instance directory with LCCFS, all compute nodes share the instance store. When migrating a virtual machine, it is only necessary to copy memory states between compute nodes, leading to an extremely fast migration without shutting down the virtual machine itself.

3 Motivation

In a typical cloud computing environment, a host maps image files in the VM instance storage backed by host file system, as virtual block devices, i.e, disks to a VM, the guest operating system will format the disks and create guest file systems on them. The host file system mostly resembles a "dumb" disk, in other words, it is enough for the host file system to implement exactly the semantics of a disk controller driver. For a disk controller, when it is serving a write request, it does not implement a transactional mechanism to ensure a write operation done atomically. Instead, it is up to the file system, who manages the disk, to adopt some techniques such as journaling to prevent inconsistency during disk failure or power down, if necessary. Similarly, in the cloud computing environment, the host file system needs only

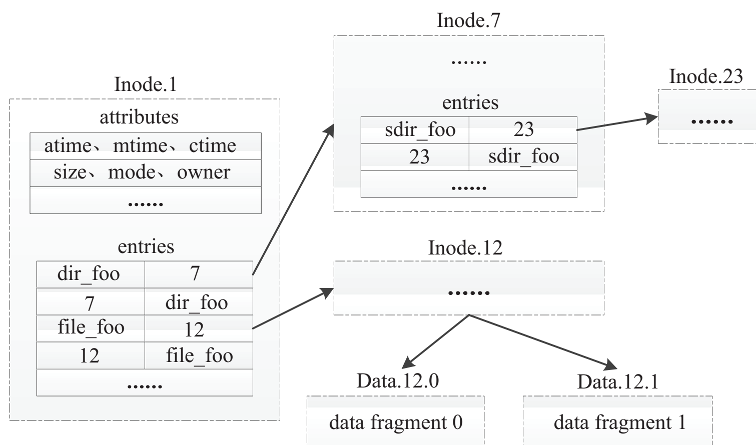

Figure 2 A namespace example.

to implement a synchronous write or flush interface, for example, fsync. It does not need to provide the data journaling mechanism to ensure a transactional data write operation.

Observation 1. Data does not need to be journaled by the file system for VM instance storage.

We analyzed carefully the file system operations invoked by typical cloud computing platform, such as OpenStack. Not surprisingly, we found the most frequently called operations are file data read and write, which correspond to the disk I/O operations performed by guest operating systems. In contrast, the metadata operations are relatively rare.

Observation 2. The workload in cloud computing imposes a low pressure on metadata performance of the file system for VM instance storage.

Object storage is a storage architecture that manages data as objects. It is easy to implement object-granularity transaction support on object storage infrastructure. Typically, there are two ways to store an object, one is to store an object as a file in conventional local file system, the other is to store an object as a key-value pair in a key-value database, with object ID being the key, and the object data as the value. For the latter, most key-value databases naturally support a series of operations on the same key executed in a transaction. For a local file system backed object storage, it could adopt the classical write-ahead logging algorithm, to implement the atomicity of a series of operations on the same object, i.e, on the same file in the underlying local file system. This is the way adopted by RADOS for object-granularity transaction support.

As a result, the Observation 2 motivates us to develop a distributed file system for cloud computing without dedicated metadata services. The Observation 1, as well as the object-granularity transaction support in object storage, enables us to further implement the file system without journaling.

4 Framework

The UNIX and Linux file systems use inode to record file metadata, one inode per file. LCCFS associates each inode with a global unique inode number ino, and stores it as the binary data in an object named Inode.(ino). For simplicity, we call such object inode object. For a regular file with its inode number being ino, its data are stored in a set of data objects named Data.(ino).(index), where each object in the set, stores the data of the file within [index×OBJECT_SIZE, (index+1)×OBJECT_SIZE). For a directory file, its data, namely the information of the files inside it are stored directly in its inode object as key-value pairs with the format (name, ino) and (ino, name).

Let us consider a namespace as shown in Figure 2. In the root directory with an inode number 1, there are one sub-directory and one file, with their inode numbers being 7 and 12, respectively. In addition, the directory with inode number 7 has a sub-directory with its inode number being 23. The size of the file with inode number 12 is 6 MB. Suppose the object size is 4 MB, then the data within [0, 4 MB) are

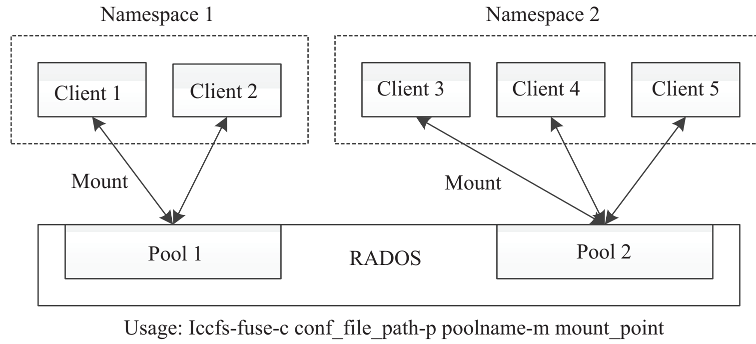

Figure 3 The usage of LCCFS.

stored in object Data.12.0, and the data within in [4 MB, 6 MB) are stored in Data.12.1.

LCCFS stores all the objects in a file system namespace into a dedicated RADOS pool, where a pool is a logical partition for storing objects. A pool can be dynamically resized easily and flexibly, and different namespaces stored in different pools ensure the namespace isolation. As shown in Figure 3, LCCFS is simple to use. For example, if the user wants to mount the name space corresponding to pool 1 to /var/lib/nova/instances, and the RADOS configuration file is stored at /etc/lccfs/rados.conf, then the command is lccfs-fuse -c /etc/lccfs/rados.conf -p 1 -m /var/lib/nova/instances, where rados.conf is the standard RADOS cluster configuration file. This configuration file provides the client the address of the monitor, which knows the whole topology of the cluster being used.

5 Non-journaling design

In this section, we discuss our approach to ensure metadata consistency, using a non-journaling design. The key idea is to implement the file system interfaces by packing as many as possible operations into a transaction, and carefully sort the execution order of transactions, so that no inconsistency will be introduced whenever crash happens. Next, let us consider the set of interfaces to be implemented by a file system, getattr(retrieve file metadata), setattr(modify file metadata), lookup(lookup a file in a given directory), readdir(list the files inside a directory) and getxattr(get file extra metadata) are only referred to a read/write operation on a single inode object in the framework of LCCFS, thus the object-granularity transaction support manifests to implement them in a transaction. In the following sections, we discuss the implementations of the interfaces with operations on two or more objects, and demonstrate why a journal is not needed.

5.1 Create and mkdir

For create and mkdir to create a file/directory, they refer to operations on two objects, the inode object corresponding to the parent directory and a newly created inode object corresponding to the new file/directory. The algorithm for create is shown in Algorithm 1. The inputs are the inode number of the parent directory and the name of the file to be created. First, in lines 3 and 4, a new global unique inode number is generated, and a reclaim entry is inserted for garbage collection which will be discussed in Section 6. The remaining process is finished in three steps, each corresponding to a transaction on one single object. First, it creates an inode object in lines 5–8. Next, it adds an entry in the parent inode object if it does not already exist in lines 9–14. Finally, if the entry already exists, it deletes the created object in lines 15–19.

Consider the namespace consistency of Algorithm 1. If it crashes within the first transaction (lines 5–8), nothing will be left. If within the second transaction (lines 9–14) or the third transaction (lines 15–19), an orphan inode object Inode.(ino) will be left but invisible from namespace, therefore causing no namespace consistency problems. The orphan inode object will be reclaimed by a garbage connection process discussed in Section 6. Mkdir is similar to create.

Algorithm 1 Create a file

```
1: procedure create
2: Input: inodeno_t pino, const char * name
3: Generate a new inode number ino;
4: Insert a reclaim entry (pino, ino);
5: begin_transaction
6: Create an object Inode.(ino);
7: Inode.(ino)→pino = pino;
8: end_transaction
9: begin_transaction
10: exist = entry_exist(Inode.(pino), name);
11: if exist == FALSE then
12:     Insert an entry name in the object Inode.(pino);
13: end if
14: end_transaction
15: if exist == TRUE then
16:     begin_transaction
17:     Delete the object Inode.(ino);
18:     end_transaction
19: end if
20: end procedure
```

5.2 Unlink and rmdir

The algorithm for unlink is given in Algorithm 2. The inputs are the inode number pino of the parent directory, and the inode number ino of the file to be deleted. The job is fulfilled in three steps. First, it adds a reclaim entry as create does. Second, done in a transaction, it removes the entry from parent inode object, which deletes the file from the perspective of namespace. Third, it relies on an asynchronous garbage collection process to reclaim the inode and data objects corresponding to the deleted file itself, leading to a simple, atomic and extremely fast unlink operation.

Algorithm 2 Remove a file

```
1: procedure unlink
2: Input: inodeno_t pino, inodeno_t ino
3: Insert a reclaim item (pino, ino);
4: begin_transaction
5: if entry_exist(Inode.(pino), ino) == TRUE then
6:     Remove the entry ino from the object Inode.(pino);
7: end if
8: end_transaction
9: end procedure
```

For rmdir, it shares the similar algorithm to unlink with the only exception that it will first check if the directory to be deleted is empty, required by the POSIX semantics. Here the check and the entry deletion could not be finished in one transaction since they are on different inode objects. This may introduce a race if a create issued in between the directory empty check and the entry deletion. However, this normally is not considered as a problem for a distributed file system, where a file created by one client could always be deleted by another client, without a notification. In addition, according to our observation to the behavior of cloud computing platform, the directories which contain the VM image files, are always exclusively accessed by the corresponding VMM processes.

5.3 Rename

The algorithm for rename is shown in Algorithm 3. The inputs are the inode number pino of parent directory, the original name oldname and the new name newname. The algorithm first makes sure that the entry oldname exists and the entry newname does not exist. Then it removes the entry oldname from the parent inode and inserts an entry newname into the parent inode. All these operations are on the same object Inode.(pino) and thus done in a transaction.

Algorithm 3 Rename a file

```
 1: procedure rename
 2: Input: inodeno_t pino, const char * oldname, const char *newname
 3: begin_transaction
 4: if entry_exist(Inode.(pino), oldname) == TRUE then
 5:     if entry_exist(Inode.(pino), newname) == FALSE then
 6:         Remove the entry oldname from the object Inode.(pino);
 7:         Insert an entry newname to the object Inode.(pino);
 8:     end if
 9: end if
10: end_transaction
11: end procedure
```

For rename, POSIX semantics allows it to move a file/directory across directories on the same file system. In our design, rename is only allowed to modify the file name, otherwise EXDEV is returned. That is, we do not implement the move semantics of rename. This is based on the following considerations. (1) According to our observation to the behavior of cloud computing platform, rename is rarely invoked, in particular, it is never called to do moving; (2) Moving operation refers to an operation on two inode objects, the old and the new parent inodes, which is not supported to be done in a transaction; (3) When such an operation is needed in future, it can be achieved by a copy operation and a delete operation.

6 Garbage collection

LCCFS relies on a runtime garbage collection process to reclaim orphan objects. The online garbage collection algorithm is described in Subsection 6.1, and it has the following properties. (1) The reclaim process is allowed to crash at any time. (2) The reclaim entries could be repeatedly processed if crash happened, before they are deleted.

Although LCCFS should not cause name space inconsistency and space leaking, in practice, all kinds of unexpected errors may happen. For in case, we also introduce an offline file system check algorithm, as most file systems provide, which is able to correct any name space inconsistency and reclaim any orphan objects, no matter how these are generated. The algorithm has a low $o(n)$ complexity, where n is the number of objects in the namespace, and is discussed in Subsection 6.2.

6.1 Online collection

The reclaim processes are periodically executed at the client side. LCCFS stores the reclaim information in a new type of object, Reclaim object, named Reclaim.(number), where number is globally unique, and increases monotonically. Besides, an object named Reclaim.counter is introduced with two values wp and rp stored. These two values are used to implement producer-consumer model. Reclaim.counter→wp points to the reclaim object being written by the producers, while Reclaim.counter→rp points to the one being read by the consumers. The file system operations which may leave orphan objects, will insert reclaim entries into the Reclaim object. The algorithm for producers to insert a reclaim item is shown in Algorithm 4. In line 4, the producers read the value of global wp pointer into their local copies. In lines 6–11, the object referred by wp is created if it does not exist, and its state is set to WRITE and timestamp updated to current time. Next, the producers check if the state of the reclaim object is WRITE (not read by consumers) and not full. If both conditions hold, the reclaim item will be successfully inserted in line 15. Otherwise, the global wp value is increased by one, by one of the producers in lines 20–25, and all the producers retry the above processes started in line 4.

The algorithm for the consumer to consume reclaim items is shown in Algorithm 5. It first checks if the reclaim object pointed to by Reclaim.counter→rp is old enough in line 7, this guarantees a reclaim object will not be consumed in REC_THRESHOLD. Then the algorithm changes its state to READ to prevent the producers from involving in line 8. The consumer exclusively owns the object and consumes all the items in the object in lines 16–19. At last, the consumer deletes the object, increases the value

Algorithm 4 Add reclaim item

```
 1: procedure add_reclaim_item
 2: Input: (pino, ino)
 3: again:
 4: wp = Reclaim.counter→wp;
 5: if object_exist(Reclaim.(wp)) == FALSE then
 6:     begin_transaction
 7:     if create_object(Reclaim.(wp)) == SUCCESS then
 8:         Reclaim.(wp)→state = WRITE;
 9:         Reclaim.(wp)→timestamp = current_time;
10:     end if
11:     end_transaction
12: end if
13: begin_transaction
14: if Reclaim.(wp)→state == WRITE && Reclaim.(wp) is not full then
15:     Insert (pino, ino) into Reclaim.(wp);
16:     Reclaim.(wp)→timestamp = current_time;
17:     Return;
18: end if
19: end_transaction
20: wp++;
21: begin_transaction
22: if Reclaim.counter→wp < wp then
23:     Reclaim.counter→wp++;
24: end if
25: end_transaction
26: goto again.
27: end procedure
```

of global rp pointer, and tries on next object. For simplicity, the algorithm currently allows only one consumer to work at a time, and it is easily improved by using multiple producer-consumer queues if necessary.

The algorithm for processing a reclaim entry is also shown in Algorithm 5. Using a reclaim item as input, it first checks if the parent inode object, inode object, and the entry in the parent inode object all exist in line 26. If this holds, it means the file system is consistent without orphan object. Otherwise, it indicates that the inode object Inode.(ino) is an orphan object, and if it has been already deleted, nothing needs to be done. Then the algorithm checks the type of the inode, if it represents a file, its data objects are reclaimed. If it represents a directory, its entries are recursively processed in a width-first manner. Finally, the inode object iteself is deleted.

Let us demonstrate why the algorithm is able to reclaim the space. Considering create in Algorithm 1, if the inode has been created, but the insertion of entry in parent inode failed, it will leave an orphan object Inode.(ino). The algorithm will delete the object in line 44. Mkdir is similar. For unlink, the data objects corresponding to the deleted file are reclaimed in lines 34–36, and the inode object is reclaimed in line 44. For rmdir, if the directory Inode.(ino) is not empty, the algorithm will iterate on the entries in Inode.(ino) in lines 39–42, for each entry, add a reclaim item to make the entries be deleted recursively, then remove the entry.

6.2 Offline collection

The algorithm for offline garbage collection is shown in Algorithm 6. It iterates on each object o in the object storage. If o represents file data, it is not an orhpan object if the corresponding inode object exists, and the data scope it represents is within the size recorded in inode object, otherwise, o is reclaimed. If o represents an inode, the algorithm checks if o is an orphan object by checking if its parent inode exists, and the corresponding entry exists, otherwise o is reclaimed. Next, if o represents a directory, the algorithm checks if each entry m in o is valid by checking if the the inode object corresponding to m exists, otherwise m is removed from o. By the above process, any namespace inconsistency could be corrected, and any orphan object could be reclaimed.

Algorithm 5 Consume reclaim item

```
1: procedure consume_reclaim_item
2: again:
3: rp = Reclaim.counter→rp;
4: begin_transaction
5: result = (Reclaim.(rp)→state == READ);
6: if result == FALSE then
7:     if (current_time−Reclaim.(rp)→timestamp) > REC_THRESHOLD then
8:         Reclaim.(rp)→state = READ;
9:         result = TRUE;
10:    end if
11: end if
12: end_transaction
13: if result == FALSE then
14:    Return;
15: end if
16: for each item m in Reclaim.(rp) do
17:    process_reclaim_item(m);
18:    Remove the item m from the object Reclaim.(rp);
19: end for
20: Delete the object Reclaim.(rp);
21: Reclaim.counter→rp++;
22: goto again;
23: end procedure
24: procedure process_reclaim_entry
25: Input: (pino, ino)
26: if object_exist(Inode.(pino)) == TRUE && object_exist(Inode.(ino)) == TRUE && entry_exist(Inode.(pino), ino) ==
    TRUE then
27:    Return;
28: end if
29: if object_exist(Inode.(ino)) == FALSE then
30:    Return;
31: end if
32: if (file_type(Inode.(ino)) == FILE) then
33:    count = Inode.(ino)→ size/OBJECT_SIZE;
34:    for m = 0 to count do
35:        Delete the object Data.(ino).(m);
36:    end for
37: end if
38: if (file_type(Inode.(ino)) == DIRECTORY) then
39:    for each entry cino in Inode.(ino) do
40:        Insert a reclaim entry (ino, cino);
41:        Remove the entry cino from the object Inode.(ino);
42:    end for
43: end if
44: Delete the object Inode.(ino);
45: end procedure
```

7 Evaluation

We have implemented LCCFS on top of RADOS object storage [4]. We measured the performance of LCCFS as a comparison to CephFS. The test bed is six workstations, with each being an Intel(R) Core(TM) i7-4770 CPU at 3.40 GHz with 8 GB memory. The operating system is Ubuntu 12.04 x86 64 server with the Linux kernel 3.8.0. The LCCFS and CephFS are both implemented based on the RADOS 0.80.5. The RADOS cluster comprises 1 MON (monitor) and 4 OSDs (object storage daemon). Besides, CephFS needs one more node as MDS. As described above, the major workload of a file system in cloud computing environment is data operations, others mainly refer to directory operations, more exactly, directory creation/deletion during VM creation and deletion.

Algorithm 6 Offline file system check

```
1: procedure fsck
2: for each object o in the object storage do
3:     if o is a data object named Data.(ino).(index) then
4:         if object_exist(Inode.(ino)) == FALSE ||
           index * OBJECT_SIZE > Inode.(ino)→size then
5:             Delete the object o; //orphan object.
6:         end if
7:     end if
8:     if o is an inode object named Inode.(ino) then
9:         pino = o→pino; //the inode number of parent inode.
10:        if object_exist(Inode.(pino)) == FALSE ||
           entry_exist(Inode.(pino), ino) == FALSE then
11:            Delete the object o; //orphan object.
12:            continue
13:        end if
14:        if file_type(Inode.(ino)) == DIRECTORY then
15:            for each entry m in Inode.(ino) do
16:                if object_exist(Inode.(m)) == FALSE then
17:                    Remove the entry m from the object Inode.(ino); //orphan entry.
18:                end if
19:            end for
20:        end if
21:    end if
22: end for
23: end procedure
```

7.1 Data performance

We evaluate the data performance by running iozone (a filesystem benchmark tool) in the VM. The guest operating system is Ubuntu 12.04 x86 64 desktop and the virtual machine monitor is KVM. Figure 4 shows the performance measured by iozone on LCCFS relative to CephFS, under the patterns of sequential read, random read, sequential write and random write, respectively. The evaluation are done under different record sizes. As the results shown, for all types of data read and write operations, LCCFS achieves very close or even a little bit higher performance compared to CephFS.

We also measured the data performance by directly running the tool dd (an utility to copy files) on LCCFS and CephFS. As the results shown in Figure 5, LCCFS exhibits slightly higher performance compared to CephFS.

7.2 Directory performance

The metadata performance of LCCFS is evaluated by measuring the time on creating/deleting directories. Figure 6 shows the time on creating/deleting 100–800 directories. The depth of each directory is one, namely, without subdirectories. As the results shown, LCCFS achieves the close performance with CephFS on directory deletion. For directory creation, LCCFS is slower. This is because CephFS has dedicated metadata service, which implements some optimizations on directory creating. Nevertheless, the total time on creating 800 directories is less than 120 s for LCCFS, which corresponds to create 800 VMs at a time. Given the VM creation is not a frequent operation, the performance of LCCFS is acceptable.

7.3 Performance scalability

We also measured the performance scalability of LCCFS under a large scale cluster with 189 nodes, with each node being an Intel(R) Xeon(R) 6-core CPU E5-2620v2 @2.10 GHz with 64 GB memory, 2 TB SATA disk. The operating system is Ubuntu 14.04 with the Linux kernel 3.13. The RADOS cluster comprises 1 MON and 177 OSDs. The test tool is fio, and we use 11 clients to generate sufficient IO pressure to RADOS, until 100% disk utilization observed. Figures 7(a) and (b) show the normalized IOPS measured

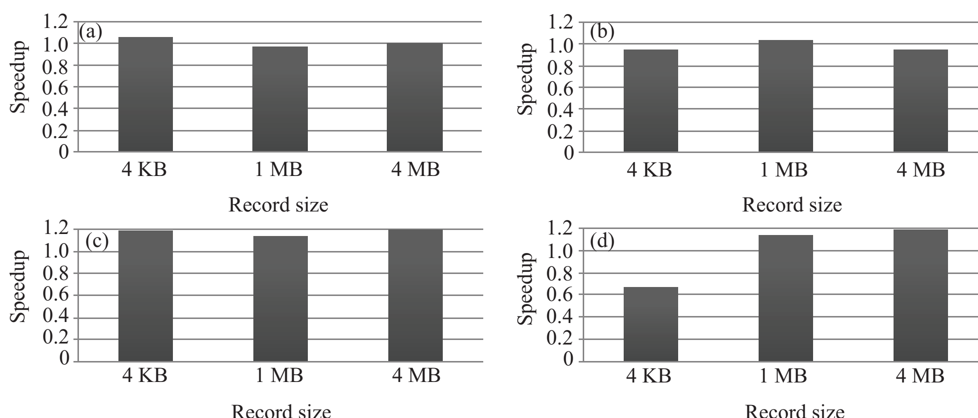

Figure 4 Performance speedups of LCCFS over CephFS under three different record sizes for (a) sequential read, (b) random read, (c) sequential write, and (d) random write.

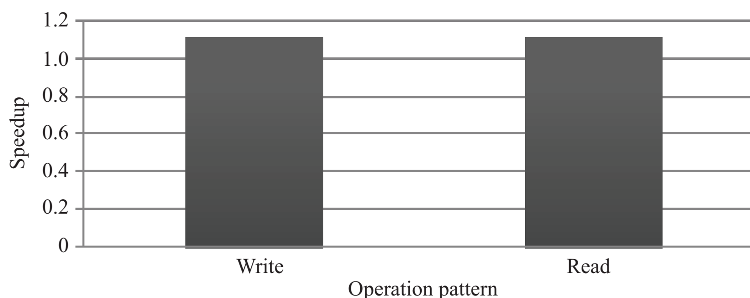

Figure 5 Performance speedups of LCCFS over CephFS for read and write measured by dd.

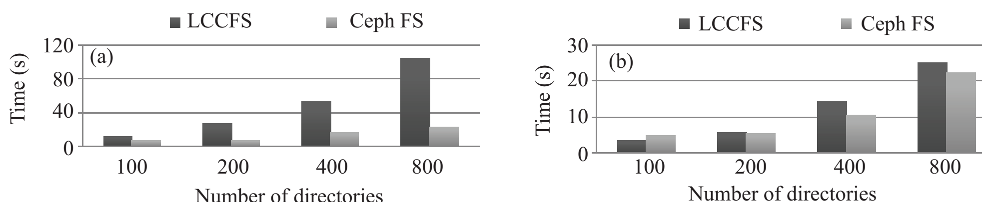

Figure 6 The time of directory creation and deletion for LCCFS and CephFS.

on LCCFS, under the patterns of random read and random write, respectively, with a small record size of 4 KB, under different numbers of OSDs. Figure 7(c)–(f) shows the normalized bandwidth measured, under the patterns of sequential read, sequential write, random read and random write, with a record size of 4 MB, under different numbers of OSDs. As the results show, with the increasing of the OSD number, LCCFS demonstrates nearly linear performance scalability, and makes good use of the disk bandwidth provided by OSDs.

8 Related work

There exist many object storage systems. The ANSI T10 OSD standard defines the object storage at the protocol and device layer [8]. Lustre [2] and Panasas [3] do little to exploit storage device intelligence, and are mainly used in high performance computing. zFS [9] and Ursa Minor [10] rely on explicit allocation maps to specify the object locations, limiting efficiency, scalability. OceanStore [11], Farsite [12] and Glacier [13] focus primarily on the scalability over a wide area, but not performance. Haystack [14] is designed paticularly for scale photo storage and management. As a comparison, RADOS leverages the intelligence present in individual storage nodes to achieve high data scalability and reliability, thus is suitable for cloud computing and it focuses mainly on local area network.

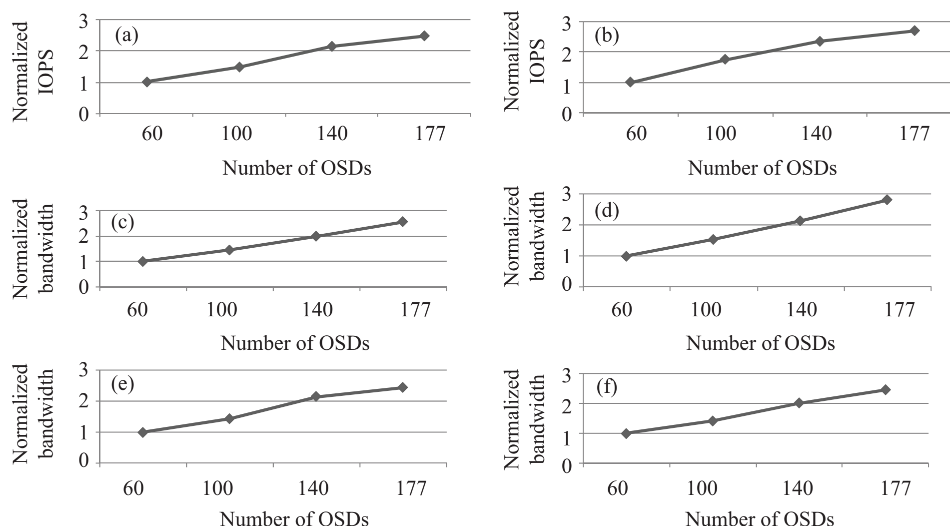

Figure 7 Normalized IOPS under random (a) read and (b) write with a record size of 4 KB. Normalized bandwidth under sequential (c) read and (d) write with a record size of 4 MB. Normalized bandwidth under random (e) read and (f) write with a record size of 4 MB.

While earlier file systems, such as ext2 [15], rely on costly consistency checks for failure recovery, the most modern file systems introduce the logging for file system consistency. There are typically the following ways to implement logging, one approach is the journaling [16–18][2)3)], which updates metadata and/or data in the journaling area before updating them to their home locations. Log-structured file systems [19, 20] use the log as the primary storage structure, and introduce a cleaner process for deallocation. Shadow paging [21, 22] uses a copy-on-write approach to first write new data to unallocated area, then simply update pointers to commit the changes. Shen et al. [23] apply two simple techniques to enhance the implementation of journaling of journal. Fryer et al. [24] present a runtime checker for the integrity of file system logging implementation. Ext4-lazy [25] optimizes journaling performance of ext4 file system on Drive-Managed SMR (Shingled Magnetic Recording) disks. Since RADOS leverages the legacy host file systems, such as ext4, to store objects in a cluster node. LCCFS could benefit from these techniques to improve performance.

Metadata performance is important for both traditional file systems and distributed file systems. Recently, some techniques, such as new virtual file system interfaces [26], many-to-one mappings of files to metadata [27], are proposed to optimize metadata performance of traditional file systems. For distributed file systems, Xu et al. [28] present a ring-based metadata management mechanism named dynamic ring online partitioning (DROP). It can preserve metadata locality using locality-preserving hashing, keep metadata consistency, as well as dynamically distribute metadata among metadata server cluster to keep load balancing. CalvinFS [29] leverages a high-throughput distributed database system for metadata management which horizontally partitions and replicates file system metadata across a shared-nothing cluster of servers, spanning multiple geographic regions. HopsFS [30] optimizes the hadoop distributed file system (HDFS) by using a distributed metadata service built on a NewSQL database. While these optimizations target metadata-intensive workloads, LCCFS proposes a non-metadata-service design based on the observation that cloud computing workloads impose a low pressure on metadata performance of the file system for virtual machine instance storage.

9 Conclusion

We presented a lightweight distributed file system, LCCFS, for cloud computing. LCCFS stores all data in an object storage, and introduces neither journaling nor metadata services. This lightweight yet efficient

2) Reiserfs. http://reiser4.wiki.kernel.org.

3) Xfs: a high-performance journaling filesystem. http://oss.sgi.com/projects/xfs/.

design makes it easy to be implemented correctly and stably. The theoretical simplicity and performance efficiency make LCCFS a promising candidate in cloud computing production environment.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 61370018).

References

1 Mesnier M, Ganger G R, Riedel E. Object-based storage. IEEE Commun Mag, 2003, 41: 84–90
2 Schwan P. Lustre: building a file system for 1000-node clusters. In: Proceedings of the Linux Symposium, Ottawa, 2003
3 Welch B, Gibson G. Managing scalability in object storage systems for HPC linux clusters. In: Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies, Greenbelt, 2004. 433–445
4 Weil S A, Leung A W, Brandt S A, et al. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In: Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing, Reno, 2007. 35–44
5 Weil S A, Brandt S A, Miller E L, et al. Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, 2006. 307–320
6 Weil S A, Pollack K T, Brandt S A, et al. Dynamic metadata management for petabyte-scale file systems. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, Pittsburgh, 2004. 6–12
7 Sevilla M. Mds has inconsistent performance. 2015. http://comments.gmane.org/gmane.comp.file-systems.ceph.devel/22674
8 Nagle D, Factor M E, Iren S, et al. The ANSI T10 object-based storage standard and current Implementations. IBM J Res Dev, 2008, 52: 401–411
9 Rodeh O, Teperman A. zFS — a scalable distributed file system using object disks. In: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, San Diego, 2003. 207–218
10 Abd-El-Malek M, Courtright II W V, Cranor C, et al. Ursa minor: versatile cluster-based storage. In: Proceedings of the 4th USENIX Conference on File and Storage Technologies, San Francisco, 2005. 59–72
11 Kubiatowicz J, Bindel D, Chen Y, et al. Oceanstore: an architecture for global-scale persistent storage. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, 2000. 190–201
12 Adya A, Bolosky W J, Castro M, et al. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, New York, 2002. 1–14
13 Haeberlen A, Mislove A, Druschel P. Glacier: highly durable, decentralized storage despite massive correlated failures. In: Proceedings of the 2nd Symposium on Networked Systems Design & Implementation, Berkeley, 2005. 143–158
14 Beaver D, Kumar S, Li H C, et al. Finding a needle in haystack: facebook's photo storage. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Vancouver, 2010. 1–8
15 Card R, Ts T, Tweedie S. Design and implementation of the second extended filesystem. In: Proceedings of the 1st Dutch International Symposium on Linux, Amsterdam, 1994
16 Mathur A, Cao M, Bhattacharya S, et al. The new ext4 lesystem: current status and future plans. In: Proceedings of the Linux Symposium, Ottawa, 2007. 21–33
17 Ts'o T Y, Tweedie S. Planned extensions to the Linux EXT2/EXT3 filesystem. In: Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, Berkeley, 2002. 235–243
18 Tweedie S. Ext3, journaling filesystem. In: Proceedings of the Linux Symposium, Ottawa, 2000
19 Konishi R, Amagai Y, Sato K, et al. The Linux implementation of a log-structured file system. SIGOPS Oper Syst Rev, 2006, 40: 102–107
20 Rosenblum M, Ousterhout J K. The design and implementation of a log-structured file system. ACM Trans Comput Syst, 1992, 10: 26–52
21 Hitz D, Lau J, Malcolm M. File system design for an NFS file server appliance. In: Proceedings of the USENIX Winter 1994 Technical Conference, San Francisco, 1994. 19
22 Rodeh O, Bacik J, Mason C. Btrfs: the linux b-tree filesystem. ACM Trans Storage, 2013, 9: 1–32
23 Shen K, Park S, Zhu M. Journaling of journal is (almost) free. In: Proceedings of the 12th USENIX Conference on File and Storage Technologies, Santa Clara, 2014. 287–293
24 Fryer D, Qin D, Sun J, et al. Checking the integrity of transactional mechanisms. In: Proceedings of the 12th USENIX Conference on File and Storage Technologies, Berkeley, 2014. 295–308
25 Aghayev A, Theodore Y, Gibson G, et al. Evolving ext4 for shingled disks. In: Proceedings of the 15th USENIX Conference on File and Storage Technologies, Santa clara, 2017. 105–119

26 Wang L, Liao X K, Xue J L, et al. Enhancement of cooperation between file systems and applications–VFS extensions for optimized performance. Sci China Inf Sci, 2015, 58: 092104

27 Zhang S, Catanese H, Wang A I A. The composite-file file system: decoupling the one-to-one mapping of files and metadata for better performance. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies, Santa Clara, 2016. 15–22

28 Xu Q, Arumugam R V, Yong K L, et al. Efficient and scalable metadata management in EB-scale file systems. IEEE Trans Parallel Distrib Syst, 2014, 25: 2840–2850

29 Thomson A, Abadi D J. Calvinfs: consistent wan replication and scalable metadata management for distributed file systems. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies, Santa Clara, 2015. 1–14

30 Niazi S, Ismail M, Haridi S, et al. Hopsfs: scaling hierarchical file system metadata using newsql databases. In: Proceedings of the 15th USENIX Conference on File and Storage Technologies, Santa Clara, 2017. 89–104