HLBS: A Random Read/Write Distributed Storage System Based on HDFS

***Abstract*—Hadoop Distributed File System (HDFS) has gained in popularity as distributed file system for both enterprise and academic research purpose because of its high fault-tolerance and the ability to be deployed upon low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. However, its once-write-many-read semantics could only be suitable for particular applications like information retrieval and so on. For some other kinds of storage applications which need random read/write interfaces, it may be hard to construct upon HDFS directly. In order to solve this problem, we have designed/implemented an open source distributed storage system, in which a wrapping layer is implemented upon HDFS borrowing the idea of Log-structured File System (LFS) without changing the source codes of HDFS, which is named Hadoop distributed file system based Log-structured Block storage System (HLBS). HLBS could not only support read/write randomly but also some useful features like snapshot, block compression and so on. We have also integrated HLBS into XEN, Network Block Device, QEMU, Openstack, Libvirt and iSCSI to widen its application scenarios. Experiments show that the write performance of HLBS is close to HDFS. However, the small read performance of HLBS is better than HDFS and the large read is almost the same. Experiments also indicate that the I/O performance of HLBS is better than Sheepdog storage system. According to our knowledge, prior published work does no jobs to support read/write randomly upon HDFS without changing the source codes of HDFS.**

I. Introduction

With the continuous increasing of computing and data storage requirements, traditional distributed system architectures have changed in last decade very much. To enable high reliability, availability and scalability storage of huge amount of data upon lower cost hardware, Apache organization has developed a new open source distributed file system *Hadoop Distributed File System* [1], which borrows the idea from *Google File System* [2]. Since HDFS is published, it has been adopted in lots of commercial and academic systems. However, HDFS only provides once-write-many-read semantics so that applications requiring random read/write semantics could not run upon HDFS. One typical example is back-end storage system for virtual machine like [3].

For these kinds of applications, local file systems could not provide enough space, high reliability, availability and scalability. To solve this problem, many projects have been launched. For instance, Amazon has developed a commercial system named Elastic Block Storage [4], which could be applied for applications including virtual machine back-end storage. The laboratory of *NTT Corporation (Nippon Telegraph and Telephone Corporation)* in Japan has developed Sheepdog system, which is an open source back-end storage system designed especially for QEMU/KVM [5], [6] virtual machine. Random read/write data access semantics are supported in these systems.

Compared with above projects, we propose a different approach to solve the problem. In our solution, we implement a new open source back-end storage system which provides random read/write data access semantics based on HDFS not requiring to modify the source codes of HDFS with the idea of *Log-structured File System*. For this reason, we name our system as *Hdfs based Log-structured Block storage System*. To achieve high reliability and availability of HLBS, we implement linear and tree snapshot subsystem based on the natural ability of LFS [7]. To save storage space, we support block-level compression and garbage collection mechanisms [8]. To improve I/O performance, we implement HLBS cache mechanism. We have also developed HLBS drivers to support *XEN* [9], *Network Block Device (NBD)* [10], *QEMU/KVM*, *Openstack* [11], *Libvirt* [12] and *iSCSI* [13] so that it could be convenient to use HLBS upon these famous software.

The rest of the paper is organized as follows. We present the design and implementation in Section II and show evaluation results in Section III. Section IV surveys related work and Section VI summarizes our conclusions.

II. Design and implementation

In order to design and implement an effective distributed back-end storage system based on Hadoop Distributed File System for virtual machine, we propose Hdfs based Log-structured Block storage System which could not only provide random read/write access semantics but also realize some useful features which inherit from the advantages of HDFS and Log-structured File System.

HLBS is designed and implemented with a subset of *Portable Operating System Interface Of Unix (POSIX)* to make the system more compact, more flexibility and have lower performance cost. To ease the complexity of debugging and deploying HLBS, we design and implement HLBS in user space. Figure 1 shows the whole structure of HLBS.

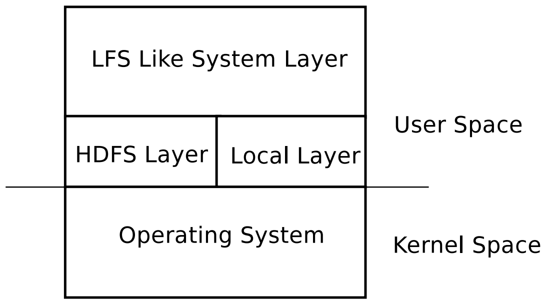


Fig. 1: Layout of HLBS

HLBS structure is shown in Fig. 1 and it has two modes which are HDFS mode and local mode. HDFS mode is designed to provide scalable storage space for applications requiring large amount of storage space. For this mode, it also provides storage service based on the underlying interfaces of HDFS. Compared with HDFS mode, HLBS could also provide local mode, which stores segment files in local file systems especially for system testing and debugging usage.

A. Random Read/Write of HLBS

HLBS supports random file access upon HDFS, which also has some advanced features like data compression, snapshot, cache, garbage collection and so on. HLBS data format has no large differences with common file system, both of which are made up of indirect block, inode, directory entry and such structures. In order to simplify implementation, HLBS just implements one file (one inode) which is split into several segments that are stored upon HDFS or local file systems. At anytime, there is only one active (latest) segment file. The segment file is organized as linear logs. When data is updated, there would be a new log appended at the end of latest segment to increase the possibility of sequential movement of disk head so that it could improve system throughput and performance. The new produced log includes five fields shown in Fig. 2. New data blocks are allocated and corresponding index addresses are updated in inode field so that random read/write is supported. Dirty data blocks would be recycled by garbage collector to save more storage space.

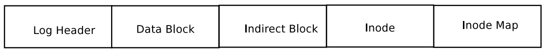


Fig. 2: Log Structure of HLBS

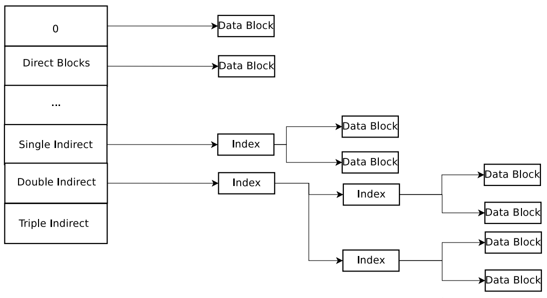


Fig. 3: Three Level Index Structure of HLBS

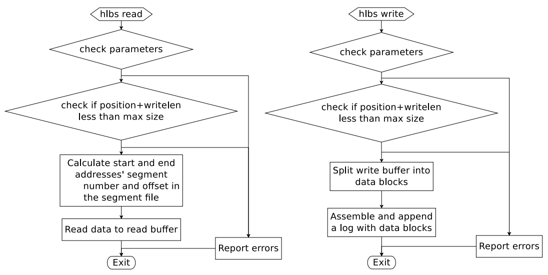
Fig. 2 shows the log structure of HLBS, which contains five fields.

1) *Log Header Field*
   Log header field describes the information of this log including log size, log creation time, data block amounts and so on.
2) *Data Block Field*
   Data block field stores user data.
3) *Indirect Block Field (see Fig. 3)*
   Indirect block field stores the address indexes of data blocks.
4) *Inode Field*
   Inode field describes the three indexes structure and file meta-data of HLBS.
5) *Inode Map Field*
   Inode map field describes the inode address and inode number.

Except data block field, other fields are all meta-data. Therefore, large data storage is more convenient for LFS. The inode data structure is shown as follows.

```
struct inode {
    /*data block is 8KB*/
    uint64_t length;
    /*time of last modification*/
    uint64_t mtime;
    /*8KB*12=96KB*/
    int64_t blocks[12];
    /*8KB/8*8KB=8MB*/
    int64_t iblock;
    /*8K/8*8K/8*8K=8GB*/
    int64_t doubly_iblock;
    /*8K/8*8K/8*8K/8*8K=8TB */
    int64_t triply_iblock;
}__attribute__((packed));
```

Any update operation would append a log in order semantics. The size of data block could be configured. Indirect blocks are index data blocks in inode. Every file has an inode so HLBS has just one inode and one inode map that could be used to find the inode location in a log. When reading the latest data, it firstly finds the latest inode map and then get the latest inode stored at the latest log in active segment file. At last, the addresses of data blocks in inode could be got from latest log. Fig. 4 indicates the flowcharts of HLBS read and write operations.


hlbs read
check parameters
check if position+writelen less than max size
Calculate start and end addresses' segment number and offset in the segment file
Read data to read buffer
Report errors
Exit
hlbs write
check parameters
check if position+writelen less than max size
Split write buffer into data blocks
Assemble and append a log with data blocks
Report errors
Exit
Fig. 4: HLBS read/write procedure flowchart

In Fig. 4, if HLBS works in HDFS mode, read/write requests would be delivered to HDFS layer. However, in local mode, requests would be sent to local file system layer.

B. Typical use case of HLBS

Fig. 5 illustrates a typical architecture of virtual machine back-end storage system constructed by HLBS. For each virtual machine, HLBS system is deployed to provide storage space so that one virtual machine could access 64 bits storage space at most. The 64 bits storage space of HLBS is split into

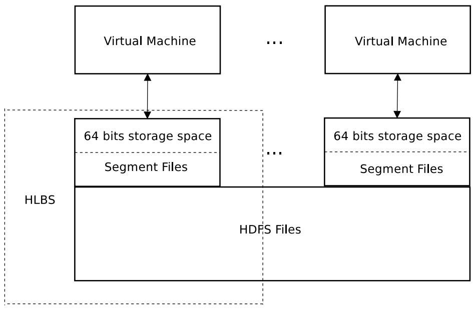


Fig. 5: Design Principle of HLBS

segment files that could be accessed with random read/write semantics. All the segment files are stored into HDFS storage space pool or local file system. The internal structure of HLBS could be divided into two parts as follows.

- *Log-based Block System (LBS) part* implements the log-based structure mechanism and takes the role to translate 64-bit linear addresses into segments (since log structure is too small to manage and recycle so we split storage space into many segment files), which are managed by HLBS upon HDFS storage pool or local file system.
- *Hook HDFS part* realizes HLBS underlying interfaces which support basic file operations about HDFS storage pool or local file system.

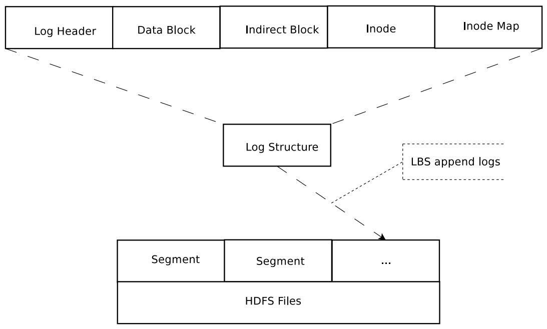


Fig. 6: Log-Structured Block Storage System

In Fig. 6, we show the architecture of LBS. In LBS, log is the update unit. When a update operation arrives at LBS, all data of the update request would be packed into a log. Then LBS will append the log into the end of the latest segment file, which is a HLBS file stored in HDFS storage pool or local file system. If a write operation updates one data block that has been created in an existing log, LBS will allocate a new data block and reorganize data between the old data block and new allocated data block in memory. Meanwhile, the new data block will be written into a new log and then LBS updates the corresponding index mapping table, which replaces old data block address with new data block address. At last, the new log would be appended into the end of latest segment file in HLBS. To recycle these invalid blocks, garbage collection mechanisms have been implemented for HLBS.

Although HLBS solves the once-write-many-read problems in HDFS, it would consume extra storage space for storing meta-data in logs. However, with support of the great scalability, performance and lower cost of HDFS and garbage collection feature of HLBS, this will not be a problem any more. We also implement a optional block-level compression feature to save more storage space. In addition, snapshot and cache features are also supported in HLBS to enhance the reliability and I/O performance.

C. HLBS snapshot

HLBS has two types of snapshots which are linear snapshot and tree snapshot. HLBS stores the address of its file inode in the latest log into a snapshot. With the idea of LFS, each appended log is a natural linear snapshot. On the contrary, users could take snapshots with specific names of any time manually, which is named tree snapshot. When users want to roll backward or forward to a time point, HLBS locates the snapshot according to the mapping between the time point and the snapshot name in each snapshot and then gets out the inode information to access required data sets. Snapshot feature of HLBS supports to recover corrupted or mis-operated data by rollback and forward recovery operations [14]. LFS has natural snapshot feature because of logs are appended sequentially. We just need roll backward or forward to the corresponding log to recover data at that time. In theory, HLBS could recover data at any time point after HLBS is started. However, it may not save all the data because of garbage collection. Users are recommended to mark log that contains important data so that it would produce a snapshot that is recorded in HLBS tree snapshot. Fig. 7 and Fig. 8 show linear and tree snapshots separately. In these two figures, one square represents one snapshot and Tn is a snapshot that is created at time Tn.

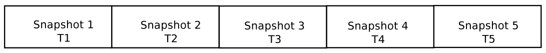


Fig. 7: An example of linear snapshot of HLBS

Tree snapshot could be produced at any time point like Fig. 8.

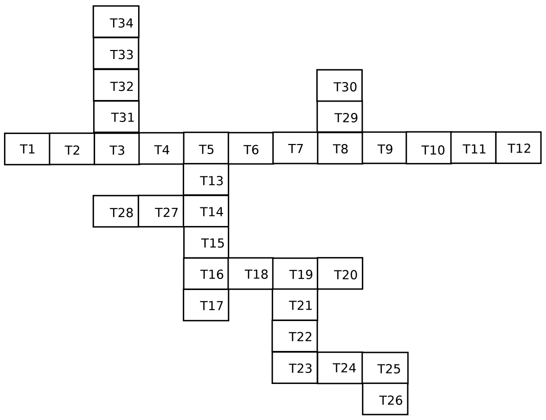


Fig. 8: An example of tree snapshot of HLBS

Snapshots in Fig. 8 are created in the following order:

1) Initialize HLBS, update online and create 12 snapshots ($T1$-$T12$), unmount HLBS.

2) Mount HLBS and rollback to the snapshot of $T5$, update online sequentially and create 5 snapshots ($T13$-$T17$), unmount HLBS.
3) Mount HLBS and rollback to the snapshot of $T16$, update online sequentially and create 3 snapshots ($T18$-$T20$), unmount HLBS.
4) Mount HLBS and rollback to the snapshot of $T19$, update online sequentially and create 3 snapshots ($T21$-$T23$), unmount HLBS.
5) Mount HLBS and rollback to the snapshot of $T23$, update online sequentially and create 2 snapshots ($T24$-$T25$), unmount HLBS.
6) Mount HLBS and rollback to the snapshot of $T25$, update online sequentially and create one snapshots ($T26$), unmount HLBS.
7) Mount HLBS and rollback to the snapshot of $T14$, update online sequentially and create 2 snapshots ($T27$-$T28$), unmount HLBS.
8) Mount HLBS and rollback to the snapshot of $T8$, update online sequentially and create 2 snapshots ($T29$-$T30$), unmount HLBS.
9) Mount HLBS and rollback to the snapshot of $T3$, update online sequentially and create 4 snapshots ($T31$-$T34$), unmount HLBS.

Tree snapshots could not only rollback/forward to any time's data sets but also update/take snapshot at any snapshot point without destroying existing snapshots. Linear snapshot is easy to design and implement because it is in chronological order, which is natural log. However, tree snapshot has to be designed and implemented with additional idea. In order to describe tree snapshot like Fig. 8, each snapshot needs to record its last snapshot name. Following data structure gives a simple model for the idea.

```
typdef struct snapshot{
    char up_ss_name[128];
    char ss_name[128];
}SNAPSHOT_T;
```

In brief, the key points of tree snapshot are how to maintain the relationship among snapshots and how to get the last snapshot name when producing a new snapshot. Our current solution is to record all the snapshot relationship in an external snapshot file in disk.

HLBS has three start modes with snapshot mechanism as follows.

- If HLBS starts without a snapshot, it would run based on the latest log. New snapshot would be added to the next place of the latest snapshot (like $T34$ in Fig. 8).
- If HLBS starts with a snapshot, it would take a new branch and update based on this branch (like $T8$ in Fig. 8). HLBS could get the snapshot according to user specified snapshot name from HLBS snapshot interface easily.
- If HLBS starts with a log between two continuous snapshots, that is to say, it would take a new branch and update based on this branch (HLBS starts based on a log between $T8$ and $T29$ in Fig. 8). HLBS has to get the latest snapshot name to do next operation.

In order to get the latest snapshot name for above first and third conditions, HLBS has to maintain an external file to record the latest alive snapshot name.

D. HLBS Garbage Collection

In order to manage and recycle HLBS storage space, HLBS storage space is splitted into segments. The size of a segment could be set according to user configuration. Each segment is mapped to a HDFS file and the format of segment file name is *segno.seg*. *Segno* is ascending from zero. HLBS has 64-bits storage space which are divided into two parts to represent segment number and offset within a segment respectively.

To verify whether a data block is garbage data, HLBS would get the valid index address of the data block from the latest log firstly. Then HLBS scans all segments one by one to check whether the latest address of the data block is the same as the old one. If they have different address, it says the old data block is dirty data block because the corresponding address has been changed.

The segment file could be removed if all the data blocks in it are dirty. In addition, if there are several data blocks are available in one segment file, copy and remove mechanism, copy valuable data blocks into a new log in the end of the latest segment file from a dirty segment and then remove the dirty segment, should be adopted. Garbage collection of HLBS works as following steps.

- *Segment usage calculation*
  During this procedure, HLBS Garbage Collector calculates the amounts of active data blocks in one segment file and save the results into a file named *segment_usage.txt*. There are two ways for segment usage calculation which are *pull mode* and *push mode*. *Pull mode* would pull segment files to local file system and then calculate all active data blocks. *Push mode* would adopt Hadoop MapReduce to run segment calculation concurrently.
- *Segment recycling*
  HLBS executes copy-remove or remove operation to recycle a dirty segment if the amount of active data blocks in one segment is more than a threshold. In order to prevent the problem of inode modification concurrently, segment recycling must be finished as a background thread daemon when there is no I/O request. Garbage collection has to be executed between two snapshots separately when snapshot mechanism is enabled.

HLBS supports two types of Garbage collection, which are *on-line* and *off-line* garbage collection. *On-line* garbage collection is provided by a daemon thread once HLBS starts. And *off-line* garbage collection is executed as an external tool.

III. Evaluation

In this section, we first present HLBS's experimental setup. Then we evaluate the I/O performance among HLBS, HDFS and Sheepdog [3]. To make experimental results more comparable with other similar storage systems, we also implement some simple tools [15] to test I/O performance.

A. Experimental Setup

Experiments in this paper are executed upon Linux Kernel 3.2.0-23-generic (Ubuntu Linux version 12.04). In order to build HLBS, users have to install some dependencies. The core third part softwares are *GLib* [16], *Snappy* [17], *Log4c* [18] and so on. *GLib* is a cross-platform software utility library [16]. *Snappy* is a fast data compression and decompression library which is the foundation of HLBS block compression feature [17]. *Log4c*is a C-based logging library and all message logging of HLBS is based on it. HDFS is most important for HLBS which provides scalable storage pool for HLBS. These dependencies have to be installed before HLBS is deployed. Generally, file systems have to be formated correctly before they could be used. HLBS also needs a format tool and we have implemented a format tool named "mkfs.hlfs" for HLBS. The experimental system should be configured with the following format command.

mkfs.hlfs -u mode:path -b block_size -s segment_size -m storage_system_size

The *mode* argument could be set as *local* or *hdfs*. The *path* gives the storage location of HLBS. The *block_size* and *segment_size* are block and segment size of HLBS whose unit is byte. *storage_system_size* is the maximum size of the whole HLBS storage space whose unit is megabyte. After the command is executed, there would produce a file named *superblock* which records above meta-data of HLBS. The file would be used when HLBS is initialized. In the following sections, we would evaluate HLBS and similar storage systems with different data block size and write size.

B. Comparison with HDFS

In this section, I/O performance of HLBS and HDFS is evaluated. In order to compare HLBS with HDFS, two programs are realized to achieve this goal, which have the same logic shown in Fig. 9.

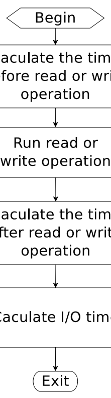

Begin
Caculate the time before read or write operation
Run read or write operation
Caculate the time after read or write operation
Caculate I/O time
Exit

Fig. 9: IO performance test tool flowchart

The IO time consumed in Fig. 9 is accurate because the unit of I/O time is microsecond. The I/O interfaces of HLBS and HDFS could be configured with different write size so that the comparison would be more comprehensive. The write and read experimental results are shown in Table I, Table II , Fig. 10 and Fig. 11.

Parameter	HLBS(MB/s)	HDFS(MB/s)	HLBS/HDFS
BS=8KB, SIZE=500MB	96.60	121.30	79.34%
BS=8KB, SIZE=100MB	81.85	101.02	80.20%
BS=8KB, SIZE=50MB	66.13	70.21	94.19%
BS=8KB, SIZE=30MB	59.57	70.67	84.29%
BS=8KB, SIZE=10MB	34.46	37.79	91.90%
BS=8KB, SIZE=1MB	5.69	14.68	35.71%

TABLE I: Write comparison between HLBS and HDFS

Parameter	HLBS(MB/s)	HDFS(MB/s)	HLBS/HDFS
BS=8KB, SIZE=500MB	103.66	148.40	70.00%
BS=8KB, SIZE=100MB	115.24	138.88	83.33%
BS=8KB, SIZE=50MB	168.97	123.73	136.59%
BS=8KB, SIZE=30MB	143.54	101.50	141.58%
BS=8KB, SIZE=10MB	310.69	75.26	413.33%
BS=8KB, SIZE=1MB	257.00	21.08	1223.80%

TABLE II: Read comparison between HLBS and HDFS

Table I illustrates that when block size is 8KB, HLBS average write performance could achieve HDFS's 79.34% 94.29%. According to Fig. 10, we could also find that HLBS and HDFS almost have the same write I/O performance between 0 50 megabyte per second. However, HDFS would have better write performance after 50 megabyte. In Table II and Fig. 11, we could find read performance of HLBS is much better than HDFS between 0 100 megabyte. However, HDFS and HLBS have the same throughput when the read size is more than 100 megabyte. In summary, I/O performance of HLBS could achieve HDFS's 79.34% 94.29% around, which we satisfy these results because we add the random read/write feature and a new layer upon HDFS.

C. Comparison with Sheepdog

In this section, both the IO efficiency and IO time are evaluated for HLBS against Sheepdog. Sheepdog is a distributed storage system for QEMU/KVM, which provides highly available block level storage volumes that can be attached to QEMU/KVM virtual machine [3]. Sheepdog is based on Corosync [19], which would simplify its design and implementation. However, it would increase Sheepdog's the optimization complexity. Meanwhile, HLBS is based on HDFS which is more popular than Sheepdog's Corosync. Sheepdog has no online data migration which limits the availability of itself.

Table III, Table IV , Fig. 12 and Fig. 13. show the I/O performance between HLBS and Sheepdog.

From Table III and Fig. 12, we could find that Sheepdog's

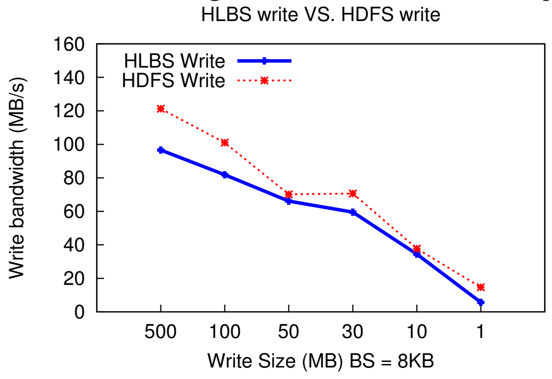


Fig. 10: HLBS write VS. HDFS write

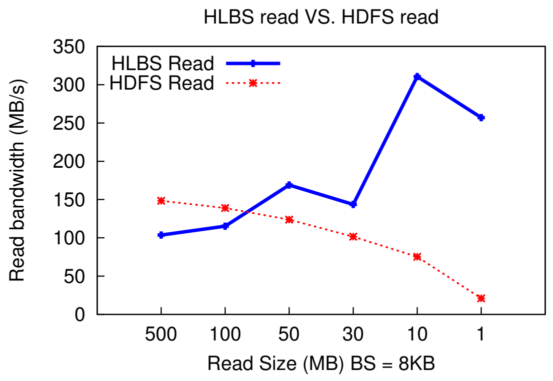


Fig. 11: HLBS read VS. HDFS read

Parameter	HLBS(MB/s)	Sheepdog(MB/s)
BS=8KB, SIZE=500MB	96.60	33.33
BS=8KB, SIZE=100MB	81.85	32.62
BS=8KB, SIZE=50MB	66.13	31.89
BS=8KB, SIZE=30MB	59.57	29.65
BS=8KB, SIZE=10MB	34.46	27.90

TABLE III: Write comparison between HLBS and Sheepdog

Parameter	HLBS(MB/s)	Sheepdog(MB/s)
BS=8KB, SIZE=500MB	103.66	0.91
BS=8KB, SIZE=100MB	115.24	0.95
BS=8KB, SIZE=50MB	168.97	1.06
BS=8KB, SIZE=30MB	143.54	1.06
BS=8KB, SIZE=10MB	310.69	0.99

TABLE IV: Read comparison between HLBS and HDFS

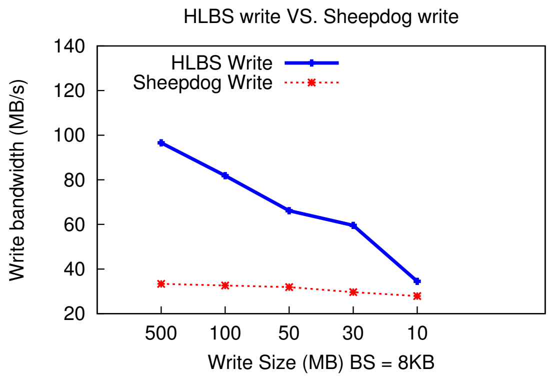


Fig. 12: HLBS write VS. Sheepdog write

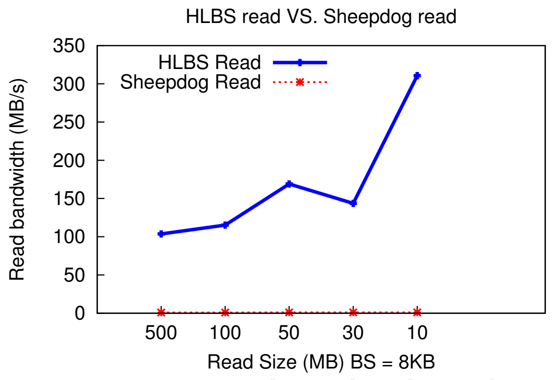


Fig. 13: HLBS read VS. Sheepdog read

write performance is stable but HLBS has better performance than Sheepdog. We could also find HLBS has better write performance when data is larger. Table IV and Fig. 13, indicate that Sheepdog has a worse read performance. In the same time, HLBS's read performance would be better when read size is lower. In summary, HLBS's I/O performance is totally better than Sheepdog.

IV. Related work

In order to widen HLBS's application scenarios, we implement several important patches [20] for HLBS to support many famous software such as *Openstack* [11], *network block device (NBD)* [10], *Quick EMUlation (QEMU)* [5], *Libvirt* [12], *Internet Small Computer System Interface (iSCSI)* [13] and *XEN* [21]. *Openstack* is a cloud computing project to provide an infrastructure as a service, which has three parts that are computing, networking and storage [22]. HLBS supports the *Cinder* subproject of *Openstack*, which is *Openstack* block storage. After HLBS is supported by *Openstack*, *Openstack* would have a strong back-end storage that have lots of advantages to store large data, which is inherited from the advantages of HLBS. *NBD* is a block device whose content is provided by a remote machine. *NBD* would be a cloud storage based network block device after HLBS is integrated, which would have a wider application scenarios. *QEMU* and *XEN* are two types of virtual machine. *QEMU* and *XEN* would support back-end storage system based on HDFS after HLBS supports them. In addition, the HLBS driver for *QEMU* and *XEN* realizes the separation between storage and computing so that it would enhance data availability. *Libvirt* is an open source application interfaces, daemon and management tool for managing platform virtualization. After *Libvirt* is supported by HLBS, *Libvirt* users would have the chances to build platform virtualization based on HLBS.

Log-Structured File System (LFS) is designed to improve I/O performance and data reliability. For LFS, each write operation produces a log structure, which should be appended sequentially when data is updated. The log may contain the updated data and meta data to maintain the system more effectively. Since the born of LFS, it has been used under many conditions. Unfortunately, there would be produced lots of meta data in Log-Structured File System so garbage collection would be important in such systems. Meanwhile, there is no time for disk seek when LFS handles I/O requirements. Hence, LFS's performance would be improved in the condition of big data writing and LFS also has other advantages like natural snapshot, quick indexing, and so on.

In the future, we would like to integrate HLBS into Google's Ganeti [23] project. The source codes of HLBS could be downloaded from [24].

V. Conclusion

HDFS is a distributed, scalable, and portable file system written in Java for the Hadoop framework. However, it just supports once-write-many-read semantics. This limits HDFS to be used in applications requiring random read/write semantics, such as back-end storage system for virtual machine and so on. In order to let HDFS support random read and write, we design and implement an infrastructure software HLBS which is based on HDFS and realize a log-structured like storage system [25]. HLBS inherits all the advantages of HDFS and LFS such as high availability, scalability, fault-tolerance and so on. HLBS has many wonderful features like snapshot, cache, garbage collection and block compression. It also supports many famous softwares including NBD, Openstack, Libvirt, XEN and so on. The average I/O time of HLBS could achieve HDFS's 79.34% 94.29%. Compared with Sheepdog, HLBS not only provides better I/O performance, but could also support more application scenarios.

VI. Acknowledgment

We would like to thank developers like Eric Blake at Redhat and Morita Kazutaka at NTT Lab who help us a lot when we develop HLBS patches for Libvirt and Sheepdog. Thanks for some anonymous reviewers for their insightful comments and detailed suggestions, which have substantially improved the content and presentation of this paper. We also thank Kang Hua for his suggestions and help for HLBS. This work is supported by the Natural Science Foundation of China (Grant No. 61272123).

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[3] K. Morita, "Sheepdog: Distributed storage system for qemu/kvm," *LCA 2010 DS&R miniconf*, 2010.

[4] S. Hazelhurst, "Scientific computing using virtual high-performance computing: a case study using the amazon elastic computing cloud," in *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*. ACM, 2008, pp. 94–103.

[5] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[7] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[8] "HLBS," *https://code.google.com/p/cloudxy/*.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[10] M. Lopez and P. Arturo Garcia Ares, "The network block device," *Linux Journal*, vol. 2000, no. 73es, p. 40, 2000.

[11] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: Toward an open-source solution for cloud computing." *International Journal of Computer Applications*, vol. 55, 2012.

[12] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehörster, and A. Brinkmann, "Non-intrusive virtualization management using libvirt," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 574–579.

[13] J. Satran and K. Meth, "Internet small computer systems interface (iscsi)," 2004.

[14] C. Lijun, L. Zhaoyuan, and J. Weiwei, "A snapshot system based on cloud storage log-structured block system."

[15] "I/O test tools for HLBS, Sheepdog and HDFS." *https://code.google.com/p/hlfs/wiki/HLFSIOPerformance*.

[16] "GLIB," *http://en.wikipedia.org/wiki/Glib*.

[17] "Snappy," *http://code.google.com/p/snappy/*.

[18] J. Lauret, G. Van Buren, and V. Fine, "Generic logging layer for the distributed computing."

[19] "Corosync," *http://corosync.github.io/corosync/*.

[20] "HLBS patches for famous software." *http://cloudxy.googlecode.com/svn/branches/hlfs/features/multi-file/patches/*.

[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of the 19th ACM symposium on Operating Systems Principles*, 2003, pp. 164–177.

[22] M. Mahjoub, A. Mdhaffar, R. B. Halima, and M. Jmaiel, "A comparative study of the current cloud computing technologies and offers," in *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*. IEEE, 2011, pp. 131–134.

[23] G. Trotter, "Ganeti: An open source high-availability cluster based on xen." in *LISA*, 2007.

[24] "HLBS source codes," *http://cloudxy.googlecode.com/svn/trunk/*.

[25] H. Kang., "Hlbs design document." *http://code.google.com/p/cloudxy/wiki/HlfsDesign*, 2012.