2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

VarFS: a Variable-sized Objects based Distributed File System

Yili Gong, Yanyan Xu
Computer School, Wuhan University
Wuhan, Hubei, China
{yiligong, xuyanyan}@whu.edu.cn

Yingchun Lei
Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China
leiyc@ict.ac.cn

Wenjie Wang
EECS, University of Michigan
Ann Arbor, MI, USA
wenjiew@umich.edu

***Abstract*—Cloud-based file systems are widely accepted and adopted for personal and business purposes in recent years. Statistics shows that approximately 25% of file operations from a typical user are random writes. Inherited from traditional disk-based file systems, most distributed file systems are also based on objects or chunks of fixed sizes, which work well for sequential writes but poorly for random writes. This paper investigates the design paradigm of variable-sized objects for a distributed file system. A novel distributed file system named VarFS, is presented to incorporate variable object indexing and support random write operations. VarFS reduces the amount of unnecessary data being read and the number of objects modified in face of updates and consequently alleviates the total amount of data transferred. The implementation is based on Ceph and the performance measurements show that it can achieve 1-2 orders of magnitude less latency than Ceph on random writes. At the same time the overhead for initial writes and re-writes is acceptable.**

***Keywords*—Distributed File System; Fixed-sized Chunk; Variable-sized Object; Random Write;**

I. Introduction

Cloud-based file systems and document storage services, such as Dropbox and Evernote, provide a reliable and convenient way for users to backup personal and/or working files in the cloud as well as update them when necessary. At the same time, they have become a base layer for a variety of applications. For instance, Facebook Messages (FM) stores its data in a distributed database (HBase [1]) atop a distributed file system (HDFS [2]). For another example, the maturity of desktop virtualization puts heavy reliance on large scale file systems.

The typical distributed file systems, like HDFS, are created to store large files in fixed sized chunks, typically 64 MB. Some file systems are based on objects in smaller sizes, e.g. 8 MB, like Ceph [3] and Lustre [4]. All these file systems follow the POSIX interface, and are optimized for sequential I/Os, such as sequential reading and appending. They assume that once created, files will rarely be modified by random writes. For a large number of applications, e.g. data analysis programs, archival data, or file processing, this assumption is valid and they are well supported by such file systems. Unfortunately for personal user file services and many other applications, the assumption does not hold. As a rule of thumb approximately 25% of a typical user's overall file access consists of random writes. 90% of the FM files are smaller than 15MB and the I/O is highly random according to [5]. Study [6] shows that VDI (Virtual Desktop Infrastructure) storage workloads are write-heavy and can take up 65% of I/O operations.

Using commonly accepted POSIX write semantics, when inserting a single byte into the midst of a file, a user has to read the data from the insertion point to the end of the file and write back the inserted byte together with the unaffected original ones. In distributed environments, where networks between users and storage servers are not always in the best condition, reading and writing unnecessary large amount of data chunks not only consume processing power on both clients and servers, but also suffer low throughput caused by network transmission overhead. The key reasons for such I/O behavior are 1) the absence of sufficient knowledge for file systems to identify unnecessary data transfer; and 2) the fixed chunk size forcing chunk re-mapping even with most content being unchanged.

Research on local file systems also try to adopt variable-sized blocks to counter the influence of fixed-sized schemes on writes. ZFS [7] manages its space with variable sized blocks in powers of two and the space for a single file is allocated with one block size. In local file systems like BTRFS [8], files are stored in extents, which hold the logical offset and the number of blocks used by this extent record. This allows performing a rewrite into the middle of an extent without having to read the old file data first.

This paper investigates the alternative design paradigm of variable-sized objects for a distributed file system, with the goal to reduce the amount of data transferred by random writes and consequentially improve the user experience. We proposes a new distributed file system, named VarFS, with a few key design novelties for variable chuck sizes, includes (1) mapping files into objects; (2) organizing metadata; (3) designing new read and write operation procedures. VarFS chooses to divide files based on their content, thus a change in the middle of a file generally does not impact the objects in the following part of the file. At the same time it provides a convenient way to identify each object by its content and thus can be used for global data de-duplication.

A prototype of VarFS is implemented and evaluated thoroughly for various file operations. With the new design, the amount of data transferred over the network is significantly reduced. The experiments show that the insertion, delete and random write performance of VarFS can outperform traditional

This work is supported by the National Natural Science Foundation of China under Grant No. 61100020 and 61373160, and Huawei Innovation Research Program.

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.54

file system by two orders of magnitude for large files. At the same time, the overhead for initial write and re-write is acceptable.

The outline of the paper is as follows. Section II explains the design of the key components of VarFS and the read/write protocols. Section III describes the implementation of VarFS. VarFS is evaluated and results are presented in Section IV. We review related work in Section V and conclude the paper in Section VI.

II. Design

The main design goal of VarFS is to allow file writes only impacting the directly modified objects or just a limited number of adjacent objects, which in turn leads to less data movement, and consequently brings the benefit of less replication updates and better performance in unfavorable wide area networks. With the new design, before a client submits some data for writing, it will try to re-group the data into proper objects and compare them for identical or unchanged ones that have already been in the server side. If the objects do not exist on server side, the objects will be written back. Otherwise they will be skipped. For example, a single byte is deleted from the very beginning of a file in a client's cache, the content of the object containing the deleted byte is changed, while the other objects in the file remain unchanged. Thus only the changed object should and will be transferred back to the storage servers for writing. In some cases, where the changed object happens to have the exact same content as another object in other files in the file system, this object does not need to be physically transferred back either. This is the benefit of content-based mapping and global de-duplication.

VarFS is compatible with standard POSIX interfaces to allow application compatibility. Existing applications can still benefit from the new design without modification. For example, in random write, the application may still use the traditional logic to read and write back the remaining part of modified file. VarFS will execute the read operations, but during write back, it may detect that after re-mapping, only the few directly impacted chucks are modified, and there is no need to write unmodified chunks back. Of course not all applications can benefit from this technique. The worst case scenario is when applications never insert or delete in the middle of a file. Nonetheless, VarFS provides significant bandwidth reduction for common mixture of write workloads.

In VarFS data are viewed in three tiers: the file tier, the object tier, and local storage tier, presented in Fig.1. Files are divided into variable-sized objects, objects are in flat structure logically and are stored as local files in local file systems. For the remainder of this section, we first discuss the overall architecture of VarFS. Then we introduce the mechanism to partition a file into variable-sized objects based on its content, called mapping in this paper. Since the object size is not fixed, it will be less straightforward to acquire the object ID or object location from the offset only. The metadata organization explains the mechanism. Last we show the process of file reading and writing in the actual VarFS protocols.

A. Mapping

One key design aspect of VarFS is to divide objects according to their content, instead of partitioning files into objects ad hoc or by positions in files. The main benefits include: 1) it keeps the boundaries between objects self-sustained by their content, i.e. an object is not or rarely affected by the modification of other objects in the file; 2) it becomes easy to tell if an object has been modified in a client and thus need be transferred back to object storage servers; 3) with these objects it provides a means to exploit similarities crossing different files, e.g. auto saved files in online backup file systems, object files output by continuous compiling, multiple revisions for a file in revision control systems, etc.

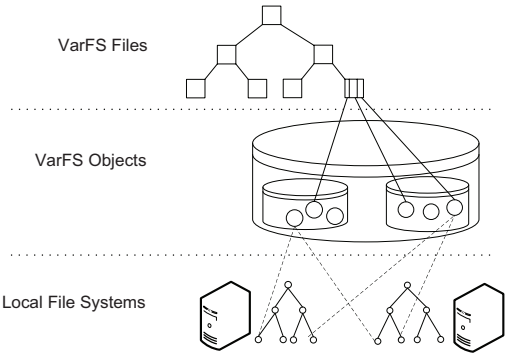

Fig. 1: The design architecture of VarFS.

We borrow the approach from [9] and use the Rabin fingerprint algorithm [10] to divide a file into variable-sized objects based on its content. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. For each object a hash value is computed by a SHA-1 hash function. It is the widely accepted practice of assumption that no hash collisions and two objects with the same hash value are considered identical. Based on the uniqueness of hash values, we can determine objects changed or not by comparing the ones in client-side cache with the original ones from servers.

With Rabin fingerprinting, a file is treated as a stream of bytes. VarFS examines every piece of data in a 48-byte window and with probability 2^{-n} considers it the end of a region as an object. The 48 byte ending zone of an object is called a *breakpoint*. The Rabin fingerprint of a window is calculated and if its low-order n bits equal to a chosen value, it is a breaking point. Here n is a pre-configured parameter, by which we adjust the expected object size correspondingly changes. Through experiments with various window sizes and ns, we find that 1) the window size has little effect thus it is set to 48 bytes; and 2) 1 MB mean object size provided best results thus n is set as 20. To avoid odd cases like too small or too large objects, VarFS defines the minimum and maximum object size, 2 KB and 8 MB respectively.

B. Metadata Organization

Metadata describe the organization and structure of a file system, usually including directory contents, file attributes, file block pointers, state information of physical space, etc. Each file has a corresponding $inode$ to record this related information. For simplicity and clarification, in this paper we focus on illustrating the aspects of the metadata design related to variable-sized objects. There are other equally important

parts not described here as they are out of the scope of this paper.

In VarFS, files are divided into objects that are stored as local files in local file systems. The core of file metadata is $inode$s which are organized as trees like in most file systems. Objects themselves are in flat structure and their metadata are well suited for key-value stores. The metadata for an object include its globally unique object ID, size, all storage locations, SHA-1 value and reference count. Every object is indexed by its SHA-1 hash and the key-value store maps these hash values to corresponding objects. In order to identify duplicated objects by their content quickly, SHA-1 values will be searched. The reference count is the number of files which contain the object. Each time an object is removed from a file, the object should be purged out from the file's inode, and its reference count should be subtract by one. When the reference count reaches down to zero, this object does not belong to any file and could be deleted immediately or cleaned up by a garbage collector later. Alternatively, in a multi-versioned file system, the deleted objects, even though no longer referenced, could be marked with version numbers for historical purpose.

Besides the common attributes, pointers to objects are also included. Keeping what is required for object locating in a file's inode will quicken metadata lookups. This brings the risk of potential inconsistency, but improves performance significantly, because if each file offset locating results in a database query it will be too expensive. VarFS adds necessary meta information of an object required for data position to its file's inode, including <object ID, start offset, end offset, location>. Accordingly with an offset in the file, it can be mapped to an offset in an object by simply looking up the inode. Locations in inodes can be considered as cache for the database and only when a request for a location fails, usually due to void or staleness, the up-to-date information will be fetched from database for future queries. If a user tries to modify the content of an object, a new object is always created and the count on the old one will be subtracted by one.

C. File Access

A VarFS client is installed on each host executing the application code and exposes applications with a POSIX-compatible file system interface. Each client maintains its own file data cache, independent of the kernel page or buffer caches, making it accessible to the applications that link to the client directly.

1) File Reads: When a user opens a file, the client sends the request to a metadata server. If the file exists and the access is granted, a metadata server traverses the file system hierarchy to translate the file name into a file handle. The handle corresponds to the file inode, which includes a unique inode number, the file owner, mode, size and other per-file metadata.

Fig. 2 shows how to read a file in VarFS. When a user acquires data from a file, the client sends a read request to a metadata server with the file handle, the file offset, and the requested size. The metadata server checks the permission for the read and if the request is valid. If so, it looks up the file's inode, searches the mapping for the objects that the requested data are mapped to, and then calculates object offsets

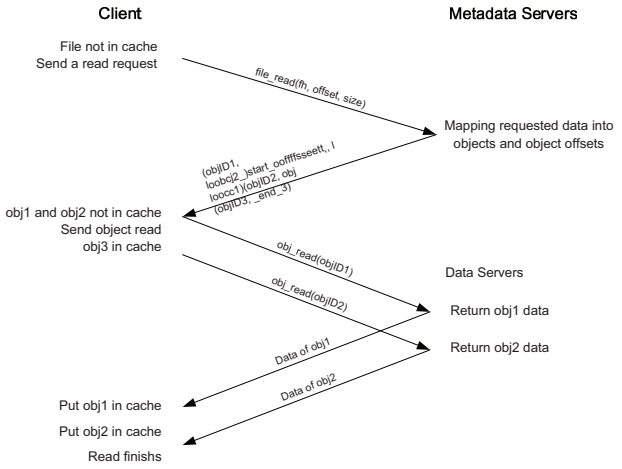

Fig. 2: Reading a file in VarFS.

for their file start offset and end offset, which are returned to the client with objects' locations. Upon receiving object and location information, the client will retrieve all returned objects from data servers that are not in local cache. An object's SHA-1 hash, in 20 bytes in VarFS, is attached whenever the object is retrieved from data servers. This design saves hash computation on clients and is convenient for hash comparison to detect object changing.

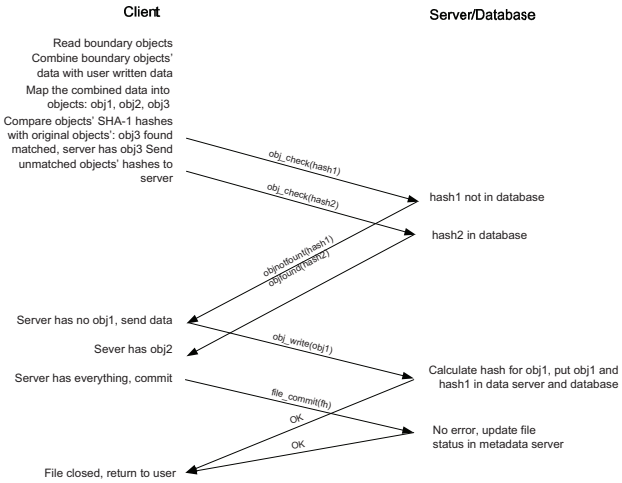

Fig. 3: Writing a file in VarFS.

2) File Writes: When a user executes a write request, the client locates the related boundary object(s), i.e. the object(s) containing the starting point, $offset$, and the ending point, $offset+nbytes$. If the boundary objects are not in the client's cache, the client will retrieve them from data servers and combine them with the written data from the user buffer. The Rabin fingerprint algorithm is applied to the combined data and mapped them into objects. Before the objects are written back to servers, VarFS checks their existence in the system by comparing their SHA-1 hashes with those of original objects for fast track and with ones in the database for slow track. Objects that are not in the system are tagged as dirty and waits to be written back to data servers. The file writing processing in VarFS is as shown in Fig. 3.

Actually any write can be transformed into a sequence of deletions and insertions. Only when a deletion or insertion involves boundaries of objects, multiple original adjacent objects will be directly changed. Otherwise only the object containing the modification point will be updated. It is rare that insertion may cause chain reactions to re-map multiple sequential objects because of the maximum size requirement in objects. Similarly, it is rare for deletion impacts multiple objects due to the minimum size requirement. In these cases, VarFS transfers more objects than the directly impacted ones, but it still saves more data movement than traditional distributed file systems.

III. Implementation

We implemented VarFS on the basis of Ceph, an open source distributed file system. Only the necessary parts are revised to implement our core design and made the system workable. Our prototype contains roughly 6000 lines of modified C++ code.

The overall architecture of VarFS implementation is shown in Fig.4. Applications access VarFS through FUSE and the kernel with the read and random write interface. Clients interpret requests and call corresponding processing procedures. File metadata servers store file related metadata and are responsible for answering corresponding requests. Object metadata servers are a key-value store holding object information and processing object adding/removing/duplication queries. OSD servers store objects and have their own local file systems.

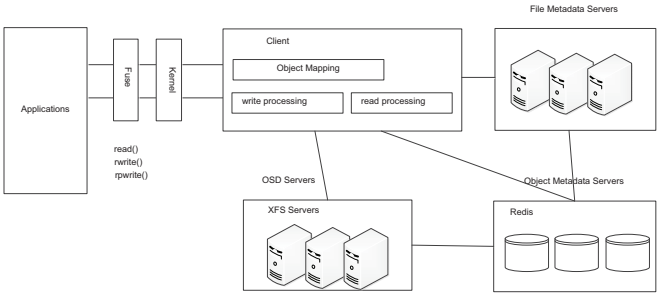

Fig. 4: Overview of the VarFS implementation.

In Ceph, every object has a globally unique object ID which consists of an inode number and an object number. An object is located by its object ID through CRUSH (Controlled Replication Under Scalable Hashing). In VarFS an object can belong to multiple files, thus an object ID should be a unique number independent of files. To exploit Ceph's code as much as possible, we set an object ID as a concatenation of an inode number and a global object stamp that will be incremented by one each time used. The inode is of the first file that introduces the object and even the object is deleted from this file it still remains; the object stamp will keep the uniqueness even after revisions. From the perspective of clients and metadata servers, the object storage cluster is viewed as a single logical object store. VarFS uses Ceph's Reliable Autonomic Distributed Object Store (RADOS) to handle object placement, object duplication, cluster expansion, failure detection and recovery. In VarFS, object storage at OSDs is the same with Ceph, except that VarFS attaches its SHA-1 value to the end of each object.

Before written to OSDs, a file has to be mapped into objects by Rabin fingerprinting. Through experiments it is found that this computation is quite time-consuming, thus we optimize it with multi-threading, specifically 2 and 4 threads. Additionally the client pipelines the Rabin fingerprint calculation with the network transfer whenever possible. We also optimize the data cache in clients to store variable-sized objects and their information for quick duplicate object identification. The object IDs, SHA-1 values and locations of recent objects are cached. Once a new object is created, the local cache is searched first. If the same object is found, the new object will be marked as clean, the new mapping information will be sent to the file metadata server, while the object data will not be transferred to OSDs. When there is no local match, the SHA-1 hash of the new object will be sent to the object metadata server for checking.

IV. Evaluation

performance of VarFS is extensively measured for its read and write performance. All the experiments are performed on a cluster which consists of machines with eight core 2.13GHz Xeon E5606 CPU, 45 GB memory, CentOS 6.3 (kernel version 3.2.51) and XFS as the local file systems. All hosts communicate using TCP over a Gigabyte network. An monitor and an MDS are installed on one server, which are responsible for collecting failure reports and managing metadata respectively. An OSD is installed on another server, which stores all file data. Dedicated machines are used as clients to generate workload and each can host tens to hundreds of client instances. the client-server bandwidth is set to 10 MB/s.

To avoid the accessing speed limitation of mechanical hard disks and the influence of prefetch mechanisms, we use RAMDisk to cache all data within the OSD memory. IOzone is used to generate and measure standard file operations, and we additionally implement the random write operations. All experiment results presented are obtained by averaging ten runs of each setting over ten 4 GB files.

A. Read, Initial Write and Re-write

Since we use RAMDisk for OSDs and all data are in memory, sequential reads and random reads perform almost the same in our experiments, thus only sequential read results are shown here. Writes in the Standard Posix I/O are similar to initial writes, so we only present the performance of initial writes due to the space limit.

Fig.5 shows the throughput of initial writes under different request sizes and mean object sizes. As the request size grows, the throughput increases, but when the request size reaches 8 MB, the predefined kernel page size, the throughput keeps steady, because a 16 MB request will always be split up into two 8 MB requests. We see that smaller mean object size always produces better performance, because an initial write is turned into several appending operations, each of which need retrieve the last object of the file, and the smaller mean object size leads to less data transferred. Smaller objects, on the other hand, generates a larger number of metadata, and reading or writing the same quantity of data causes more metadata accesses and object operations. In the following experiments, the mean object size is set to 1 MB.

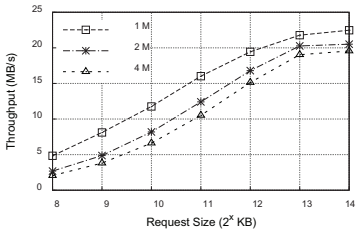

Fig. 5: The initial write throughput of VarFS.

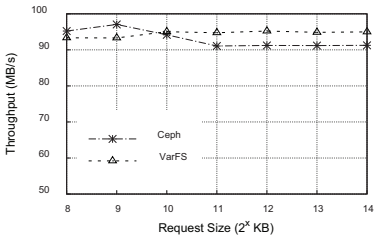

Fig. 6: The read throughput of VarFS and Ceph.

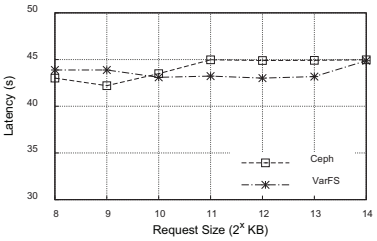

Fig. 7: The read latency of VarFS and Ceph.

By design, the read performance of VarFS should not be very different with that of Ceph. Our experiment confirms this expectation, shown in Fig.6 and 7. Fig.8 and Fig.9 demonstrate the initial write throughput and latency when the network bandwidth between clients and servers is set as 10 MB/s. Since initial writes in VarFS include object mapping computation, for a smaller request size the computation is dominant and as the request size increases, the network becomes saturated and the performance of VarFS and Ceph gets similar. On one side the overhead are amortized over the followup random writes operations; on the other side in real scenarios that the network between clients and servers is limited, VarFS can achieve comparable write performance with Ceph for larger request sizes.

B. Insertion and Deletion

Since insertion and deletion share similar system behavior, we only show the insertion results. Fig.10 plots the insertion latency of different inserted data sizes for both VarFS and Ceph in log-scale. The latency of Ceph is significantly longer than that of VarFS because not only the inserted data but the data from the inserting point to the end of the file need to be transferred. The larger the file, the more the unnecessarily data transmitted. For VarFS, the file size affects little on the latency, because most of the time only the object containing the inserting point and the inserted data is transferred. When the file size is 4 GB, the latency of VarFS is just 2% of Ceph. Even for a smaller file of 256 MB, the insert latency is only 45% of Ceph.

The reason of VarFS outperforming Ceph on operations is the amount of data transferred. VarFS transfers much less data than Ceph for insertion, only 1% to 9%, presented in Fig.11. This is particularly true with large files. Even though, for insertion operations, VarFS still computes the Rabin fingerprint of the inserted data, but the saving on data transferring outweighs substantially.

C. Random Write

The random write operation equals to deleting data from a file and inserting some other data to the same position. The latency shown in Fig. 12 confirms the observation from our design that VarFS performs considerably better than Ceph for random writes. The latency of Ceph increases dramatically with the file size, nevertheless, VarFS is quite inert to the file size and keeps the latency within 0.5 second or less. Fig. 13 show that in the worst case, VarFS's latency is about 25% of Ceph's result and its data transfer volume is about 5% of Ceph's. The performance becomes even more prominent with larger files. In the best case, VarFS's random write latency is only 2% of Ceph and its data transfer volume is about 0.5%.

V. Related Work

ZFS [7] is a filesystem originally developed by SUN™for its Solaris OS. In ZFS, space is managed with variable sized blocks in powers of two; the space for a single file is allocated with one block size. BTRFS [8] divides a file into extents, which are contiguous on-disk area, page aligned and of multiple page sizes. The concept of variable-sized extents inspired the surfacing of VarFS. In BTRFS, the logical offset and the extent size are stored together with data on disk, thus data insertion will cause all offsets in the following extents to change. VarFS separates the logical information and the physical storage to keep distributed data from changing as little as possible.

GFS [11] and its open source implementation HDFS [2] target to store very large files and adapted to sequential I/Os, such as sequential reading and appending operations. Files are divided into fixed-sized chunks, typically 64 MB, which are duplicated and distributed across chunk servers or data nodes. The file systems based on objects, like Ceph [3], Lustre [4], map file data onto a sequence of objects. The object sizes for different files theoretically may vary but in practice remain the same. It is by default set to 8 MB. VarFS distinguishes itself from these file systems in that files are mapped into variable-sized objects based on content, which additionally provides easy means for de-duplication.

Athicha Muthitacharoen et al. present LBFS for low-bandwidth networks, which exploits similarities between files to save bandwidth [9]. This paper is based on the concept of breaking files into variable-sized chunks onto distributed file systems in data centers as well as WANs to improve random write performance.

Besides file systems, database is another active battle field for data management. For record insertions and deletions, one branch of effort focuses on adopting auxiliary structures to divide updates into partitions by their storage locality and to organize random updates into disk friendly sequential accesses, e.g. [12] [13]. VarFS could benefit from these approaches in client or server cache to sort and combine updates before commit. Another branch is to crack database storage into manageable pieces, usually by key ranges, to reduce update maintenance effort [14] [15].

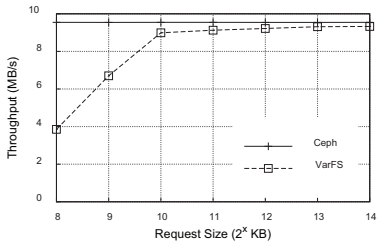

Fig. 8: The initial write throughput with limited network bandwidth.

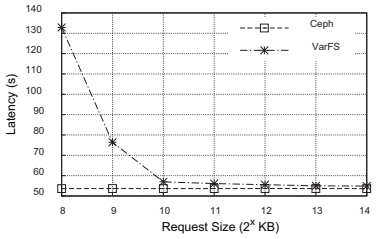

Fig. 9: The initial write latency with limited network bandwidth.

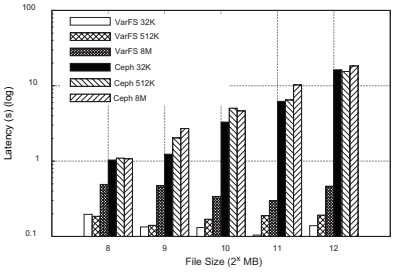

Fig. 10: The insertion latency of VarFS and Ceph (log).

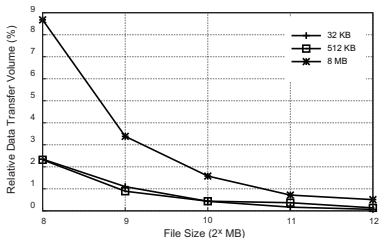

Fig. 11: The data transfer volume of VarFS over Ceph for insertions.

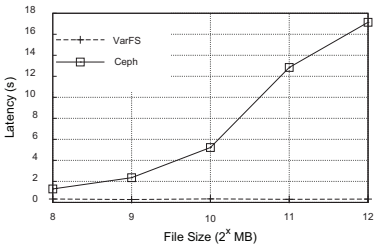

Fig. 12: The random write latency of VarFS and Ceph.

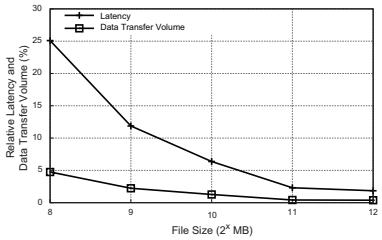

Fig. 13: The random write latency and data transfer volume of VarFS over Ceph.

VI. Conclusion

In this paper we have presented a variable-sized object based file system, VarFS, in which we choose mapping files into objects by their content. VarFS could minimize objects to be read and written back by keeping changes to minimal number of objects. We implemented VarFS based on Ceph and for random writes it achieves significantly better throughput and latency than Ceph by 50 times. As part of the future works, we will explore more efficient content-based mapping mechanism.

References

[1] "Hbase," http://hbase.apache.org/.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, Lake Tahoe, NV, May 2010.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, Nov. 2006, pp. 307–320.

[4] "Lustre," http://lustre.org/.

[5] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Santa Clara, CA, Feb. 2014, pp. 199–212.

[6] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield, "Capo: Recapitulating storage for virtual desktops," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*, San Jose, CA, Feb. 2011.

[7] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," in *the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[8] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," IBM Research Division Almaden Research Center and FusionIO, Tech. Rep., 2012.

[9] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Alberta, Canada, Oct. 2001, pp. 174–187.

[10] M. O. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Technology, Harvard University, Tech. Rep. TR-15-81, 1981.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, Oct. 2003, pp. 29–43.

[12] C. Jermaine, E. Omiecinski, and W. G. Yee, "The partitioned exponential file for database storage management," *The VLDB Journal*, vol. 16, no. 4, pp. 417–437, Oct. 2007.

[13] C. Jermaine, A. Datta, and E. Omiecinski, "A novel index supporting high volume data warehouse insertion," in *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*, San Francisco, CA, Edinburgh, Scotland 1999, pp. 235–246.

[14] S. M. Martin Kersten, "Cracking the database store," in *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, Jan. 2005.

[15] S. M. Stratos Idreos, Martin L. Kersten, "Database cracking," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*, Asilomar, CA, Jan. 2007.