Advance Access publication on 8 August 2016

doi:10.1093/comjnl/bxw057

A Distributed File System with Variable Sized Objects for Enhanced Random Writes

Yili Gong[1,2], Chuang Hu[2], Yanyan Xu[2] and Wenjie Wang[3*]

[1]*State Key Laboratory of Software Engineering, Wuhan University, Wuhan, Hubei, P.R. China*
[2]*Computer School, Wuhan University, Wuhan, Hubei, P.R. China*
[3]*EECS, University of Michigan, Ann Arbor, MI, USA*
[*]*Corresponding atuhor: wenjiew@eecs.umich.edu*

Cloud-based file systems are widely accepted and adopted for personal and business purposes in recent years. Statistics shows that ~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, where a new file write interface is proposed to provide rich write semantics. A novel distributed file system named VarFS, is presented to incorporate variable object indexing, support the random write interface and remain POSIX compatible. 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. VarFS is implemented 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

Received 12 December 2015; revised 5 July 2016
Handling editor: Mark Josephs

1. INTRODUCTION

Cloud-based file systems and document storage services, such as Dropbox [1] and Evernote [2], 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 [3], derived from BigTable [4]) atop a distributed file system Hadoop Distributed File System (HDFS) [5], derived from the Google File System (GFS) [6]). For another example, the maturity of desktop virtualization puts heavy reliance on large scale file systems.

The existed distributed file systems, like GFS and HDFS, are created to store very large files in fixed size file chunks, typically in 64 MB size. Some file systems are based on objects in smaller size, e.g., in 8 MB size[7], like Ceph [8] and Lustre [9]. All these file systems follow the Portable Operating System Interface (POSIX) interface [10] and are optimized for sequential I/Os, such as sequential reading and appending operations. They assume that once created, files will usually be read or written sequentially, and rarely be modified by random writes. Given this access pattern on huge files, appending is the main write operation that the systems are optimized for. 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 ~25% of a typical user's overall file access consists of random writes. Around 90% of the FM files are smaller than 15 MB and the I/O is highly random according to [11]. Study [12] shows that Virtual Desktop Infrastructure (VDI) storage workloads are write-heavy and can take up 65% of I/O operations. On the other side, in Hadoop for metadata management efficiency small files are merged into large files. Consequently any write with a size change to a interjacent file will incur change to the following small files. Though the modification can be kept alongside the original file and combined later for faster writing, it slows down subsequent reads and re-merging of the files can be time-consuming or even temporarily suspends the service.

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 original following content. This is reasonable and acceptable in face of directly accessible mechanical disks, which are organized by blocks or sectors. However in distributed environments where users and storage servers are only connected by local-area or even wide-area networks instead of buses, which are not always in the best condition, reading and writing back 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 (i) the absence of sufficient knowledge for file systems to identify unnecessary data transfer and (ii) the fixed chunk size forcing chunk re-mapping even with most content being unchanged.

Research on local file systems has been tried to adopt variable-sized blocks to counter the influence of fixed-sized schemes on writes. ZFS [13] 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 B-Tree File System (BTRFS) [14], files are stored in extents, which hold the logical offset and the number of blocks used by this extent record, which 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 propose a new distributed file system, named VarFS, with a few key design novelties for variable chunk size, includes (i) mapping files into objects, (ii) organizing metadata and (iii) designing a new write operation interface. 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. Additionally, we propose a new file operation interface for random writes, which keeps complete compatibility with POSIX and further allows users to specify the exact part of data being modified. The interface is powerful enough to express re-write, append, insert and delete operations, and consequently allows a file system better controlling the write behavior.

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 even if we remove bandwidth bottleneck, the random write performance of VarFS can outperform traditional file system by two orders of magnitude for large files.

In summary, the contributions of this work are: (i) a new POSIX-compatible file write interface, *rwrite()* and *rpwrite()*, is proposed for clients to enrich the application level semantics as well as providing file systems more information for optimization; (ii) a new file system based on variable-sized objects, VarFS, is designed and optimized for random writes and data de-duplication and (iii) the proposed interface and system are implemented and evaluated extensively.

The outline of the paper is as follows. The Posix interface is revisited and the new compatible random write interfaces are proposed in Section 2. Section 3 explains the design of the key components of VarFS and the read/write protocols. Section 4 describes the implementation of VarFS. VarFS is evaluated and results are presented in Section 5. We review related work in Section 6 and conclude the paper in Section 7.

2. FILE SYSTEM WRITE INTERFACE

POSIX [10] defines a standard operating system interface and environment, including a command interpreter (or 'shell') and common utility programs to support application portability at the source code level. The POSIX write interface has been followed and implemented by most distributed file systems of the present time. In spite of its popularity, the semantics of the POSIX write interface is too simple to provide sufficient information for file systems of the present time to take advantages of for optimization. After the analysis of the original interface, we explain the supplemental APIs *rwrite()* and *rpwrite()* that enrich the POSIX semantics and provide means to improve support for file systems.

2.1. POSIX interface revisited

According to the POSIX standard, when a process writes one or more bytes to a file, the system interface *write()* or *pwrite()* is called to transmit the data to the device.

- *ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);*
- *ssize_t write(int fildes, const void *buf, size_t nbyte);*

The *write()* function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*. On a regular file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with *fildes*. Before successful return from *write()*, the file offset shall be incremented by the number of bytes actually written. If the position of the last byte written is greater than or equal to the length of the file, the length of the file shall be set to this position plus one.

The *pwrite()* function shall be equivalent to *write()*, except that it writes into a given position and does not change the file offset. The first three arguments to *pwrite()* are the same as *write()* with the addition of a fourth argument offset for the desired position inside the file.

If a user wants to insert, delete or substitute some bytes with other bytes of a different size in the middle of a file, it has to rewrite the data from the modification point to the end of the file. It is reasonable in the days of mechanical disks, since files are stored in sequential blocks on disks and the bit change in the middle causes the bits after it to be moved along. However, for modern distributed file systems, files are divided into fixed-sized chunks and chunks are further stored in local file systems as files on storage servers. A small change in the middle of a file will lead to all subsequent chunks to be read, modified and written back, including their duplicated copies distributed in the whole storage system. This kind of design is no longer justified by current physical properties of storage devices, let alone the huge overhead it brings.

To change this situation, writes need to be specified in a way that provides more information to help a file system to understand the write behavior.

2.2. Random write

The behavior of the new *rwrite()* and *rpwrite()* APIs is random write, emphasizing writing at any position in a file, known from appending operations. The following definitions depict the semantics more accurately:

- *ssize_t rpwrite(int fildes, const void *buf, size_t nbyte, off_t offset, size_t mbyte);*
- *ssize_t rwrite(int fildes, const void *buf, size_t nbyte, size_t mbyte);*

A new parameter, *mbyte*, is added and represents the number of bytes will be overridden by the *nbyte* data in the buffer *buf* . The other parameters remain the same with the ones in the standard write interface. When *mbyte* is identical to *nbytes*, *rwrite()* degenerates to *write()*. Further on in this paper, we refer the behavior of *write()*, substituting exactly the same number of new bytes with the original ones, as re-write to distinguish it from the general term write. The new APIs are named by random write. It has been noticed that in other literature random writes have been referred in contrast to sequential writes. Here the term emphasizes writing at any position in a file known from appending operations only at the end.

If *mybte* is equal to zero, *rwrite()* acts actually as inserting *nbyte* data into the file at the position of the file offset and is an *insert* operation. If the offset is the end of the file, this operation is equivalently an *append*. If *mbytes* is greater than zero and *nbytes* equals to zero, it means to delete *mbytes* from the file, which is a random write operation. If *mbytes* and *nbytes* are both greater than zero and different from each other, it indicates the replacement of *nbytes* bytes original data by *mbytes* data from the buffer.

Though functionally speaking the new interface is capable of replacing the traditional one, it can also work as a supplement to the standard with full compatibility. We choose not to alter current standard POSIX APIs in order to maintain the compatibility with legacy applications. For these legacy applications, the traditional *write()* and *pwrite()* can be easily implemented with the new APIs by setting *mbyte* to *nbyte*. The transformation can be applied by an automatic tool to source codes before compilation or detecting and converting at the runtime system level. Such transition will be a per-application decision after evaluation of the benefit of the new system.

3. DESIGN

The new proposed random write interface enables applications to control write operations with more accuracy and flexibility. Given sufficient information, the new underlying design, named VarFS, allows file writes only impacting the directly modified objects or just a minimum 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's written content is finally submitted, the file system will try to re-group the data into proper objects and compare them for identical or unchanged ones that have already been on the server side. If they do not exist on server side, the objects will be written back, otherwise 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 intact. 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 advantage and our motivation for content-based mapping and global de-duplication.

VarFS is compatible with the standard POSIX interface to accommodate legacy applications. Unmodified applications can still benefit from the new design, but in a limited scale. For example, 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 a few chunks 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.

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 a flat structure logically and are stored as files in local file systems on storage servers.

For the remainder of this section, we first discuss the overall architecture of VarFS, then 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 VarFS protocol.

3.1. 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: (i) 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; (ii) it becomes easy to tell if an object has been modified in a client and thus need be transferred back to object storage servers and (iii) 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.

We borrow the approach from [15] and use the Rabin fingerprint algorithm [16] 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 [17]. 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.

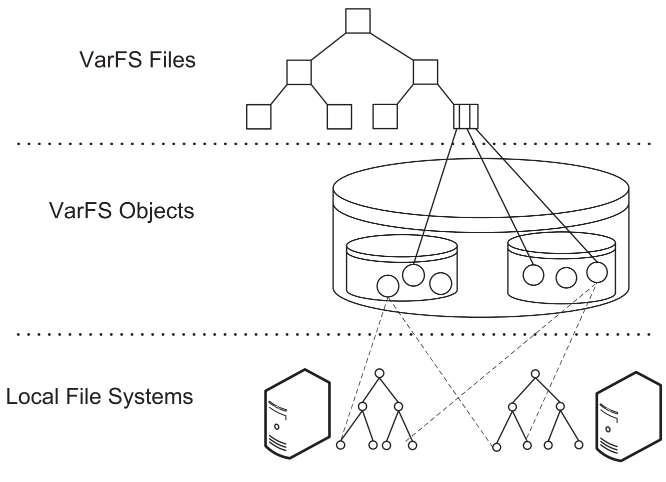

FIGURE 1. The design architecture of VarFS.

With Rabin fingerprinting, a file is treated as a stream of bytes. Starting from the very beginning of the file, every piece of data in a 48-byte window is examined and with probability 2^{-n} considers it the end of a region as an object. The Rabin fingerprint of a window is calculated and if its low-order n bits is not equal to a chosen value, the window slides a byte forward and continues to calculate. If yes, it is a breakpoint and a new object is recognized. Then the window slides 48 bytes starting right after the new object and the mapping process proceeds. Here n is a pre-configured parameter, by which we adjust the expected object size correspondingly changes. As will be discussed in Section 5.3, we experiment with various window sizes and ns, and find that (i) the window size has little effect thus it is set to 48 bytes and (ii) the mean object size of 1 MB provides best results thus n is set as 20.

It is possible that the approach should generate too small or too large objects, which will further incur metadata management inconvenience, network transfer inefficiency and data cache issues. To avoid such odd cases, VarFS defines the minimum and maximum object size, 2 KB and 8 MB, respectively.

3.2. Metadata organization

File's metadata describe the organization and structure of a file system. In VarFS files are organized as trees as in most file systems, further file data are divided into objects, which are in flat structure. The metadata of objects are suitable for a key-value store and object data are stored as local files in local file systems. For simplicity and clarification, 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.

Each file keeps its metadata in an inode, including an unique inode number, directory contents, file attributes, etc. Besides the common attributes, VarFS adds necessary meta information of objects required for data position, in format of <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. Pointers to objects are also included and can be considered as cache for the object database and only when a request for an object location fails, usually due to void or staleness, the up-to-date information will be fetched from database for future queries. 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.

The attributes of an object include its globally unique object ID, a size, an SHA-1 value, a reference count and storage locations. Object are also indexed by their SHA-1 hash values and a key-value store maps these hash values to corresponding objects. Though searching the SHA-1 values, duplicate objects can be identified. The reference count is the number of files, which contain the object. Each time an object is removed from a file, it should be purged out from the file's object list, and its reference count should be subtract by one. If a user tries to modify a shared object, a new object is created and the count on the old one decreases accordingly. 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, whenever an object is modified, a new version of this object is created. Meanwhile the deleted objects, even though no longer referenced, could be marked with version numbers for historical purpose.

3.3. File access

A VarFS client is installed on each host executing application code and exposes applications with the new proposed 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.

3.3.1. *File reading*

When a user opens a file, the client sends the request to a metadata server (MDS). If the file exists and the access is granted, a MDS 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, a object list and other per-file metadata.

Figure 2 shows how to read a file in VarFS. When a user acquires some data from a file, the client sends a read request to a MDS with the file handle, the file offset and the requested size. The MDS checks the permission for the read and if the request is valid. If so, it looks up the file's inode, searches the object list for the object(s) that the requested data are mapped to, and then calculates object offsets for their file start and end offset. The object ID(s) and object offsets are returned to the client together with objects' locations. Upon receiving the object and location information, the client will retrieve all mapped objects from data servers that are not in local cache.

An object's SHA-1 hash value, in 20 bytes in VarFS, is attached whenever the object is retrieved from data servers.

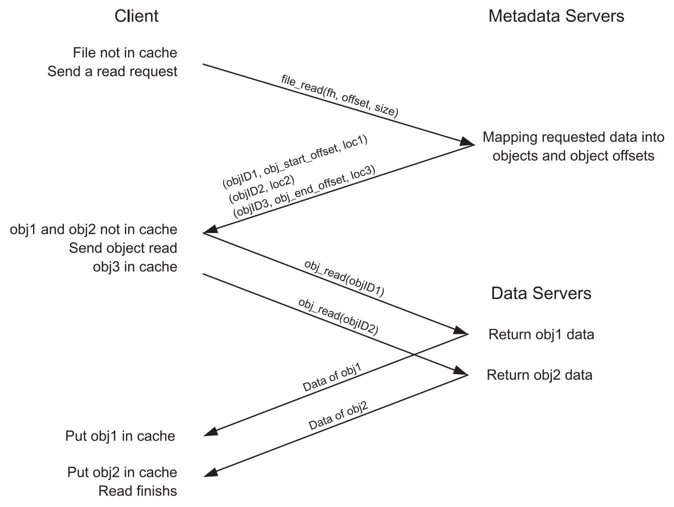

FIGURE 2. Reading a file in VarFS.

This design saves hash computation on clients and is convenient for hash comparison to detect object changing.

3.3.2. *File writing*

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, $offset + nbytes$, respectively. 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 new 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, two original 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.

4. IMPLEMENTATION

We implemented VarFS on the basis of Ceph [18], an open source distributed file system. We only revised the necessary

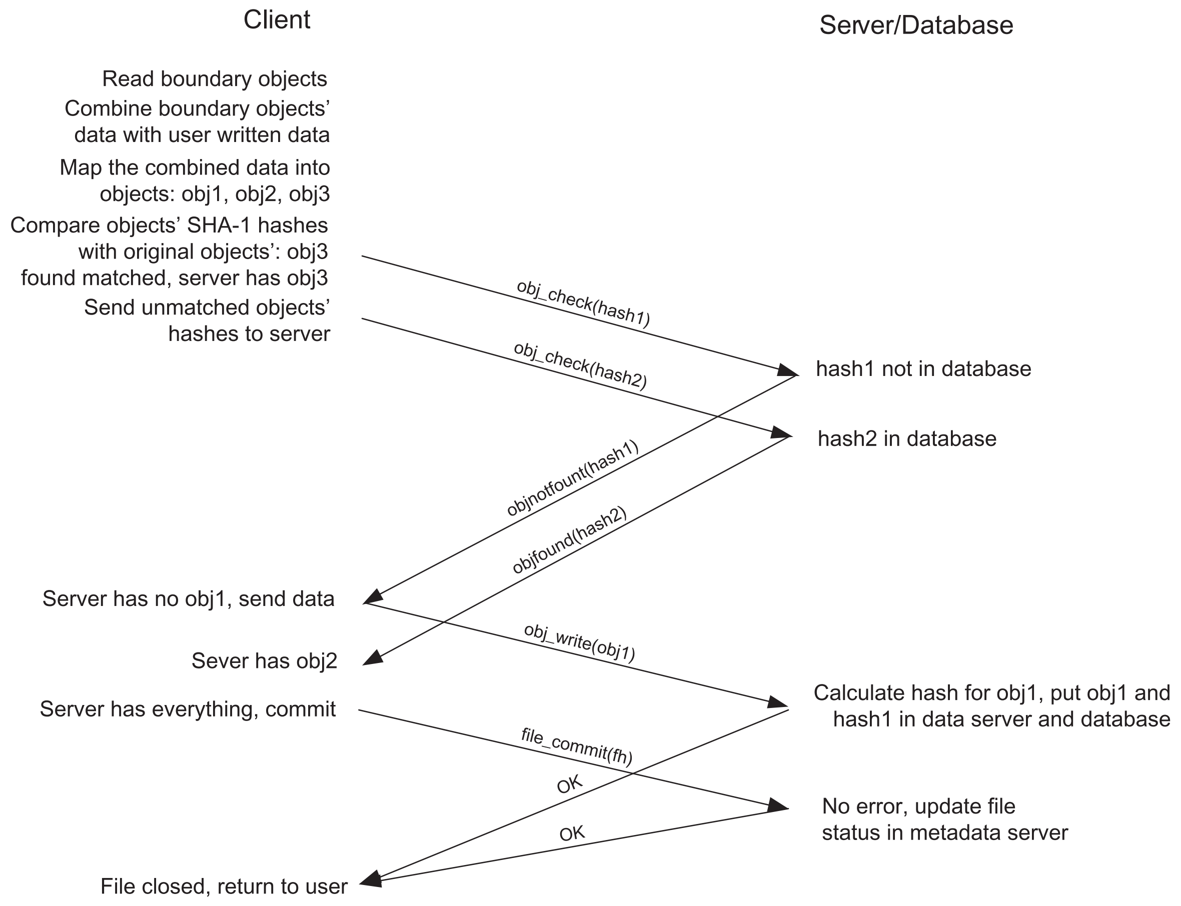
Client
Server/Database
Read boundary objects
Combine boundary objects' data with user written data
Map the combined data into objects: obj1, obj2, obj3
Compare objects' SHA-1 hashes with original objects': obj3 found matched, server has obj3
Send unmatched objects' hashes to server
obj_check(hash1)
obj_check(hash2)
hash1 not in database
hash2 in database
objnotfount(hash1)
objfound(hash2)
Server has no obj1, send data
Sever has obj2
obj_write(obj1)
Calculate hash for obj1, put obj1 and hash1 in data server and database
Server has everything, commit
file_commit(fh)
OK
OK
No error, update file status in metadata server
File closed, return to user

FIGURE 3. Writing a file in VarFS.

parts of Ceph 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 Filesystem in Userspace (FUSE) and the kernel with the read and the new random write interface. Clients interpret requests and call corresponding processing procedures. File MDSs store file related metadata and are responsible for answering corresponding requests. Object MDSs are a key-value store holding object information and processing object adding/removing/duplication queries. Object Storage Device (OSD) servers store objects, and have their own local file systems. We use XFS as the local file systems.

In Ceph, every object has a globally unique object ID, which consists of an inode number (a globally unique number for every file's inode) and an object number (an object sequence number in the file). 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 file revisions.

From the perspective of clients and MDSs, 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.

Our key modification lies in two components: the client library and the MDS.

4.1. Client implementation

The major revisions in a client are 3-folded:

(1) providing the new *random write* interface to applications;

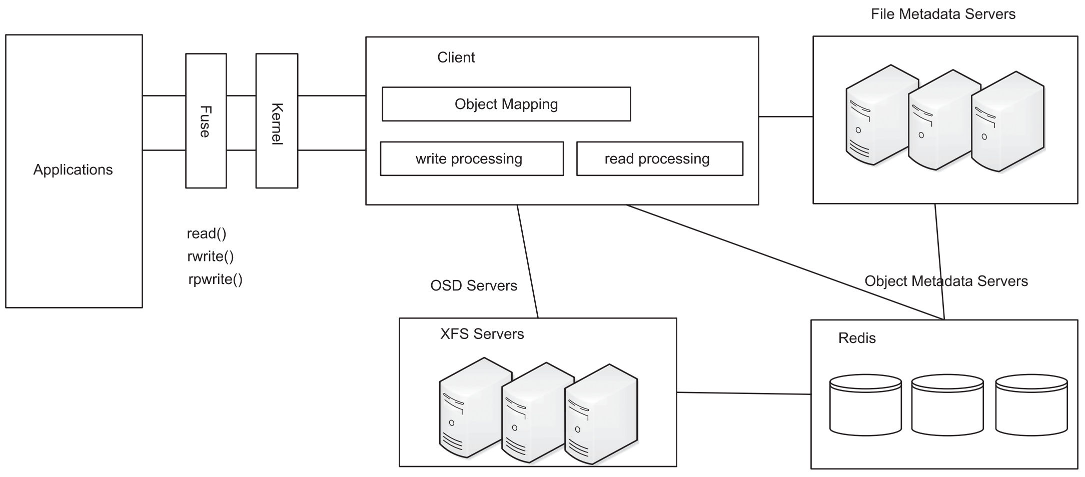

FIGURE 4. Overview of the VarFS implementation.

(2) adding a file mapping module using the Rabin fingerprint algorithm;
(3) implementing the read and write processing functions, including interacting with the file MDS and the object MDS (the communication with the OSD server remains the same as Ceph).

Before written to OSDs, a file has to be mapped into objects by Rabin fingerprinting. Through experiments we find that this computation is quite time-consuming. 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.

In a VarFS client, we also optimize the data cache 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 MDS, 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 MDS for checking.

4.2. MDS implementation

VarFS has its file metadata in a metadata cluster and its object metadata in a data store. The object metadata data store, implemented on Redis [19], records information about objects, answers queries on objects and processes requests for adding/removing objects.

The MDSs are based on Ceph's MDS. Besides the regular attributes, such as the size of the file in bytes, the user ID of the file's owner, etc., a VarFS *inode* includes a list of objects that the file is mapped to and each object information consists of:

- object ID,
- object size,
- SHA-1 hash value of the object.

One of the most important function is to determine the corresponding object by the requested file offset and data size. The MDS will traverse the object list to locate the offset. Though start and end offsets of objects are not recorded, since for insertion and deletion objects' positions in files may change, they can always be calculated by objects' position in file with object sizes. If a file is huge and mapped to a large number of objects, this traversal will take time. In this case, the techniques like index pointers could be adopted to expedite this lookup.

Different from Ceph, which determines object locations through CRUSH by object IDs, thus unnecessary to store locations, VarFS caches the previously fetched object locations in the metadata for fast object locating. Only when the location information is void or stale, the object metadata store is queried.

5. EVALUATION

A variety of factors play roles in the overall performance of a distributed file system. The performance of VarFS is extensively measured for its read and write performance. There are some tradeoff made for the experiments, which are explained in Section 5.2.

5.1. Experiment setup

All the experiments are performed on a cluster that consists of machines with eight core 2.13 GHz 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 Transmission Control Protocol 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. A VarFS client is accessed as a mounted file system via FUSE.

To avoid the accessing speed limitation of mechanical hard disks and the influence of prefetch mechanisms, we use RAMDisk [20] to cache all data within the OSD memory and disable the client-side prefetch feature.

IOzone [21], a widely used filesystem benchmark tool, is used to generate and measure a variety of file operations, and is used to evaluate the performance of VarFS. Besides standard reads and writes, we additionally implement the random write operation as described in Section 2.

We have no intention to investigate the de-duplication strength of VarFS since it mainly comes from the choice of Rabin fingerprint algorithm and has been discussed in [15]. Thus all experiment results presented are obtained by averaging 10 runs of each setting over 10 randomly selected 4 GB files instead of related ones.

5.2. Experimental choices and caveats

To evaluate the performance of the designed system in real deployment scenarios, we also simulate the case where there are limited bandwidth between clients and servers. Since the actual bandwidth and latency between clients and servers vary greatly, it is difficult to choose representative values for the evaluation. However, one conclusion is both intuitively and provably true, that is when the bandwidth between clients and servers becomes bottleneck, the amount of data transferred becomes the dominant factor in system throughput. In such scenarios, the design principle of VarFS, including content-based object mapping through Rabin fingerprints, with the goal to reduce network traffic, will benefit the most and present the best performance.

To evaluate the advantage and potential limitation of VarFS more thoroughly and systematically, in our experiments we intentionally eliminate the bandwidth bottleneck. We only present one case where the client-server bandwidth constraint is enforced, in order to demonstrate the rational and advantage of Rabin fingerprint mechanism in real world scenarios. In that case, the client-server bandwidth is set to 10 MB/s.

It is also possible to replace Rabin fingerprint algorithm with alternatives, such as random blocking and heuristic-based algorithms, to reduce the computational overhead. These alternatives are very welcome as long as they perform similarly well in limited bandwidth scenario. In most of evaluation, we use Rabin fingerprint to demonstrate that even with its relatively high demand on computation, VarFS still clearly achieves its desired performance target.

Additionally, early prototypes without extensive tuning, like the one being evaluated here, are hard to compete with mature commercial ready systems. To ensure fair comparison, we adopt similar implementation and optimization approaches to implement both the proposed design and existing systems. Similarly, the qualitatively trends of the measurements are more meaningful than the quantitative numbers, as they demonstrate the design efficiencies, particularly when they are orders of magnitude performance improvement.

Since existing file access traces are based on existing POSIX interfaces where redundant data are being transferred, we plan to collect real world random writes patterns once we have a more reliable implementation deployed. We have the Chinese national educational cloud platform for our experiment where over 100 servers and over 100 K clients can be leveraged when we have the reliable version ready. Currently, in our experiment, positions and contents of file operations data are randomly generated by the IOzone benchmark.

5.3. Read, initial write and re-write

We begin by measuring read, initial write and re-write performance. Since we use RAMDisk for OSDs and all data set is in memory, sequential reads and random reads perform almost the same in our experiments, thus only sequential read results are shown here. Initial writing is to create a file and keep on writing data of the kernel page size to the end of the file until completing the whole file. Writes in the standard Posix I/O are referred as re-writes to distinguish from our proposed random writes. The system behavior of re-write is similar to that of initial write without caching and prefetching and the actual results also confirm identical trend, thus we only present the performance of initial writes.

5.3.1. Mapping files into objects

In VarFS, files are divided into variable-sized objects based on their content. First of all, we investigate how the mean object size and the request size influence the performance of VarFS. Figure 5 shows the throughput of initial writes with the mean object size of 1, 2 and 4 MB.

As the request size grows, the throughput increases, but when the request size reaches 8 MB, the kernel page size that we use, the throughput keeps steady. Before hitting the page limit, the whole request could be sent from FUSE through the kernel to VarFS one-time, but a 16 MB request will always be split up to two 8 MB requests sent to VarFS, thus the performance for 16 MB is similar to that of 8 MB.

It can also be seen that smaller mean object size always produces better performance, because initial writing is actually implemented as a sequence of appending in the file system and appending need retrieve the last object of the file, consequently the smaller mean object size leads to less data transferred. On the other hand, smaller objects generates a larger number of metadata, and reading or writing the same quantity of data causes more metadata accesses and object operations. Table 1 shows object numbers generated with different mean object sizes for 4 GB files. Considering both application scenarios and performance, in the following experiments, we use 1 MB as the mean object size.

5.3.2. *Read and initial write*

It is important that the proposed new design does not introduce extra overhead for read operations. By design, the read performance of VarFS should not be very different with that of Ceph. Our experiment confirms this expectation that the throughput and latency of both VarFS and Ceph are comparable shown in Figs 6 and 7.

For initial write, a certain amount of overhead can be tolerated as additional logic is put in place to create various size objects. The overhead amortized over the followup random writes operations, should become minimal. We first evaluate VarFS with Rabin fingerprinting on real application scenario where there are limited bandwidth between clients and servers. Then we remove the bandwidth limitation and study the potential overhead generated by Rabin fingerprinting.

Figures 8 and 9 demonstrate the initial write throughput and latency when the network bandwidth between clients and servers is set as 10 MB/s. As has been explained in Fig. 5, a smaller request size incurs more appending operations, more redundant computation and more time consumed on computation. As the computation decreases with a larger request size, the network becomes saturated and the performance of VarFS and Ceph gets similar. In real scenarios that clients and servers are not int the same LAN, VarFS can achieve comparable initial write performance with Ceph for larger request sizes.

When the network bottleneck is removed, computing the data's fingerprints and mapping them into objects become the new bottleneck for initial writes. It is understandable that its performance is lower than that of Ceph, shown in Figs 10 and 11.

Figure 12 shows that with sequential single thread implementation, the computation dominates 82–91% of the write latency and the ratio increases with the request size. If we chose random object blocking instead of using Rabin fingerprints, the computational overhead will be void and the initial write performance of VarFS will become similar to that of Ceph. Even so we still believe Rabin offers greater advantage

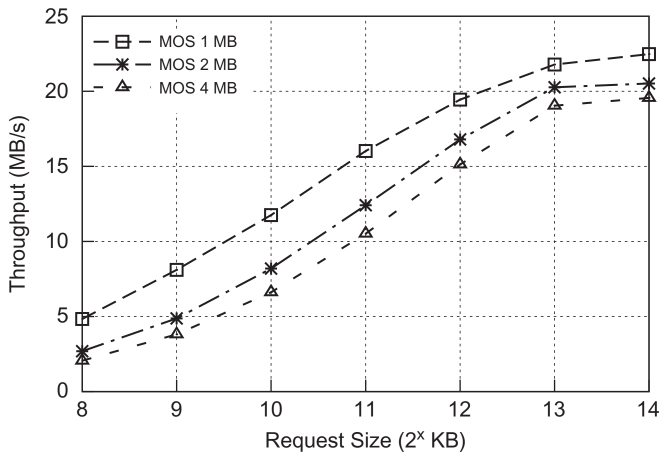

FIGURE 5. The initial write throughput with different mean object sizes and request sizes.

TABLE 1. Average object numbers of 4 GB files with varying mean object sizes.

	Mean object size			
Object number	512 KB	1 MB	2 MB	4 MB
	7236	3848	2182	1479

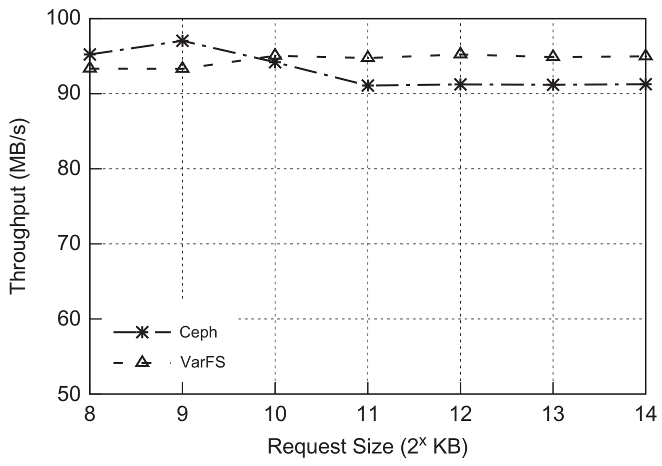

FIGURE 6. The read throughput of VarFS and Ceph.

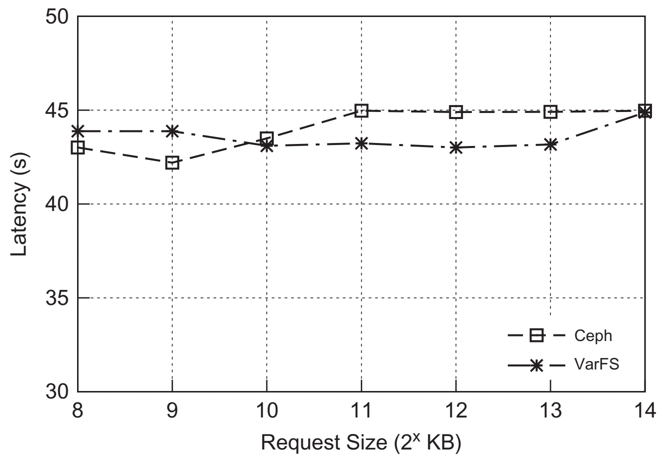

FIGURE 7. The read latency of VarFS and Ceph.

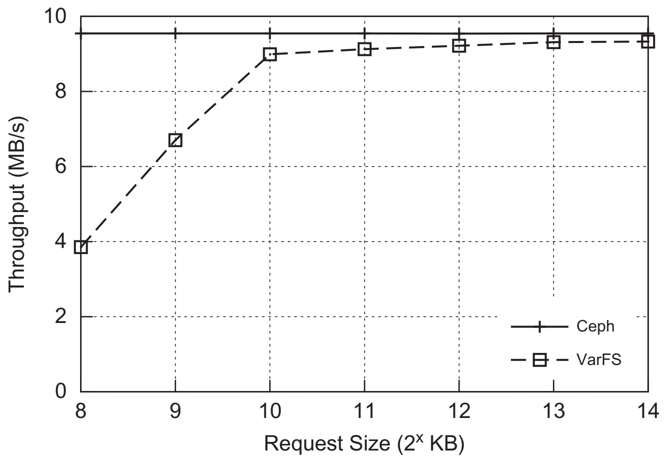

FIGURE 8. The initial write throughput with limited network bandwidth.

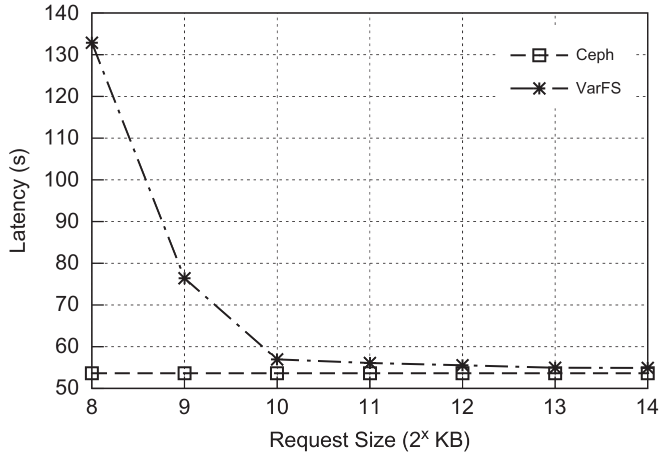

FIGURE 9. The initial write latency with limited network bandwidth.

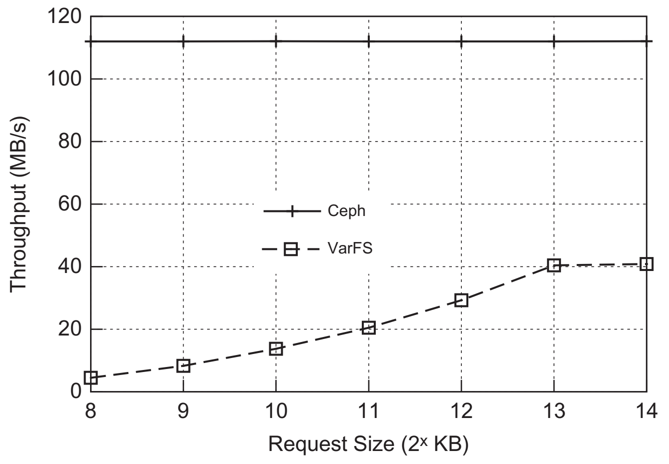

FIGURE 10. The initial write throughput without network bandwidth constraint.

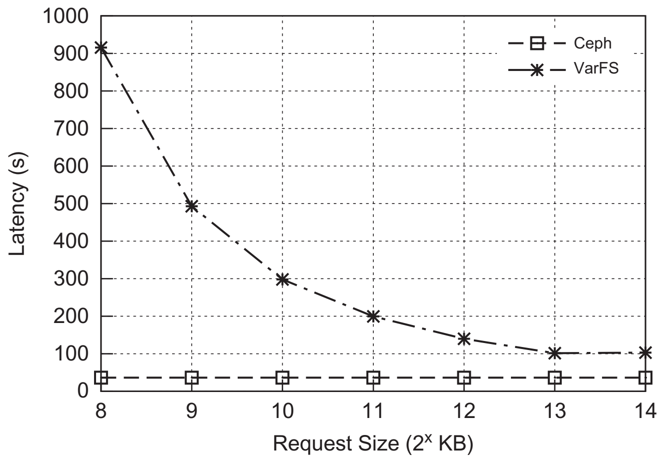

FIGURE 11. The initial write throughput without network bandwidth constraint.

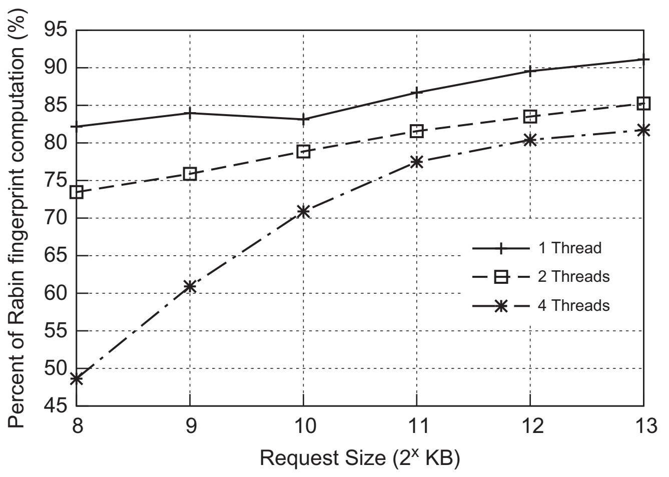

FIGURE 12. The proportion of Rabin fingerprint computation in the initial write latency.

in distributed client–server scenarios. Our experiments show that the additional overhead on initial write, amortized over the performance gain on random write, become very acceptable.

5.3.3. *Improving Rabin fingerprinting*

A number of techniques can be used to improve the computation latency incurred during fingerprinting calculation, such as hardware acceleration, or delayed writing where the server gradually applies the re-mapping in the background.

In this section, we study the effect of the parallelization.

We implement the algorithm in two and four threads and throughputs are shown in Fig. 13. When the request size is relatively small, e.g. 256 or 512 KB, multi-threading does not help much. However, for request size of 4 MB or larger, multi-threading outperform the single thread version by 48–

86%. Experiments show that with more threads and larger request size, the performance improvements are even more evident.

In spite of multi-threading, Rabin computation may still take up over 80% of the write latency for the 8 MB mean object size seen from Fig. 12. Further optimization on the Rabin fingerprint processing is one of our future works.

5.4. Insertion and deletion

Figure 14 presents the insertion latency of VarFS and Ceph. The insertion latency of Ceph increases dramatically when the file size increases 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 transmitted data. For VarFS, the file size affects little on the latency, because only the object containing the inserting point and the inserted data is transferred. As expected, VarFS's insertion latency increases slightly with the increasing size of inserted data. 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 1–9%, presented in Fig. 15. 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.

In VarFS, deleting some data from a file is to retrieve the affected objects, delete the data and then merge the remaining data, and write the new object to data servers. While for Ceph it is to rewrite the file's data from the starting point of deletion to the end. If the deleted data are at the front part of the file, almost the whole file will be rewritten, which costs huge overhead.

The deletion latency of VarFS and Ceph is presented in Fig. 16. The deletion latency of Ceph increases dramatically when the file size increases, while VarFS is less affected and the latency remains stable with various file size. With file size of 4 GB, the latency of VarFS is only 1% of Ceph, and for 256 MB it is 16%. It is obvious that the performance of VarFS benefits prodigiously by leaving the file data after the deleted area untouched.

5.5. Random write

The random write operation is to delete data from a file and insert other data to the same position. In Ceph, it writes the file's data to data servers from the deleting point to the end in addition to the new data. While for VarFS, it writes the objects containing the start point and the end point of the deleted data besides the inserted data.

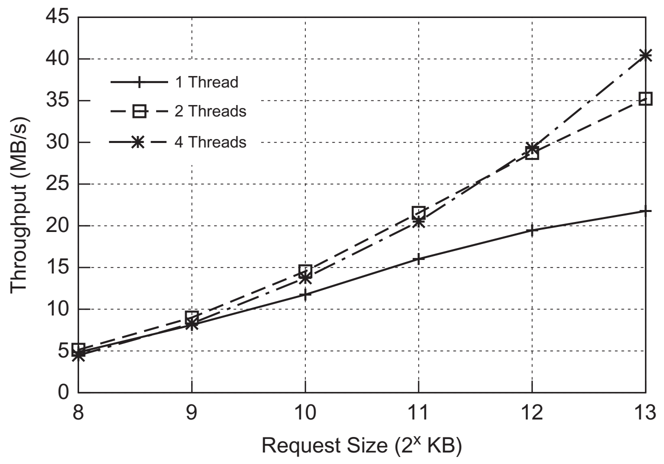

FIGURE 13. The initial write throughput with multi-threading.

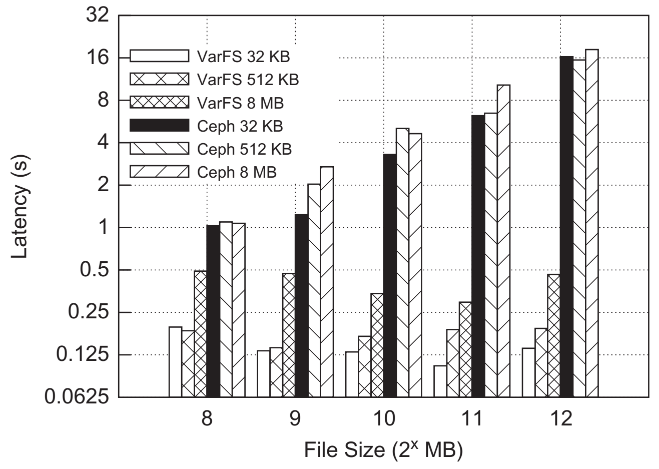

FIGURE 14. The insertion latency of VarFS and Ceph.

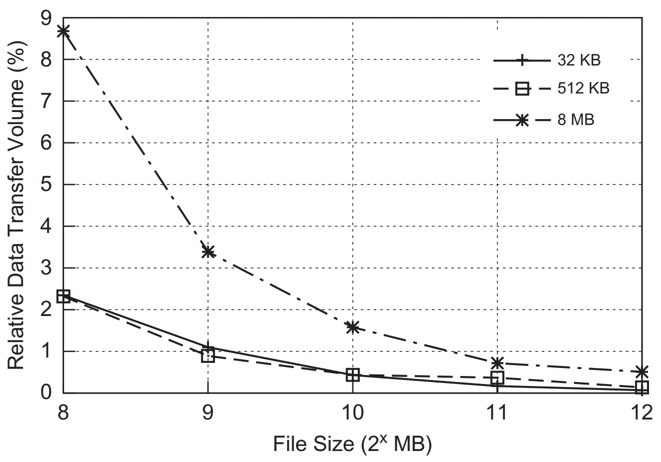

FIGURE 15. The data transfer volume percentage of VarFS over Ceph for insertions.

Figure 17 shows the total amount of data transferred of VarFS and Ceph in the random write. For writing the same amount of new data, Ceph transfers significantly more data than VarFS, by 30–500 times. Those additional data transferred are moving existing file data around, which waste network and disk bandwidth.

The latency shown in Fig. 18 shows the latency of Ceph increases dramatically with the file size. This is because large amount of network bandwidth is wasted on moving existing file data around, increasing the time used for the write operation. VarFS is quite inert to the file size and keeps the latency within 0.5 second or less.

Figure 19 shows that in the worst case, VarFS'(s) latency is ~25% of Ceph's result and its data transfer volume is ~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 ~0.5%.

5.6. OSD performance

To evaluate the performance of VarFS on OSD servers, we set up various number of clients to send insertion and random write requests to the system as fast as possible. These two types of operations are chosen because they are typical operations and our extensive experiments with other operations demonstrated similar results.

The insertion operation is to insert a block of 8 MB into a 1 GB file. Figure 20 shows the insertion throughput on OSD servers of Ceph and VarFS. The throughput of VarFS increases with higher number of clients, while that of Ceph remains unchanged. VarFS achieves the peak throughput as the server's network bandwidth is fully occupied. Since Ceph transfers a large amount of data even with a small number of clients to saturate the OSD's network, its throughput does not vary much. In the peak cases, VarFS performs ~100 times better than Ceph in throughput.

For random writes, *mbytes* (the deleting size) and *nbytes* (the inserting size) are generated randomly up to 8 MB. The results are shown in Fig. 21. The VarFS's throughput improves with the increasing number of client, and reaches the peak until the network becomes the bottleneck. Similar to the insertion case, Ceph's random write throughput remains unchanged. In the peak case, VarFS's throughput is ~70 times to Ceph. The random write throughput of VarFS is less than the insertion throughput, because deleting data consumes additional time.

6. RELATED WORK

Variable sized block based file systems. ZFS [13] is a filesystem originally developed by SUN^{TM}for its Solaris OS. In ZFS, space is managed with variable-sized blocks in powers

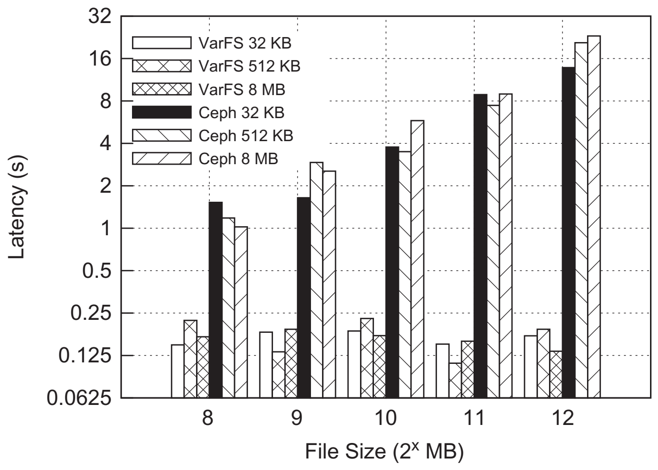

FIGURE 16. The deletion latency of VarFS and Ceph.

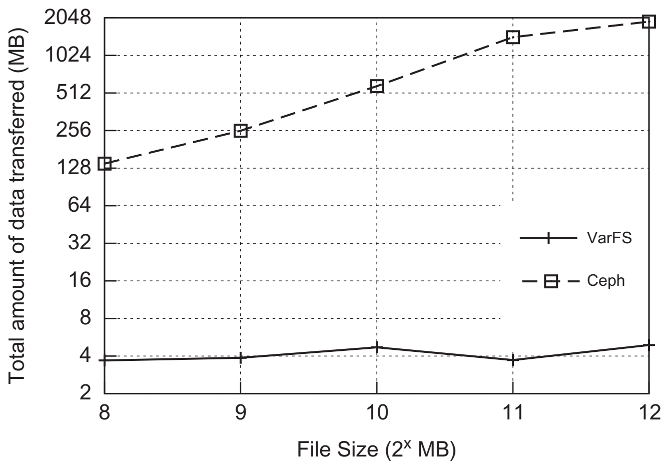

FIGURE 17. Total amount of data transferred of VarFS and Ceph in the random write.

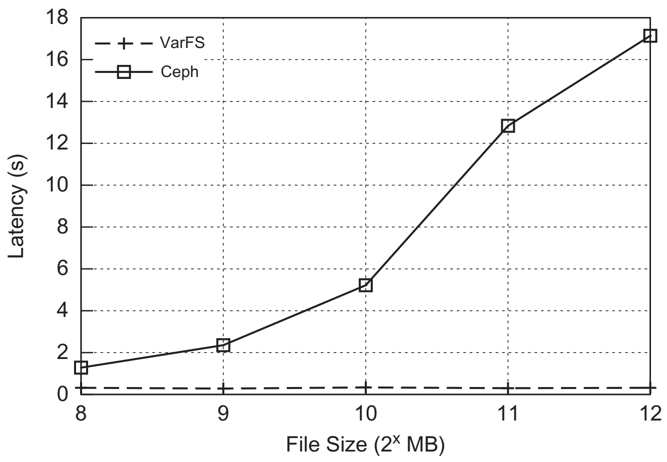

FIGURE 18. The random write latency of VarFS and Ceph.

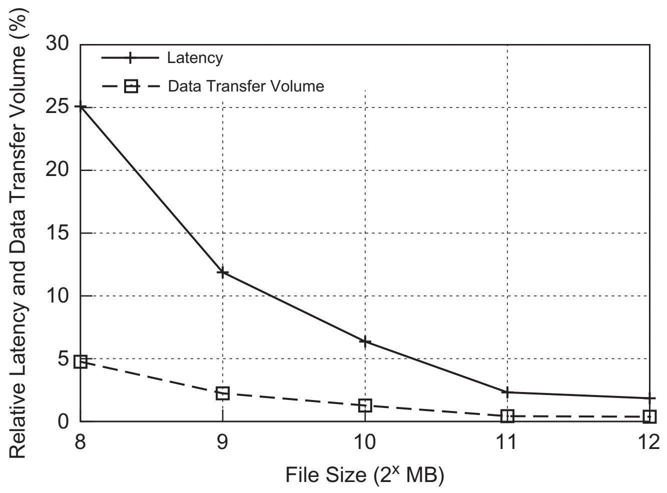

FIGURE 19. The random write latency and data transfer volume of VarFS over Ceph.

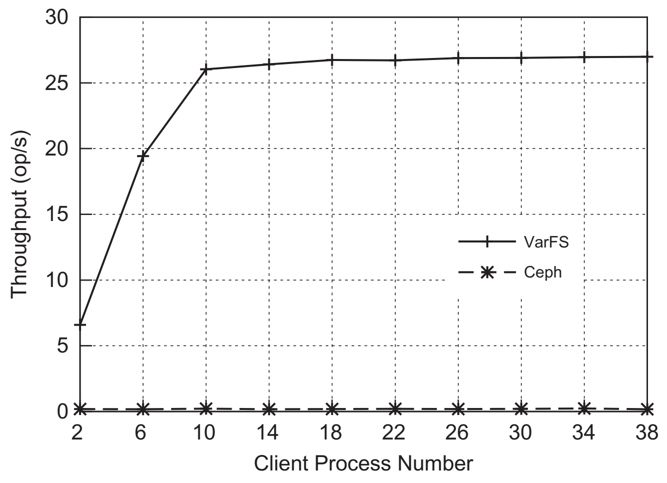

FIGURE 20. The insertion throughput of OSDs in Ceph and VarFS.

of two; the space for a single file is allocated with one block size. BTRFS [14] 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.

Fixed sized chunk-based distributed file systems. In past two decades, distributed file systems have gained their popularity in academia as well as industry. GFS [6] and its open source implementation HDFS [5] target to store very large files and adapted to sequential I/Os, such as sequential

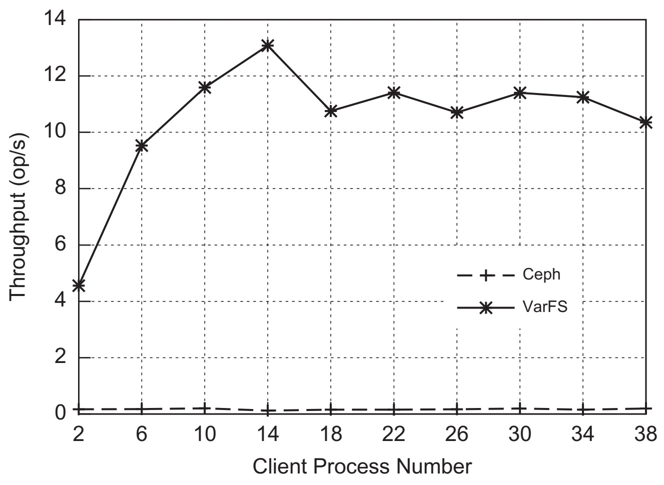

FIGURE 21. The random write throughput of OSDs in Ceph and VarFS.

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 [8], Lustre [9] and PVFS2 [22], 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 in Ceph [7]. For Lustre, even though a striped file can be stored in multiple object storage targets with different file block sizes. The block size is not variable within one stripe. PVFS2 provides message passing interface. It can be accessed like a POSIX system, but not fully POSIX-compliant. The data are accessed by fixed block size, where larger block size usually performs better.

IBM's General Parallel File System (GPFS) [23] is another widely used distributed and parallel file systems. GPFS provides a switching fabric to allow cluster nodes connecting to the file system distributed over thousands of disks. Large files are divided into equal-sized blocks, typical in size of 256 KB. GPFS is POSIX compliant.

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.

Muthitacharoen *et al.* [15] present Low-Bandwidth Network File System (LBFS) for low-bandwidth networks, which exploits Rabin fingerprints to detect similarities between files to save bandwidth. LBFS inspires us to apply 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. Another effort from HP Labs [24] focuses improving the content-based chunking algorithm for low-bandwidth network file system. The two thresholds and two divisors algorithm is proposed to choose the best chunk size. The chunk fingerprint calculation is also based on Rabin fingerprint algorithm. VarFS adopts the Rabin fingerprint approach to map files based on their content and share the benefits of easy identical object detection.

Database. 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. [25–27]. 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 [28–30].

7. 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. Combined with the new file write interface, VarFS could minimize objects to be read by write semantics by not transferring unchanged objects. At the same time, since changes in the middle of files do not impact sequential objects, object mapping of files becomes comparatively stable and could be used for data de-duplication. We implemented VarFS based on Ceph and for random writes it achieves better throughput and latency than Ceph by 50 times. As part of the future works, we will explore more efficient content-based mapping mechanism. Additionally, we will implement the file system prototype targeted for solid state drive, to evaluate actual application performance like VDI with large file rewrite requirements.

FUNDING

The National Natural Science Foundation of China (Grant No. 61100020 and 61572373), and Huawei Innovation Research Program.

REFERENCES

[1] Dropbox. http://www.dropbox.com.
[2] Evernote. http://www.evernote.com.
[3] Hbase. http://hbase.apache.org/.
[4] Chang, F. *et al.* (2006) Bigtable: a distributed storage system for structured data. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, November, pp. 205–218.
[5] Shvachko, K., Kuang, H., Radia, S. and Chansler, R. (2010) The hadoop distributed file system. In *Proc. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, Lake Tahoe, NV, May, pp. 1–10.
[6] Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003) The google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, October, pp. 29–43.
[7] Weil, S.A. (2007) Ceph: reliable, scalable, and high-performance distributed storage. PhD Thesis, University of California, Santa Cruz.
[8] Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E. and Maltzahn, C. (2006) Ceph: a scalable, high-performance distributed file system. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, November, pp. 307–320.
[9] Lustre. http://lustre.org/.
[10] Posix.1-2008. http://pubs.opengroup.org/onlinepubs/-9699919799/.
[11] Harter, T. *et al.* (2014) Analysis of HDFS under HBase: a Facebook messages case study. In *Proc. 12th USENIX Conf. File and Storage Technologies (FAST'14)*, Santa Clara, CA, Febuary, pp. 199–212.
[12] Shamma, M. *et al.* (2011) Capo: recapitulating storage for virtual desktops. In *Proc. 9th USENIX Conf. File and Stroage Technologies (FAST'11)*, San Jose, CA, Febuary, pp. 3–3.
[13] Bonwick, J., Ahrens, M., Henson, V., Maybee, M. and Shellenbaum, M. (2002) The zettabyte file system. Technical report. Sun Microsystems, Santa Clara, CA.
[14] Rodeh, O., Bacik, J. and Mason, C. (2012) BTRFS: the linux b-tree filesystem. Technical report. IBM Research Division Almaden Research Center and FusionIO, San Jose, CA.
[15] Muthitacharoen, A., Chen, B. and Mazières, D. (2001) A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Banff, Alberta, Canada, October, pp. 174–187.
[16] Rabin, M. O. (1981) Fingerprinting by random polynomials. Technical Report TR-15-81. In *Center for Research in Computing Technology*, Harvard University, Cambridge, MA.
[17] Burrows, J. H. (1995) Secure hash standard. Technical report. DTIC Document, Fort Belvoir, VA.
[18] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. (2006) CRUSH: controlled, scalable, decentralized placement of replicated data. In *Proc. 2006 ACM/IEEE Conf. Supercomputing*, Tampa, FL, November.
[19] Redis. http://redis.io/.
[20] Nielsen, M. How to use a RAMdisk for Linux. http : //www. linuxfocus, org/English/Novemberl999/articlel24 .html.
[21] Iozone. http://www.iozone.org.
[22] Team, P.D. (2003). Parallel virtual file system, version 2. http://www.pvfs.org/pvfs2/pvfs2-guide.html.
[23] Schmuck, F.B. and Haskin, R.L. (2002) GPFS: a shared-disk file system for large computing clusters.In *Proc. 1st USENIX Conf. File and Storage Technologies (FAST'02)*, Monterey, CA, January, pp. 231–244.
[24] Eshghi, K. and Tang, H.K. (2005) A framework for analyzing and improving content-based chunking algorithms. Technical report. Hewlett-Packard Labs, Palo Alto, CA.

[25] Jermaine, C., Omiecinski, E. and Yee, W. G. (2007) The partitioned exponential file for database storage management. *The VLDB J.*, **16**, 417–437.

[26] Jermaine, C., Datta, A. and Omiecinski, E. (1999) A novel index supporting high volume data warehouse insertion. In *Proc. 25th Int. Conf. Very Large Data Bases (VLDB'99)*, Edinburgh, Scotland, September, pp. 235–246.

[27] Jagadish, H.V. *et al.* (1997) Incremental organization for data recording and warehousing. In *Proc. 23rd Int. Conf. Very Large Data Bases (VLDB'97)*, Athens, Greece, August, pp. 16–25.

[28] Kersten, M. and Manegold, S. (2005) Cracking the database store. In *Proc. 2nd Biennial Conf. Innovative Data Systems Research (CIDR'05)*, Asilomar, CA, January, pp. 4–7.

[29] Stratos Idreos, S.M. and Kersten, M.L. (2007) Database cracking. In *Proc. 3rd Biennial Conf. Innovative Data Systems Research (CIDR'07)*, Asilomar, CA, January, pp. 1–8.

[30] Graefe, G. and Kuno, H. (2010) Self-selecting, self-tuning, incrementally optimized indexes. In *Proc. 13th Int. Conf. Extending Database Technology (EDBT'10)*, Lausanne, Switzerland, March, pp. 371–381.