PAPER

Distributed and Scalable Directory Service in a Parallel File System

Lixin WANG[†a)], **Yutong LU**[†], **Wei ZHANG**[†], *Nonmembers*, *and* **Yan LEI**[†], *Student Member*

SUMMARY One of the patterns that the design of parallel file systems has to solve stems from the difficulty of handling the metadata-intensive I/O generated by parallel applications accessing a single large directory. We demonstrate a middleware design called SFS to support existing parallel file systems for distributed and scalable directory service. SFS distributes directory entries over data servers instead of metadata servers to offer increased scalability and performance. Firstly, SFS exploits an adaptive directory partitioning based on extendible hashing to support concurrent and unsynchronized partition splitting. Secondly, SFS describes an optimization based on recursive split-ordering that emphasizes speeding up the splitting process. Thirdly, SFS applies a write-optimized index structure to convert slow, small, random metadata updates into fast, large, sequential writes. Finally, SFS gracefully tolerates stale mapping at the clients while maintaining the correctness and consistency of the system. Our performance results on a cluster of 32-servers show our implementation can deliver more than 250,000 file creations per second on average.

key words: *parallel file system, distributed and scalable directory service, concurrent and unsynchronized splitting, recursive split-ordering, write-optimized index structure*

1. Introduction

As computing breaches petascale and approaches exascale limits both in processor performance and storage capacity, the computing revolution has created growing storage challenges as traditional storage methods struggle to keep pace with the speed and parallel access demands of scalable compute environments. To reply to these challenges, a variety of parallel file systems are being employed to extract the highest possible performance from underlying storage hardwares.

File system designers have long sought to look for new architectures to improve and scale the performance. Over the last two decades, the predominant focus of current file system research, which is driven by application workloads, emphasizes access to large files and tends to be in the realm of access bandwidth and data redundancy. While providing scalable file I/O bandwidth to large files, most parallel file systems can not scale well to support efficient concurrent access to a single shared directory that stores millions to billions of files.

In this thesis, we present a prototype middleware file system called Strawberry File System (SFS) to support efficient concurrent access to large directories. An adaptive directory partitioning scheme based on extendible hashing is used to enable high concurrency for partition splitting while eliminating system-wide serialization and synchronization. To achieve higher concurrency and scalability, we distribute directory entries over a cluster of data servers instead of metadata servers. Directory partitions can dynamically grow with usage. We also avoid the expensive splitting process by employing a recursive split-ordering mechanism. In order to accelerate metadata updates on data servers, we use Log-Structured Merge (LSM) tree [1] to be an ordered, persistent index structure for metadata storage. Despite the stale copies of mappings in client caches, we ensure that the client's requests are forwarded to the correct servers with minimal communication and synchronization overheads. Finally, we verify the effectiveness of all these mechanisms in SFS.

In the rest of the paper, we present the motivation, background and related work in Sect. 2. Section 3 demonstrates the design choices of our distributed and scalable directory service, followed by evaluation results in Sect. 4. Section 5 concludes the paper.

2. Motivation, Background and Related Work

This section summarizes the motivation calling for scalable and high-performance directory service and shows how current file systems are ill-suited to scale.

2.1 Motivation

Modern parallel file systems are designed to adapt to various kinds of workloads. In addition to efficient operations in general cases, the file systems should additionally handle the extreme usage patterns which are common to scientific computing and general purpose workloads. Recently, many modern parallel file systems have adopted architectures which are based on closely-related object-based storage paradigm [2]. This compelling architecture diverges traditional storage architectures from decoupling metadata transactions from I/O operations and delegating low-level block allocation decisions to object-based storage devices. However, the best way to leverage this storage architecture is to perform high-bandwidth access to large files, rather than struggling with workloads involving access to large numbers of files.

While some applications have applied this I/O best practice and delivered scalable storage bandwidth, there are

Manuscript received January 15, 2015.
Manuscript revised July 11, 2015.
Manuscript publicized October 26, 2015.

[†]The authors are with the National University of Defense Technology (NUDT), China.
a) E-mail: wanglixin08@nudt.edu.cn

DOI: 10.1587/transinf.2015EDP7009

also a number of miscellaneous metadata-intensive applications, such as gene sequencing and photo storage [3], are not well served at very large scale. These applications apply an I/O pattern that creates a file per thread/process in a single shared directory that are accessed by large amounts of clients simultaneously. Another notable I/O workload in HPC clusters which creates similar directory access scenario is checkpointing that applications insulate themselves from inevitable system failures by periodically saving application states to persistent storage. Unfortunately, applications that adopt this per-process (or per-thread) checkpointing are subject to poor file create rates from the underlying file system which is optimized for large, non-shared file. In the impending exascale-era, applications running on computing clusters with up to billions of CPU cores may impose significant access challenges when checkpointing. As a result, what really mattered is to build a scalable directory service for parallel file systems to support efficient concurrent access to even larger directories in the future.

2.2 Background

Parallel file system which is one of the most important layers in I/O stack of large computing systems is designed for parallel applications that share data across many clients in a coordinated manner. Parallel file systems which apply the object-based storage architecture always consist of many *data servers* that provide object-based data storage, one or more *metadata server(s)* that manage namespace hierarchy and files metadata, and many load generating clients, all of which are connected by a shared network.

Modern parallel file systems are designed exploit parallelism for I/O operations. Files are always being striped across multiple selected data servers to facilitate parallel access. A file is made up of a *metadata* object and several *datafile* objects each of which is a block of actual file contents and stored in a separate data server. Metadata objects store data about files or directories. In addition to the common file attributes like owner, group and permissions, metadata object also keeps file distribution information, such as the location of datafile objects and its distribution layout.

Meanwhile, directories are used to manage the global namespace and system hierarchy. Traditionally, each directory is a special file which contains a list of key/value pairs called directory entries to identify all files within the directory. A key is the user-visible name of the file, while a value could be a handle to a file, a sub-directory, or a symbolic link to some other place in file system. Files with identical name are not permitted to exist in a single directory. In the absence of an explicit distributed directory implementation, parallel creation of a large quantity of files from multiple clients in a single shared directory may induce a system bottleneck and low overall performance. File systems which do not distribute single large directory are limited by the speed of the single metadata server that manages the entire directory.

Recently, many studies have applied an index structure called LSM tree to be the metadata's storage backend. LSM-tree is a data structure that is preferred for random updates, inserts, and deletes. In a simple understanding of LSM-trees, they are multi-component data structures composed of several in-memory and on-disk tree-like components. Incoming updates are completely sequentially stored in an operation-log which is pushed to disk periodically and asynchronously by default. When the log has the modification saved, updates are applied to an active in-memory component (called *memtable*) that holds the most recent inserts. Once the memtable is filled up, batched entries are indexed and flushed to disk as a new on-disk component (called *SSTable*). At this point, the updates in the log can be thrown away, as all modifications have been persisted.

2.3 Related Work

Managing a large shared directory offers significant challenges both in terms of performance and scalability. This section shows the related work.

Local file systems such as TableFS [4] and KVFS [5] are developed to organize all metadata into a write-optimized LSM-tree layout. TableFS represents metadata in one all-encompassing table, and only writes large objects to the local disk. They show that even an inefficient FUSE based implementation of TableFS can perform comparably to Ext4, XFS and Btrfs on data-intensive benchmarks while providing substantial performance improvements on metadata-intensive workloads. KVFS uses a transactional variation of LSM-trees called VT-tree to organize metadata. They use stitching to avoid always copying old SSTables into new SSTables in the presence of LSM-tree compaction. KVFS can offer concurrent access with transactional guarantees, and consequently provides efficient and scalable access to both large and small data items regardless of the access pattern.

Distributed file systems, including HDFS [6], GoogleFS [7], Lustre [8], PVFS [9] use a single dedicated metadata server to manage a globally shared namespace. While simple, this design limits scalability, resulting in the metadata server becoming a bottleneck and a single point of failure.

File systems tend to distribute the directory tree over different servers to increase the metadata concurrency. However, for some of them, such as PanFS [10], parallel creations on a shared directory still have an important overhead due to that all files in the directory are managed by a single dedicated metadata server. Meanwhile, some file systems and studies go a step further, eliminating this inefficiency by distributing single large directory through several metadata servers in order to scale.

GPFS [11] is a parallel, shared-disk file system that support large directory with millions of files. GPFS uses extendible hashing [12] to organize directory entries within a directory. Unfortunately, the mechanism to coordinate the global view and ensure consistency is complex and expensive. GPFS employs a distributed locking mechanism that works well as long as different nodes operate on different pieces of metadata. In case of parallel creations, all

the concurrent writes need to acquire write locks from the lock manager before updating the directory blocks. As a result, bouncing the directory lock between simultaneous nodes leads to a far-from-optimal performance.

Patil and Gibson [13] propose GIGA+, a scalable directory design which uses hash-based indexing to incrementally divide each directory into a growing number of partitions that are distributed over multiple metadata servers. A partition that is too full to insert any files will be spilt into two by moving half of the partition to a new partition on another server. Each server splits independently and preserves its own split history to prevent a system-wide serialization or synchronization. GIGA+ can achieve high concurrency due to that manipulations on different partitions can be proceeded in parallel.

SkyeFS [14] implements Giga+ distributed directories on top of PVFS. SkyeFS builds on the FUSE module and the PVFS client library to provide file system services. To split a partition, the split initiator locks the entire partition, moves the entries and drops the lock until split is done. It shows that Giga+ is capable of achieving near linear speedup once a directory is at load balance.

Based on GIGA+, Xing [15] present a scalable directory scheme that aims to maintain billions of files in a directory. They employ an adaptive two-level partitioning method to split directory into partitions when the size of directory is small and enlarge partitions with more chunks when the size of directory is large. These two levels of splitting are done automatically and alternately to adapt to different size of directories and the growth of directories. Compared with GIGA+, this scheme reduces the amount of partitions that need to be migrated among the servers when directory grows and improves the performance.

Yang [16] demonstrates an implementation of a scalable distributed directory service based on extensible hashing technique and the splitting strategy of GIGA+. When a directory is created, an array of *dirdata objects* is allocated with each dirdata object on each metadata server. Directory entries are then spread across the dirdata objects. In contrast to GIGA+, the initial number of active dirdata objects is configurable to enjoy better scalability from the beginning, rather than starting from one partition. As a matter of fact, the splitting process is found to be expensive when increasing the number of partitions gradually [17].

IndexFS [18] also uses the GIGA+ binary splitting technique to distribute directory entries. It embeds inode attributes and small files into directory entries and stores them into a single LSM tree. When the split target server receive the migrated entries (in form of a SSTable), it adds the SSTable as a file at level 0 of current LSM tree directly, avoiding the overhead of write-ahead log and in-memory cache. IndexFS also implements write back caching at the client for creation of large directories. IndexFS clients can complete creation locally and later bulk insert all file creation operations into IndexFS servers using a single SSTable insertion, eliminating the one-RPC-per-file-create overhead and enabling total throughput to scale linearly with the number of clients. However, their ultra high throughput bulk insertion comes at the expense of weak file system semantics. First, they suppose any created files are new to the file system. Second, another client asks for access to the localized files of current client will fail until the write lease period expires and the completion of its remaining bulk inserts. Third, their lease-based client cache protocol increases the degree of difficulty of failure tolerance.

In our design, despite that we employ the similar splitting strategy of GIGA+, it differs in several aspects. Firstly, we have decided to distribute the directory entries over a cluster of data servers rather than traditional metadata servers, preventing us from being limited to the relative rare number of metadata servers. Secondly, we provide an optimization to facilitate a fast splitting process. Thirdly, inspired by the TableFS and IndexFS, SFS employs LSM tree to be our ordered and persistent metadata storage, leading to fast metadata updates.

3. Distributed and Scalable Directory Design of SFS

This section describes designs to build distributed and scalable directory services for SFS to support efficient concurrent access to large directories and provides more detail on the internal components that enable high-performance and scalability.

3.1 Object Structure of Large Directory

Some studies have indicated that 99.99% of the directories contain fewer than 8,000 files [19]. Most file system directories are small and remaining small. An empty or a small directory in SFS is initially stored on one metadata server, avoiding to degrade small directory performance. For directories that are known to be large before creation, we introduce a new directory hint called *large-dir*, which can be used to turn the directory optimizations on and off on a per-directory basis. Since only a few directories grow to really large, the hint will not be set in a default situation.

As illustrated in GIGA+, splitting to create more than one partition per server significantly improves system load balancing for non power-of-two numbers of servers. Meanwhile, more partitions per server takes longer for the system to stop splitting and increases client addressing errors. SFS decides to distribute large directory entries across a cluster of data servers rather than metadata servers. This design has several features that make it attractive. First, the total number of data servers is always several magnitudes of order of metadata servers. With more servers, we can achieve more concurrency. Second, for a really large directory, we can set the number of servers that the directory is about to split to as a power-of-two number to achieve server load balancing. Now the number of servers can be chosen from a greater scope than the initial number of metadata servers. Third, we can set the partition per server to a small value to avoid too much splitting. Each server in SFS owns only one partitions of a large directory. Finally, we can exploit the usage of data

servers to facilitate optimizations on file create operations. For example, for each file in SFS, we currently allocate a file in the local file system to store the actual data (described as datafile object below).

SFS organizes large directories in terms of several storage objects that are identified by *handles*, which are unique, opaque, integer-like identifiers. Some important objects are

- *Directory metadata* objects include data about directories and are stored on metadata servers. Aside from the usual file attributes like owner, group, etc., the metadata object of a large directory keeps an array of dirdata handles and a dirdata bitmap (described below).
- *Datafile* objects contain contents of files. Each datafile object is a byte-addressed sequence of bytes, plus a set of attributes, accessed via a file-like interface (e.g., CREATE, READ, and so on). Datafile objects are stored in the local file system on data servers.
- *Dirdata* objects store directory entries on the data server (or can be stored in a shared distributed file system). SFS oragnizes entries of sub-directories and subfiles of current large directory in different ways. The entry for a sub-directory is a pair of entry name and the identifier to its directory metadata object. For a subfile, we store the metadata of the file directly with directory entry that link to it. On a per-directory basis, SFS can also be configured to embed data of small files into metadata (dirdata objects) or store them in datafile objects. A dirdata object is also referred to as a partition.

The structures of these objects are visualized in Fig. 1. For a directory that is expected to be large, we construct a server list to record data servers that a directory splits to, which is unique for each directory. The server list can be expanded if new data servers are added to the system. When a directory is created, an array of partitions is allocated with each on a data server in the list and a partition bitmap and an array of partition handles are initialized in the directory metadata object. The bitmap keeps a one-to-one mapping between the bit position and the index of the partition handle array. A bit value of '1' indicates an active partition while '0' indicates an inactive partition. Each partition also keeps a copy of bitmap and array of handles as shown in Fig. 1. The bitmap on the directory metadata object is most up-to-date to keep up with changes in every partition.

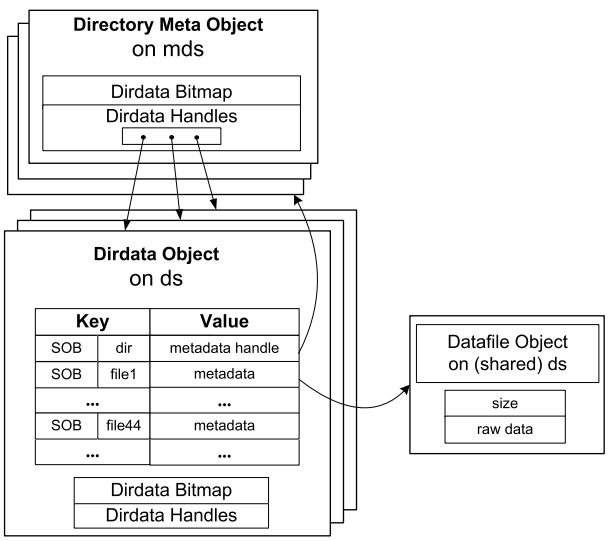

Fig. 1 Diagram of object structures of large directories. A large directory is split into a cluster of data servers with dirdata handles array and dirdata bitmap are added to the directory metadata object.

3.2 LSM Tree Based Metadata Storage

SFS uses an open-source key-value storage library called LevelDB for on-disk metadata storage. As shown in Fig. 1, SFS records keys that consist of two parts to support fast entry operations in the database: the recursive split-ordering-bits (SOB) and the filename. SOB is devoted to speed up the splitting process and will be introduced in later section. The value part consists of metadata and may contain file data.

Every dirdata object in SFS is represented as one LevelDB instance. This per-dirdata design prevents the LevelDB database from growing too large, facilitating faster file lookups and directory scans than storing all dirdata objects of directories in a single LevelDB instance. On each data server, we maintain an opened dirdata list with LRU eviction policy. A dirdata that is not accessed for a threshold time will be evicted from the list to reclaim memory. We also have an option that control the maximum number of opened dirdata objects. When reaching the maximum, the least recently accessed dirdata object will be released to satisfy the new incoming file operations on other dirdata.

LSM tree supports "upsert", an efficient method for updating a key-value pair in the tree. However, when creating a file, the POSIX semantics require the file system to check that the file doesn't already exist. As the entries in a dirdata grow, the creation rate will slow down because the non-exist test in each create is applied to ever larger on-disk data structure. SFS provides an error-free solution that uses per-dirdata bloom filter to eliminate most unnecessary disk accesses. For a "possibly in set" test, SFS asks the LevelDB to indicate whether the file really exists. When creating a file, SFS first set the right position of bloom filter, no matter whether the operation succeeds or not. For file renames and deletions, SFS does not change the bloom filter to ensure the correctness, although increasing the probability of false positives. The size of each bloom filter is set to 4MiB by default, which can also be configured on a per-directory basis. As the directory entries grow with usage, this filter changes - and, at high insert rates it changes rapidly.

SFS currently does not support "hard links" in large directories. Despite multiple "hard links" to individual files are relatively rare, and are used primarily for temporary files [20], we are looking forward to solve this problem in the future.

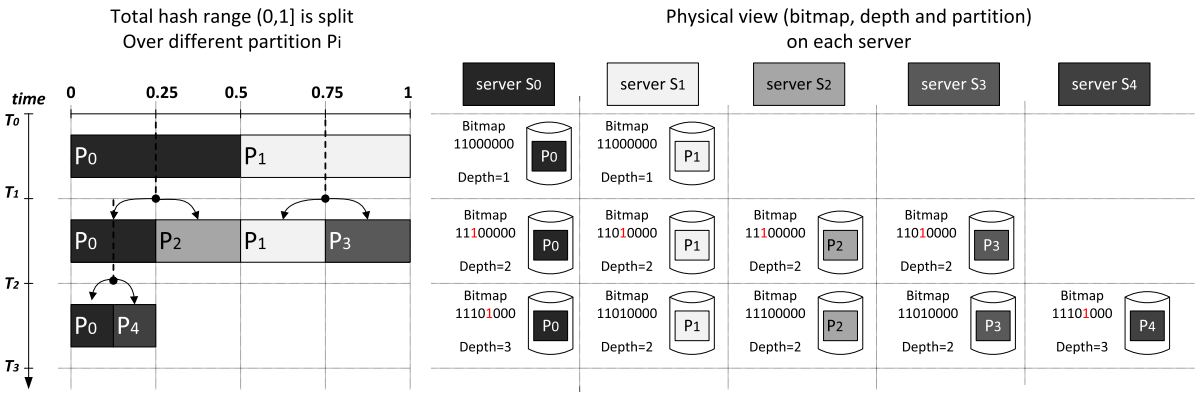

Fig. 2 Concurrent and unsynchronized directory splitting in SFS.

3.3 Concurrent and Unsynchronized Splitting

Like GIGA+, SFS employs extendible hashing technique to incrementally divide large directory into numerous partitions. A directory entry is hashed and then assigned to an active partition handle based on its name. Our implementation uses a strong hash algorithm MD5 to encode the directory entry name to facilitate a random distribution. The hashed value then serves as the key to partition selection.

Because each data server maintains only one partition for a directory, the maximum number of partitions of a large directory is fixed at the number of data servers in the corresponding server list. In our case, the initial number of active partitions is configurable and set to 1 as default. As new entries get added and the number of entries in a partition exceeds a threshold value, the partition splits by moving about half of its entries (including files data if exist) onto an inactive partition on another data server if possible. Splitting is no longer necessary when no inactive partitions can be used to split to. In that case, the number of directory entries stored in the partition is allowed to exceed the threshold, regardless of potential slowdowns in performance.

Figure 2 shows the directory splitting process. In this example, a directory is to be spread over five servers $\{S_0, S_1, S_2, S_3, S_4\}$ in five shades of gray color. P_i denotes the hash-space range held by a partition with an unique index i. Initially, at time T_0, the partition on each server and the handles that contain identifiers to these five partitions are initialized. We suppose the initial number of active partitions is set to two. Thus, the directory entries that the least significant bit of hashed value of the name equals to '0' will be assigned to P_0 and those equal to '1' will be assigned to P_1. Meanwhile, the first two bits of partition bitmaps will be set to '1' and the depths of the corresponding partitions (P_0 and P_1)will be calculated as one. As the directory grows and the number of entries exceeds a threshold, the partitions start to split. For example, at time T_1, P_0 is split into two by moving the great half of its hash range to a new partition P_2 on S_2. SFS calculates the identifier of a split's target partition using well-know radix-based techniques as GIGA+ does. Specifically, if a partition has an identifier i and depth of r, the index of the new partition is $i + 2^r$. Afterwards, both partitions will be at depth $r + 1$. Note that the splitting course can not proceed if partition handle with index $i + 2^r$ does not exist.

The corresponding position of partition bitmap is also updated to indicate an active partition. As each partition can split independently without global coordination, no other than the original and new target partition need to update their bitmaps. The original partition is also responsible for updating the bitmap in the directory metadata object, thus keeping it most up-to-date at all times.

3.4 Speeding Up the Splitting Process

There are several steps involved in the source server when splitting: (i) the splitting process first locks the original directory partition, (ii) scans all entries in the partition to find which entries to be moved to a new partition, (iii) then transactionally migrates these entries (also including bitmap, depth and files data if exist and needed) to the partition hosted on another server, (iv) and updates the bitmap and depth before releasing the local lock. For a partition with depth of r, directory entries with a '1' on the $r + 1$ bit position of their hash values will be moved to the new partition when splitting. The write lock acquired by the splitting process is used to protect both the directory entries and the bitmap. This implies that all other create threads would suspend and wait for the competition of the splitting, and has been found to be expensive [17].

However, if we sort the entries in a partition carefully, our modulo-based extensible hashing will keep all the entries destined for a given new partition together in the same old partition. The core of this mechanism is *recursive split-ordering* [21], a way of ordering elements to allow splitting repeatedly without any reordering. It is achieved by simple *binary reversal* so that the new key's most significant bits are those that are originally its least significant. As mentioned before, we record directory entries in partition with key consisting of two parts: the recursive split-ordering-bits (SOB) and the name. The SOB is calculated by reversing the bits

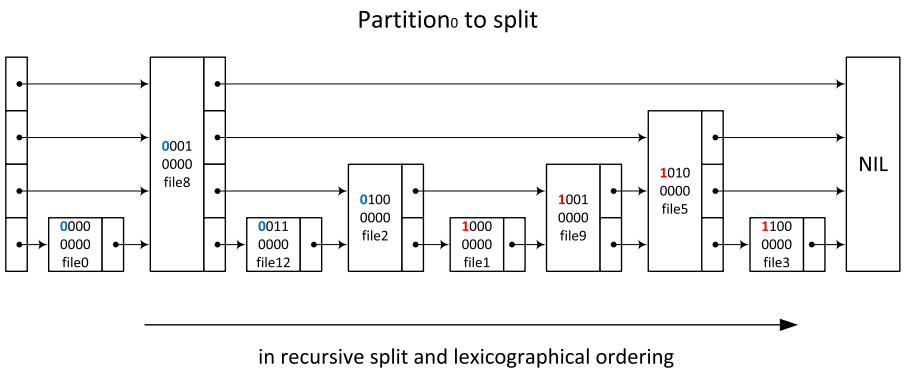

Fig. 3 Partition splitting with recursive split-ordering mechanism.

of the hash value of the filename (with the same MD5 hash function). For instance, the split-ordering value of 5 is the bit-reverse of its binary representation, which is 10100000 (in this example, we use 8-bit words). In SFS, we currently use 32-bit words to represent the SOB part of a key.

LevelDB provides iterators, a way to combine the different data-sources to supply a global, unified, iterable, and sorted view of the database. It simply merges iterators provides by the memtables (a maximum of 2), all SSTables in level-0 (LevelDB tries to keep to a number no more than 4), and level 1 to max-level (a maximum of max-levels minus 1), and returns the next value in order. A iterator will operate on a "snapshot" of the database which is a consistent read-only views over the entire state of the store, disregarding any further new updates.

Figure 3 shows an example of the splitting process of a partition P_0 whose both index and depth equal to 0. Because our threshold to split is relative small (as default as 8,000), we only show the key-value pairs in memtables here for simplicity. File entries are organized as a skip-list structure in memtable and are depicted by the combination of split-ordering-bits and files name. In this example, we assume that the hash value of "file**x**" modulo 256 is **x** (in 8-bit words). As shown, file entries are sorted in a recursive split and lexicographical ordering. When splitting, entries are differentiated by the 1st binary digit of their keys: those with '0' will reside on the old partition, while those with '1' to the new partition with index 1. The next splitting will cause each partition to split again into two differentiated by 2nd bit, and so on. To summarize, when a partition with index of i and depth of r is going to split, a simple prefix lookup (using iterator) with a given prefix that equals to first $r + 1$ bits of the split-ordering value of $i + 2^r$ will return all records that need to be migrated to the new partition. We can deduce from Fig. 3 that scanning of entries will not require the protection of the overall write lock any longer, because the clients are informed with the updated bitmap and the position of any new incoming inserts would be in front of "file1".

Directory splitting in SFS is protected by a distributed transaction protocol to preserve consistency. It is implemented as a two-phase commit and failure protection with write-ahead log:

- *Phase 1:* The split initiator locks the partition, updates the bitmap and depth, releases the lock, performs a range scan on its LevelDB instance and then sends the bitmap, depth, and entries to the split receiver. The split receiver then completes the bulk insert operation, notifies the initiator and then blocks to wait for the confirmed message from the initiator.
- *Phase 2:* The initiator receives the notification from the receiver, begins to clean up the migrated entries and then responses to the receiver to inform that the receiver now can start responding to clients.

The status of split initiator and receiver remains consistency even if it releases the partition lock before the splitting completes. Our new splitting scheme based on recursive split-ordering mechanism can prevent us from blocking to scan and transfer the entries and benefit us from splitting the partition much more quickly than the original scheme.

3.5 Locating Directory Entry with Inconsistent Mapping

When locating a directory entry, the SFS client contacts the corresponding metadata server to fetch the relative directory attributes, including the partition bitmap and handles array. The client caches these mappings and uses them to find the appropriate partition and the data server it maps to. A directory entry name is hashed and the hashed value is used as the key to partition selection. The lower R bits ($R = \lceil \log_2(N) \rceil$, where N is the size of handles array) server as the initial matching index I. Given the index I, client checks the bit position I in the bitmap. If the bit is set, the Ith partition handle is picked and the later operations can be processed on the corresponding partition. Otherwise, either the partition is inactive or the client bitmap is stale. In both cases, the highest bit of I is taken off and the bitmap is checked again with the new matching index. The bit position '0' in the bitmap is always set which guarantees an active partition is selected in any circumstance.

In fact, the bitmap in the client cache may be out-of-date because of splitting. Despite the inconsistent copies of bitmap, SFS ensures that the client's requests are forwarded

to the correct server. To minimize communication and synchronization overheads, SFS updates and synchronizes the bitmaps in a lazy manner, without using a traditional, synchronous cache consistency protocol. A client with stale mappings sends its request to an "incorrect" server that no longer holds the desired part of entries. However this "incorrect" server keeps a most up-to-date part of the bitmap and is responsible for updating the client's bitmap. The client uses this updated bitmap to probe again. In the worst case, this process could take $O(\log N)$ incorrect probes, where N is the number of partitions. But the client updates its bitmap with each probe, so the number of times of this action is bounded. Thus, though each client may cache an inconsistent copies of bitmap, the correctness and consistency of the system are still maintained. In contrast to GIGA+, our design leads to far less probes due to each data server maintains only one partition for a large directory.

3.6 Handling Failures

SFS is designed as a middle-ware layer on top of an underlying failure-tolerant and cluster file system. SFS does not add major challenges when it is integrated into a cluster file system that has already presented fault tolerance for data and services. In fact, on the server side, SFS's fault tolerance strategy uses full data journaling to flush metadata updates to the files (the SSTables) in the underlying file system. The recent changes are firstly written into write-ahead logs (WAL) which are rotated and committed to disk every few seconds or when a memtable buffer overflows. The SFS servers processes are monitored by scout server process (such as Nagios). In case of a process failure, we can reboot the process or use a standby server process to replace the failed process. The disk failure tolerance is based on RAID encoded and replications. Data files (such as SSTables and logs) can also be configured to be stored in a underlying shared distributed file systems, thus can be accessed by any data server when we need to reconstruct the directory entries in a standby server.

Each data server also maintains a separate write-ahead log that record the mutation status such as the bitmap, depth, migrated entries and which step we have reached when splitting. In case of a process failure when splitting, the reboot or standby process can choose to rollback the status or continue the splitting, depends on which step the splitting has reached and the time of process failure. For instance, if the reboot or standby process found that all the migrated entries have been sent but the target server has no response (it has to response a success flag), the source server deems that the migration fails and rollbacks to the status before splitting. After a threshold time, it can start a new splitting request. However, there is still a situation that may imply an inconsistency occurred in phase 2 described in Sect. 3.4. If the split target dose not receive a notification from the initiator in a threshold time, and it deems that the split fails. Now, the status between initiator and target becomes inconsistency. We plan to use a more robust consistency protocol in the further to solve this problem.

Other issues, such as client-side reboots and network partitions are relatively easy because SFS tolerate stale mappings in client's cache. SFS does not cache directory entries in client process. Directory-specific bitmap can be rebuilding by contacting metadata server and further updated by the data servers through incorrect addressing of partitions during normal operations.

3.7 Handling Server Additions and Removals

When new data servers are added to an existing configuration, overloaded partitions can resume splitting to partitions on new servers. For a large directory, these new servers are added to the server list and each of them allocates a partition for later use. For every partition that wishes to but can not split due to the target partition does not exist, it sends a splitting request to the directory metadata server (for one depth, it sends only once). We uses a FIFO (First In First Out) splitting request queue to record these requests. The new added partition handle will be set to the relevant position in the handles array which depends on the first request in the FIFO queue. Thus the directory metadata server inform the corresponding partition that the splitting operations can be performed now. Thus new servers are made use of properly.

When deciding to remove a data server or directory service in a data server, the situation is trickier. If the server to be removed has not contains any directories entries, it can be safely removed. All things we need do are updating the server lists. However, if the server has entries of large directories, the administrator should first change the mode of all affected directories to read-only (this can be done easily because each server saves the list of large directories in this server). Then we copy all the data files (including LevelDB files and WALs) to the new server if needed. The new server can be any server in our server configuration, including the original servers that have been used. It starts a new directory service, loads all the data files and is ready to work. Then the administrator updates the server lists and notifies all other data servers the change of server configuration. At last, the administrator can change the mode of all affected directories to writable, and server removal is done.

Currently, we provide a set of dedicated commands in favor of server additions and removals, and our scalable directory service is carefully design so that it is aware of server number changes and always adapts to them.

4. Experimental Evaluation

We have built a skeleton prototype file system to evaluate our directory design under a range of workloads to demonstrate its performance and scalability. Clients, data servers, and metadata server are entirely implemented in user-space, and communicated over TCP using socket.

All experiments are performed on a cluster of 65 machines connected with 10GigE switches. Each node has two six-core 2.10GHz Intel Xeon processors, 16GB memory, a

Seagate 7200rpm disk of 2TB, and a 10GigE NIC. All nodes are running the 64-bit Linux 2.6.32 kernel with ext4 file system. We assign 32 nodes as data servers, 1 node as metadata server and the remaining 32 nodes as load generating clients. The threshold for splitting a partition is configured to 8,000 entries as same as GIGA+ does. We use version 1.15 of LevelDB, with asynchronous log commits and default snappy compression on.

We generate a concurrent create workload that creates a large number of files simultaneously in an empty directory. In all cases, operations on files are evenly distributed among the clients and each client spawns eight processes to concurrently create files in a shared directory. All results are the average of, at least, five runs of each test.

4.1 Baseline Performance

We start with a baseline for the performance of various file systems running the `mdtest` benchmark. We use the synthetic `mdtest` benchmark to insert zero-byte files into a single directory. In the beginning, we compare `mdtest` running on SFS to `mdtest` running on Linux ext4 and a separate client and single server instance of PVFS cluster file system and a mature commercial NFSv4 filer (all on ext4). The single client uses eight processes to create 320,000 files in a common directory. LevelDB is configured with enough memory to store all entries in memtable (100MiB).

We use three machines with ext4 on the server (one as client, one as metadata server and on as data server) and a direct library linking way to bind `mdtest` to SFS. We modify the codes of `mdtest` to allow it to use our non-POSIX custom object create calls (such as `sfs_creat ()`).

Table 1 plots the baseline performance. As shown, ext4 deliver the highest directory insert rate, which can sustain about 45,095 file creates per second. Our SFS configuration creates at about 80% of the rate of ext4 with local load generating processes. Although the LevelDB is effectively adapted to metadata insert-intensive workloads, the baseline performance of SFS is subject to datafile creation in local ext4 file system. This comparison also shows that the remote RPC is not a huge penalty for SFS. We also compare SFS with other network file systems, SFS generally outperforms all, probably for most network file systems perform metadata operations in the same way all the time regardless of the size of the directory.

Table 1 File create rate in a single directory on a single server and 320,000 files in total (with 1% standard deviation).

	File System	File creates/second in one directory
Local file system	Linux ext4	45,095
Networked file systems	NFSv4 filer	1,506
	PVFS	1,184
SFS (layered on ext4)	Library API	35,848

4.2 Incremental Metadata Scalability

In this section, the experiment is a stress test to analyze the scaling behavior of our directory service in SFS. We run the `mdtest` benchmark to create a large number of zero-byte files in an empty directory and measure the aggregate throughput (file creates per second) continuously throughout the benchmark. We answer two questions: (1) how does our design scale with increasing number of clients and data servers, (2) What are the tradeoffs of SFS's design involving memtable size and choice of embedding file data into metadata; and discuss about scale-out heuristics of SFS: (3) What causes may limit the scalability?

For the first question, we create six test scenes with each owning a distinct number of data servers which is 1, 2, 4, 8, 16, and 32. Within each test scene, we scale the number of clients from 1 to 32. The total number of files created in every scene is proportional to the number of data servers and clients. For instance, in the occasion of 1 data server, a single client with 8 processes will create 10,000 unique files, two clients with 16 processes will create 20,000 files and up to 32 clients will create 320,000 files on one data server. Likewise, with 2 data server, a single client will create 20,000 files and up to 32 clients will create 640,000 files on the two data servers. To test the on-disk performance, memtable in LevelDB is configured with 8MiB, too small to store all metadata in memtable (about 64MiB metadata for the 32 clients test scene).

Figure 4 plots the result in terms of aggregate operation throughput. The curves for each test scene are similar. As the directory gradually splits to more data servers, the create rate grows. SFS demonstrates scalable performance for the concurrent create workload delivering a maximum throughput at the point of 20 clients. This is because when creating with 20 clients, the directory entries will consume about 40MiB space that generated about 4 SSTables in LevelDB. As SSTables grow more than 4 (the default maximum number of SSTables in level-0 in LevelDB), LevelDB begins to merge a range of SSTables ,compact them to higher levels and sequentially write back to disk. With our 32-server

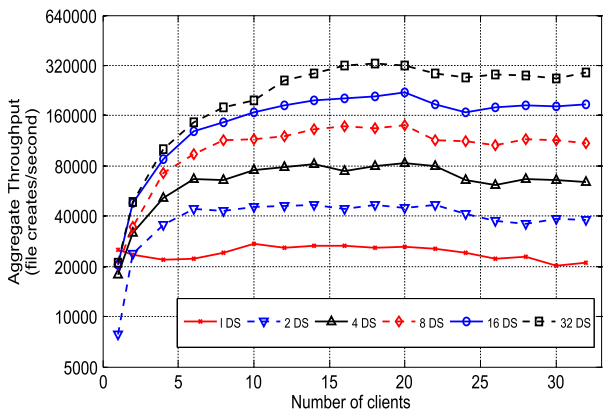

Fig. 4 Scalability of SFS directories.

configuration, SFS can sustain a peak throughput of about 330,000 file creates per second - this satisfies today's most rigorous scalability demands.

It is observed that with a larger number of clients, which introduce higher creation request rate, the throughput scales better and keeps to scale. With 22 and more clients, the peak throughput degrades over time on account of a bottleneck of LevelDB which induces more frequently memtable flushes and SSTable compactions.

It is also noticeable that when the number of clients is relatively small ($\leqslant 4$), increasing the number of data servers does not dramatically improve the throughput. This is because the arrival rate of request from the clients has not achieve the overall service rate, the servers are "hungry" and are not fully utilized.

In the context of the second question, we run the concurrent create benchmark with options that differentiate from our previous settings. We create two test scenes. First, we increase the memetable size that can gracefully hold all entries in memory. Second, we use the inode stuffing technique to ingest data of small files into dirdata object, omitting the overhead of creating files in local file system. Within each test, we set the number of data servers to 32 and scale the number of clients from 2 to 32.

Figure 5 compares the effect of different policies for the size of memtable and inode stuffing on the system throughput. As shown, the gap between the new policy and our default setting enlarges as clients increase. The graph shows a performance upgrades when adopting the larger memtable and inode stuffing. However, the advantage comes at other costs. For example, larger memtables increase the usage of system memory, resulting in reduced capabilities of concurrent creates on different directories. The larger the memtable is, the smaller number of dirdata objects in one server SFS can manage concurrently. On the other side, inode stuffing brings additional overhead during LSM Tree's compactions since embedding small files increase the data volume processed by every compaction. Furthermore, the advantage of reduced random disk seeks for lookup operations like `getattr` and `read` for small files now diminishes, owing to more SSTables in the underlying storage. SFS currently presents these configurations optionally for the user, on a per-directory basis.

SFS has demonstrated scalable performance for the concurrent create workload, delivering a peak throughput of more than 300,000 file creates per second for our 32 server configurations. SFS is expected to continue to scale with even larger number of servers. However, good metadata scalability comes at the expense of degraded performance for a single operation. There are still some factors that may limit the scalability. First, as illustrated before, the local file system and LevelDB can incur different kinds of overhead in the separate server. For instance, the small datafile created in local file system is the main cause that prevents SFS from good single partition performance. We plan to use ReiserFS (with `-notail` option), a file system that packs the data inside i-node for high small file performance usage. For LevelDB, a new variation of LSM trees called RocksDB can be employed to enable concurrent compactions and more options tuning. Second, Flash storage (like SSD) can also benefit our scalable design of SFS. LSM-trees implementations, such as LevelDB and RocksDB, are optimized for Flash storage with extremely low latencies. We leave these feasible improvements in the near future, due to that our present computing cluster configuration does not include any SSD devices at all.

4.3 Efficiency of Recursive Split-Ordering Based Splitting

In this section, to better understand the impact of splitting scheme on our performance, we run create test to verify the efficiency between the original and our new splitting scheme. In order to evaluate the result on small I/O-intensive workloads, we use a micro-write benchmark (written by ourself) to create 10,240,000 500-byte files in a shared directory, with 32 clients (256 processes) on 32 data servers.

Figure 6 shows the first 20 seconds of the concurrent workload. As shown, the throughput of both cases experiences interim drops in the initial seconds when the servers are busy splitting and both of them can make maximum use of 32 servers at last. However, our new scheme encounters much faster splitting course and uses much litter time to utilize the remaining data servers. The original scheme uses about 10 seconds to catch up with our new scheme. Our new scheme outperforms the original one with the reduction of approximate 10% of the total running time. This hap-

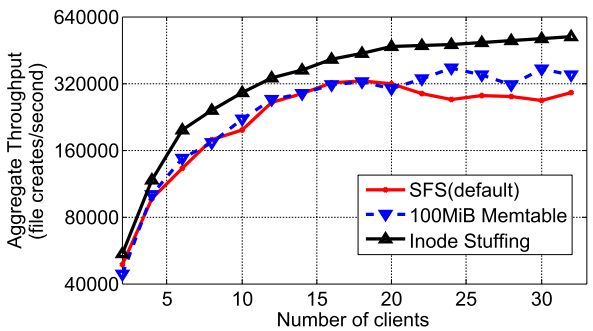

Fig. 5 The effect of discrepant options in SFS.

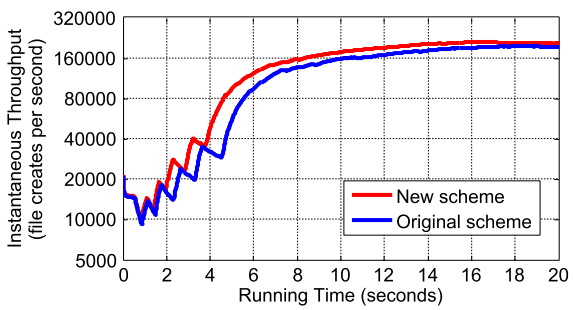

Fig. 6 Impact of splitting scheme on directory growth.

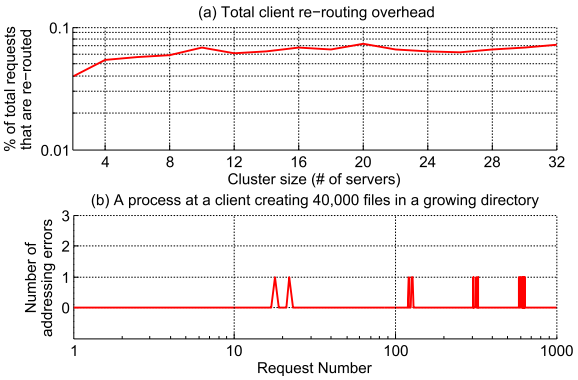

Fig. 7 Cost of using inconsistent mapping at the clients.

pens because the partition can continue to serve the requests without waiting for the competition of the splitting when adopting our recursive split-ordering based scheme.

4.4 Cost of Inconsistent Mapping

In this section, we rerun the micro-write benchmark in previous section to measure the overhead of our weak consistency mapping at the clients. First, we measure the percentage of all clients requests that are re-routed when using 32 clients to create different scale of files on different scale of data servers. Figure 7(a) depicts the overhead incurred by clients when their cached mappings become stale. As shown, in absolute terms, fewer than 0.08% of the requests are addressed incorrectly; this is only about 240 requests per client when each client is doing 320,000 file creates.

We study further the worst case in Fig. 7(a), 32 servers with 10,240,000 file creates, to learn when addressing errors occur. Figure 7(b) shows when a process of a client encounter the addressing errors when it creates 40,000 files in the test. Figure 7(b) suggest two conclusions. First, each process can find the correct server with at most 2 probes. Second, the process makes no more than 30 addressing errors total and make no more addressing errors after the 637th request. This indicates that more than 98% of the work is done without any addressing errors. Hence, using our weak consistency mapping strategies has a very negligible overhead on client performance.

5. Conclusion

Dedicated metadata servers limit the performance and scalability of parallel file systems. In this paper we describe a scalable and distributed directory service that distributes directory entries over data servers rather than metadata servers to offer increased scalability and performance. The number of partitions of a large directory increases as the directory grows in a concurrent and asynchronous manner. Files are assigned to a specific partition by hashing their names and clients seeking a specific filename find the appropriate partition by probing servers based on their cached mapping. To accelerate metadata updates, we employ LSM tree to be an ordered, persistent index structure for metadata storage. We have also describe an optimization based on recursive split-ordering to accelerate the splitting process.

We used these principles to prototype a distributed directory implementation that scales linearly on a 32-server configuration. Our analysis also shows that SFS achieves faster partition splitting than the traditional splitting strategy and incurs a negligible performance penalty when allowing to use weak consistent mappings at the clients.

Acknowledgments

This material is based upon work supported in part by the National Natural Science Foundation of China under grant 61120106005 and the National High Technology Research and Development 863 Program of China under grant 2012AA01A301. We thank Guang Suo, Enqiang Zhou, Tao Gao and Rongdong Hu for their helpful insight and improving in this paper. We are also grateful to anonymous reviewers for their valuable comments and suggestions.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree," Acta Informatica, vol.33, no.4, pp.351–385, 1996.
[2] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," Proc. Symposium on Operating Systems Design and Implementation (OSDI'06), Seattle, WA, USA, Nov. 2006.
[3] D. Beaver, S. Kumar, H.C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," Proc. Symposium on Operating Systems Design and Implementation (OSDI'10), Vancouver, BC, Canada, Oct. 4–6, 2010.
[4] K. Ren and G.A. Gibson, "Tablefs: Enhancing metadata efficiency in the local file system," Proc. USENIX Annual Technical Conference, pp.145–156, 2013.
[5] P. Shetty, R.P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," Proc. USENIX Conference on File and Storage Technologies (FAST'13), pp.17–30, 2013.
[6] K. Shvachko, H. Huang, S. Radia, and R. Chansler, "The hadoop distributed file system," Proc. IEEE Symposium on Mass Storage Systems and Technologies (MSST'10), Incline Village, NV, pp.1–10, May 3–7, 2010.
[7] S. Ghemawat, H. Gobioff, and S.T. Lueng, "Google file system," Proc. ACM Symposium on Operating Systems Principles (SOSP'03), Bolton Landing, NY, Oct. 2003.
[8] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," Proc. Linux Symposium, Ottawa, Canada, July 2003.
[9] I.F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," Linux Journal, vol.2000, p.5, 2000.
[10] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," Proc. USENIX Conference on File and Storage Technologies (FAST'08), 2008.
[11] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," Proc. USENIX Conference on File and Storage Technologies (FAST'02), Monterey, Canada, Jan. 2002.
[12] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible hashing – a fast access method for dynamic files," ACM Trans. Database Syst., vol.4(3), Sept. 1979.
[13] S. Patil and G.A. Gibson, "Scale and concurrency of giga+: File system directories with millions of files," Proc. USENIX Conference

on File and Storage Technologies (FAST'11), San Jose, USA, Feb. 2011.
[14] A. Chivetta, S. Patil, and G. Gibson, "Skyefs: Distributed directories using giga+ and pvfs," Tech. Rep. CMU-PDL-12-104, Carnegie Mellon University, 2012.
[15] J. Xing, J. Xiong, N. Sun, and J. Ma, "Adaptive and scalable metadata management to support a trillion files," Proc. SC'09, 2009.
[16] S. Yang, W.B. Ligon III, and E.C. Quarles, "Scalable distributed directory implementation on orange file system," 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O, Citeseer, 2011.
[17] Y. Wu, A study for scalable directory in parallel file systems, Master's thesis, Clemson University, Clemson, SC, USA, 2009.
[18] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for, pp.237–248, IEEE, 2014.
[19] S. Dayal, "Characterizing hec storage systems at rest," Tech. Rep. CMU-PDL-08-109, Carnegie Mellon University, July 2008.
[20] S.A. Weil, Ceph: Reliable, Scalable, and High-performance Distributed Storage, Ph.D. thesis, Univ. of California, Dec. 2007.
[21] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," J. ACM, vol.53, pp.379–405, May 2006.

Lixin Wang is currently a Ph.D candidate at the College of Computer, National University of Defense Technology, Changsha, Hunan, China. His research interests include distributed metadata management and object-based storage system.

Yutong Lu is the director of the System Software Laboratory, College of Computer, National University of Defense Technology, Changsha, Hunan, China. She is also a professor in the State Key Laboratory of High Performance Computing, China. Her research interests include parallel operating system (OS), high speed communications, global file systems, and advanced programming environments with MPI.

Wei Zhang is a Ph.D at the College of Computer, National University of Defense Technology, Changsha, Hunan, China. His research interests include file and storage systems, manageability and performance analysis.

Yan Lei is a Ph.D candidate at the College of Computer, National University of Defense Technology, Changsha, Hunan, China. His research interests include software debugging and system software.