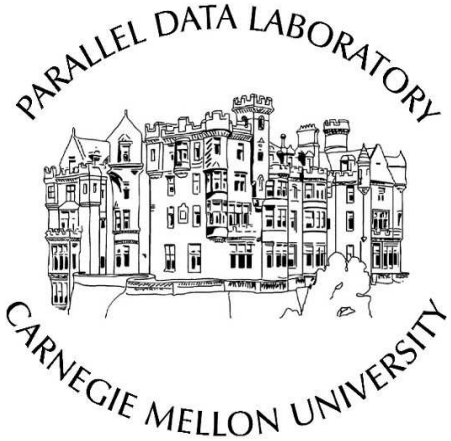

SkyeFS: Distributed Directories using Giga+ and PVFS

Anthony Chivetta, Swapnil Patil & Garth Gibson

anthony @ chivetta.org, {swapnil.patil , garth.gibson} @ cs.cmu.edu

CMU-PDL-12-104
May 2012

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

There is growing set of large-scale data-intensive applications that require file system directories to store millions to billions of files in each directory and to sustain hundreds of thousands of concurrent directory operations per second. Unfortunately, most cluster file systems are unable to provide this level of scale and parallelism. In this research, we show how the GIGA+ distributed directory algorithm, developed at CMU, can be applied to a real-world cluster file system. We designed and implemented a user-level file system, called SkyeFS, that efficiently layers GIGA+ on top of the PVFS cluster file system. Our experimental evaluation demonstrates how an optimized interposition layer can help PVFS achieve the desired scalability for massive file system directories.

1 Overview

PVFS, the Parallel Virtual File System, stores all metadata for a particular directory on a single metadata server[2] resulting in poor scalability in the case of large or high traffic directories. Giga+ is a scheme for partitioning directory metadata across a set of servers while still maintaining performance for small directories.[1] SkyeFS implements the Giga+ algorithm on top of an unmodified PVFS file system.

SkyeFS consists of a client (`skye_client`) which functions as a FUSE filesystem and a server (`skye_server`) which provides synchronization for metadata operations, controls directory placement and coordinates splitting.[5]

As a directory grows, Giga+ incrementally splits the directory into multiple partitions which are load-balanced across available servers. In SkyeFS, we represent each partition as a distinct PVFS directory. A SkyeFS server located on the same host as each PVFS metadata server (MDS) provides synchronization and coordination.

2 Implementation Overview

We chose to implement Giga+ as an overlay filesystem on top of PVFS, instead of modifying PVFS directly to keep the implementation small and simple. FUSE makes developing a VFS-compatable filesystem easy by removing the need to write a kernel-mode filesystem drier. Additionally, the pvfs2fuse application distributed with the PVFS source provides an example of how to implement FUSE operations using PVFS. However, unlike pvs2fuse, we use the lowlevel FUSE API. This allows us to directly specify PVFS handles as the inode numbers returned to the kernel, avoiding extraneous pathname resolution in the client.

In PVFS, every object is assigned a PVFS metadata handle and each PVFS server is statically assigned a range of handles for which it is responsible. Directory entries are stored with the directory's metadata on the server responsible for its metadata handle. We exploit this property of PVFS to control metadata placement for each Giga+ partition.

2.1 Filesystem Layout

SkyeFS establishes a direct mapping between Giga+ partitions and a set of PVFS directories. This allows us to use Giga+ for bothload-balancing and to keep directory size small. Rename operations in PVFS are relatively cheep as they only move object metadata and not entire files. This limits the cost of the extra renames required by this storage scheme. In systems without efficient rename, a single directory per metadata server might be used instead.

When a logical directory is created, we also create a PVFS directory on the same server called "p00000" inside the logical directory to represent the first Giga+ partition. When this partition splits, a new directory "p00001" will be created adjacent to it to store the files in the new partition.

The PVFS client uses round-robin assignment for selecting the metadata server on which it will create a new object. Unfortunately, the PVFS client API does not currently provide any way to influence this server selection. Therefore, when creating PVFS directories for new partitions, we must repeatedly create new directories until the resulting directory is on the desired server. Future versions of SkyeFS might cache these directories to limit extraneous creates.

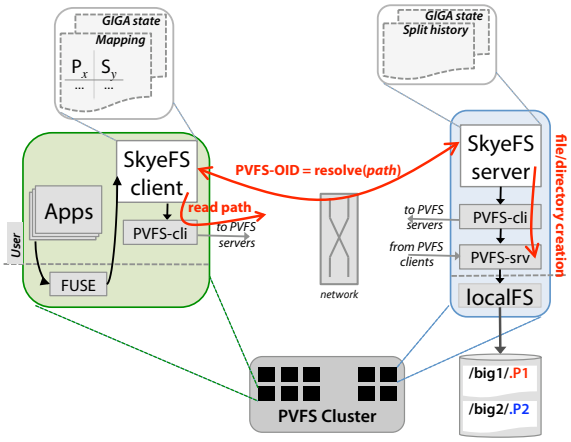

Figure 1. SkyeFS Architecture Diagram

```
struct skye_directory {
    giga_mapping mapping;
    int reference_count;
    PVFS_object_ref PVFS_handle;
    int splitting_index;
    pthread_rwlock_t rwlock;
    UT_hash_handle hashtable_handle;
};
```

Figure 2. Giga+ Directory Metadata

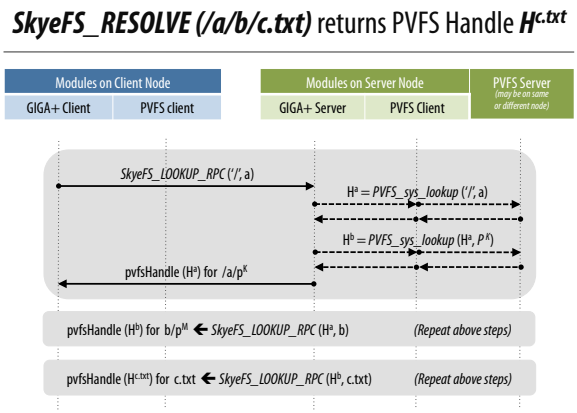

Figure 3. SkyeFS Name Resolution

2.2 Client/Server Architecture

Giga+ has metadata that it must maintain in order to manage each directory's partitions. SkyeFS uses a small server process, `skye_server`, located on each PVFS server to manage this metadata and ensure consistency for the directories and partitions resident on that server. The `skye_server` has a cache of `skye_directory` structures which store this metadata. This metadata can be regenerated from PVFS at any time, allowing the `skye_server` to fail or evict items from its cache at any time without the risk of leaving the system in an inconsistent state.

Each client runs a `skye_client` process that provides a FUSE filesystem and issues requests to the `skye_server`s using ONC RPC and PVFS using the PVFS system interface.[4] For most operations, the client is a simple adaption of the `pvfs2fuse` code to use the lowlevel FUSE API. However, metadata operations which require path name resolution or object creation are forwarded through the `skye_server` to prevent race conditions. To speed operation, the client also maintains a cache of `giga_mapping`s for recently accessed directories.

For performance and load-balance reasons, locality between the `skye_server` process and the PVFS server to which requests are being issued is maintained whenever possible. Upon receiving a RPC request, the server will first check if the partition to be accessed by the request is local. If not, the server will return the error EAGAIN and provide the client with a copy of its own `giga_mapping` for the directory. Upon receiving this error, the client will read the provided mapping and merge that information with its local version. The client will then reattempt the RPC to the new `skye_server`, as per the updated mapping. This ensures that locality is maintained and that the client's cached mapping will be updated to reflect new partitions.

The most important job of the `skye_server` is to prevent race conditions during splitting. Any operation that must resolve a pathname (such as a lookup, create, or rename) needs to have an accurate Giga+ bitmap for the directory so that they can operate on the correct partition. By forwarding these operations to the `skye_server`, the server can ensure that these operations proceed without interference from changing bitmaps. There do exist optimistic schemes that would allow client applications to perform some of these operations without server involvement, however we believe that they would incur unacceptable performance overhead and require significantly more complex operations.

2.3 Path Resolution

The FUSE lowlevel API uses a `lookup(inode, name)` callback to perform path resolution. We use an object's PVFS handle as the FUSE inode to avoid keeping a lookup table on the client. PVFS provides a `PVFS_sys_ref_lookup(handle, name)` which is analogous to the FUSE `lookup` function. Our `skye_server` wraps this PVFS function to both resolve the Giga+ partition and the object itself and exposes a `lookup` RPC to accomplish this. When the `skye_client` receives a `lookup` callback from FUSE, it consults its cached bitmap (if any) and then makes an RPC call to the responsible `skye_server`, retrying in the event that an EAGAIN is returned. By resolving both the partition and requested object in the server, we avoid race conditions due to concurrent splits.

As PVFS metadata handles do not change when their objects are moved between directories, the handle returned by `lookup` is guaranteed to continue to be a valid reference to the object for the duration of the object's existence. This is true even if the partition holding an object splits or the logical directory is moved elsewhere in the directory tree. As a result, we can safely return a PVFS handle to FUSE in the form of an inode umber.

2.4 Metadata Persistence

All of the metadata required by Giga+ is persisted directly in PVFS. Whenever possible, we prefer to rederive metadata from PVFs instead of storing new Giga+ metadata. While this may come at a performance cost, preventing opportunities for inconsistencies makes the system simple to implement and highly resilient to failures.

The only filesystem-wide metadata is the server list. Each SkyeFS client and server queries this information from PVFS directly. This allows the system administrator to change server host names or add new servers by modifying only the PVFS configuration.

The most important piece of per-directory metadata is the Giga+ bitmap. This consists of bits indicating the presence (or absence) of each possible partition, the server number of the zeroth server and the number of servers at the time of creation. To determine the partitions present in a logical directory, we issue a `readdir()` against the directory and parse the resulting directory names to determine the partition. Future versions of SkyeFS will use a different prefix for currently splitting partitions to avoid prematurely adding these to the bitmap and to enable resuming splitting after a failure. The zeroth server is always the server responsible for the logical directory. To support server additions, the number of servers at creation can be stored as an extended attribute on the directory.

2.5 Client Bootstrap

To avoid creating additional load on PVFS, the `skye_client` only loads a subset of the metadata from PVFS. The first time a directory is accessed, it is assumed that there is only one partition. The client can determine the identity of the zeroth server without incurring any PVFS operations. If the client issues a RPC to the `skye_server` for the directory which results in an EAGAIN error, it takes the bitwise OR of the current bitmap and the server provided bitmap, allowing the client bitmap to be filled-in by the server without additional PVFS operations.

Populating the initial mapping in this way causes the `lookup()` code to always start at the zeroth server in the cache-cold state. While this ensures that some progress towards finding the correct server is always made, it has the potential to cause unbalanced load on servers which host the zeroth partition for a popular directory. A future optimization might include having the

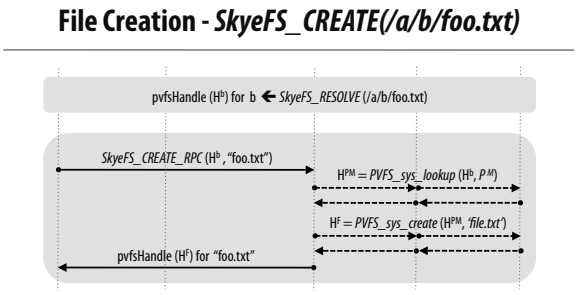

Figure 4. SkyeFS Create Procedure

`skye_server` provide a bootstrap bitmap to clients when returning a `lookup()` result for a directory that is already cached on the server.

3 Filesystem Operations

3.1 Splitting and file creation

The `skye_server` provides a `create()` RPC for the `skye_client` to create files in a given directory. As with all `skye_server` RPCs, only the server responsible for a partition will service create requests for that partition. Each server manages splits for its partitions and coordinates operations that happen concurrently with a split.

Splits are triggered in Giga+ when the partition size exceeds a threshold. The current splitting status is maintained by the `splitting_index` field of the `skye_directory` structure. When no split is happening, this field has the value -1; other times it has the number of the currently splitting partition. Only one partition per directory may split at a time under this scheme, however this is a desirable property for performance reasons.

The `skye_directory` structure contains a read-write lock which is used to synchronize changes in the splitting state. Operations that act on the directory, such as `lookup` and `create`, take a read lock for their duration. When a partition is to be split, the write lock is taken, the `splitting_index` updated, and then the write lock is dropped. Similarly, when splitting is complete, the write lock is taken while updating the `splitting_index`. This scheme ensures that each operation acting on a directory has a consistent view of the splitting state throughout the operation. Without this synchronization, a complete bitmap comparison would be required to determine if a split had interfered with a concurrent operation making retry and recovery complex. The read-write lock is initialized with the `PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP` option to prevent starving writers.

With a one percent probability, we evaluate the partition of a new object at the end of each `create` operation to see if the partition needs to be split. This check is implemented using the PVFS `getattr` operation which will return the number of directory entries in a directory. If the directory size is greater than the split threshold, the partition is added to a queue of partitions to split and the `create` call returns.

Each `skye_server` has a dedicated thread for splitting partitions. This thread waits on a condition variable associated with the split queue. When a partition is added to the queue, the adding thread signals on the condition variable to wake up the splitter thread. Upon waking, the splitter thread will iterate through the queue, confirming that the partitions are still over threshold

and splitting them if needed.

To split a partition, the splitter thread performs repeated `readdir` operations on the partition to determine the set of candidate entries. For each entry, it hashes the name to determine if the object is to be moved to the new partition and, if so, moves the entry using the PVFS `rename` operation. This process continues until a complete `readdir` of the directory finds no new entries to move. In practice, this usually requires three passes. When the split is complete, a `bucket_add` RPC is sent to the server responsible for the new partition. This RPC will cause the new server to add the partition to its bitmap and, therefore, assume responsibility for the partition.

While a split is occurring, any thread that accesses the directory will notice that `splitting_index` is not set to -1 allowing the operation to modify its behavior to operate correctly despite the split. In the case of object creation, objects are created in the child partition if that is where they would be migrated. In the case of `lookup`, both the child and parent partitions are checked for the object.

We use a separate splitting thread to allow splitting to take place asynchronously with respect to the triggering `create` operation. Our locking semantics allow filesystem operations to continue on a partition during its split. These features can result in significantly improved performance over implementations that must block all operations during a split.

3.2 Directory Creation

In most respects, `mkdir` is implemented identically to `create`. When the `skye_server` creates a directory it also creates that directory's first partition. No additional work is done at creation time to setup Giga+ metadata as the server which creates the directory is usually not the zeroth server for the new directory. The first time the new directory is accessed, the zeroth server will load the Giga+ metadata as if it was simply an old, but empty, directory. In the event that a server fails between creating a directory and its zeroth partition, the directory will be in an inconsistent state. However, this can be easily repaired by the server when such a directory is first accessed.

3.3 Object Removal

Removing a file is a relatively simple operation, we simply issue a PVFS remove for the file. Currently, we do not remove empty partitions or coalesce partitions which shrink to below the split threshold.

However, `rmdir` presents a challenge because the directory removal must be coordinated between all servers which are responsible for a partition of the directory. Additionally, we must ensure that the remove operation on a directory will fail if there exist any files in the directory.

To coordinate removal of a directory, we implement a server to server RPC call called `bucket_remove`. The client attempting to remove a directory will send a `remove` RPC to the server holding the directory entry for the directory. The server will then issue a `bucket_remove` RPC for each bucket in descending order (i.e. the highest numbered bucket first).

When a server receives this call it will attempt to remove the specified partition by issuing a PVFS `remove` for the partition. If the operation succeeds then the partition will be removed from the owner's bitmap and the owner will return a positive response to the caller. The caller will also remove the partition from its bitmap and then issue the next `bucket_remove`.

If the PVFS `remove` fails (for any reason other than `ENOENT`) the partition will remain in the bitmap and a negative response will be returned to the server. The coordinating server will then halt the operation and return the error to the client. In this case care must be taken to ensure that the system is left in a consistent state. By removing the partitions in descending order we ensure

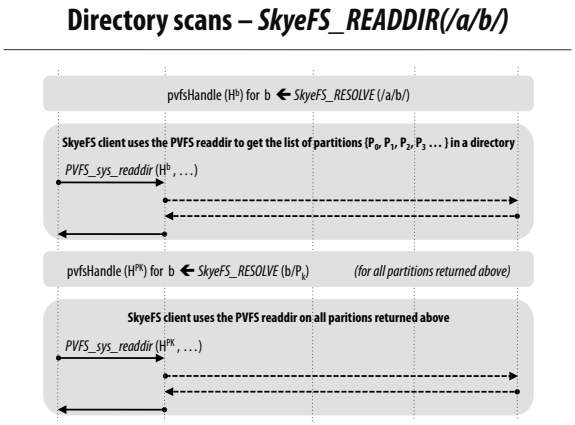

Figure 5. SkyeFS readdir()

that we always remove the child of a split before its parent. Therefore, each intermediate bitmap (produced between `bucket_remove()` calls) is a valid bitmap in itself.

However, for a client to behave correctly after an aborted `rmdir` operation, we need a mechanism to remove the deleted partitions from the client bitmap. As the standard merge operation performed by the client does a logical OR of the old and new bitmaps it is unable to handle a removal. Instead, we rely on a fail safe mechanism. If a client receives an `EAGAIN` from a server but merging the returned bitmap does not add any new partitions, the client will assume that a partition was removed and replace its bitmap with the server provided bitmap. This behavior is not currently implemented in SkyeFS, but may also allow for removal or coalescing of partitions as a directory shrinks.

3.4 Rename

The `rename` operation is also challenging because it must operate on two distinct objects that may not be located on the same server. To solve this we use an optimistic retry strategy. The `skye_server` implements a `partition` RPC that is similar to `lookup` but instead of returning the handle of the requested object it returns the handle of the encompassing partition. The client uses `partition` to determine the handle of the source's parent partition. This handle (and the source and destination file names) are passed to the destination partition's `skye_server`. The `skye_server` will attempt to move the file using the provided source handle and name into the correct destination partition.

In the event that a split has happened between the `partition` call and the `rename` call, causing the source partition to change, the `rename` will return `ENOENT`. The client will then retry the `partition` RPC to see if the partition's handle has changed. If so, the rename is retried with the new handle.

This technique eliminates the need for any locks to be held across multiple RPC calls but is vulnerable to starvation in the event of high rates of splitting. We believe starvation is unlikely to occur in practice.

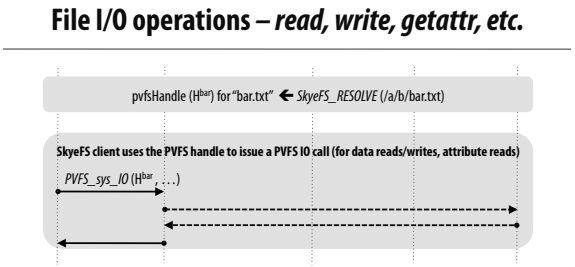

Figure 6. SkyeFS IO Operations

3.5 Directory Listing

To simplify `readdir` and ensure correct semantics, the `skye_client` implements the actual directory reading during the FUSE `opendir` callback at which point the contents of the directory are loaded into an array from which each subsequent `readdir` command is serviced.

Because each Giga+ partition is implemented as a distinct PVFS directory, it would be very difficult to implement a fully consistent readdir. Instead, we make a best effort and avoid synchronization. When the `skye_client` receives an `opendir` call it first issues a PVFS `readdir` on the logical directory itself to get a list of partitions (both complete and splitting). The client then iterates through each of these partitions and issues a PVFS `readir` for each partition, concatenating the results.

While this technique is prone to returning inconsistent results in the case of concurrent modifications, it places a minimum load on the PVFS servers. However, it is important to note that this is only an issue for directories larger than the split threshold and most directories will fall well below this threshold. In the future, `skye_client` could aggressively fetch directory entries from multiple PVFS servers in parallel to complete the readdir more quickly.

3.6 Other Operations

All other operations are relatively straightforward modifications of their pvfs2fuse counterparts with the PVFS `lookup` operation replaced by our RPC. This includes both data operations such as `read` and `write` as well as metadata operations like `stat` and `chmod`. Because we resolve all objects to PVFS handles inside their Giga+ partitions, these operations can proceed on the client without concern for future splitting.

4 Other Considerations

4.1 Multi-step Lookup

In the current system, a `lookup` call can only descend one step in a path traversal. Future work could extend the operation to provide the entire path to the `skye_server` and allow the server to descend as many directories in the path as it owns. This would be of limited value in the current system where the servers for a parent and child directory are chosen independently. At the cost of load balancing, parent and child directories could be placed often on the same server to allow this mechanism to speed directory lookups. One example scheme to achieve this would be to place all new directories on the same server as their parent. When a directory splits the first time the zeroth server would be moved to a new server chosen at random. This would have the effect of

keeping strings of small directories on the same server while still ensuring that large directories are load-balanced.

4.2 Server Addition

PVFS includes very limited support for server addition. While new servers can be added to the configuration file for a filesystem and brought online they must take a previously unoccupied part of the handle space and there exists no mechanism for automatically migrating either data or metadata to the new server. The current SkyeFS implementation includes no specific provisions for addition of servers, however future work could use SkyeFS to support the migration of some data to new PVFS MDS. In particular, by splitting overfull but already load-balanced directories to these new servers some load can be moved to the servers in already existing directories. Because SkyeFS will store the number of servers existing at the time of creation in each directory as an extended attribute of that directory, no specific mechanism is needed to add a new server other than adding the server to the PVFS cluster and restarting all `skye_server` processes.

4.3 Fault Tolerance

PVFS is not a redundant file system and has very limited support for fault tolerant operation. As a result, we do not make any attempts to provide redundant services or fall over support in the case of failures. We assume that `skye_server` processes and the PVFS servers fail together and that any failure will render the system unusable until resolved.

However, we do make every effort to leave the system in a consistant state at all times should any component fail. Any skye process can fail at any time and the resulting state will be repaired transparently upon restart. This is primarily due to the lack of additional SkyeFS metadata that is required on top of the PVFS file system. By carefully controlling the sequence of actions we take on the PVFS file system we are able to ensure that any state of PVFS is recognizable as either complete or the result of an unfinished action which can then be resumed or aborted.

5 Correctness

To ensure correctness of the system, we tested against a modified version of FreeBSD's fstest suite.[6] Our version was limited to the tests for `chflags`, `chmod`, `chown`, `mkdir`, `open`, `readdir`, `rename`, `rmdir` and `unlink`. We removed tests that required hard links, which SkyeFS does not support.

6 Performance Analysis

6.1 Method

Tests were performed on the Marmot cluster consisting of 128 nodes each with two single core Opteron CPUs, 16GiB of RAM, one SATA disk and Gigabit Ethernet. PVFS was configured with default settings. In production configurations, we expect that PVFS would be run on fast disk arrays. To prevent the single 7200 RPM disks available from artificially limiting throughput, we ran all experiments with PVFS configured to use a tmpfs for storage.[7] In each experiment, both the clients and servers were located on the same set of machines

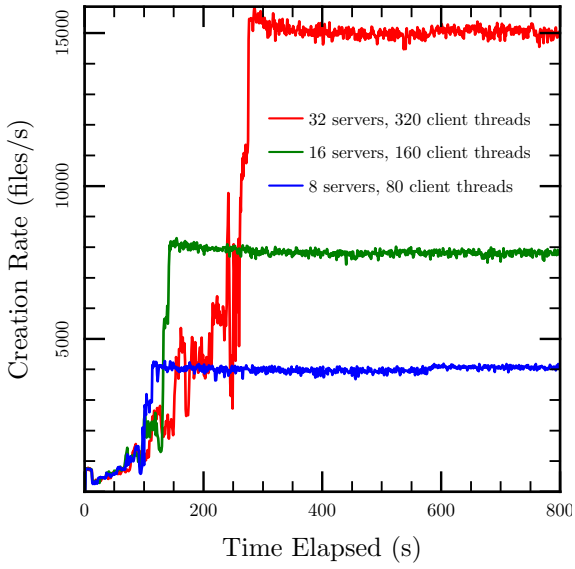

Figure 7. SkyeFS Empty File Creation Throughput

6.2 File Creation

To test directory insert rates we started with an empty filesystem and created a single directory. For each server, we started 10 clients simultaneously which all attempted to insert uniquely named files into the directory. We recorded the overall rate of inserts in the entire system.

For 32 servers, the system reached a peak throughput of over 15k creates per second after 4 minutes and 30 seconds. For 16 servers, peak throughput was 8k creates per second after 2 minutes and 25 seconds. And for 8 servers, peak throughput was 4k creates per seconds after 1 minutes and 55 seconds.

This experiment provides the expected linear scaling of throughput after the system has become load balanced. We believe the increased time required to get to a load-balanced state for the 32 server experiment is due to higher resource contention during the first few splits. Future work might look at ways to quicken these first few splits in the event of very high load to prevent this problem.

We note that the use of FUSE turns each `mknod` into a pair of `lookup` and `mknod` requests. The use of the SkyeFS operations directly, i.e. as part of a shared library, might be able to drastically improve this performance by avoiding the superfluous `lookup`.

6.3 Readdir

One of the desirable properties of the Giga+ algorithm is that it keeps the overhead for a `readdir` small in the case of small directories. To test this, we created directories of various sizes and measured the time it took a single client to read all the entries in the directory. We repeated this experiment with 8, 16, and 32 servers.

The client was able to complete a `readdir` for a directory with 1000 entries in 0.1 seconds. As expected, this increased slightly super linearly as the number of entries and partitions increased. In the case of 200k entries, a `readdir` took approximately 2.4 seconds. No significant difference in performance was noted between the different server configurations.

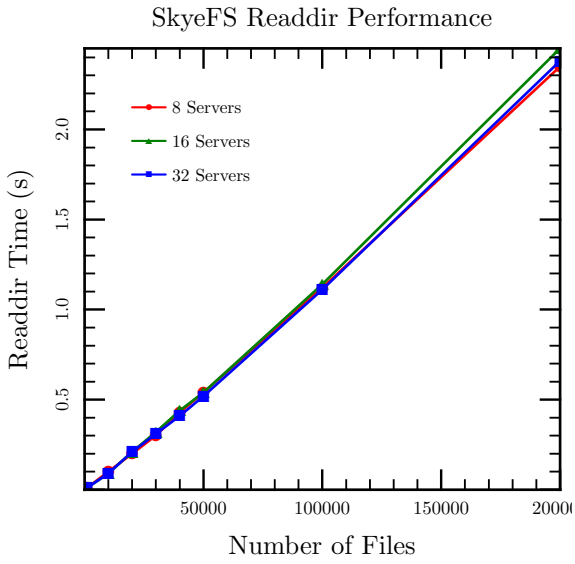

Figure 8. SkyeFS Readdir Performance

Further work might improve upon this performance by issuing the `readdir` for each partition in parallel.

6.4 FUSE Low Level API

Our initial prototype system used the standard FUSE "high level" API. This API provides a full pathname in every callback. As a result, each `mknod` call in our file creation experiment required resolving the PVFS path from the root of the filesystem. This created an excessive amount of traffic on the server responsible for the filesystem root and bottlenecked the system. We were able to resolve this problem by switching to the lowlevel API and running our benchmark utility from within the directory in which the files are to be created.

6.5 SkyeFS Client and PVFS

Our initial prototype included a fully multi-threaded client. However, we quickly noticed that executing many operations in parallel (e.g. using `make -j8`) would result in long stalls while executing PVFS calls. To work around this problem, we currently run the client in the single threaded FUSE mode.

The FUSE lowlevel API allows the filesystem to return from the callback without returning a result to FUSE. After consulting with the PVFS developers, it has been suggested that using this functionality along with the asynchronous PVFS operations might allow concurrent operation without experiencing these stalls. This was not attempted for this project because of the large engineering effort required to convert each operation to persist its state across asynchronous PVFS calls.

6.6 SkyeFS Server and PVFS

The `skye_server` is based on the initial Giga+ prototype and uses a single thread to handle each incoming RPC connection. As a result, it is easy in our experiments for upwards of 100 client operations to be in flight concurrently. The PVFS system interface keeps an array of state machines for all outstanding operations in the address space. All threads which have an in-flight operation attempt to take a lock on this array and the thread which acquires the lock drives progress on all state machines in the array.

The first problem that results from this design is that the array is of limited size. Currently, the PVFS code statically defines the array of state machines to be of length 256. If an incoming operation would overflow this array, an assertion is tripped. To avoid tripping this assertion we implemented flow control in the form of a semaphore. When the server starts the semaphore is initialized to a small value. When a thread receives an RPC request, it downs the semaphore before issuing any PVFS requests. When the thread completes its work, it will up the semaphore. In this way, we prevent more than the semaphore's initial value threads from issuing PVFS requests concurrently. In practice, we found that we needed to set this initial value as low as 32 to avoid tripping the assert.

The other problem with the PVFS client design is that it does not guarantee fairness to the requests on the array. This means that while a request may have completed, the requesting thread may not notice this for a very long time. This introduces significant jitter into client response times and slows the entire system's throughput as a client is unnecessarily blocked on a server response. To solve this problem, we further restrict the number of concurrent operations in flight by initializing our flow control semaphore to 12. Our testing indicated that this value is optimal in the 32 server case.

6.7 Split Performance

During a split, the split thread must issue `readdir` calls to the partition which is being split. While this is happening, additional creates may be issued against that partition. We found that without external synchronization, this pattern of requests results in very poor PVFS performance.

We wrote a test program to isolate this particular workload. The program connects to a provided PVFS server using the system interface. It then spawns 10 threads which all create files in that directory until 5000 files are created. It also spawns another thread that will list out the contents of that directory repeatedly, reporting the time taken each time. The program supports three synchronization models for the threads to test different levels of interleaving of requests.

In the unsynchronized model all threads are allowed to run without any synchronization. On our test system, this required 42 seconds and the directory listings took between 2 and 12 seconds to complete with a median of 7 seconds.

In our create-synchronized model, each of the 10 threads that issue creates synchronizes on a single mutex such that only one create is ever in flight at a time. With this model, 41 seconds were required to create the files and the listings took between 2 and 20 seconds to complete.

In our final, everything synchronized model, all create threads and the readdir thread synchronized on the same mutex. With this model, the creates take 43 seconds to complete and all listings take less than 200ms.

To overcome the PVFS stalls in the unsynchronized and create-synchronized models, our splitter thread initially drops the write lock after updating the `splitting_index` but then reacquires the write lock for each set of `readdir + rename` operations. This allows some degree of concurrency (outstanding operations can complete between each set of operations) while still preventing

the observed stall.

7 Related Work

SkyeFS implements the Giga+ technique for distributing filesystem metadata presented in [1]. As discussed in [1], Giga+ builds on a wide range of previous work on distributed file systems and data structures. We adapt the code from the initial Giga+ prototype for use on a distributed file system (DFS) and show that the scalability results seen by the initial prototype can be seen when Giga+ is implemented in the context of a DFS.

SkyeFS also builds on the FUSE module for the Linux kernel and the PVFS client library to provide filesystem services and consume PVFS services respectively. The `pvfs2fuse` application distributed with PVFS provided valuable insight on how to bridge between the FUSE API and the PVFS API. When possible, portions of code from this application were used verbatim.

OrangeFS is a native implementation of Giga+ in PVFS.[3] OrangeFS shows similar scaling performance to SkyeFS, however it lacks the ability to incrementally grow the number of partitions in a directory.

8 Conclusion

We successfully implemented Giga+ distributed directories on top of PVFS. We demonstrated that the capabilities provided by PVFS are sufficient for a high performance implementation of distributed directories without requiring modification to PVFS itself. This shim technique may be adaptable to implementation of distributed metadata on other distributed filesystems. We've shown that Giga+ is capable of achieving near linear speedup once a directory is at load balance. We've also demonstrated that Giga+ preserves the desired responsiveness properties of PVFS in the case of small directories. We note that future work is needed to overcome performance problems in PVFS exposed by the SkyeFS workloads.

Acknowledgements

We'd like to thank Sam Lang, Phil Carns and Rob Ross for their advice and assistance in working with PVFS. This material is based upon research supported in part by the Notional Science Foundation under contract NFS CNS-1042543 (PRObE) and the Los Alamos National Lab under contract number DE-AC52-06NA25396 (IRHPIT). We also thank the members and companies of the PDL Consortium (including Actifio, American Power Conversion, EMC Corporation, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc., Oracle Corporation, Panasas, Riverbed, Samsung Information Systems America, Seagate Technology, STEC, Inc., Symantec Corporation, VMware, Inc., Western Digital) for their interest and support.

Source Code

The source code for SkyeFS is available under the terms of the GNU Lesser General Public License. It can be downloaded at https://github.com/achivetta/skyefs.

References

[1] Patil, S. and Gibson, G. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. Proceedings of FAST '11: 9th USENIX Conference on File and Storage Technologies.

[2] P. H. Carns, R. B. W. B. Ligon III, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. Proceedings of the 4th Annual Linux Showcase and Conference, 2000.

[3] S. Yang, W. Ligon, and E. Quarles. Scalable Distributed Directory Implementation on Orange File System. 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O, 2011.

[4] Srinivasan, R. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Aug. 1995.

[5] FUSE. Filesystem in Userspace. http://fuse.sf.net

[6] Pawel Dawidek. fstest from FreeBSD. http://people.freebsd.org/~pjd/fstest/

[7] tmpfs: a file system which keeps all files in virtual memory. http://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt