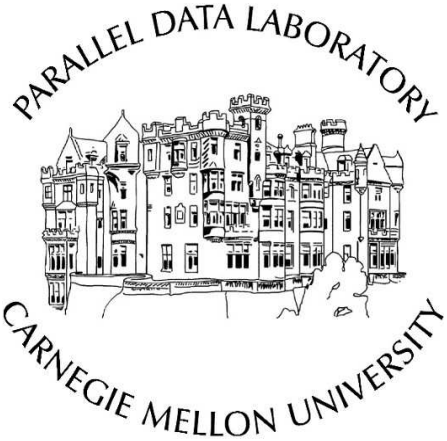

GIGA+ : Scalable Directories for Shared File Systems

Swapnil Patil Garth Gibson

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

CMU-PDL-08-110
October 2008

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

Acknowledgements: We would like to thank several people who made significant contributions in improving this paper. Ruth Klundt put in a significant effort and time to run our experimental evaluation at Sandia National Labs, especially getting it working few days before a deadline; thanks to Lee Ward who offered us Sandia's resources. James Nunez and Alfred Torrez helped us run experiments at Los Alamos National Lab. Aditya Jayaraman, Sanket Hase, Vinay Perneti and Sundar Sundaraman built the first FUSE prototype as their course project. Discussions with Milo Polte, Sam Lang, Rob Ross, Greg Ganger and Christos Faloutsos improved various aspects of our design. This material is based upon research sponsored in part by the Department of Energy under Award Number DE-FC02-06ER25767, the Petascale Data Storage Institute (PDSI) and by the Los Alamos National Lab under Award Number 54515-001-07, the CMU/LANL IRHPIT initiative. We also thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft Research, NetApp, Oracle, Seagate, Symantec, and VMWare) for their interest and support.

Keywords: scalability, weak consistency, directories, concurrency

Abstract

Traditionally file system designs have envisioned directories as a means of organizing files for human viewing; that is, directories typically contain a few tens to thousands of files. Users of large, fast file systems have begun to put millions of files into single directories, for example, as simple databases. Furthermore, large-scale applications running on clusters with tens to hundreds of thousands of cores can burstily create files using all compute cores, amassing bursts of hundreds of thousands of creates or more.

In this paper, we revisit data-structures to build large file system directories that contain millions to billions of files and to quickly grow the number of files when many nodes are creating concurrently. We extend classic ideas of efficient resizeable hash-tables and inconsistent client hints to a highly concurrent distributed directory service. Our techniques use a dense bitmap encoding to indicate which of the possibly created hash partitions really exist, to allow all partitions to split independently, and to correct stale client hints with multiple changes per update.

We implement our technique, GIGA+*, using the FUSE user-level file system API layered on Linux ext3. We measured our prototype on a 100-node cluster using the UCAR Metarates benchmark for concurrently creating a total of 12 million files in a single directory. In a configuration of 32 servers,* GIGA+ *delivers scalable throughput with a peak of 8,369 file creates/second, comparable to or better than the best current file system implementations.*

1 Introduction

1.1 Motivation

Demand for scalable storage I/O continues to grow rapidly as more applications begin to harness the parallelism provided by massive compute clusters and "cloud computing" infrastructures [34, 20]. Much of the research in storage systems has focused on improving the scale and performance of the "data-path", operations that read and write large amounts of file data. Scalable file systems do a good job of scaling large file access bandwidth by striping or sharing I/O resources across many servers or disks [16, 18, 35, 28]. The same cannot be said about scaling file metadata operation rates.

Two trends motivate the need for scalable metadata services in shared file systems. First, there is a growing set of applications in scientific computing and Internet services (examples in §2.1) that use a file system as a fast, lightweight "database" with files as unstructured records in a large directory [34]. The second trend – growing application-level parallelism – increases the potential concurrency seen by the metadata service. Clusters today consist of thousands of nodes growing to tens and hundreds of thousands, and each node has one to eight or more multi-core processors. Highly parallel applications can be expected to be soon executing more than hundreds of thousands to millions of concurrent threads. At this high level of real concurrency, even simple output file creation, one per thread, can induce intense metadata workloads.

Most file systems, like NFS [32], NTFS [6], SGI's XFS [43], Lustre [28] and PanFS [49], manage an entire directory through a single metadata server (MDS), thus limiting the overall scalability of mutations to the system's metadata service. Customers in need of more metadata mutation throughput usually mount more independent file systems into a larger aggregate, but each directory or directory subtree is still managed by one metadata server. Some systems cluster metadata servers in pairs for failover, but not increased throughput [11]. Some systems allow any server to act as a proxy and forward requests to the appropriate server; but this also doesn't increase metadata mutation throughput in a directory [24]. Symmetric shared disk file systems, like Redhat GFS [15] and IBM GPFS [35], that support concurrent updates to the same directory use complex distributed locking and cache consistency semantics, both of which have significant bottlenecks for concurrent create workloads, especially from many clients working in one directory. Moreover, file systems that support client caching of directory entries for faster read only workloads, generally disable client caching during concurrent update workload to avoid excessive consistency overhead.

1.2 Our contributions

In this paper, we describe techniques, called GIGA+, for a simple, highly decentralized metadata service. We have built a POSIX-compliant directory implementation by stacking GIGA+ techniques over Ext3 on multiple servers using FUSE [14]. Our long term goal is to push the limits of metadata scalability by building directories containing billions to trillions of files and handling more than 100,000 metadata operations/second. In contrast to several attractive "domain-specific" systems that achieve similar scalability (like Google's BigTable [5] and Amazon's Dynamo [9]), GIGA+ builds file system directories that offer UNIX file system semantics, for example, no duplicates, no range queries, and unordered `readdir()` scans.

The central tenet of our research is to avoid system-wide consistency or synchronization. GIGA+ starts with well-known, out-of-core indexing techniques that incrementally divide a directory into non-cacheable[1] hash partitions in a way that achieves incremental growth and load-

[1]While we have not implemented read-only optimizations, it is common practice in distributed systems to enable

balancing across all the servers [13, 12, 26, 27]. The distinguishing feature of our distributed index is that each server expands its portion of the index without any central co-ordination or synchronization between servers or clients.

GIGA+ uses a simple, dense, fine-grain bitmap to map filenames to directory partitions and to a specific server. Clients use it to lookup the server that stores the partition associated with a given filename. Servers use it to identify another server for a new partition to receive a portion of the local partition that overfilled and to update a client's view of the index. A bitmap is fast and simple to maintain, and is compact enough (a few bytes to a few kilobytes for a billion file directory) to be cached in-memory effectively and to be shipped over the network piggybacked with operation status results as needed.

Our indexing scheme delegates different partitions of the directory to different servers. In order to build its view of each directory's bitmap, each server keeps track of the destination server of all prior split operations on their partitions. This "split history" is stored locally as an attribute of the respective partition. Thus, by not using any globally shared state on the servers, GIGA+ servers can grow (or shrink) their partitions without any overhead from lock contention and synchronization bottlenecks.

The drawback of not using a global consistency mechanism is that clients wanting to contact a server about a file may use an out-of-date copy of a bitmap, specifically, with workloads that perform high rates of concurrent updates. GIGA+ tolerates the use of stale, inconsistent state at the client without affecting the correctness of their operations. Clients may send a request to an "incorrect" server, which will correct the client. On detecting that a request has been incorrectly addressed to it, a server uses its bitmap to update the client's cached copy. This ability to lazily update the client's bitmap avoids the need to keep synchronized mapping state at the clients. In addition, correctable state at clients simplifies the client's failure recovery semantics.

We have built a prototype implementation of GIGA+ in a cluster of Linux nodes using the FUSE (filesystem in user-space API) [14]. We use the UCAR Metarates benchmark to measure our prototype on a large 100-node cluster configured with up to 32 servers. Our prototype results show that untuned GIGA+ delivers 8,369 file creates per second, significantly higher than some production systems.

The rest of this paper describes the design, implementation, and evaluation of GIGA+. We continue in the next section by motivating the rationale for huge directories and elaborate on how GIGA+ differs from current related work. §3 presents an overview of GIGA+ before describing the details of our indexing technique. §4 describes our prototype implementation, experimental setup and results of our system.

2 Background and Related Work

2.1 Rationale for large FS directories

Among file system vendors and users, there is a small but persistent voice calling for huge directories and fast create rates. So we decided to explore alternatives that have been proposed in the past and implementations that are currently available. Unfortunately, as described later in §2.2, the best production system does not scale concurrent creates into one directory. Because concurrent creates in a single directory is largely neglected in research and not well implemented in production, we decided to develop a framework for exploring the scalability roadblocks when scaling up with the

client-side caching of non-changing directories (or in our case, partitions of a directory) under the control of each server independently

largest computers. In the rest of the section, we motivate the need for large directories and then argue the need for a file system interface, instead of other interfaces like databases.

Applications, often seen in long running scientific computing applications and Internet services, create a large number of small files in a single directory either steadily or burstily.

- **Steady small file creation:** Phone companies monitor and record information about their subscribers' calls for billing purposes. Typically for every call, this monitoring application creates a file that logs the start and end time of that call. Todays telecom infrastructure can support more than hundred thousand calls per second [45]. Even a system that is running at one-third utilization can easily create more than 30,000 files per second. Similarly, there are applications that store the output generated from scientific experiments in domains like genomics and high-energy physics. For instance, a file may be created for every atomic particle created in a physics experiment or a file may store information from each splice of a gene sequencing microarray experiment.

- **Bursty small file creation:** Another example comes from applications that perform per-process memory state checkpointing, where every core in a large cluster runs a process. An application running on PetaFLOP scale computers, for example a 25,000-node cluster with 16-32 cores per node, may nearly simultaneously create more than half a million files.

Our goal is to build a huge directory implementation for a POSIX-compliant file system and support traditional UNIX file system semantics. This rules out certain otherwise attractive interfaces; we ruled out using a database interface for the following reasons. First, the VFS interface enables a faster adoption of our techniques by making them backward compatible for legacy applications written for a file system interface. Second, traditional "one size fits all" databases aren't always the best solution for all problems [41, 40, 39]. Specialized solutions often outperform the traditional RDBMS approaches; there is an growing trend of re-writing "from scratch" specialized databases for high availability (like Amazon's Dynamo [9]) and high performance (like Google's BigTable [5] built on the Google file system [16]). In addition, most applications generate very heterogeneous data; extracting value from semi- and un-structured data is not well supported by traditional database indices. Large database systems are also often coupled with additional features like query optimizers, transactional semantics and lock hierarchies. Applications may not need most of this functionality; they may want simpler, faster semantics. Finally, we want to see how far we can push highly concurrent file system directories.

2.2 Related Work

We present the related work in two parts: out-of-core indexing structures and practical challenges in using these indices to build scalable storage systems.

2.2.1 Out-of-core Indexing

Out-of-core indexing structures, such as hash tables and B-trees, are widely used to index the disk blocks of large datasets between disk and memory in both databases and file systems. File systems have used both types of structures (or their variants) for directory indexing; e.g., SGI's XFS uses B-trees [43] and Linux Ext2/3 uses hash-tables [44]. For extreme scale, indexing structures should have four properties: load-balanced key distribution, incremental growth, ability to support high concurrency, and the ability to be distributed over hundreds to hundreds of thousands of cores or more.

To eliminate hot-spots, the indexing technique should uniformly distribute the keys over all the buckets. In a distributed system, it is also important to achieve uniform load distribution with respect to the servers. In fact, buckets don't need to have the same load provided that the servers each holding a collections of buckets have the same aggregate load. We use a strong hash function (MD5) that achieves load-balancing of keys with respect to both, buckets and servers.

The directory index should grow incrementally with usage. Particularly in file systems, where most directories are small and they see the bulk of accesses [3, 8], it is important that small directory performance should not be penalized. B-trees naturally grow in an incremental manner but require logarithmic partition fetches per lookup, while hash-table which support lookups in a single partition fetch do not. If a hash-table bucket is full, all keys need to be re-hashed into a bigger hash-table. This can be inefficient in terms of both space utilization and re-hashing overhead for a large number of keys. We look at three variants that have addressed this problem and are most closely related to our work: extendible, linear, and consistent hashing.

Extendible hashing uses a specific family of hashes in order to defer rehashing keys in most buckets after the number of buckets is doubled [13]. This technique doubles the number of bucket headers in one step, with two headers pointing to each bucket. And then it splits each overflowing bucket as needed by creating a new bucket, transferring half the keys from the old bucket and updating one of the bucket pointers to point to this new bucket. The main drawback of this scheme is that it assumes that the hash-table pointers are globally available on a single machine. As far as we can tell the best distributed version of this algorithm is in development as an extension of IBM's GPFS [35, 1] (discussed in §2.3), and that it will still have interesting cache consistency issues for the global pointer table.

Linear hashing (and its variant LH*) is another technique that grows a similar hash table dynamically [26, 27]. LH* allows clients to have stale hash table pointers by restricting the bucket splitting to happen in a strict serial ordering. The growth of the index is controlled by two variables, a radix and a split token. The radix is used to identify the number of times a bucket has doubled and the split token enforces the order of splitting by referencing the bucket that is supposed to split next. This serial split order needs a central co-ordinator, and this order may delay splitting an overflowing bucket for a long time. Moreover, servers traversed in the past do not know about later updates to the split token until it gets back to them, so updates to stale clients are also stale. While LH* has inspired us to use stale information, GIGA+ provides more concurrency by allowing multiple splits to happen simultaneously without any specific order. This is enabled by using additional state, represented as a bitmap, that is used to record the splits executed by each server and to update the clients when they address an incorrect server. Finally, we believe that GIGA+ is simpler than LH* and its extensions.

Consistent hashing forms the basis of distributed hash-tables (DHTs) used widely for Internet-scale P2P systems. Consistent hashing dynamically divides the keyspace into multiple regions and assigns a node to manage that region based on the hash of an unique name of the node [25]. Each node knows about a few other nodes in the system based on their order of the keyspace range managed by that nodes. The principal advantage of consistent hashing, which makes it attractive for Internet scale systems, is that the addition and removal of any node only affects its immediate neighbors. This property is crucial for designing systems where one expects a high rate of node arrival and departure, as one should in Internet P2P systems. GIGA+ targets cluster environments where the number of nodes is much smaller than end hosts on the Internet and the availability of servers in a cluster changes much less often than Internet end-hosts. Thus, it is feasible for all servers to know about each other, and for lookups to take O(1) instead of O(log N) as in consistent hashing.

2.3 Related File Systems

Most file systems – local and distributed – store a directory on a single server [28, 49, 17, 37, 16]. The most sophisticated huge directory indexing is implemented in IBM's GPFS [35]. GPFS starts from Fagin's extendible hashing [13], stores its buckets in disk blocks and builds a distributed implementation that maintains strong consistency on the disk block of the shared directory buckets across all the nodes using a distributed lock managers and cache consistency protocols. This is significantly different from GIGA+'s weak consistency semantics where clients do not cache directory entries and send all directory operations to the servers. In case of concurrent writers, the GPFS lock manager initiates token transfers between the writers before anyone of them can update the shared directory block. The lock acquire and release phase can cause multiple disk I/Os on the underlying shared disk system on every write. Experiments performed in 2005 by NCAR show GPFS read-only lookups scaling very well but concurrent creates limited by write through to disk [7]. GPFS authors tell us that they are changing the cache consistency protocol to send requests to the lock holder rather than sending changes to the client through the shared disk [1]. Even with this improvement, GPFS will use whole directory locking to handle the growth of the directory's storage. This offers easy fault tolerance semantics at the cost of reduced performance at every client with every bucket split. GIGA+ does not use a distributed lock protocol at all.

Farsite is a distributed file system intended for commodity, desktop machines in an enterprise, providing the semantics of a central file server [2]. It redesigned its centralized directory service to be distributed for server load balancing by partitioning the metadata based on the file identifier, instead of the file path name [10]. Farsite's goals are fundamentally different from GIGA+ in that it explicitly assumes directory updates are rare and never highly concurrent.

Ceph [46] is an object-based research cluster file system that proposes to dynamically partition metadata based on the directory namespace tree. Ceph distributes "hot spot" directories by keeping track of the popularity of every metadata object [47]. At the cost of keeping a per object (file/inode) counter, their partitioning technique preserves locality while achieving load distribution over all directories. GIGA+ uses hash functions to distribute each directory without using additional per-file state.

Lustre is a production cluster file system based on object-based storage that at present has very limited support for distributed metadata [28]. The 2005 experiments by NCAR show that Lustre scaled to about 1000 creates/second [7]. However, Lustre is planning to implement a hash-based distributed metadata service, supporting one level of splitting. Prototype experiments performed in 2005 suggest that Lustre's future scalable metadata service may support over 10,000 file creates per second [42].

Several other research projects have used "scalable distributed data structures" (SDDSs) to build scalable storage abstractions for cluster computing – Gribble et. al. use a hash-table interface [19] and Boxwood uses B-link trees [30]. Both these projects used SDDSs to build a block-level persistent storage abstraction for different goals; Gribble's work focused on building cluster-based Internet services and Boxwood focused on building storage abstractions to ease the development of distributed file systems. Our work is different in that we use SDDSs to build a metadata service that provides high scalability, high concurrency, and fault tolerance.

3 Giga+ Design

GIGA+ divides each directory into a scalable number of partitions that are distributed across multiple servers in the cluster. Directory partitions grow dynamically with usage; that is, when a directory is small, it is represented as a single partition on one server. This is necessary to

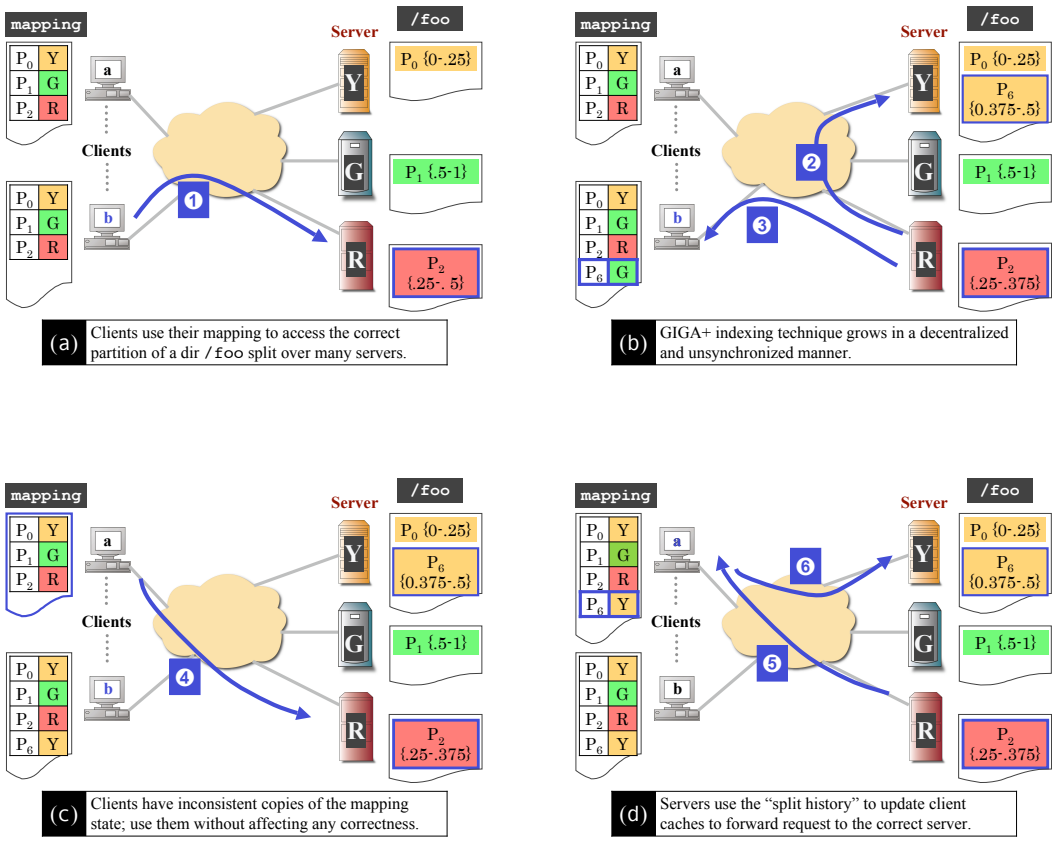

Figure 1. Example of inserts and lookups in Giga+ – *Directory* `/foo` *is divided into partitions that are distributed on three servers, such that each partition holds a particular range in the hashspace (denoted by* $\{x - y\}$*). (1) Client* `b` *inserts a file* `test.log` *in the directory. Clients hash the filename to find the partition that stores the name. Assume,* $hash(\text{“}\texttt{test.log}\text{”}) = 0.4321$*, the filename gets hashed to partition* P_2*. Client uses its partition-to-server mapping to send the request to server* R *that holds the partition* P_2*. (2) Server* R *receives the request and observes that partition* P_2 *is full. It uses the* GIGA+ *indexing technique to split* P_2 *to create a new partition* P_6 *on server* Y *(indexing details in §3.2 and Figure 2) On this partition split, half the hashspace range of* P_2 *is moved to* P_6*. The filename* `test.log` *is also moved to* P_6*. (3) Once the split is complete, server* R *sends a response to client* `b`*. The client updates its partition-to-server map to indicate the presence of* P_6 *on server* Y*. (4) Other clients have inconsistent copies of the mapping information but they continue to use it for directory operations. Client* `a` *wants to lookup the file* `test.log` *and its old mapping indicates (incorrectly) that the entry is located on* P_2 *on server* R*. (5) The "incorrect" server* R *receives client* `a`*'s request and detects that the range of the hash desired by the client has been moved to another partition* P_6 *on server* Y*. Server* R *uses the split history of* P_2 *to update the client's stale cache. (6) Client* `a` *then sends its request to the "correct" server. In* GIGA+ *we use a bitmap representation to lookup the partition-to-server mapping.*

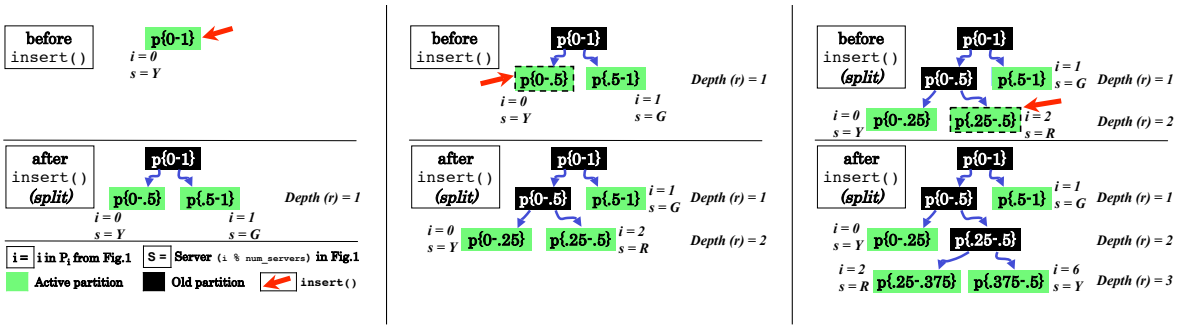

Figure 2. Giga+ index tree – GIGA+ *splits a directory into multiple partitions that are distributed across many servers. Each server grows a directory independently by splitting a partition half in two partitions. Each partition stores a the hash(filename) space, indicated as* $\{x - y\}$*. Initially, directories are small and stored in a single partition on one server. As they grow in size, the server splits half of each partition into a new partition, possibly on another server (the right child of a node) The old partition, with a decreasing hash-space range, remains on the original server (left child of a split node).*

ensure that performance on small directories , which form the bulk of directory operations [8, 3], is not penalized. Striping a small directory on many servers will incur a significant overhead when `readdir()` operations have to gather many tiny partitions instead of one small directory.

GIGA+ uses hashing to spread the directory incrementally over a set of servers. Since hashing based schemes achieve a uniform distribution of hash values, GIGA+ load-balances files with respect to both partitions and servers. In GIGA+, the directory entry name (filename) serves as the hash input. We use the MD5 hash algorithm because of its good collision resistance and randomness properties [33].

All directory operations in GIGA+ are handled at the server; clients send their requests to an appropriate server that operates on the given file. In other words, clients do not cache any directory entries. Even in read only directories this may be appropriate with large directories because these will be too big to fit in memory. Moreover, caching can incur a high consistency overhead in workloads with concurrent creates in the same directory. For instance, IBM's GPFS allows nodes to cache directory partitions. While this greatly improves the read only lookup performance, it suffers from lock contention and cache update traffic in the face of concurrent inserts to the same directory [1], unless it is disabled when concurrent updates are observed [23].

Figure 1 presents an example of an insert and lookup operations on GIGA+ directories and Figure 2 shows a tree representation of an incrementally expanding directory index. The example in Figure 1 shows a directory `/foo` that is managed by three servers, Y, G, and R. In Figure 1, we use the notation $P_i\{x - y\}$ to denote the range of the hashspace ($\{x - y\}$) held by a partition P_i. As the directory grows in size, partitions get filled up and they *split* half the hash range into a new partition. §3.2 describes the details of how GIGA+ splits partitions with high concurrency.

3.1 Bootstrapping

Each large directory is striped over a set of servers recorded in a *server list* that is stored as an attribute of that directory or its enclosing volume. The server list becomes known to a server when it recovers a partition of that directory. Clients learn about the directory's zeroth partition and server list when they first access the directory through a parent directory during pathname lookup.

This history of prior splits is called the *split history*.

Split history contains references to partitions that themselves have their own split history. Thus, the complete history of the directory growth is the transitive closure over all partitions. We can enumerate all the partitions of a directory by traversing the split history pointers starting at the zeroth pointer. This is analogous to enumeration of file's blocks by following the direct-, indirect- and doubly indirect-pointers of the file's i-node, or to `fsck`, where we traverse all the block pointers, starting from the superblock, to check the consistency of the whole file system.

3.2 Mapping and splitting partitions

GIGA+ servers use a *bitmap* to represent a tree of partitions and the corresponding servers. A bit position in the bitmap serves as a partition identifier and each position, i, is deterministically mapped to a specific server in the directory's server list (i `mod` *number_of_servers*). When the directory is newly created it is stored on a single partition and is represented by a bit-value '1' at the zero-th bit position in the bitmap, while all other bits are set to '0' indicating the absence of any other partitions.

As a directory grows, an overflowing partition is split to create a new partition and the bitmap is updated by setting the bit-value for this new partition to '1'. Because there is a deterministic relationship between parent and child partition and servers for each in the bitmap, GIGA+ encodes the lineage of all its partitions in a single representation.

Specifically, if a partition has an index i and is at depth r in the tree, meaning that it is the result of r splits from the zeroth partition, then in the next split it will move filenames in the larger half of filenames in its hash space to a partition $i + 2^r$, and both partitions will be at depth $r + 1$. For example, on the far right in Figure 2, partition `p{.25-.5}` (with index $i = 2$) is at a tree depth of $r = 2$. A split causes this partition to move the larger half of its hash space ({0.375-0.5}) to the newly created partition, `p{0.375-0.5}` (with index $i = 6$). And now both the partitions are at an increased tree depth of $r = 3$.

The depth r of a partition i responsible for a filename whose hash is K, is closely related to radix algorithms because $i = K$ `mod` 2^r. But it is important to recognize that every partition in GIGA+ can be at a different depth, and would therefore have its own radix. GIGA+ does not need to record the depth of each partition because it can determine r from inspecting the bitmap; partition i is at a depth r if the bit-value at position i is a '1' and the bit-value at position $i + 2^r$ is a '0'.

Looking up the appropriate partition, i, for a given filename with hash K can be done in the bitmap by finding the value of r for which, `bitmap`$[K$ `mod` $2^r] =$ '1' and `bitmap`$[K$ `mod` $2^{r+1}] =$ '0'. For the r that satisfies the above condition, partition i is K `mod` 2^r, and the server that holds the partition is i `mod` *number_of_servers*.

Because each server manages its partitions independently, it can split their partitions in parallel, without any client-side or server-side synchronization or consistency except for communicating to the server receiving the split partition. Figure 2 illustrates this with an example. Over a period of time, servers will have diverging copies of the bitmap indicating only their own partitions and splits. Performing a bit-wise "OR" operation on the bitmaps of all the servers, gives a map of all the partitions of that directory, which is what clients strive to do as a result of incorrect addressing updates.

For fast lookups and efficient space management, GIGA+ bitmaps maintain a local variable r such that all the bitmap positions above 2^r are a '0' and begins all depth searches at i (= K `mod` 2^r). As long as GIGA+ uses a hash that does a good job of distributing values evenly, all partitions will be at a depth close to r, which is the maximum depth of any partition. Figure 3 shows how

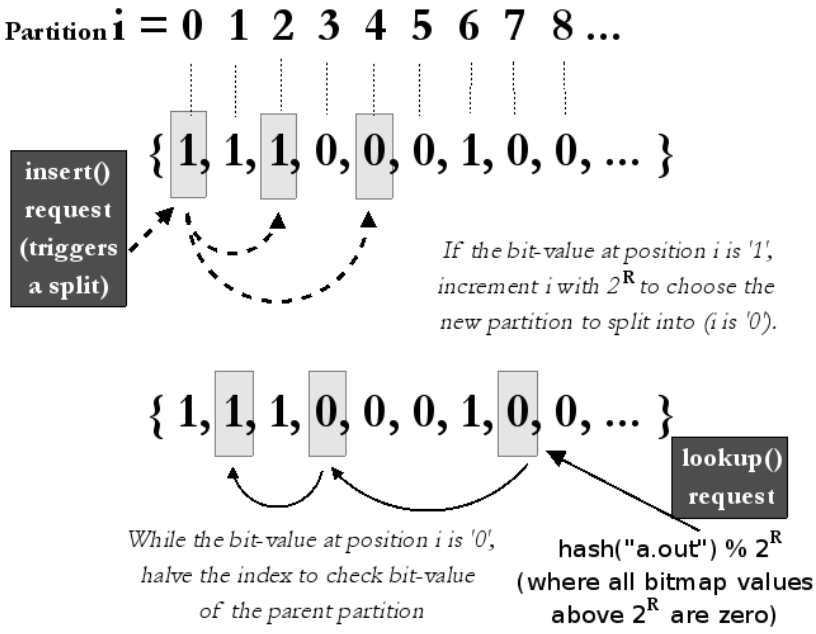

Figure 3. Using bitmap to lookup partitions – *A bit-value of "1" indicates the presence of a partition on a server, and bit-value "0" indicates the absence of the partition on a server. This example show how bitmaps are used to choose a new partition to split into and to lookup the partition that holds the filename ("a.out") you are searching.*

---

bitmaps are searched for lookups and for splits.

Thus, bitmaps in GIGA+ serve as simple, complete representation of all the partitions stored on all servers. Bitmaps are also attractive from a system design perspective for two reasons: they are a compact representation and they allow fast update operations. For example, a directory with a billion files and a partition size of 8,000 files, can be represented in a 16KB bitmap! Such small bitmaps should be easily cached in memory at all times. They can also be shipped to clients with small network traffic overhead because they are only needed when a client's bitmap has led to an incorrectly addressed server. Finally, such bitmaps can be updates very quickly by a simple bit-wise OR of a server's bitmap onto a stale client's copy.

3.3 Tolerating inconsistent clients

In absence of any explicit synchronization, clients will often end-up with a stale, out-of-date cache. This happens more frequently at high insert rates when more splits create newer partitions on the servers. A client with an out-of-date copy of the bitmap may send a request to an "incorrect" server; that is, a server that once was responsible for this file but has since passed responsibility to another server. An incorrect server discovers the stale mapping by recomputing the file's hash and testing it against its partitions. After detecting incorrect addressing, a server sends its bitmap to inform the stale client of all that server's partitions and the partitions in other servers that were created as a result of splits on this server, and the updated client retries its lookup.

The cost of this lazy client update mechanism is that the clients need additional probes to reach the correct server. Since the bitmaps at each server contain information about partitions

on other servers (resulting from prior splits on the server), every update message tells the clients about the state of more than one partition split, thus updating the clients quickly.

3.4 Power of 2 specialization

In the special case where the number of servers is a power of 2, two powerful optimizations result. If the number of servers is 2^S then beyond depth S in the split lineage tree (Figure 2), all partitions will split with both halves being assigned to the same server and that later split actions not generate network traffic. This implies that splitting is only being used to ensure that the underlying file system storing each partition as a directory never sees directories larger than the split threshold. Moreover, because servers only update client bitmaps when the client addresses an incorrect server, client bitmaps with no stale bits in the first 2^S bits will not generate incorrect server addresses no matter how many new partitions are beyond depth S. This means that client bitmaps will remain small (and could be truncated to 2^S bits by servers) and will generate few incorrectly addressed servers.

3.5 Adding new servers

In real-world deployments, the total size of a system tends to grow over time and administrators are likely to want to increase the set of servers. Even if the total system does not grow, conservative administrators may find that too few of the nodes had been servers in the original configuration. A scalable directories scheme should support increasing the pool of servers efficiently.

In the general case, changing the server list associated with a GIGA+ directory changes almost all partition to server assignments. This redistribution work can be easily parallelized, but it can be a large amount of overhead.

Given that in GIGA+ each directory can have its own server list, we can choose which directories are most appropriate to redistribute and when. For example, we may only redistribute the partitions of a directory when each server holds many partitions and when the number of servers is increasing from one power of 2, say 2^s, to the next, 2^{s+1}, to effectively double the operation rate. With this choice half of the partitions of every current server of that directory are moved to the new servers, which would have been the split destinations in the first place if the larger number of servers had been available. Updating clients' bitmaps is disabled at each server while that server is redistributing partitions, for simplicity in this rare expansion phase, and each server can instead proxy operations to the new servers for partitions it have moved. When all partitions have been moved, updating clients' bitmaps can be re-enabled and the now longer server list made available so that clients will be updated to identify the new distribution of partitions.

For non-power of two numbers of servers more complicated redistribution schemes are possible, but mostly seem to have much more complicated client bitmap handling after the redistribution, or suffer from slow and unbalanced redistribution phases. This may be a topic for future work if experience shows that power of two numbers of servers is not sufficient.

3.6 Handling failures

In this section, we describe how GIGA+ handles node failures, common events in systems with thousands or more nodes [16, 36].

Handling client failures can be subdivided into two recovery processes. Request or reply packet loss is a client recovery action in most distributed systems. GIGA+ uses best in class solutions such as sequence numbers in requests and responses to distinguish new from retransmitted requests, and a server reply cache to ensure that non-idempotent commands, like create, can return the

original command's response when a reply packet is lost. If a client reboots, however, it loses all its state with respect to GIGA+. This is not a problem as all client state for a GIGA+ directory is reconstructed by reopening the directory from the zeroth partition named in a parent directory entry, refetching its server list and rebuilding the bitmaps through incorrect addressing of server partitions during normal operations.

GIGA+ servers contain a large amount of state, so server failures require more mechanisms. GIGA+ uses a chained-declustering scheme [22] to increase directory availability in the case of permanent server failures. This technique replicates partitions so that there are two copies of every partition and these two are stored on adjacent servers in the server lost order. For example, if a directory is spread on 3 servers, all the primary copy partitions on server 1 will be replicated on server 2, partitions primary in server 2 replicated on server 3, and partitions primary in server 3 will be replicated on server 1. Chained declustering makes it simple to shift a portion of the read workload of each primary to its secondary so that the secondary of a failed node does not have a higher load than the other servers [22]. While some systems have used chained de-clustering to load balance read requests on a specific partition over one of its two servers [**?**], GIGA+ doesn't need to do this because hashing ensures us uniform load across all servers.

On normal reads and writes, clients send their request to the server that holds the primary copy. A non-failed primary handles reads directly and replicates writes to the secondary before responding. If the client's request times out too many times, the client will send the request marked as a failover request than an incorrectly addressed request to the server that holds the replica. A server receiving a failover request participates in a membership protocol among servers to diagnose and confirm the failover [4, 48]. While a node is down or being reconstructed, its secondary executes all of its writes and reads, and uses chained declustering to shift some of its read workload over other servers. This shifting is done by notifying clients in reply messages to cache a hint that a server is down and execute chained declustering workload shifts. Clients either try the failed primary first and failover to learn about the failure or try the secondary first and be corrected to retry at the primary.

In this scheme, if the replica's server also fails (along with the primary) then the requested data becomes unavailable; one way to avoid this is by keeping more replicas, a practice adopted by large file systems like the Google file system which keeps 3-6 copies of data [16].

4 Experimental Evaluation

4.1 Giga+ Prototype

We have built a user-space implementation of GIGA+ using the FUSE API [14]. The advantages of a user-space implementation are ease of implementation and portability across various file systems: we can overlay our prototype on top of any existing file system [50, 21]. The disadvantage is the potential reduced performance compared to a native parallel file system implementation; however, this should not affect the scaling properties of the GIGA+ indexing techniques.

In our implementation, we layer FUSE on top of the Linux ext3 file system. The application performs a file operation that is intercepted by the VFS layer in the kernel. The VFS layer forwards the request to the *FUSE* kernel module. This FUSE layer manages the application's view of the file system by maintaining in-memory state of the files created by the application through FUSE. These dynamic structures keep a mapping of the i-nodes associated with the files created by the application; this mapping is evicted if the file is no longer in use or if its associated timeout (set internally by FUSE) value expires. FUSE uses these mappings for path-name translation; for a file descriptor used by the application, FUSE uses the mapping to find the matching i-node. If an

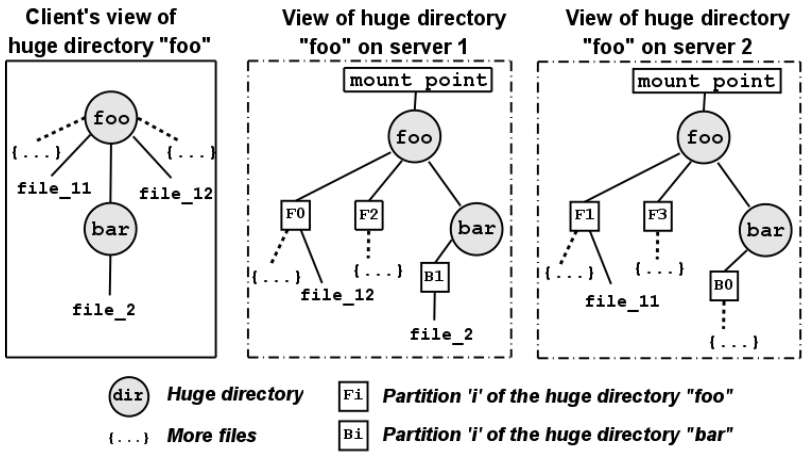

Figure 4. Local representation of huge directory in Giga+ prototype layered on Ext3 – *Currently, we replicate the huge directory tree structure (not the files or partitions) for ease of namespace management through common path-name manipulation across all servers.*

---

operation is performed on a file descriptor that is not in the mapping, FUSE does a complete path-name lookup to find the respective i-node, which is then cached for later use. This kernel module bounces the request out to the user-level GIGA+ *library*. This library contains the core indexing technique that selects the destination server. The request is then handed over to the GIGA+ *RPC layer* (built using ONC RPC [38]) that marshals the request and sends it to the server.

At the server, the GIGA+ *RPC layer* unmarshals the request and sends it to the appropriate message handler. While a client sees a "single" large directory, the server creates sub-directories to represent the partitions associated with a huge directory (as shown in 4). Our current prototype replicates the huge directory tree structure (not the files or partitions) on all the servers. While this replication is inefficient for many large directories, it enables easy namespace management by performing common path-name manipulation across all servers. If a partition splits, our prototype performs a `readdir` on the directory and rehashes every entry to check whether it needs to be moved to the new partition. Because of layering on a local file system, we also have to copy the entire file from an old partition to the new partition; this won't be an issue if we layer our prototype on parallel file systems that typically have a separate metadata manager. Not counting comments, our prototype implementation consists of little more than 7,000 LOC.

4.2 Experimental Setup

Our testbed is a 100-node cluster of HP XW9400 Workstations, each containing a two dual-core 2.8GHz AMD Opteron processor, 8GB of memory, and a LSI/Symbios Logic SAS1068E PCI-Express Fusion-MPT serial attached SCSI storage controller connected to one 7200 rpm SATA 80 GB Samsung HD080HJ disk with 8MB buffer DRAM size. Nodes have a nVidia Corporation MCP55 GigE controller and are interconnected using a HP Procurve 2824 switch on a Gigabit Ethernet backplane with 20 μsecond latency. All machines run Linux 2.6.9-55.0.9.EL_lustre.1.4.11.1smp (Redhat release) and use the ext3 file system to manage its disk.

We evaluate the performance of our prototype with the UCAR metarates benchmark [31], previously used by several other parallel file systems [49, 7]. Metarates is an MPI application that

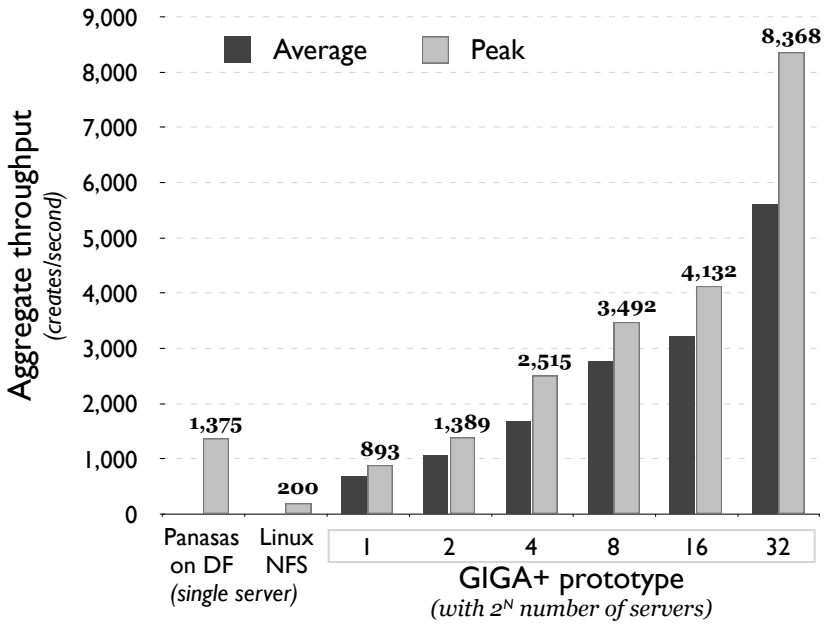

Figure 5. Scale and performance of Giga+ using UCAR Metarates benchmark.

---

manages multiple clients creating files in a directory. Once all files are created, it also performs phases that `stat` each file (to get file attributes) and `utime` (to set the timestamp) on each file. Most published results of Metarates we have found report concurrent create rates in one directory of about 1,000 creates per second [49, 7].

For the experiments below, we use a power-of-two number of servers, a setup that greatly reduces the number of server-to-server split and server-to-client bitmap update traffic. GIGA+ is configured with a partition size of 8,000 entries per partition by default – a number based on a study that observed that 99.99% of directories in HPC file systems contain less than 8,000 entries [8]; thus, directories smaller than 8,000 files are stored on a single server. When experimenting with scaling we increase the size of the directory and number of clients running the benchmark as we increase the number of servers. In general, we employ two clients generating load for each server.

4.3 Scale and performance

Figure 5 shows the scalability of our prototype implementation when each server creates 370,000 files in a common directory. The graph shows three systems: a commercial parallel file system (Panasas), a Linux NFSv3 server, and our GIGA+ prototype, where the first two are reported from a prior published result [49]. The single server performance of our system (peak at 893 creates/second) is faster than Linux NFS, but about 33% slower than Panasas's file system on their Directflow protocol and hardware. Panasas's PanFS file system manages each directory with a single metadata server. During the file creation process, its metadata server also creates the two file objects on storage before returning to the client; this allows clients to then start writing data directly on the two storage nodes, although Metarates in fact does no data writing [49]. PanFS buffers writes in two battery backed memories and batches directory updates after journaling them.

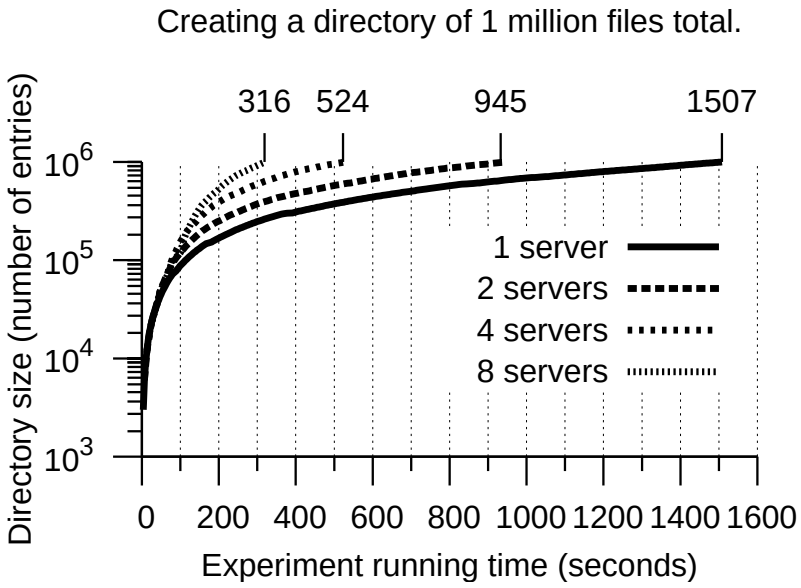

Figure 6. Growth rate of directories in Giga+ – *Initially when the directory is small and stored on a single server, the overall growth rate is limited by the performance of that server. For a constant 1,000,000 files to be created when the threshold when the threshold for the partition splitting is 8,000 files, an 8-server system goes little faster than a single server system during the first 100,000 creates and has barely reached saturation at 1,000,000 created files.*

While it is not our intention to compete with commercial product, many improvements could be made to our prototype, such as the use of a thread-pool based server, asynchronous execution of partition splits at the server, and dynamic partition resizing. However, the performance of a GIGA+ single server, between Linux NFSv3 and PanFS single servers, indicates that it is not vastly slow.

The key contribution of GIGA+ is the throughput scale-up achieved by incrementally distributing a directory on all the available servers – our prototype delivers 55-60% average scaling, i.e., twice the number of servers give 1.55-1.6 times more creates/second, and a peak throughput of 8368 inserts/second, using a 32 server configuration to construct a single directory with 12 million files (Figure 5). This is a significant improvement over some of the existing parallel file systems, based on a 3-year old study of single directory file creation rates [7]. In that study, when multiple clients perform concurrent inserts, LustreFS achieved a peak rate of 1027 inserts/second and GPFS v2.3 achieved a peak rate of 167 inserts/second. This study used the UCAR Metarates benchmark on a 64-node cluster of dual core Xeon 2.4GHz processors and 2GB of memory, running Linux 2.4.26. Since this study was published, both GPFS and LustreFS have built more sophisticated directory implementations to handle concurrent access. GPFS is optimizing its hash directory implementation by using strong consistency only after split operations (instead of every insert) and reducing lock protocol traffic and disk I/Os for concurrent inserts [1]. LustreFS strategy is closer to GIGA+ in that it splits a directory into multiple storage objects but apparently only once [29]. In a presentation by a LustreFS partner, an early prototype was reported to achieve as much as 10,000 creates/seconds [42]. These experiments indicate that GIGA+ techniques produce

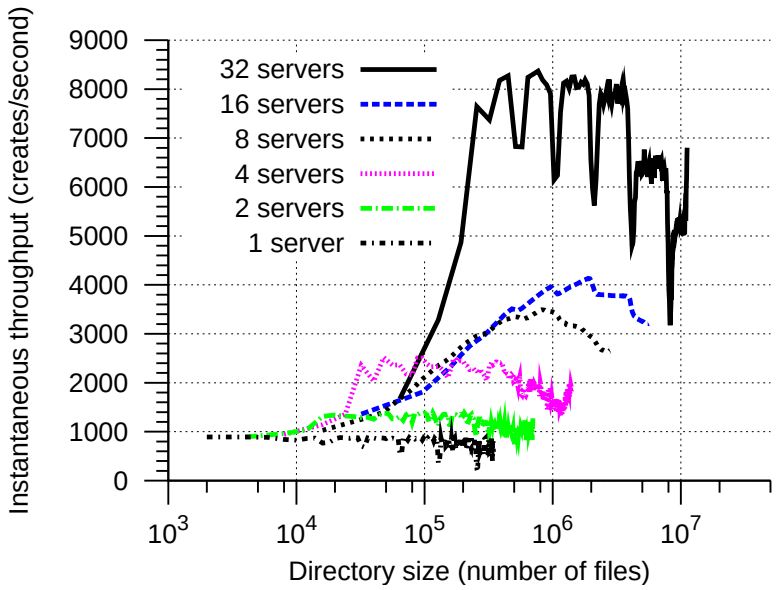

Figure 7. Analyzing the Giga+ behavior – *Initially, throughput increases in "steps" when the directory grows on to more servers. Uniform load distribution causes all the partitions, all have same size, to fill up at the same time and that causes all the servers to split the partitions; this "dips" the throughput of the system for that period. The gradual "slow-down" towards the end of the experiment happens when all servers reach their saturation point.*

---

comparable results even with little tuning or optimization.

To understand how GIGA+ scales over time with fixed (non-scaling) amount of files to create, we measured the time required to add one million files total to a directory distributed on one to eight servers. Figure 6 shows the size of the directory at any instant of time during the experiment. The directory starts small, stored on a single server, so the larger system's throughput is limited by the single and small server performance. In the experiment of Figure 6, the directory needs to be bigger than 100,000 files to see much difference in create rates of various numbers of servers, but at 1,000,000 files large differences in speed become apparent. To achieve parallelism more quickly, the system could be optimized to split each partition into N-chunks on N-servers, instead of 2 chunks of 2 servers, but this decreases the size of worst case partitions resulting in slower `readdir()` performance because more servers will have to be invoked for less entries each. A similar drawback can result for a 2-way split if the split threshold size is smaller.

In summary, GIGA+ throughput scales with more number of servers, however it's not a linear 2X scale-up. We now analyze a long running system to identify the source of throughput degradation. In the following sections we look at the overhead involved in split operations at the server and the use of weak consistency semantics at the clients.

4.4 Cost of partition splits

Figure 7 shows the instantaneous throughput over the experiment shown in Figure 5, in terms of the growth in total directory size. At each client, we measure the insert rate for the last 1,000 files created. The sum over N clients gives us the aggregate instantaneous insert rate for the last N thousand files. Because we employ 2 clients for each server, the 1 server case is sampled every

Entries per partition	8 servers, each creating .. 10^3 files	8 servers, each creating .. 10^4 files	16 servers, each creating .. 10^3 files	16 servers, each creating .. 10^4 files	32 servers, each creating .. 10^3 files	32 servers, each creating .. 10^4 files
1K	78.2%	64.4%	80.4%	64.8%	75.8%	64.2%
8K	75.2%	61.8%	76.8%	63.8%	71.1%	62.9%
24K	46.2%	91.1%	51.2%	93.4%	56.2%	92.1%

Figure 8. Redundant work done at the servers – *Servers repeatedly rehash and move the entries from one partition to another partition, during splitting. This table shows the total number of moves of directory entries as a fraction of the total number of files created.*

2,000 files, 2-servers every 4,000 files, 4-servers every 8,000 files, 16-servers every 32,000 files, and 32-server every 64,000 files. GIGA+ system behavior can be broken up into three parts: a "step-up" in throughput when splitting adds new servers, a "dip" in throughput when servers are generally splitting at the same time, and a gradual "slow-down" once all servers are saturated.

GIGA+ spreads a growing directory over additional servers in an incremental manner. With every additional server we should see a "step-up" in the throughput. For instance, in the 4-server case, after 8,000 files, the directory is striped on 2 servers, and after 24,000 files, the system uses all the servers. We don't always see the "step-up" in Figure 7 however, and instead some curves appear to smoothly grow. In case of 32 servers, we are sampling at every 64,000 files, but with a split threshold of 8,000 entries per partition, the directory is already split into 8 partitions at the first sample point. The next two sample points for the 32 server case are at 128,000 files and 256,000 files. And at that point, GIGA+ has striped the directory on all servers – running at more than 7,500 inserts/second. Our sampling method under samples during the "step-up" phase for 32 servers so the graph appears smoother than it should be.

The main reason for deep throughput degradation, indicated by the "troughs" in the figure, is when all the servers are busier splitting their partitions at nearly the same time. Because GIGA+ uses fixed size partitions, the uniform distribution of the hash function will tend fill up all the partitions at about the same time. These "dips" are only apparently worse for a large system (the 32 server case). One optimization to avoid this behavior would be to add randomization to the period between splits or the split threshold for each partition.

While splits are important to grow the directory on all servers, they are also expensive server-side operations. To split a partition into a new partition, GIGA+ implementation takes the following steps: first, the servers creates a new partition; if the new partition is on a remote server, an RPC is sent to that server. The server then reads the entire partition (`readdir()`) and rehashes all the directory entries to decide whether it moves to the new partition. The entries are copied to the new partition and then deleted (`unlink()` from the current partition. In our current prototype, the server blocks to complete a split operation. We chose this approach for its simplicity and robustness; however, the server is unable to service requests that may be addressed to any other partition. A better implementation might use Non-blocking split operations, allowing the servers to split a partition in the background, while continuing to service requests that are not addressed to the partition being split.

A central property of GIGA+ and its predecessors extendible hashing [13] and linear hashing [27] is the incremental use of more partitions. The advantage is small directories, by far more common, are not distributed and no partition is ever allowed to grow very large. The disadvantage is that splitting causes the server to perform "redundant" work by repeatedly rehashing and moving

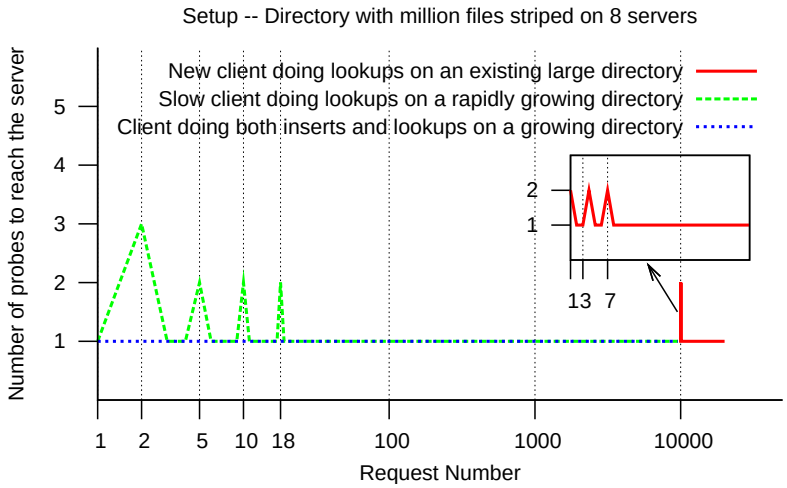

Figure 9. Cost of using inconsistent client hints – *This graph shows the number of probes required to learn about the entire system using three types of clients: a "new client" that does lookups on a large directory, a "slow client" that does lookups at a rate two orders of magnitude slower than a rapidly growing directory, and a client that is doing both creates and lookups in a growing directory. Every server's bitmap has information about its partitions and a few partitions on other servers created by prior split operations. Thus, clients learn more information about the entire system when it is updated using a server's bitmap.*

(copy and unlink) entries to a new partition. We measure the redundant work done by servers as the fraction of extra `mknod()` operations compared to the total number files created in the experiment. Figure 8 shows how the amount of redundant work per server is sensitive to the partition size, number of files created per server, and the total number of servers used in the system.

It should not be surprising that the amount of redundant work is not larger than 100%, though we were initially expecting that it might be much larger. As Figure 2 shows we are essentially dealing with a binary tree and we know that the number of non-leaf nodes in a binary tree is slightly smaller than the number of the leaf nodes. Since the redundant work in GIGA+ is about half a full partition for each interior nodes and the size of each leaf nodes is between half a full partition and a full partition, the redundant work should never exceed the number of files.

This logic and our experiments did not include any redistribution as new servers are added to an existing directory. As we have discussed that is another source of additional work that is also about half of the total number of filenames in the directory.

4.5 Cost of weak consistency

In this section, we analyze the cost of using inconsistent bitmaps at the client in terms of the number of hops taken by a client request to reach the correct server. Figure 9 shows the number of probes required by clients; a single probe indicates that the client has addressed the correct server, while more than one probe indicates that the client has a stale bitmap that has been updated by one or more servers before it reaches the correct server. We use three types of clients: a "new client" that does lookups after the directory is striped on all servers, a "slow client" that does lookups at a rate two orders of magnitude slower than a rapidly growing directory, and a client that is doing both

creates and lookups in a growing directory.

In a growing system, where there is a mix of lookups and creates, the behavior of the system depends on the workload. If there are more inserts than lookups, the directory grows faster by continuously spreading on more servers. Clients doing lookups will have more incorrect probes to learn about the partitions on the new servers. In Figure 9, this phenomenon is verified by the "slow client" that emulates an insert-intensive workload that where the insert rate is significantly higher than the lookup rate. This client sends lookup requests at a rate that is 100 times slower than create rate of the system. On the other hand, in a lookup-intensive workload where the rate of lookups is more than inserts, the clients will always have updated information because the directory is growing very slowly. Once the clients are updated by an incorrect server, the continue to address the correct server until the directory splits on to a new server.

A "new client" that does lookups on a existing large directory learns about the entire bitmap in the first few requests (inset figure in Figure 9). When a lookup request is addressed to an incorrect server, that server sends its bitmap to update the client's copy. A server's bitmap encodes not only the partitions stored at that server, but also the partitions created from prior split operations. For a directory striped on all N servers, the clients will never incur any additional incorrect forwarding after they have contacted the $log_2(N)$ servers. Thus, GIGA+'s bitmap encoding helps servers correct the stale clients with more information per update.

5 Summary and Future Work

Today file systems are being used in interesting ways; one such example is using file system directories as a fast, lightweight "database" to store millions to billions of files in a single directory and supporting insert, delete, lookup and unordered scans (not range queries). In this paper, we explore building highly scalable traditional file system directories that allow high update concurrency and parallelism. Our research is aimed at pushing the limits of scalability by minimizing serialization, eliminating system-wide synchronization, not caching directory entries and using weak consistency semantics for client's caches of mapping that describes which server manages each directory entry.

Our work builds on dynamically resizeable hash tables so that the bulk of (small) directories are stored on one server and can be accessed or scanned efficiently, and on server-resolution hints handed out to clients without guarantees of correctness. The former suffers from synchronization needed to share the changing hash table headers. The latter suffers from serialization, on the pattern that hash tables headers are changed, needed to make client hints correctable.

In GIGA+, our experimental implementation shows how all hash table headers can be distributed as partition split histories and how all partitions can be independently split without global synchronization because client hints represent all partitions separately. One hint representation, a bitmap of the presence or absence of each partition in the growing binary tree of splits, is small enough to be easily cached, transmitted and updated efficiently with a simple bit-wise OR. Clients resolving a server address incorrectly are redirected with a bitmap from a server that used to hold the request partition and knows at least one more definitive server. We also show how the special case of power of two numbers of servers allows bitmaps to be no larger than the number of servers and rarely incorrect, and how adding twice the number of servers can support migration of half of the partitions quickly.

Our user-level prototype implementation built in FUSE layered on Linux ext3 and ran on a 100-node cluster with 32 directory servers delivers a peak throughput of more than 8,000 file creates/second during which clients can expect only a handful of server redirections. We intend to release our prototype as it becomes more full featured and robust.

References

[1] Private Communication with Frank Schmuck and Roger Haskin, IBM.

[2] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., an Jon Howell, J. R. D., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)* (Boston MA, Nov. 2002).

[3] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. A Five-Year Study of File-System Metadata. In *Proc. of the FAST '07 Conference on File and Storage Technologies* (San Jose CA, Feb. 2007).

[4] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle WA, Nov. 2006).

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle WA, Nov. 2006).

[6] Cluster, H. Inside the Windows NT File System. Tech. rep., Microsoft Press, Aug. 1994.

[7] Cope, J., Oberg, M., Tufo, H. M., and Woitaszek, M. Shared Parallel File Systems in Heterogeneous Linux Multi-Cluster Environments. In *Proc. of the 6th LCI International Conference on Linux Clusters: The HPC Revolution* (Apr. 2005).

[8] Dayal, S. Characterizing HEC Storage Systems at Rest. Tech. Rep. CMU-PDL-08-109, Carnegie Mellon University, 2008.

[9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of 21st ACM Symposium on Operating Systems Principles (SOSP '07)* (Stevenson WA, Oct. 2007).

[10] Douceur, J. R., and Howell, J. Distributed Directory Service in the Farsite File System. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle WA, Nov. 2006).

[11] Eisler, M., Corbett, P., Kazar, M., Nydick, D. S., and Wagner, J. C. Data ONTAP GX: A Scalable Storage Cluster. In *Proc. of the FAST '07 Conference on File and Storage Technologies* (San Jose CA, Feb. 2007).

[12] Ellis, C. Extendible Hashing for Concurrent Operations and Distributed Data. In *Proc. of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta GA, Mar. 1983).

[13] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible Hashing – A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems 4*, 3 (Sept. 1979).

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

[15] GFS. Red Hat Global File System. http://www.redhat.com/gfs.

[16] Ghemawat, S., Gobioff, H., and Lueng, S.-T. Google File System. In *Proc. of 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Rochester NY, Oct. 2003).

[17] Gibson, G., and Corbett, P. pNFS Problem Statement. Internet Draft, July 2004.

[18] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)* (Cambridge MA, Oct. 1998).

[19] Gribble, S., Brewer, E., Hellerstein, J., and Culler, D. Scalable Distributed Data Structures for Internet Service Construction. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)* (San Diego CA, Oct. 2000).

[20] Hand, E. Head in the Clouds. *Nature 449*, 963 (2007).

[21] Heidemann, J. S., and Popek, G. J. File System Development with Stackable Layers. *ACM Transactions on Computer Systems 12*, 1 (Feb. 1994).

[22] Hsaio, H.-I., and DeWitt, D. J. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proc. of the 6th International Conference on Data Engineering (ICDE '90)* (Washington DC, 1990).

[23] IETF. NFS v4.1 specifications. http://tools.ietf.org/wg/nfsv4/.

[24] ISILON. Isilon Systems Inc. http://www.isilon.com/.

[25] Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., and Panigrahy, R. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of the ACM Symposium on Theory of Computing* (El Paso TX, May 1997).

[26] Litwin, W. Linear Hashing: A New Tool for File and Table Addressing. In *Proc. of the 6th International Conference on Very Large Data Bases (VLDB '80)* (Montreal, Canada, Sept. 1980).

[27] Litwin, W., Neimat, M.-A., and Schneider, D. A. LH*–A Scalable, Distributed Data Structure. *ACM Transactions on Database Systems 21*, 4 (Dec. 1996).

[28] Lustre. Lustre File System. http://www.lustre.org.

[29] LustreFS. Clustered Metadata. http://arch.lustre.org/index.php?title=ClusteredMetadata.

[30] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)* (San Francisco CA, Dec. 2004).

[31] Metarates. UCAR Metarates Benchmark. www.cisl.ucar.edu/css/software/metarates/.

[32] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D. NFS version3: Design and implementation. In *Proc. of Summer USENIX Conference '94* (Boston MA, 1994).

[33] Rivest, R. A. The MD5 Message Digest Algorithm. RFC 1321, Apr. 1992.

[34] Ross, R., Felix, E., Loewe, B., Ward, L., Nunez, J., Bent, J., Salmon, E., and Grider, G. High end computing revitalization task force (HECRTF), inter agency working group (HECIWG) file systems and I/O research guidance workshop. http://institutes.lanl.gov/hec-fsio/docs/HECIWG-FSIO-FY06-Workshop-Document-FINAL6.pdf, 2006.

[35] Schmuck, F., and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the FAST '02 Conference on File and Storage Technologies* (Monterey CA, Jan. 2002).

[36] Schroeder, B., and Gibson, G. A. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. of the FAST '07 Conference on File and Storage Technologies* (San Jose CA, Feb. 2007).

[37] Soltis, S. R., Ruwart, T. M., and O'Keefe, M. T. The Global File System. In *Proc. of the 5th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies* (College Park MA, Sept. 1996).

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

[39] Stonebraker, M., Bear, C., Çetintemel, U., Cherniack, M., Ge, T., Hachem, N., Harizopoulos, S., Lifter, J., Rogers, J., and Zdonik, S. B. One Size Fits All? Part 2: Benchmarking Studies. In *Proc. of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)* (Aislomar CA, 2007).

[40] Stonebraker, M., and Çetintemel, U. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *Proc. of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, 2005).

[41] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proc. of the 33rd International Conference on Very Large Data Bases (VLDB '07)* (Vienna, Austria, Sept. 2007).

[42] Studham, R. S. Lustre: A Future Standard for Parallel File Systems. Invited presentation at International Supercomputer Conference. Heidelberg, Germany. June 24, 2005.

[43] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In *Proc. of USENIX Conference '96* (San Jose CA, 1996).

[44] Ts'o, T. Y. Planned Extensions to the Linux Ext2/Ext3 Filesystem. In *Proc. of USENIX Conference '02, FREENIX Track* (Monterey CA, 2002).

[45] VERIZON. 'Trans-Pacific Express' to Offer Greater Speed, Reliability and Efficiency. http://newscenter.verizon.com/press-releases/verizon/2006/verizon-business-joins.html, Dec. 2006.

[46] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle WA, Nov. 2006).

[47] Weil, S. A., Pollack, K., Brandt, S. A., and Miller, E. L. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC '04)* (Pittsburgh PA, Nov. 2004).

[48] Welch, B. Integrated System Models for Reliable Petascale Storage Systems. In *Proc. of the Petascale Data Storage Workshop (at Supercomputing '07)* (Reno NV, Nov. 2007).

[49] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. Scalable Performance of the Panasas Parallel File System. In *Proc. of the FAST '08 Conference on File and Storage Technologies* (San Jose CA, Feb. 2008).

[50] Zadok, E., and Nieh, J. FiST: A Language for Stackable File Systems. In *Proc. of USENIX Conference '00* (San Diego CA, 2000).