Scale and Concurrency of GIGA+: File System Directories with Millions of Files

Swapnil Patil and Garth Gibson
Carnegie Mellon University
{firstname.lastname @ cs.cmu.edu}

Abstract – We examine the problem of scalable file system directories, motivated by data-intensive applications requiring millions to billions of small files to be ingested in a single directory at rates of hundreds of thousands of file creates every second. We introduce a POSIX-compliant scalable directory design, GIGA+, that distributes directory entries over a cluster of server nodes. For scalability, each server makes only local, independent decisions about migration for load balancing. GIGA+ uses two internal implementation tenets, asynchrony and eventual consistency, to: (1) partition an index among all servers without synchronization or serialization, and (2) gracefully tolerate stale index state at the clients. Applications, however, are provided traditional strong synchronous consistency semantics. We have built and demonstrated that the GIGA+ approach scales better than existing distributed directory implementations, delivers a sustained throughput of more than 98,000 file creates per second on a 32-server cluster, and balances load more efficiently than consistent hashing.

1 Introduction

Modern file systems deliver scalable performance for large files, but not for large numbers of files [18, 67]. In particular, they lack scalable support for ingesting millions to billions of small files in a single directory - a growing use case for data-intensive applications [18, 44, 50]. We present a file system directory service, GIGA+, that uses highly concurrent and decentralized hash-based indexing, and that scales to store at least millions of files in a single POSIX-compliant directory and sustain hundreds of thousands of creates insertions per second.

The key feature of the GIGA+ approach is to enable higher concurrency for index mutations (particularly creates) by eliminating system-wide serialization and synchronization. GIGA+ realizes this principle by aggressively distributing large, mutating directories over a cluster of server nodes, by disabling directory entry caching in clients, and by allowing each node to migrate, without notification or synchronization, portions of the directory for load balancing. Like traditional hash-based distributed indices [17, 36, 52], GIGA+ incrementally hashes a directory into a growing number of partitions. However, GIGA+ tries harder to eliminate synchronization and prohibits migration if load balancing is unlikely to be improved.

Clients do not cache directory entries; they cache only the directory index. This cached index can have stale pointers to servers that no longer manage specific ranges in the space of the hashed directory entries (filenames). Clients using stale index values to target an incorrect server have their cached index corrected by the incorrectly targeted server. Stale client indices are aggressively improved by transmitting the history of splits of all partitions known to a server. Even the addition of new servers is supported with minimal migration of directory entries and delayed notification to clients. In addition, because 99.99% of the directories have less than 8,000 entries [4, 14], GIGA+ represents small directories in one partition so most directories will be essentially like traditional directories.

Since modern cluster file systems have support for data striping and failure recovery, our goal is not to compete with all feature of these systems, but to offer additional technology to support high rates of mutation of many small files.[1] We have built a skeleton cluster file system with GIGA+ directories that layers on existing lower layer file systems using FUSE [19]. Unlike the current trend of using special purpose storage systems with custom interfaces and semantics [6, 20, 54], GIGA+ directories use the traditional UNIX VFS interface and provide POSIX-like semantics to support unmodified applications.

Our evaluation demonstrates that GIGA+ directories scale linearly on a cluster of 32 servers and deliver a throughput of more than 98,000 file creates per second – outscaling the Ceph file system [63] and the HBase distributed key-value store [26], and exceeding petascale scalability requirements [44]. GIGA+ indexing also achieves effective load balancing with one to two orders of magnitude less re-partitioning than if it was based on consistent hashing [30, 58].

In the rest of the paper, we present the motivating use cases and related work in Section 2, the GIGA+ indexing design and implementation in Sections 3-4, the evaluation results in Section 5, and conclusion in Section 6.

[1] OrangeFS is currently integrating a GIGA+ based distributed directory implementation into a system based on PVFS [2, 45].

2 Motivation and Background

Over the last two decades, research in large file systems was driven by application workloads that emphasized access to very *large files*. Most cluster file systems provide scalable file I/O bandwidth by enabling parallel access using techniques such as data striping [20, 21, 25], object-based architectures [21, 39, 63, 66] and distributed locking [52, 60, 63]. Few file systems scale metadata performance by using a coarse-grained distribution of metadata over multiple servers [16, 46, 52, 63]. But most file systems cannot scale access to a *large number of files*, much less efficiently support concurrent creation of millions to billions of files in a single directory. This section summarizes the technology trends calling for scalable directories and how current file systems are ill-suited to satisfy this call.

2.1 Motivation

In today's supercomputers, the most important I/O workload is checkpoint-restart, where many parallel applications running on, for instance, ORNL's CrayXT5 cluster (with 18,688 nodes of twelve processors each) periodically write application state into a file per process, all stored in one directory [7, 61]. Applications that do this per-process checkpointing are sensitive to long file creation delays because of the generally slow file creation rate, especially in one directory, in today's file systems [7]. Today's requirement for 40,000 file creates per second in a single directory [44] will become much bigger in the impending Exascale-era, when applications may run on clusters with up to billions of CPU cores [31].

Supercomputing checkpoint-restart, although important, might not be a sufficient reason for overhauling the current file system directory implementations. Yet there are diverse applications, such as gene sequencing, image processing [62], phone logs for accounting and billing, and photo storage [6], that essentially want to store an unbounded number of files that are logically part of one directory. Although these applications are often using the file system as a fast, lightweight "key-value store", replacing the underlying file system with a database is an oft-rejected option because it is undesirable to port existing code to use a new API (like SQL) and because traditional databases do not provide the scalability of cluster file systems running on thousands of nodes [3, 5, 53, 59].

Authors of applications seeking lightweight stores for lots of small data can either rewrite applications to avoid large directories or rely on underlying file systems to improve support for large directories. Numerous applications, including browsers and web caches, use the former approach where the application manages a large logical directory by creating many small, intermediate sub-directories with files hashed into one of these sub-directories. This paper chose the latter approach because users prefer this solution. Separating large directory management from applications has two advantages. First, developers do not need to re-implement large directory management for every application (and can avoid writing and debugging complex code). Second, an application-agnostic large directory subsystem can make more informed decisions about dynamic aspects of a large directory implementation, such as load-adaptive partitioning and growth rate specific migration scheduling.

Unfortunately most file system directories do not currently provide the desired scalability: popular local file systems are still being designed to handle little more than tens of thousands of files in each directory [43, 57, 68] and even distributed file systems that run on the largest clusters, including HDFS [54], GoogleFS [20], PanFS [66] and PVFS [46], are limited by the speed of the single metadata server that manages an entire directory. In fact, because GoogleFS scaled up to only about 50 million files, the next version, ColossusFS, will use BigTable [12] to provide a distributed file system metadata service [18].

Although there are file systems that distribute the directory tree over different servers, such as Farsite [16] and PVFS [46], to our knowledge, only three file systems now (or soon will) distribute single large directories: IBM's GPFS [52], Oracle's Lustre [38], and UCSC's Ceph [63].

2.2 Related work

GIGA+ has been influenced by the scalability and concurrency limitations of several distributed indices and their implementations.

GPFS: GPFS is a shared-disk file system that uses a distributed implementation of Fagin's extendible hashing for its directories [17, 52]. Fagin's extendible hashing dynamically doubles the size of the hash-table pointing pairs of links to the original bucket and expanding only the overflowing bucket (by restricting implementations to a specific family of hash functions) [17]. It has a two-level hierarchy: buckets (to store the directory entries) and a table of pointers (to the buckets). GPFS represents each bucket as a disk block and the pointer table as the block pointers in the directory's i-node. When the directory grows in size, GPFS allocates new blocks, moves some of the directory entries from the overflowing block into the new block and updates the block pointers in the i-node.

GPFS employs its client cache consistency and distributed locking mechanism to enable concurrent access to a shared directory [52]. Concurrent readers can cache the directory blocks using shared reader locks, which enables high performance for read-intensive workloads. Concurrent writers, however, need to acquire write locks from the lock manager before updating the directory blocks stored on the shared disk storage. When releasing (or acquiring) locks, GPFS versions before 3.2.1 force the directory block to be flushed to disk (or read back from disk) inducing high I/O overhead. Newer releases of GPFS have modified the cache consistency protocol to send the directory

insert requests directly to the current lock holder, instead of getting the block through the shared disk subsystem [1, 22, 27]. Still GPFS continues to synchronously write the directory's i-node (i.e., the mapping state) invalidating client caches to provide strong consistency guarantees [1]. In contrast, GIGA+ allows the mapping state to be stale at the client and never be shared between servers, thus seeking even more scalability.

Lustre and Ceph: Lustre's proposed clustered metadata service splits a directory using a hash of the directory entries only once over all available metadata servers when it exceeds a threshold size [37, 38]. The effectiveness of this "split once and for all" scheme depends on the eventual directory size and does not respond to dynamic increases in the number of servers. Ceph is another object-based cluster file system that uses dynamic sub-tree partitioning of the namespace and hashes individual directories when they get too big or experience too many accesses [63, 64]. Compared to Lustre and Ceph, GIGA+ splits a directory incrementally as a function of size, i.e., a small directory may be distributed over fewer servers than a larger one. Furthermore, GIGA+ facilitates dynamic server addition achieving balanced server load with minimal migration.

Linear hashing and LH:* Linear hashing grows a hash table by splitting its hash buckets in a linear order using a pointer to the *next* bucket to split [34]. Its distributed variant, called LH* [35], stores buckets on multiple servers and uses a central split coordinator that advances permission to split a partition to the next server. An attractive property of LH* is that it does not update a client's mapping state synchronously after every new split.

GIGA+ differs from LH* in several ways. To maintain consistency of the split pointer (at the coordinator), LH* splits only one bucket at a time [35, 36]; GIGA+ allows any server to split a bucket at any time without any coordination. LH* offers a complex partition pre-split optimization for higher concurrency [36], but it causes LH* clients to continuously incur some addressing errors even after the index stops growing; GIGA+ chose to minimize (and stop) addressing errors at the cost of more client state.

Consistent hashing: Consistent hashing divides the hash-space into randomly sized ranges distributed over server nodes [30, 58]. Consistent hashing is efficient at managing membership changes because server changes split or join hash-ranges of adjacent servers only, making it popular for wide-area peer-to-peer storage systems that have high rates of membership churn [13, 42, 48, 51]. Cluster systems, even though they have much lower churn than Internet-wide systems, have also used consistent hashing for data partitioning [15, 32], but have faced interesting challenges.

As observed in Amazon's Dynamo, consistent hashing's data distribution has a high load variance, even after using "virtual servers" to map multiple randomly sized hash-ranges to each node [15]. GIGA+ uses threshold-based binary splitting that provides better load distribution even for small clusters. Furthermore, consistent hashing systems assume that every data-set needs to be distributed over many nodes to begin with, i.e., they do not have support for incrementally growing data-sets that are mostly small – an important property of file system directories.

Other work: DDS [24] and Boxwood [40] also used scalable data-structures for storage infrastructure. While both GIGA+ and DDS use hash tables, GIGA+'s focus is on directories, unlike DDS's general cluster abstractions, with an emphasis on indexing that uses inconsistency at the clients; a non-goal for DDS [24]. Boxwood proposed primitives to simplify storage system development, and used B-link trees for storage layouts [40].

3 GIGA+ Indexing Design

3.1 Assumptions

GIGA+ is intended to be integrated into a modern cluster file system like PVFS, PanFS, GoogleFS, HDFS etc. All these scalable file systems have good fault tolerance usually including a consensus protocol for node membership and global configuration [9, 29, 65]. GIGA+ is not designed to replace membership or fault tolerance; it avoids this where possible and employs them where needed.

GIGA+ design is also guided by several assumptions about its use cases. First, most file system directories start small and remain small; studies of large file systems have found that 99.99% of the directories contain fewer than 8,000 files [4, 14]. Since only a few directories grow to really large sizes, GIGA+ is designed for incremental growth, that is, an empty or a small directory is initially stored on one server and is partitioned over an increasing number of servers as it grows in size. Perhaps most beneficially, incremental growth in GIGA+ handles adding servers gracefully. This allows GIGA+ to avoid degrading small directory performance; striping small directories across multiple servers will lead to inefficient resource utilization, particularly for directory scans (using `readdir()`) that will incur disk-seek latency on all servers only to read tiny partitions.

Second, because GIGA+ is targeting concurrently shared directories with up to billions of files, caching such directories at each client is impractical: the directories are too large and the rate of change too high. GIGA+ clients do not cache directories and send all directory operations to a server. Directory caching only for small rarely changing directories is an obvious extension employed, for example, by PanFS [66], that we have not yet implemented.

Finally, our goal in this research is to complement existing cluster file systems and support unmodified applications. So GIGA+ directories provide the strong consistency for directory entries and files that most POSIX-like file systems provide, i.e., once a client creates a file in a directory all other clients can access the file. This strong

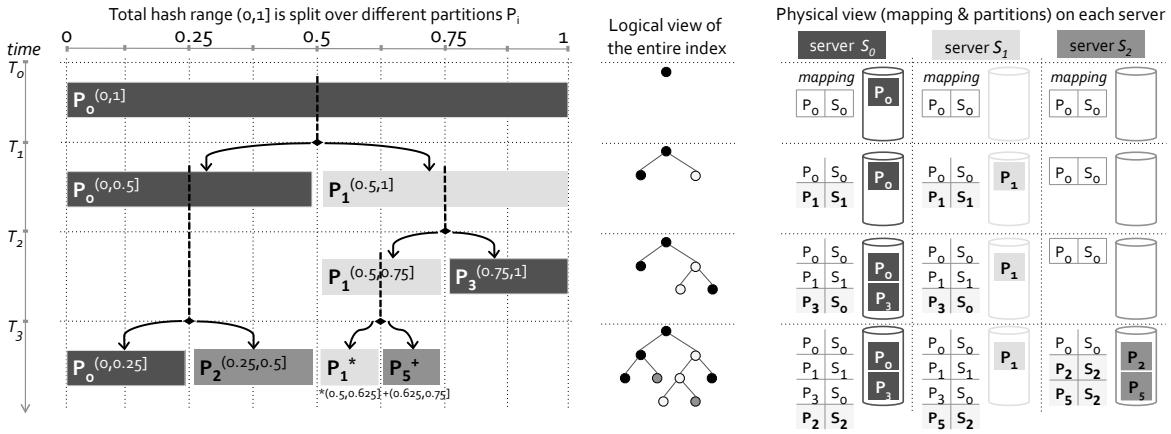

Figure 1 – Concurrent and unsynchronized data partitioning in GIGA+. *The hash-space $(0,1]$ is divided into multiple partitions (P_i) that are distributed over many servers (different shades of gray). Each server has a local, partial view of the entire index and can independently split its partitions without global co-ordination. In addition to enabling highly concurrent growth, an index starts small (on one server) and scales out incrementally.*

consistency API differentiates GIGA+ from "relaxed" consistency provided by newer storage systems including NoSQL systems like Cassandra [32] and Dynamo [15].

3.2 Unsynchronized data partitioning

GIGA+ uses hash-based indexing to incrementally divide each directory into multiple partitions that are distributed over multiple servers. Each filename (contained in a directory entry) is hashed and then mapped to a partition using an index. Our implementation uses the cryptographic MD5 hash function but is not specific to it. GIGA+ relies only on one property of the selected hash function: for any distribution of unique filenames, the hash values of these filenames must be uniformly distributed in the hash space [49]. This is the core mechanism that GIGA+ uses for load balancing.

Figure 1 shows how GIGA+ indexing grows incrementally. In this example, a directory is to be spread over three servers $\{S_0, S_1, S_2\}$ in three shades of gray color. $P_i^{(x,y]}$ denotes the hash-space range $(x,y]$ held by a partition with the unique identifier i.[2] GIGA+ uses the identifier i to map P_i to an appropriate server S_i using a round-robin mapping, i.e., server S_i is i `modulo` *num_servers*. The color of each partition indicates the (color of the) server it resides on. Initially, at time T_0, the directory is small and stored on a single partition $P_0^{(0,1]}$ on server S_0. As the directory grows and the partition size exceeds a threshold number of directory entries, provided this server knows of an underutilized server, S_0 splits $P_0^{(0,1]}$ into two by moving the greater half of its hash-space range to a new partition $P_1^{(0.5,1]}$ on S_1. As the directory expands, servers continue to split partitions onto more servers until all have about the same fraction of the hash-space to manage (analyzed in Section 5.2 and 5.3). GIGA+ computes a split's target partition identifier using well-known radix-based techniques.[3]

The key goal for GIGA+ is for each server to split independently, without system-wide serialization or synchronization. Accordingly, servers make local decisions to split a partition. The side-effect of uncoordinated growth is that GIGA+ servers do not have a global view of the partition-to-server mapping on any one server; each server only has a partial view of the entire index (the mapping tables in Figure 1). Other than the partitions that a server manages, a server knows only the identity of the server that knows more about each "child" partition resulting from a prior split by this server. In Figure 1, at time T_3, server S_1 manages partition P_1 at tree depth $r = 3$, and knows that it previously split P_1 to create children partitions, P_3 and P_5, on servers S_0 and S_2 respectively. Servers are mostly unaware about partition splits that happen on other servers (and did not target them); for instance, at time T_3, server S_0 is unaware of partition P_5 and server S_1 is unaware of partition P_2.

Specifically, each server knows only the split history of its partitions. The full GIGA+ index is a complete history of the directory partitioning, which is the transitive closure over the local mappings on each server. This full index is also not maintained synchronously by any client. GIGA+ clients can enumerate the partitions of a directory by traversing its split histories starting with the zeroth partition P_0. However, such a full index constructed and

[2] For simplicity, we disallow the hash value zero from being used.

[3] GIGA+ calculates the identifier of partition i using the depth of the tree, r, which is derived from the number of splits of the zeroth partition P_0. Specifically, if a partition has an identifier i and is at tree depth r, then in the next split P_i will move half of its filenames, from the larger half of its hash-range, to a new partition with identifier $i + 2^r$. After a split completes, both partitions will be at depth $r+1$ in the tree. In Figure 1, for example, partition $P_1^{(0.5,0.75]}$, with identifier $i = 1$, is at tree depth $r = 2$. A split causes P_1 to move the larger half of its hash-space $(0.625, 0.75]$ to the newly created partition P_5, and both partitions are then at tree depth of $r = 3$.

cached by a client may be stale at any time, particularly for rapidly mutating directories.

3.3 Tolerating inconsistent mapping at clients

Clients seeking a specific filename find the appropriate partition by probing servers, possibly incorrectly, based on their cached index. To construct this index, a client must have resolved the directory's parent directory entry which contains a cluster-wide i-node identifying the server and partition for the zeroth partition P_0. Partition P_0 may be the appropriate partition for the sought filename, or it may not because of a previous partition split that the client has not yet learned about. An "incorrectly" addressed server detects the addressing error by recomputing the partition identifier by re-hashing the filename. If this hashed filename does not belong in the partition it has, this server sends a split history update to the client. The client updates its cached version of the global index and retries the original request.

The drawback of allowing inconsistent indices is that clients may need additional probes before addressing requests to the correct server. The required number of incorrect probes depends on the client request rate and the directory mutation rate (rate of splitting partitions). It is conceivable that a client with an empty index may send $O(log(N_p))$ incorrect probes, where N_p is the number of partitions, but GIGA+'s split history updates makes this many incorrect probes unlikely (described in Section 5.4). Each update sends the split histories of all partitions that reside on a given server, filling all gaps in the client index known to this server and causing client indices to catch up quickly. Moreover, after a directory stops splitting partitions, clients soon after will no longer incur any addressing errors. GIGA+'s eventual consistency for cached indices is different from LH*'s eventual consistency because the latter's idea of independent splitting (called pre-splitting in their paper) suffers addressing errors even when the index stops mutating [36].

3.4 Handling server additions

This section describes how GIGA+ adapts to the addition of servers in a running directory service.[4]

When new servers are added to an existing configuration, the system is immediately no longer load balanced, and it should re-balance itself by migrating a minimal number of directory entries from all existing servers equally. Using the round-robin partition-to-server mapping, shown in Figure 1, a naive server addition scheme would require re-mapping almost all directory entries whenever a new server is added.

GIGA+ avoids re-mapping all directory entries on addition of servers by differentiating the partition-to-server mapping for initial directory growth from the mapping for additional servers. For additional servers, GIGA+ does not use the round-robin partition-to-server map (shown in Figure 1) and instead maps all future partitions to the new servers in a "sequential manner". The benefit of round-robin mapping is faster exploitation of parallelism when a directory is small and growing, while a sequential mapping for the tail set of partitions does not disturb previously mapped partitions more than is mandatory for load balancing.

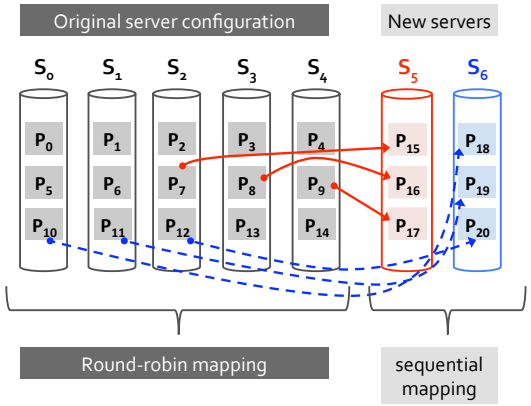

Figure 2 – Server additions in GIGA+. *To minimize the amount of data migrated, indicated by the arrows that show splits,* GIGA+ *changes the partition-to-server mapping from round-robin on the original server set to sequential on the newly added servers.*

Figure 2 shows an example where the original configuration has 5 servers with 3 partitions each, and partitions P_0 to P_{14} use a round-robin rule (for P_i, server is i `mod` N, where N is number of servers). After the addition of two servers, the six new partitions P_{15}-P_{20} will be mapped to servers using the new mapping rule: i `div` M, where M is the number of partitions per server (e.g., 3 partitions/server).

In GIGA+ even the number of servers can be stale at servers and clients. The arrival of a new server and its order in the global server list is declared by the cluster file system's configuration management protocol, such as Zookeeper for HDFS [29], leading to each existing server eventually noticing the new server. Once it knows about new servers, an existing server can inspect its partitions for those that have sufficient directory entries to warrant splitting and would split to a newly added server. The normal GIGA+ splitting mechanism kicks in to migrate only directory entries that belong on the new servers. The order in which an existing server inspects partitions can be entirely driven by client references to partitions, biasing migration in favor of active directories. Or based on an administrator control, it can also be driven by a background traversal of a list of partitions whose size exceeds the splitting threshold.

[4] Server removal (i.e., decommissioned, not failed and later replaced) is not as important for high performance systems so we leave it to be done by user-level data copy tools.

4 GIGA+ Implementation

GIGA+ indexing mechanism is primarily concerned with distributing the contents and work of large file system directories over many servers, and client interactions with these servers. It is not about the representation of directory entries on disk, and follows the convention of reusing mature local file systems like ext3 or ReiserFS (in Linux) for disk management found as is done by many modern cluster file systems [39, 46, 54, 63, 66].

The most natural implementation strategy for GIGA+ is as an extension of the directory functions of a cluster file system. GIGA+ is not about striping the data of huge files, server failure detection and failover mechanism, or RAID/replication of data for disk fault tolerance. These functions are present and, for GIGA+ purposes, adequate in most cluster file systems. Authors of a new version of PVFS, called OrangeFS, and doing just this by integrating GIGA+ into OrangeFS [2, 45]. Our goal is not to compete with most features of these systems, but to offer technology for advancing their support of high rates of mutation of large collections of small files.

For the purposes of evaluating GIGA+ on file system directory workloads, we have built a skeleton cluster file system; that is, we have not implemented data striping, fault detection or RAID in our experimental framework. Figure 3 shows our user-level GIGA+ directory prototypes built using the FUSE API [19]. Both client and server processes run in user-space, and communicate over TCP using SUN RPC [56]. The prototype has three layers: unmodified applications running on clients, the GIGA+ indexing modules (of the skeletal cluster file system on clients and servers) and a backend persistent store at the server. Applications interact with a GIGA+ client using the VFS API (e.g., `open()`, `creat()` and `close()` syscalls). The FUSE kernel module intercepts and redirects these VFS calls the client-side GIGA+ indexing module which implements the logic described in the previous section.

4.1 Server implementation

The GIGA+ server module's primary purpose is to synchronize and serialize interactions between all clients and a specific partition. It need not "store" the partitions, but it owns them by performing all accesses to them. Our server-side prototype is currently layered on lower level file systems, ext3 and ReiserFS. This decouples GIGA+ indexing mechanisms from on-disk representation.

Servers map logical GIGA+ partitions to directory objects within the backend file system. For a given (huge) directory, its entry in its parent directory names the "zeroth partition", $P_0^{(0,1]}$, which is a directory in a server's underlying file system. Most directories are not huge and will be represented by just this one zeroth partition.

GIGA+ stores some information as extended attributes on the directory holding a partition: a GIGA+ directory ID

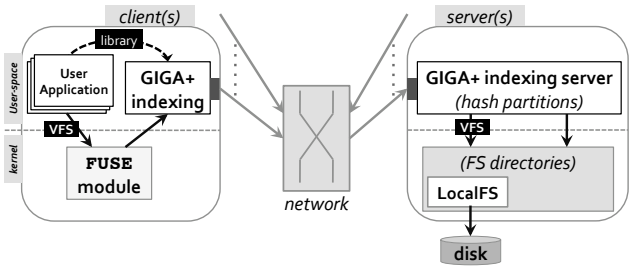

Figure 3 – GIGA+ experimental prototype.

(unique across servers), the the partition identifier P_i and its range $(x, y]$. The range implies the leaf in the directory's logical tree view of the huge directory associated with this partition (the center column of Figure 1) and that determines the prior splits that had to have occurred to cause this partition to exist (that is, the split history).

To associate an entry in a cached index (a partition) with a specific server, we need the list of servers over which partitions are round robin allocated and the list of servers over which partitions are sequentially allocated. The set of servers that are known to the cluster file system at the time of splitting the zeroth partition is the set of servers that are round robin allocated for this directory and the set of servers that are added after a zeroth partition is split are the set of servers that are sequentially allocated.[5]

Because the current list of servers will always be available in a cluster file system, only the list of servers at the time of splitting the zeroth server needs to be also stored in a partition's extended attributes. Each split propagates the directory ID and set of servers at the time of the zeroth partition split to the new partition, and sets the new partition's identifier P_i and range $(x, y]$ as well as providing the entries from the parent partition that hash into this range $(x, y]$.

Each partition split is handled by the GIGA+ server by locally locking the particular directory partition, scanning its entries to build two sub-partitions, and then transactionally migrating ownership of one partition to another server before releasing the local lock on the partition [55]. In our prototype layered on local file systems, there is no transactional migration service available, so we move the directory entries and copy file data between servers. Our experimental splits are therefore more expensive than they should be in a production cluster file system.

4.2 Client implementation

The GIGA+ client maintains cached information, some potentially stale, global to all directories. It caches the current server list (which we assume only grows over time)

[5] The contents of a server list are logical server IDs (or names) that are converted to IP addresses dynamically by a directory service integrated with the cluster file system. Server failover (and replacement) will bind a different address to the same server ID so the list does not change during normal failure handling.

and the number of partitions per server (which is fixed) obtained from whichever server GIGA+ was mounted on. For each active directory GIGA+ clients cache the cluster-wide i-node of the zeroth partition, the directory ID, and the number of servers at the time when the zeroth partition first split. The latter two are available as extended attributes of the zeroth partition. Most importantly, the client maintains a bitmap of the global index built according to Section 3, and a maximum tree-depth, $r = \lceil log(i) \rceil$, of any partition P_i present in the global index.

Searching for a file name with a specific hash value, H, is done by inspecting the index bitmap at the offset j determined by the r lower-order bits of H. If this is set to '1' then H is in partition P_j. If not, decrease r by one and repeat until $r = 0$ which refers to the always known zeroth partition P_0. Identifying the server for partition P_j is done by lookup in the current server list. It is either $j \bmod N$, where N is the number of servers at the time the zeroth partition split), or $j \operatorname{div} M$, where M is the number of partitions per server, with the latter used if j exceeds the product of the number of servers at the time of zeroth partition split and the number of partitions per server.

Most VFS operations depend on lookups; `readdir()` however can be done by walking the bitmaps, enumerating the partitions and scanning the directories in the underlying file system used to store partitions.

4.3 Handling failures

Modern cluster file systems scale to sizes that make fault tolerance mandatory and sophisticated [8, 20, 65]. With GIGA+ integrated in a cluster file system, fault tolerance for data and services is already present, and GIGA+ does not add major challenges. In fact, handling network partitions and client-side reboots are relatively easy to handle because GIGA+ tolerates stale entries in a client's cached index of the directory partition-to-server mapping and because GIGA+ does not cache directory entries in client or server processes (changes are written through to the underlying file system). Directory-specific client state can be reconstructed by contacting the zeroth partition named in a parent directory entry, re-fetching the current server list and rebuilding bitmaps through incorrect addressing of server partitions during normal operations.

Other issues, such as on-disk representation and disk failure tolerance, are a property of the existing cluster file system's directory service, which is likely to be based on replication even when large data files are RAID encoded [66]. Moreover, if partition splits are done under a lock over the entire partition, which is how our experiments are done, the implementation can use a non-overwrite strategy with a simple atomic update of which copy is live. As a result, recovery becomes garbage collection of spurious copies triggered by the failover service when it launches a new server process or promotes a passive backup to be the active server [9, 29, 65].

While our architecture presumes GIGA+ is integrated into a full featured cluster file system, it is possible to layer GIGA+ as an interposition layer over and independent of a cluster file system, which itself is usually layered over multiple independent local file systems [20, 46, 54, 66]. Such a layered GIGA+ would not be able to reuse the fault tolerance services of the underlying cluster file system, leading to an extra layer of fault tolerance. The primary function of this additional layer of fault tolerance is replication of the GIGA+ server's write-ahead logging for changes it is making in the underlying cluster file system, detection of server failure, election and promotion of backup server processes to be primaries, and reprocessing of the replicated write-ahead log. Even the replication of the write-ahead log may be unnecessary if the log is stored in the underlying cluster file system, although such logs are often stored outside of cluster file systems to improve the atomicity properties writing to them [12, 26]. To ensure load balancing during server failure recovery, the layered GIGA+ server processes could employ the well-known chained-declustering replication mechanism to shift work among server processes [28], which has been used in other distributed storage systems [33, 60].

5 Experimental Evaluation

Our experimental evaluation answers two questions: (1) How does GIGA+ scale? and (2) What are the tradeoffs of GIGA+'s design choices involving incremental growth, weak index consistency and selection of the underlying local file system for out-of-core indexing (when partitions are very large)?

All experiments were performed on a cluster of 64 machines, each with dual quad-core 2.83GHz Intel Xeon processors, 16GB memory and a 10GigE NIC, and Arista 10 GigE switches. All nodes were running the Linux 2.6.32-js6 kernel (Ubuntu release) and GIGA+ stores partitions as directories in a local file system on one 7200rpm SATA disk (a different disk is used for all non-GIGA+ storage). We assigned 32 nodes as servers and the remaining 32 nodes as load generating clients. The threshold for splitting a partition is always 8,000 entries.

We used the synthetic `mdtest` benchmark [41] (used by parallel file system vendors and users) to insert zero-byte files in to a directory [27, 63]. We generated three types of workloads. First, a *concurrent create* workload that creates a large number of files concurrently in a single directory. Our configuration uses eight processes per client to simultaneously create files in a common directory, and the number of files created is proportional to the number of servers: a single server manages 400,000 files, a 800,000 file directory is created on 2 servers, a 1.6 million file directory on 4 servers, up to a 12.8 million file directory on 32 servers. Second, we use a *lookup workload* that performs a `stat()` on random files in the directory. And

	File System	File creates/second in one directory
GIGA+ *(layered on Reiser)*	Library API	17,902
	VFS/FUSE API	5,977
Local file systems	Linux ext3	16,470
	Linux ReiserFS	20,705
Networked file systems	NFSv3 filer	521
	HadoopFS	4,290
	PVFS	1,064

Table 1 – File create rate in a single directory on a single server. *An average of five runs (with 1% standard deviation).*

finally, we use a mixed workload where clients issue create and lookup requests in a pre-configured ratio.

5.1 Scale and performance

We begin with a baseline for the performance of various file systems running the `mdtest` benchmark. First we compare `mdtest` running locally on Linux ext3 and ReiserFS local file systems to `mdtest` running on a separate client and single server instance of the PVFS cluster file system (using ext3) [46], Hadoop's HDFS (using ext3) [54] and a mature commercial NFSv3 filer. In this experiment GIGA+ always uses one partition per server. Table 1 shows the baseline performance.

For GIGA+ we use two machines with ReiserFS on the server and two ways to bind `mdtest` to GIGA+: direct library linking (non-POSIX) and VFS/FUSE linkage (POSIX). The library approach allows `mdtest` to use custom object creation calls (such as `giga_creat()`) avoiding system call and FUSE overhead in order to compare to `mdtest` directly in the local file system. Among the local file systems, with local `mdtest` threads generating file creates, both ReiserFS and Linux ext3 deliver high directory insert rates.[6] Both file systems were configured with `-noatime` and `-nodiratime` option; Linux ext3 used write-back journaling and the `dir_index` option to enable hashed-tree indexing, and ReiserFS was configured with the `-notail` option, a small-file optimization that packs the data inside an i-node for high performance [47]. GIGA+ with `mdtest` workload generating threads on a different machine, when using the library interface (sending only one RPC per create) and ReiserFS as the backend file system, creates at better than 80% of the rate of ReiserFS with local load generating threads. This comparison shows that remote RPC is not a huge penalty for GIGA+. We tested this library version only to gauge GIGA+ efficiency compared to local file systems and do not use this setup for any remaining experiments.

To compare with the network file systems, GIGA+ uses the VFS/POSIX interface. In this case each VFS file `creat()` results in three RPC calls to the server: `getattr()` to check if a file exists, the actual `creat()` and another `getattr()` after creation to load the created file's attributes. For a more enlightening comparison, cluster file systems were configured to be functionally equivalent to the GIGA+ prototype; specifically, we disabled HDFS's write-ahead log and replication, and we used default PVFS which has no redundancy unless a RAID controller is added. For the NFSv3 filer, because it was in production use, we could not disable its RAID redundancy and it is correspondingly slower than it might otherwise be. GIGA+ directories using the VFS/FUSE interface also outperforms all three networked file systems, probably because the GIGA+ experimental prototype is skeletal while others are complex production systems.

[6] We tried XFS too, but it was extremely slow during the create-intensive workload and do not report those numbers in this paper.

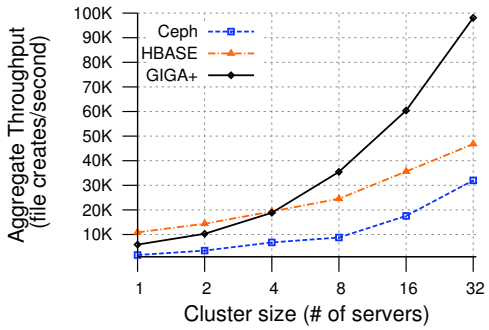

Figure 4 – Scalability of GIGA+ FS directories. GIGA+ *directories deliver a peak throughput of roughly 98,000 file creates per second. The behavior of underlying local file system (ReiserFS) limits* GIGA+*'s ability to match the ideal linear scalability.*

Figure 4 plots aggregate operation throughput, in file creates per second, averaged over the complete *concurrent create* benchmark run as a function of the number of servers (on a log-scale X-axis). GIGA+ with partitions stored as directories in ReiserFS scales linearly up to the size of our 32-server configuration, and can sustain 98,000 file creates per second - this exceeds today's most rigorous scalability demands [44].

Figure 4 also compares GIGA+ with the scalability of the Ceph file system and the HBase distributed key-value store. For Ceph, Figure 4 reuses numbers from experiments performed on a different cluster from the original paper [63]. That cluster used dual-core 2.4GHz machines with IDE drives, with equal numbered separate nodes as workload generating clients, metadata servers and disk servers with object stores layered on Linux ext3. HBase is used to emulate Google's Colossus file system which plans to store file system metadata in BigTable instead of internally on single master node[18]. We setup HBase on a 32-node HDFS configuration with a single copy (no replication) and disabled two parameters: blocking while the HBase servers are doing compactions and write-ahead logging for inserts (a common practice to speed up insert-

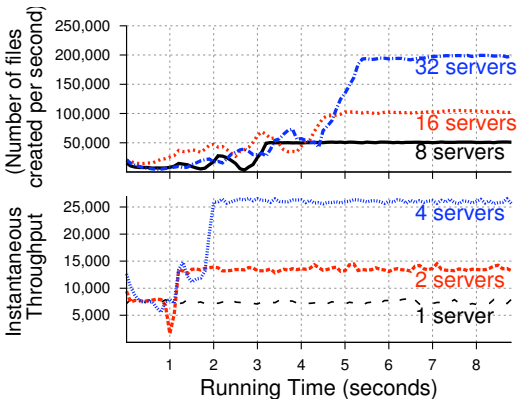

Figure 5 – Incremental scale-out growth. GIGA+ *achieves linear scalability after distributing one partition on each available server. During scale-out, periodic drops in aggregate create rate correspond to concurrent splitting on all servers.*

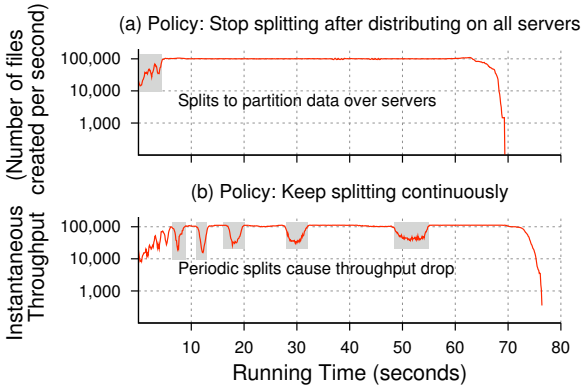

Figure 6 – Effect of splitting heuristics. GIGA+ *shows that splitting to create at most one partition on each of the 16 servers delivers scalable performance. Continuous splitting, as in classic database indices, is detrimental in a distributed scenario.*

ing data in HBase). This configuration allowed HBase to deliver better performance than GIGA+ for the single server configuration because the HBase tables are striped over all 32-nodes in the HDFS cluster. But configurations with many HBase servers scale poorly.

GIGA+ also demonstrated scalable performance for the *concurrent lookup* workload delivering a throughput of more than 600,000 file lookups per second for our 32-server configuration (not shown). Good lookup performance is expected because the index is not mutating and load is well-distributed among all servers; the first few lookups fetch the directory partitions from disk into the buffer cache and the disk is not used after that. Section 5.4 gives insight on addressing errors during mutations.

5.2 Incremental scaling properties

In this section, we analyze the scaling behavior of the GIGA+ index independent of the disk and the on-disk directory layout (explored later in Section 5.5). To eliminate performance issues in the disk subsystem, we use Linux's in-memory file system, `tmpfs`, to store directory partitions. Note that we use `tmpfs` only in this section, all other analysis uses on-disk file systems.

We run the *concurrent create* benchmark to create a large number of files in an empty directory and measure the aggregate throughput (file creates per second) continuously throughout the benchmark. We ask two questions about scale-out heuristics: (1) what is the effect of splitting during incremental scale-out growth? and (2) how many partitions per server do we keep?

Figure 5 shows the first 8 seconds of the *concurrent create* workload when the number of partitions per server is one. The primary result in this figure is the near linear create rate seen after the initial seconds. But the initial few seconds are more complex. In the single server case, as expected, the throughput remains flat at roughly 7,500 file creates per second due to the absence of any other server. In the 2-server case, the directory starts on a single server and splits when it has more than 8,000 entries in the partition. When the servers are busy splitting, at the 0.8-second mark, throughput drops to half for a short time.

Throughput degrades even more during the scale-out phase as the number of directory servers goes up. For instance, in the 8-server case, the aggregate throughput drops from roughly 25,000 file creates/second at the 3-second mark to as low as couple of hundred creates/second before growing to the desired 50,000 creates/second. This happens because all servers are busy splitting, i.e., partitions overflow at about the same time which causes all servers (where these partitions reside) to split without any co-ordination at the same time. And after the split spreads the directory partitions on twice the number of servers, the aggregate throughput achieves the desired linear scale.

In the context of the second question about how many partitions per server, classic hash indices, such as extendible and linear hashing [17, 34], were developed for out-of-core indexing in single-node databases. An out-of-core table keeps splitting partitions whenever they overflow because the partitions correspond to disk allocation blocks [23]. This implies an unbounded number of partitions per server as the table grows. However, the splits in GIGA+ are designed to parallelize access to a directory by distributing the directory load over all servers. Thus GIGA+ can stop splitting after each server has a share of work, i.e., at least one partition. When GIGA+ limits the number of partitions per server, the size of partitions continue to grow and GIGA+ lets the local file system on each server handle physical allocation and out-of-core memory management.

Figure 6 compares the effect of different policies for the number of partitions per server on the system throughput (using a log-scale Y-axis) during a test in which a large directory is created over 16 servers. Graph (a) shows a split policy that stops when every server has one partition, caus-

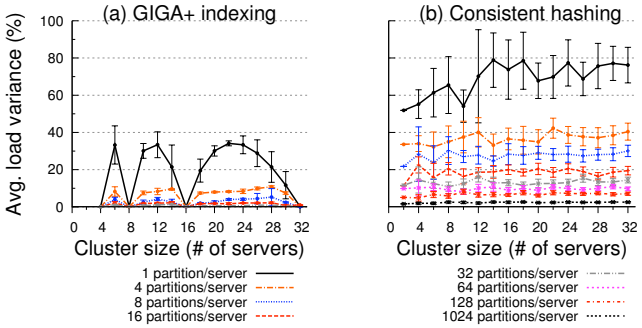

Figure 7 – Load-balancing in GIGA+. *These graphs show the quality of load balancing measured as the mean load deviation across the entire cluster (with 95% confident interval bars). Like virtual servers in consistent hashing,* GIGA+ *also benefits from using multiple hash partitions per server.* GIGA+ *needs one to two orders of magnitude fewer partitions per server to achieve comparable load distribution relative to consistent hashing.*

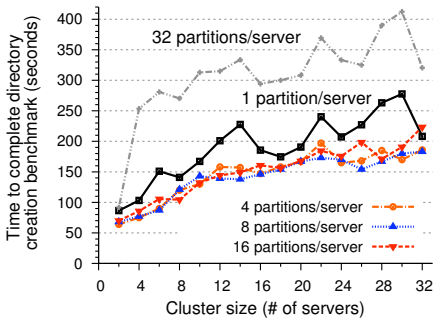

Figure 8 – Cost of splitting partitions. *Using 4, 8, or 16 partitions per server improves the performance of* GIGA+ *directories layered on Linux ext3 relative to 1 partition per server (better load-balancing) or 32 partitions per server (when the cost of more splitting dominates the benefit of load-balancing).*

ing partitions to ultimately get much bigger than 8,000 entries. Graph (b) shows the continuous splitting policy used by classic database indices where a split happens whenever a partition has more than 8,000 directory entries. With continuous splitting the system experiences periodic throughput drops that last longer as the number of partitions increases. This happens because repeated splitting maps multiple partitions to each server, and since uniform hashing will tend to overflow all partitions at about the same time, multiple partitions will split on all the servers at about the same time.

Lesson #1: To avoid the overhead of continuous splitting in a distributed scenario, GIGA+ stops splitting a directory after all servers have a fixed number of partitions and lets a server's local file system deal with out-of-core management of large partitions.

5.3 Load balancing efficiency

The previous section showed only configurations where the number of servers is a power-of-two. This is a special case because it is naturally load-balanced with only a single partition per server: the partition on each server is responsible for a hash-range of size 2^r-th part of the total hash-range $(0,1]$. When the number of servers is not a power-of-two, however, there is load imbalance. Figure 7 shows the load imbalance measured as the average fractional deviation from even load for all numbers of servers from 1 to 32 using Monte Carlo model of load distribution. In a cluster of 10 servers, for example, each server is expected to handle 10% of the total load; however, if two servers are experiencing 16% and 6% of the load, then they have 60% and 40% variance from the average load respectively. For different cluster sizes, we measure the variance of each server, and use the average (and 95% confidence interval error bars) over all the servers.

We compute load imbalance for GIGA+ in Figure 7(a) as follows: when the number of servers N is not a power-of-two, $2^r < N < 2^{r+1}$, then a random set of $N - 2^r$ partitions from tree depth r, each with range size $1/2^r$, will have split into $2(N-2^r)$ partitions with range size $1/2^{r+1}$. Figure 7(a) shows the results of five random selections of $N-2^r$ partitions that split on to the $r+1$ level. Figure 7(a) shows the expected periodic pattern where the system is perfectly load-balanced when the number of servers is a power-of-two. With more than one partition per server, each partition will manage a smaller portion of the hash-range and the sum of these smaller partitions will be less variable than a single large partition as shown in Figure 7(a). Therefore, more splitting to create more than one partition per server significantly improves load balance when the number of servers is not a power-of-two.

Multiple partitions per server is also used by Amazon's Dynamo key-value store to alleviate the load imbalance in consistent hashing [15]. Consistent hashing associates each partition with a random point in the hash-space $(0,1]$ and assigns it the range from this point up to the next larger point and wrapping around, if necessary. Figure 7(b) shows the load imbalance from Monte Carlo simulation of using multiple partitions (virtual servers) in consistent hashing by using five samples of a random assignment for each partition and how the sum, for each server, of partition ranges selected this way varies across servers. Because consistent hashing's partitions have more randomness in each partition's hash-range, it has a higher load variance than GIGA+ – almost two times worse. Increasing the number of hash-range partitions significantly improves load distribution, but consistent hashing needs more than 128 partitions per server to match the load variance that GIGA+ achieves with 8 partitions per server – an order of magnitude more partitions.

More partitions is particularly bad because it takes longer for the system to stop splitting, and Figure 8 shows how this can impact overall performance. Consistent hash-

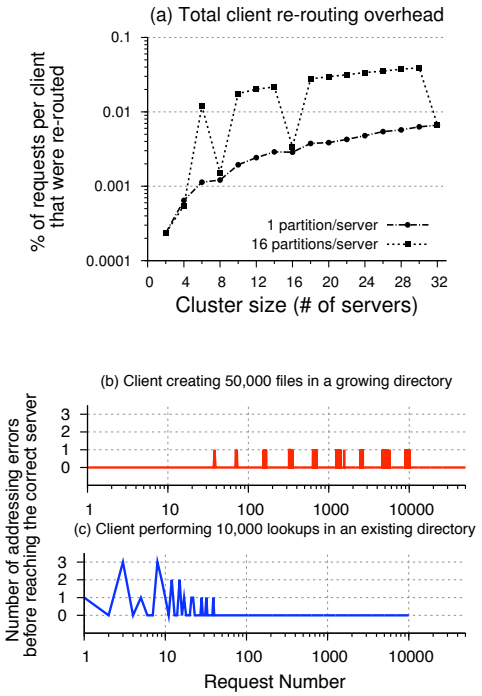

Figure 9 – Cost of using inconsistent mapping at the clients. *Using weak consistency for mapping state at the clients has a very negligible overhead on client performance (a). And this overhead – extra message re-addressing hops – occurs for initial requests until the client learns about all the servers (b and c).*

ing theory has alternate strategies for reducing imbalance but these often rely on extra accesses to servers all of the time and global system state, both of which will cause impractical degradation in our system [10].

Since having more partitions per server always improves load-balancing, at least a little, how many partitions should GIGA+ use? Figure 8 shows the *concurrent create* benchmark time for GIGA+ as a function of the number of servers for 1, 4, 8, 16 and 32 partitions per server. We observe that with 32 partitions per server GIGA+ is roughly 50% slower than with 4, 8 and 16 partitions per server. Recall from Figure 7(a) that the load-balancing efficiency from using 32 partitions per server is only about 1% better than using 16 partitions per server; the high cost of splitting to create twice as many partitions outweighs the minor load-balancing improvement.

Lesson #2: Splitting to create more than one partition per server significantly improves GIGA+ load balancing for non power-of-two numbers of servers, but because of the performance penalty during extra splitting the overall performance is best with only a few partitions per server.

5.4 Cost of weak mapping consistency

Figure 9(a) shows the overhead incurred by clients when their cached indices become stale. We measure the percentage of all client requests that were re-routed when running the *concurrent create* benchmark on different cluster sizes. This figure shows that, in absolute terms, fewer than 0.05% of the requests are addressed incorrectly; this is only about 200 requests per client because each client is doing 400,000 file creates. The number of addressing errors increases proportionally with the number of partitions per server because it takes longer to create all partitions. In the case when the number of servers is a power-of-two, after each server has at least one partition, subsequent splits yield two smaller partitions on the same server, which will not lead to any additional addressing errors.

We study further the worst case in Figure 9(a), 30 servers with 16 partitions per server, to learn when addressing errors occur. Figure 9(b) shows the number of errors encountered by each request generated by one client thread (i.e., one of the eight workload generating threads per client) as it creates 50,000 files in this benchmark. Figure 9(b) suggests three observations. First, the index update that this thread gets from an incorrectly addressed server is always sufficient to find the correct server on the second probe. Second, that addressing errors are bursty, one burst for each level of the index tree needed to create 16 partitions on each of 30 servers, or 480 partitions ($2^8 < 480 < 2^9$). And finally, that the last 80% of the work is done after the last burst of splitting without any addressing errors.

To further emphasize how little incorrect server addressing clients generate, Figure 9(c) shows the addressing experience of a new client issuing 10,000 lookups after the current create benchmark has completed on 30 servers with 16 partitions per server.[7] This client makes no more than 3 addressing errors for a specific request, and no more than 30 addressing errors total and makes no more addressing errors after the 40th request.

Lesson #3: GIGA+ clients incur neglible overhead (in terms of incorrect addressing errors) due to stale cached indices, and no overhead shortly after the servers stop splitting partitions. Although not a large effect, fewer partitions per server lowers client addressing errors.

5.5 Interaction with backend file systems

Because some cluster file systems represent directories with equivalent directories in a local file system [39] and because our GIGA+ experimental prototype represents partitions as directories in a local file system, we study how the design and implementation of Linux ext3 and ReiserFS local file systems affects GIGA+ partition splits. Although different local file system implementations can be expected to have different performance, especially for emerging workloads like ours, we were surprised by the size of the differences.

Figure 10 shows GIGA+ file create rates when there are 16 servers for four different configurations: Linux ext3

[7] Figure 9 predicts the addressing errors of a client doing only lookups on a mutating directory because both `create(filename)` and `lookup(filename)` do the same addressing.

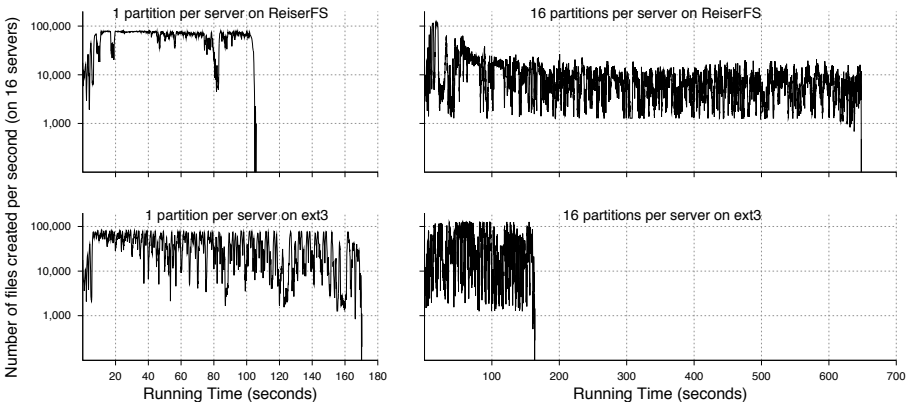

Figure 10 – Effect of underlying file systems. *This graph shows the concurrent create benchmark behavior when the* GIGA+ *directory service is distributed on 16 servers with two local file systems, Linux ext3 and ReiserFS. For each file system, we show two different numbers of partitions per server, 1 and 16.*

or ReiserFS storing partitions as directories, and 1 or 16 partitions per server. Linux ext3 directories use h-trees [11] and ReiserFS uses balanced B-trees [47]. We observed two interesting phenomenon: first, the benchmark running time varies from about 100 seconds to over 600 seconds, a factor of 6, and second, the backend file system yielding the faster performance is different when there are 16 partitions on each server than with only one.

Comparing a single partition per server in GIGA+ over ReiserFS and over ext3 (left column in Figure 10), we observe that the benchmark completion time increases from about 100 seconds using ReiserFS to nearly 170 seconds using ext3. For comparison, the same benchmark completed in 70 seconds when the backend was the in-memory `tmpfs` file system. Looking more closely at Linux ext3, as a directory grows, ext3's journal also grows and periodically triggers ext3's `kjournald` daemon to flush a part of the journal to disk. Because directories are growing on all servers at roughly the same rate, multiple servers flush their journal to disk at about the same time leading to troughs in the aggregate file create rate. We observe this behavior for all three journaling modes supported by ext3. We confirmed this hypothesis by creating an ext3 configuration with the journal mounted on a second disk in each server, and this eliminated most of the throughput variability observed in ext3, completing the benchmark almost as fast as with ReiserFS. For ReiserFS, however, placing the journal on a different disk had little impact.

The second phenomenon we observe, in the right column of Figure 10, is that for GIGA+ with 16 partitions per server, ext3 (which is insensitive to the number of partitions per server) completes the create benchmark more than four times faster than ReiserFS. We suspect that this results from the on-disk directory representation. ReiserFS uses a balanced B-tree for *all objects* in the file system, which re-balances as the file system grows and changes over time [47]. When partitions are split more often, as in case of 16 partitions per server, the backend file system structure changes more, which triggers more re-balancing in ReiserFS and slows the create rate.

Lesson #4: Design decisions of the backend file system have subtle but large side-effects on the performance of a distributed directory service. Perhaps the representation of a partition should not be left to the vagaries of whatever local file system is available.

6 Conclusion

In this paper we address the emerging requirement for POSIX file system directories that store massive number of files and sustain hundreds of thousands of concurrent mutations per second. The central principle of GIGA+ is to use asynchrony and eventual consistency in the distributed directory's internal metadata to push the limits of scalability and concurrency of file system directories. We used these principles to prototype a distributed directory implementation that scales linearly to best-in-class performance on a 32-node configuration. Our analysis also shows that GIGA+ achieves better load balancing than consistent hashing and incurs a neglible overhead from allowing stale lookup state at its clients.

Acknowledgements. This work is based on research supported in part by the Department of Energy, under award number DE-FC02-06ER25767, by the Los Alamos National Laboratory, under contract number 54515-001-07, by the Betty and Gordon Moore Foundation, by the National Science Foundation under awards CCF-1019104 and SCI-0430781, and by Google and Yahoo! research awards. We thank Cristiana Amza, our shepherd, and the anonymous reviewers for their thoughtful reviews of our paper. John Bent and Gary Grider from LANL provided valuable feedback from the early stages of this work; Han Liu assisted with HBase experimental setup; and Vijay Vasudevan, Wolfgang Richter, Jiri Simsa, Julio Lopez and Varun Gupta helped with early drafts of the paper. We also thank the member companies of the PDL Consortium (including APC, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support.

References

[1] Private Communication with Frank Schmuck and Roger Haskin, IBM, February 2010.

[2] Private Communication with Walt Ligon, OrangeFS (`http://orangefs.net`), November 2010.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB '09)*, Lyon, France, August 2009.

[4] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose CA, February 2007.

[5] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The Claremont report on database research. *ACM SIGMOD Record*, 37(3), September 2008.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[7] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the ACM/IEEE Transactions on Computing Conference on High Performance Networking and Computing (SC '09)*, Portland OR, November 2009.

[8] P. Braam and B. Neitzel. Scalable Locking and Recovery for Network File Systems. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07)*, Reno NV, November 2007.

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

[10] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley CA, February 2003.

[11] M. Cao, T. Y. Ts'o, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the Art: Where we are with the ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, June 2007.

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

[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[14] S. Dayal. Characterizing HEC Storage Systems at Rest. Technical Report CMU-PDL-08-109, Carnegie Mellon University, July 2008.

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

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

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

[18] A. Fikes. Storage Architecture and Challenges. Presentation at the 2010 Google Faculty Summit. Talk slides at `http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf`.

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

[20] S. Ghemawat, H. Gobioff, and S.-T. Lueng. Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing NY, October 2003.

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

[22] GPFS. An Introduction to GPFS Version 3.2.1. `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp`, November 2008.

[23] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

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

[25] J. H. Hartman and J. K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, Asheville NC, December 1993.

[26] HBase. The Hadoop Database. `http://hadoop.apache.org/hbase/`, December 2010.

[27] R. Hedges, K. Fitzgerald, M. Gary, and D. M. Stearman. Comparison of leading parallel NAS file systems on commodity hardware. Poster at the Petascale Data Storage Workshop 2010. `http://www.pdsi-scidac.org/events/PDSW10/resources/posters/parallelNASFSs.pdf`, November 2010.

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

[29] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, Boston MA, June 2010.

[30] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the ACM Symposium on Theory of Computing (STOC '97)*, El Paso TX, May 1997.

[31] P. Kogge. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. DARPA IPTO Report at `http://www.er.doe.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf`, September 2008.

[32] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. In *Proceedings of the Workshop on Large-Scale Distribued Systems and Middleware (LADIS '09)*, Big Sky MT, October 2009.

[33] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks.

In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, Cambridge MA, October 1996.

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

[35] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - Linear Hashing for Distributed Files. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington D.C., June 1993.

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

[37] Lustre. Clustered Metadata Design. `http://wiki.lustre.org/images/d/db/HPCS_CMD_06_15_09.pdf`, September 2009.

[38] Lustre. Clustered Metadata. `http://wiki.lustre.org/index.php/Clustered_Metadata`, September 2010.

[39] Lustre. Lustre File System. `http://www.lustre.org`, December 2010.

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

[41] MDTEST. mdtest: HPC benchmark for metadata performance. `http://sourceforge.net/projects/mdtest/`, December 2010.

[42] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston MA, November 2002.

[43] NetApp-Community-Form. Millions of files in a single directory. Discussion at `http://communities.netapp.com/thread/7190?tstart=0`, February 2010.

[44] H. Newman. HPCS Mission Partner File I/O Scenarios, Revision 3. `http://wiki.lustre.org/images/5/5a/Newman_May_Lustre_Workshop.pdf`, November 2008.

[45] OrangeFS. Distributed Directories in OrangeFS v2.8.3-EXP. `http://orangefs.net/trac/orangefs/wiki/Distributeddirectories`.

[46] PVFS2. Parallel Virtual File System, Version 2. `http://www.pvfs2.org`, December 2010.

[47] H. Reiser. ReiserFS. `http://www.namesys.com/`.

[48] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the Oceanstore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco CA, March 2003.

[49] R. A. Rivest. The MD5 Message Digest Algorithm. Internet RFC 1321, April 1992.

[50] R. Ross, E. Felix, B. Loewe, L. Ward, J. Nunez, J. Bent, E. Salmon, and G. Grider. High End Computing Revitalization Task Force (HECRTF), Inter Agency Working Group (HECIWG) File Systems and I/O Research Guidance Workshop 2006. `http://institutes.lanl.gov/hec-fsio/docs/HECIWG-FSIO-FY06-Workshop-Document-FINAL6.pdf`, 2006.

[51] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[52] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey CA, January 2002.

[53] M. Seltzer. Beyond Relational Databases. *Communications of the ACM*, 51(7), July 2008.

[54] K. Shvachko, H. Huang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Transactions on Computing Symposium on Mass Storage Systems and Technologies (MSST '10)*, Lake Tahoe NV, May 2010.

[55] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger. A Transparently-Scalable Metadata Service for the Ursa Minor Storage System. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '10)*, Boston MA, June 2010.

[56] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. Internet RFC 1831, August 1995.

[57] StackOverflow. Millions of small graphics files and how to overcome slow file system access on XP. Discussion at `http://stackoverflow.com/questions/1638219/`, October 2009.

[58] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, San Diego CA, August 2001.

[59] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, Trondheim, Norway, September 2005.

[60] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997.

[61] TOP500. Top 500 Supercomputer Sites. `http://www.top500.org`, December 2010.

[62] D. Tweed. One usage of up to a million files/directory. Email thread at `http://leaf.dragonflybsd.org/mailarchive/kernel/2008-11/msg00070.html`, November 2008.

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

[64] S. A. Weil, K. Pollack, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the ACM/IEEE Transactions on Computing Conference on High Performance Networking and Computing (SC '04)*, Pittsburgh PA, November 2004.

[65] B. Welch. Integrated System Models for Reliable Petascale Storage Systems. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07)*, Reno NV, November 2007.

[66] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose CA, February 2008.

[67] R. Wheeler. One Billion Files: Scalability Limits in Linux File Systems. Presentation at LinuxCon '10. Talk Slides at `http://events.linuxfoundation.org/slides/2010/linuxcon2010_wheeler.pdf`, August 2010.

[68] ZFS-discuss. Million files in a single directory. Email thread at `http://mail.opensolaris.org/pipermail/zfs-discuss/2009-October/032540.html`, October 2009.