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

Swapnil Patil   Garth Gibson

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

CMU-PDL-10-110

September 2010

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

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 that make only local, independent decisions about migration.* Giga+ *uses two tenets, asynchrony and inconsistency, to: (1) partition the index among all servers without any synchronization or serialization, and (2) minimize stale and inconsistent mapping state at the clients. Applications are provided traditional strong data consistency semantics, and cluster growth requires minimal directory entry migration. 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

Most file systems deliver scalable performance for large files, but not for large numbers of files [63, 18]; in particular, they lack scalable support for ingesting millions to billions of small files in a single directory - a growing requirement for data-intensive applications [48, 43]. We present the Giga+ file system directory service that uses highly concurrent and decentralized indexing, and that scales to store millions of files in a single directory and sustain hundreds of thousands of insertions per second.

The key feature of the Giga+ approach is to enable higher concurrency for index mutations (particularly inserts) by eliminating system-wide serialization and synchronization. Giga+ realizes this principle by aggressively distributing large, mutating directories, in a load-balanced manner, on a cluster of nodes and allowing each node to independently manage its own out-of-core data indexing. Like traditional distributed indices [35, 17, 50], Giga+ incrementally hashes a directory into partitions spread over multiple servers. But unlike prior work, Giga+ allows each server to re-partition its portion of the directory independently without global co-ordination, shared state or gossip about status traffic. As a directory grows, a server independently decides when to split its local partition by moving half the keys to a new partition on another server and independently decides when not to split because neither parallelism nor load-balancing would be improved. Each server maintains a history of splits its partitions have undergone. Clients avoid multi-hop resolution of path-name to server mapping by improving a stale cache of this mapping only as needed. Giga+ tolerates the use of inconsistent mapping state by clients without affecting the correctness of any operation; this state gets updated as needed, when any server detects that it has been incorrectly addressed. Client state is aggressively improved by such a server by transmitting its complete partition split history. Even the addition of new servers is supported with minimal migration of directory entries and delayed notification to clients.

We have implemented Giga+ file system directories as an interposition layer on any cluster-wide file system deployment. Unlike the current trend of using special purpose file systems with custom interfaces and semantics [27, 20], Giga+ directories use the traditional UNIX VFS interface and provide POSIX-like semantics to support unmodified applications. Giga+ has several file system specific optimizations including incremental growth such that a small directory is stored on a single server (an important requirement because 99.99% of the directories have less than 8,000 entries [14, 5]) and a compact bitmap-based encoding of the split histories and the partition index that minimizes the memory footprint and network messaging overhead of Giga+.

Our experimental evaluation demonstrates that Giga+ file system directories scale linearly on a cluster of 32 servers and delivers a throughput of more than 98,000 file creates per second – exceeding the performance of IBM GPFS [50], Lustre [38], and Ceph [59] file systems, and the HBase table store [26], and exceeding peta-scale scalability requirements [43]. Giga+ indexing also achieves effective load balancing with one to two orders of magnitude less re-partitioning than a similar system employing consistent hashing [54, 29]. We also study the cost-benefit tradeoffs of our design decisions including incremental growth, weak mapping consistency and pushing the out-of-core indexing to local file systems at the server.

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*. Today most cluster file systems provide scalable file I/O bandwidth by enabling parallel access using techniques such as data striping [25, 21, 20], object-based architectures [38, 59, 62] and distributed locking [56, 50, 59]. Few file systems scale metadata access by distributing metadata over multiple servers [44, 50, 16, 59]. And most file systems cannot scale access to a *large number of files*, much less support concurrently creating 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 a parallel application running on, for instance, LLNL's Bluegene/L cluster (with 64,000 nodes of two processors each) periodically writes application state into a file per process, all stored in one directory [59, 7]. Applications that do this per-process checkpointing already suffer due to the underlying file system's poor file creation rate (especially in one directory) [7]. This problem will be worse in a few years: in the Exascale-era clusters are expected to have up to billions of CPU cores [30].

Supercomputing checkpoint-restart, although important, would not be a sufficient reason for overhauling today's file systems. Yet there are diverse applications, such as gene sequencing, image processing [57], and phone logs for accounting and billing [58], that essentially generate an unbounded number of small files in one logical directory. Although these applications are treating the file system as a fast, lightweight "data store", replacing the underlying file system with a database is an oft-rejected option because it is infeasible to port existing code to use a new API (like SQL) and because traditional databases do not match the desired scalability and availability of file systems running on thousands of nodes [51, 6, 55, 3]. Authors of these applications repeatedly ask file system developers to improve support for small files.

Unfortunately traditional file system directories do not 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 [64, 42, 53] and even distributed file systems that run on the largest clusters, including HDFS [27, 12], GoogleFS [20], Panasas's PanFS [62] and PVFS [44], 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, uses a distributed metadata layer based on BigTable [11] for small file metadata [18].

To our knowledge, only four file systems, prior to ColossusFS, support distributed directories: IBM's GPFS, Oracle's Lustre, Microsoft's Farsite and UCSC's Ceph.

2.2 Related work

Giga+ draws inspiration from the scalability and concurrency limitations of several distributed indices and their implementations.

IBM GPFS and Farsite: IBM GPFS is a shared-disk file system that uses a distributed implementation of Fagin's extendible hashing for its directories [17, 50]. Fagin's extendible hashing dynamically resizes a hash-table, expanding only the overflow bucket by rehashing its keys using a specific family of hash functions [17]. It has a two-level hierarchy: buckets (to store the keys) 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-nodes. When the directory grows in size, GPFS allocates new blocks, moves the directory entries from the overflow block into the new block and updates the block pointers in the i-node.

GPFS uses its cache consistency and distributed locking mechanisms to enable concurrent access to the shared directory [50]. 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 tokens from the lock manager before updating the directory blocks stored on the shared disk storage. When releasing (or acquiring) locks, GPFS forces the directory block to be flushed to disk (or read back from disk) inducing high I/O overhead. Newer releases have modified the cache consistency protocol to send requests directly to the lock holder, instead of going through the shared disk subsystem [2, 1]. Still GPFS continues to synchronously write the directory's i-node (i.e., the mapping state) to provide strong consistency guarantees [2]. In contrast, GIGA+ allows the mapping state to be inconsistent and avoids the overhead of strongly consistent mapping.

Farsite, a file system for untrustworthy enterprise settings, proposed a distributed directory service using file identifier based metadata partitioning, and fine-grained lock hierarchies and lease mechanisms to allow multiple writers to have concurrent access to an object (e.g., shared directory) [16, 4]. Although it was not designed for data-intensive tasks, if used, its distributed locking semantics will limit the scalability under concurrent workloads.

Lustre and Ceph: Lustre's new clustered metadata service distributes a large directory over all the available metadata servers [36, 37]. It splits a directory only once when it exceeds a threshold size; effectiveness of this "split once and for all" scheme depends on the knowledge of the expected directory size distribution and access pattern. Ceph is another object-based cluster file system that uses dynamic sub-tree partitioning of the namespace and hashes individual directories when they gets too many accesses [59, 60]. Compared to Lustre and Ceph, GIGA+ splits a directory incrementally as a function of size, i.e. a small directory may be split on to less servers than a larger one. Furthermore, GIGA+ indexing facilitates server addition with minimal rehashing (there's no evidence of how Lustre and Ceph handle this).

Linear hashing and LH:* Linear hashing grows its table by splitting its hash buckets in a round-robin order using a pointer to the *next* bucket to split [33]. Its distributed variant, called LH* [34], stores buckets on servers and uses a central split co-ordinator (SC) that keeps a consistent split pointer to the next server to split a partition. An attractive property of LH* is that it does not update a client's mapping state synchronously after every new split; GIGA+ uses this property to allow weak consistency of the mapping state at the clients.

GIGA+ differs from LH* in several ways. First, to maintain consistency of the split pointer (at the SC), LH* splits only one bucket at a time [34, 35]; GIGA+ allows any server to split its bucket at any time without any SC. Second, because LH* chose to use only two state variables for index growth, LH* clients continuously incur some addressing errors even long after the index stops growing; GIGA+ chose to optimize for ways that minimize (and stop) the addressing errors at the cost of more state (but only tens of bytes in size). Optimizations to improve LH* concurrency through decentralized splitting incur unbounded number of addressing errors at the clients and adds significant complexity during split operations [35].

Consistent hashing: Consistent hashing divides the hash-space randomly into multiple ranges that are distributed over many nodes such that keys are mapped to nodes based on their proximity in the hash-space [54, 29]. Consistent hashing is efficient at managing membership changes, and hence is widely used by wide-area P2P systems that have high membership churn [13, 49, 41, 46, 24]. Cluster systems, even though they have much lower churn than P2P systems, have also used consistent hashing for data partitioning [15, 31]. However, the resulting data distribution has a high load variance, even after using "virtual servers" to map multiple hash-ranges to each node [15]. Furthermore, consistent hashing systems assume that data-sets need to be distributed over many nodes to begin with; they do not account for incremental growing data-sets that are distributed one node at a time - an important requirement for directories that GIGA+ provides.

Other work: DDS and Boxwood 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

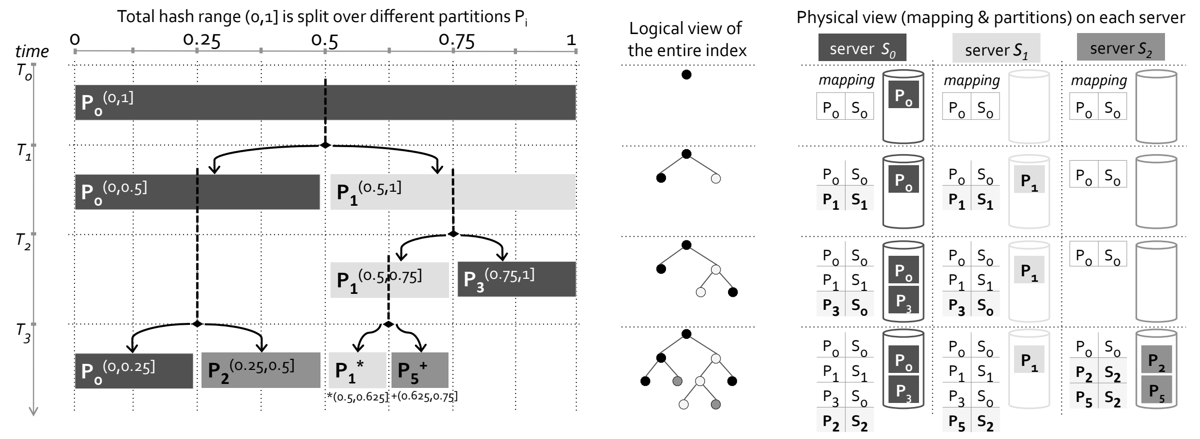


Figure 1. Concurrent and unsynchronized data partitioning in Giga+. *The hash-space* $(0, 1]$ *is divided into multiple partitions* (P_i) *that are distributed over many servers (different shades of gray). Each server has a local, partial view of the entire index and can independently split its partitions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

cluster abstractions, with an emphasis on indexing that uses inconsistency at the clients; a non-goal for DDS [23]. Boxwood proposed primitives to simplify storage system development, and used B-link trees for storage layouts[39].

3 Giga+ architecture

Giga+ is designed for incremental growth, i.e., an empty or a small directory is initially stored on one server and is partitioned over multiple servers as it grows in size. Such growth is appropriate for file system directories because most start small and remain small; studies of large file systems have found that 99.99% of the directories contain fewer than 8,000 files [14, 5]. 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. Giga+ does not cache directories at the clients, i.e., clients will send all directory operations to a server. Directory caching only for small directories ([62]) is an obvious extension that we did not yet implement because Giga+ is targeting concurrently shared directories with up to billions of files. Caching huge directories at each client is impractical; the directories are too large and the rate of change too high.

3.1 Unsynchronized data partitioning

Giga+ uses hash-based indexing to incrementally divide each directory into multiple partitions that are distributed over multiple servers. Each filename (directory entry) is hashed to its respective partition (our implementation uses the cryptographic MD5 hash function) allowing Giga+ to rely on an important 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 [47].

Figure 1 shows how Giga+ indexing starts small and grows incrementally in a decentralized and concurrent manner. 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. [1] GIGA+ uses this 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* (§3.3 addresses growing the number of servers). The color of each partition indicates the (color of the) server where it resides. Initially, at time T_0, the directory is small and stored on a single partition $P_o^{(0,1]}$ on server S_0. As the directory grows and the partition size exceeds a threshold, provided this server knows of another available server to increase parallelism or balance load, S_0 splits into half by moving the greater half of its hash-range to a new partition $P_1^{(0.5,1]}$ on S_1. As the directory grows over time, servers continue to split partitions onto more servers until all servers have about the same fraction of the hash-space range to manage. GIGA+ computes partition identifiers using classic radix-based techniques. [2]

The key idea in GIGA+ is that each server splits independently in a completely decentralized manner. Data partitioning happens without any serialization or synchronization across the system. Servers make local decisions about splitting a partition which enables highly parallel scale-out growth, and growing without shared-state or co-ordinator nodes does not need to use any system-wide synchronization mechanisms. This "shared-nothing" partitioning approach yields high concurrency and differentiates GIGA+ from prior distributed indexing work.

The side-effect of such uncoordinated growth is that GIGA+ does not have a global view of the partition-to-server mapping on any servers; 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 about the identity of another server that knows more about each "child" partition that was created by splitting a particular partition on that server. In Figure 1, at time T_3, server S_1 manages partition P_1, and knows about its children, partitions P_3 and P_5. But servers are often unaware about partition splits that happened 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 maintains only the split history of its partitions. The global partition mapping, or the complete history of the directory growth, is the transitive closure over the local mapping on each server and is not maintained synchronously by any server. GIGA+ can enumerate all the partitions of a directory by traversing the split history tables starting with the zeroth partition P_0. However, a GIGA+ client accessing a large directory that is not changing may construct the global mapping in its map cache.

3.2 Tolerating inconsistent mapping at clients

Clients find the appropriate partition by probing servers, possibly incorrectly, based on their map cache which starts with only the root partition identified in the directory's parent directory entry. By allowing unsynchronized growth at the servers, GIGA+ clients may have stale and inconsistent split histories, particularly when the directory is mutating rapidly. A client may address a request to an "incorrect" server, i.e. a server that is no longer responsible for the desired filename (in the partitions it holds) because of a previous partition split that the client has not learned about. This "incorrect" server detects the addressing error by recomputing the partition identifier (by hashing the filename) for each operation, and on a mismatch sending a split history update to the client. The client updates its merged version of the global split history, and retries the original request.

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

[2] 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 (from the available history). 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 filenames, from the larger half of its hash-range, to a new partition with identifier $i + 2^r$. After this 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 (with identifier $i = 5$). And now both partitions are at an increased tree depth of $r = 3$.

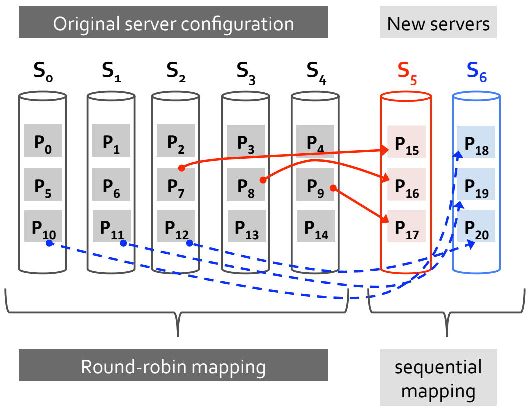


Figure 2. Server additions in Giga+. *To minimize the amount of data migrated, indicated by the arrows that show splits,* GIGA+ *changes the partition to server mapping from round-robin on the original server set to sequential on the newly added servers.*

The drawback of an inconsistent map cache 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 possible that a client with an empty map cache 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. Each update sends the split histories of all partitions that reside on a server, far more than the minimal information, which helps client map caches to catch up quickly. Moreover, after a directory stops splitting partitions, clients will soon have the complete split histories and will not incur any addressing errors. This eventual correctness differentiates GIGA+ from prior distributed indices, particularly LH* that chose to maintain less client state while incurring addressing errors even when the system is not mutating [35].

3.3 Reconfiguration and recovery

This section describes how GIGA+ handles addition of new servers, and failure of existing clients and servers.

Handling server addition –

When new servers are added to an existing configuration, the system should re-balance itself by migrating the minimal number of directory entries to the new servers. Using the round-robin partition to server mapping, shown in Figure 1, a naive server addition scheme would require re-mapping all the directory entries whenever a new server is added.

GIGA+ avoids re-mapping all directory entries by adding rules to the partition-to-server mapping that differentiates additional servers. For all new servers added to the original server set GIGA+ does not use the round-robin partition-to-server striping and instead maps all future partitions on to the new servers in a "sequential manner". The benefit is that a round-robin mapping enables faster parallelism, when a directory is small and growing, but a sequential mapping does not disturb previously mapped partitions more than is mandatory for load balancing.

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 (P_i `mod` N, where N is number of servers). After the addition of two servers, all new partitions (P_{14} onwards) will be mapped on to servers using a new mapping rule: P_i `div` M, where M is the number of partitions per server (i.e. 3 partitions/server).

Data is migrated when the new servers notify the existing servers about their arrival. In GIGA+ even the number of servers can be stale at servers and clients. New servers seeking work trigger old servers and then clients to discover mapping changes. Each new server contacts the server that holds the zeroth partition (P_0) to declare its arrival; this zeroth server assigns each server a unique identifier (which is the position of the server in the server ordering). Newly added servers then use their identifier to calculate the partition identifiers (P_{new}) that they will be responsible to hold and contacts servers that hold the parent partitions of P_{new} using the mechanisms described in prior sections. If large enough to warrant splitting, the parent partition will be split by the existing server in the normal way because the existing server becomes aware of new servers onto which to load balance.

Handling failures –

We describe how to tolerate two kinds of failures, GIGA+ operation failures and general (storage) system failures, at the clients and the servers.

Client failures: If a client reboots, it can only lose operations in flight and the state (bitmap and server list) with respect to GIGA+ directories. GIGA+ uses existing techniques for lost operations: sequence numbers are used to distinguish new messages from retransmitted messages and a server reply cache ensures that non-idempotent commands, like create, can return the original command's response when a reply packet is lost. Lost mapping state is relatively easy to handle since all client state for a directory can be reconstructed by contacting the zeroth partition named in a parent directory entry, re-fetching its server list and rebuilding the bitmaps through incorrect addressing of server partitions during normal operations.

Server failures: GIGA+ stores data persistently in the server's underlying (local) file system and relies on RAID or replication in the underlying storage for durability. However, GIGA+ needs to explicitly handle cases when a server process fails. This affects both the server-side operations and handling client requests. To handle server process failure during a split operation, GIGA+ treats the split as a multi-stage atomic operation; the split-initiator process writes all the operations in an on-disk write-ahead log which can be replayed either by the split-initiator after it reboots (from a crash) or by a backup process that replaces the primary process.

To effectively service client requests after server failures, GIGA+ server processes rely on chained-declustering server replication [28] for high availability. This technique replicates partitions so that there are two copies of every partition and these two are stored on adjacent servers in the server list 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 other servers [28]. Some systems have used chained declustering to spread "hot"partition reads across its two servers [32, 56], but GIGA+'s hashing implicitly ensures uniform load distribution across all servers so this is not needed.

On normal lookups and mutations, clients send their requests to the server that holds the primary copy. A non-failed primary handles lookups directly and replicates mutations 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 rather 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 [8, 61]. While a node is down or being reconstructed, its secondary executes all of its operations, 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 GoogleFS which keeps 3 to 6 copies of data [20].

3.4 Bitmap-based encoding

Giga+ implementation uses a bitmap-based encoding to represent a partial or complete split history of a directory. In this bitmap, a value '1' at the i^{th} bit indicates that P_i has been created, and value '0' indicates it is absent and has not been created (yet); as partitions split, the bit value at the position corresponding to the new partition's identifier is set to '1' at the server that does the split. For each directory, Giga+ servers create and maintain a bitmap of the partial split history that they know; and since servers splits independently, they will have different bitmaps that encode their respective split histories. This compact representation reduces the memory footprint and message sizes for client updates. In a simple scheme, the bitmap size is proportional to the number of partitions in the directory, but this can be highly compressed using run-length encoding. Giga+ servers send these bitmaps to update the clients' stale state using a quick bit-wise `OR` of bitmaps.

4 Experimental Evaluation

Our evaluation answers two questions: (1) How does Giga+ scale compared to other existing systems? and (2) What are the cost and benefits of Giga+'s design choices including incremental growth, weak mapping consistency and using a server's local file system for out-of-core indexing (when partitions become very large)?

Prototype: Our Giga+ indexing prototype is written in C and runs on the Linux platform using the FUSE user-level file system [19]. Both the client and server processes run in user-space, and communicate using SUN RPC [52]. The prototype has three layers: unmodified applications running on clients, the Giga+ indexing modules (on clients and servers) and a backend persistent store at the server. Applications interact with the Giga+ client using the VFS API (e.g., `open()`, `creat()` and `close()` syscalls). The FUSE kernel module intercepts and redirects these VFS calls to the client-side Giga+ indexing module. This module determines the partition and server responsible, and forwards each operation. Clients find a server list associated with each directory maintained as a per-directory attribute because the directory entry in the parent directory names the first server.

On the server, for every request, the server-side indexing module hashes the filename and uses its split history bitmap to identify the partition responsible for the request, and checks if it is the correct server (and if not, replies to the client with `retry` code and its bitmap). Servers map logical Giga+ partitions to physical representations specific to the backend file system: each Giga+ partition is stored as a file system directory. Our implementation allows administrators to specify any VFS file system as the back-end store; we experiment with Linux `tmpfs`, ext3 and ReiserFS but cluster file systems have obvious durability advantages.

Each partition split is handled by the server-side Giga+ indexing module by locking the particular directory partition and scanning all its files using `readdir()`. After the server-side indexing module decides which file names should be moved to the new partition, these files are transferred to the new server, and upon success, the server will `unlink()` the original copies of transferred files

	File System	**File creates/second in one directory**
GIGA+	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. Single-node file system directory performance.

from the directory partition being split. This is much more effective if the underlying file system can `rename()` without copying data, as in PVFS [44].

Platform/Workloads: 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 on one 7200rpm SATA disk. We assigned 32 nodes as servers and the remaining as load generating clients.

We used the synthetic `mdtest` benchmark [40] (used by parallel file system vendors and users) to generate three 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 receives 400,000 file creates, 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 a directory. And finally, we use a mixed workload where clients issue create and lookup requests in a pre-configured ratio.

4.1 Scale and performance

We compare the performance of GIGA+ directories with various local file systems (ext3 and Reiser), PVFS cluster file system [44], Hadoop's HDFS [27] and a (commercial) NFSv3 filer. Except for the NFS filer, all distributed file systems used ext3 and were created new for these experiments. Although none of these file systems distribute directories over multiple servers, this test allows us to compare single server "baseline" performance. Furthermore the networked file systems are real production file systems with complex code paths to handle complex error and failure handling. We have configured these file systems to be functionally equivalent to the GIGA+ prototype, e.g., we disabled the write-ahead log and replication in HDFS, for a fair comparison.

Table 1 shows a comparison of all the above file systems. GIGA+ is shown using FUSE and using a library module compiled into the load generating process. The former is a valid comparison to the network file systems where each VFS file `creat()` results in three calls – `getattr()` to check if a file exists, the actual `creat()` and another `getattr()` – each of which is an RPC sent to the server. This also results from the aggressive `getattr()` checks performed by FUSE [19], something that may be avoided using an in-kernel client module. The library approach allows GIGA+ to linked into the applications that want to use custom object creation calls (such as `giga_creat()`);

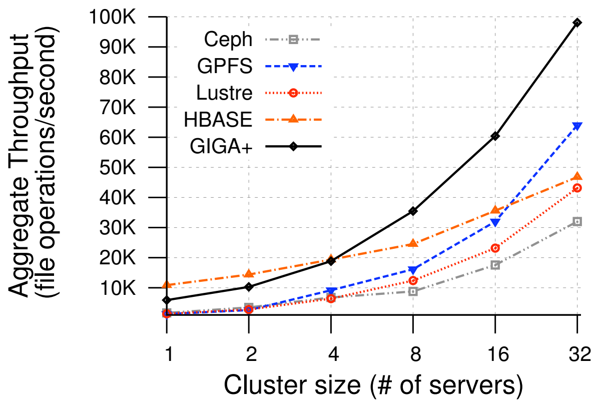


Figure 3. Scalability of Giga+ FS directories. Giga+ *directories deliver a peak throughput of roughly 98,000 file creates per second. The behavior of underlying local file systems limits* Giga+*'s ability to match the ideal linear scalability*

we test the library library version only to gauge Giga+ efficiency.

Among the local file systems, with local threads generating the creates, both ReiserFS and Linux ext3 deliver high directory insert rates. [3] All three were configured with `-noatime` and `-nodiratime` option, while ext3 used write-back journaling and the `dir_index` option to enable hashed-tree indexing, and ReiserFS was configured with `-notail`, a small-file optimization that packs the data inside an i-node for high performance [45]. Giga+ with its load 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+ while remote VFS's use of three RPCs is a large penalty. Giga+ directories using the VFS/FUSE interface also outperforms all three networked file systems.

Using this baseline, we evaluate the scalability of Giga+ directories (layered on ReiserFS) using the *concurrent create* benchmark. Figure 3 plots aggregate operation throughput averaged over the complete benchmark run as a function of the number of servers (on a log-scale X-axis). Giga+ over ReiserFS scales linearly up to the size of our 32-server configuration, and can sustain 98,000 file creates per second - to the best of our knowledge, this exceeds the most rigorous scalability demands in high-performance computing systems [43].

We compare Giga+ with the scalability of IBM GPFS, Lustre, Ceph and HBase. HBase is used to emulate Google's approach in their Colossus file system of storing small file (metadata) in BigTable instead of the file system [18]. We setup HBase on a 32-node HDFS configuration with single copy (no replication) and modified two parameters: we disabled blocking while the HBase servers are doing compactions and disabled write-ahead logging for inserts (a common practice to speed up inserting data in HBase). This modifications allowed HBase to deliver peak performance for the single server configuration. Still HBase scales poorly when running on multiple servers. For the three file systems with distributed directory implementations, we are presenting numbers from experiments performed on different clusters [59, 1]. The experimentation of two production file systems, GPFS and Lustre, was performed recently on a high-end cluster where the physical storage was a DDN9550 disk array system and metadata servers were each attached to a 16-disk JBOD of 15K RPM SAS drives, and both file systems were tuned by expert users [1]. We observe that GPFS outperforms Lustre and scales well up to 32 servers. This benefit comes from the fine-

[3]We used XFS too, but it was extremely slow during this create-intensive workload.

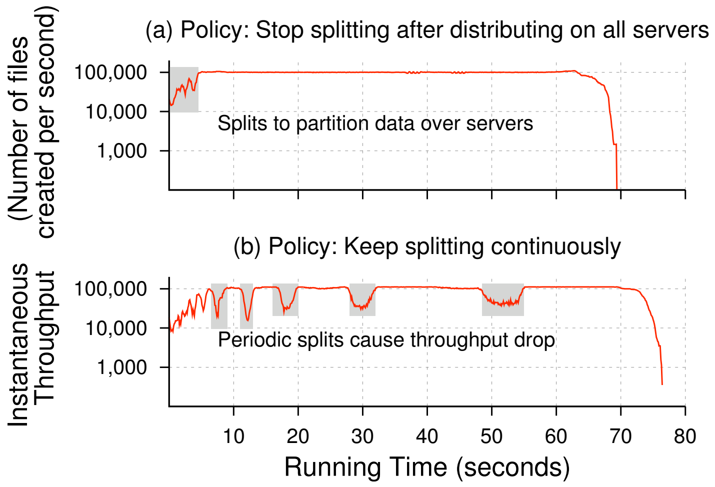


Figure 4. Effect of splitting heuristics. GIGA+ *shows that splitting to create at most one partition on each server can deliver scalable performance. Continuous splitting, used in classic database indices, is detrimental in a distributed scenario.*

grained directory locking optimizations included in the newer versions of GPFS (described earlier in §2). Despite using a cluster with slower commodity disks and networking interconnect, GIGA+ outperforms all three file systems and shows promising scalability.

GIGA+ also demonstrated scalable performance for *concurrent lookup* workload delivering a throughput of more than 600,000 file lookups per second (not shown in any graph). This 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.

4.2 Incremental scaling properties

In this section, we analyze the scaling behavior of the GIGA+ index and not the on-disk directory service (explored later in §4.5). To eliminate the performance interference from the disk subsystem, we use Linux's in-memory file system, `tmpfs`, for each partition of a large directory. In this experiment, we use the *concurrent create* benchmark to create large number of files in an empty directory and measure the aggregate throughput (number of files created per second) continuously throughout the benchmark. We ask two questions about scale-out heuristics: (1) when should we stop splitting? and (2) what is the effect of splitting during incremental scale-out growth?

In the context of the first question, classic indices, such as extendible and linear hashing [17, 33], used for out-of-core indexing in single-node databases keep splitting the index forever whenever a partition overflows, probably because the partitions correspond to disk allocation blocks in their early designs [22]. However, the splits in GIGA+ are only required to parallelize access to a directory by distributing the directory load on all servers; thus GIGA+ can stop splitting after each server has at least one partition and the local file system can handle physical allocation and out-of-core cache management.

Figure 4 compares the effect of different split policies on the system throughput (using a log-scale Y-axis) for the entire time while a large directory is created over 16 servers. Graph (a) shows the GIGA+ split policy and graph (b) shows the continuous split policy used by classic database indices where a split happens whenever a partition has more than 8,000 files. Graph

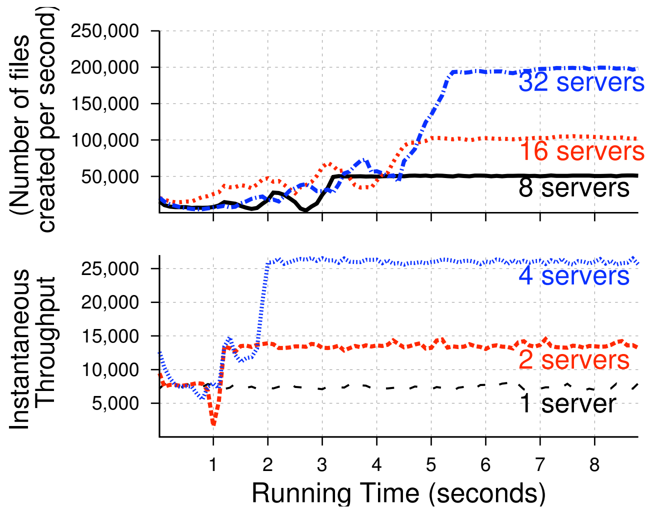


Figure 5. Incremental scale-out growth. GIGA+ *achieves linear scalability after distributing partitions to all available servers. During the scale-out, periodic drops in aggregate create rate correspond to concurrent splitting on all servers.*

(b) shows that 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. Continual splitting is detrimental for a distributed system, like GIGA+, where each split consumes resources on two machines and the network, provided that the underlying file system can handle very large partitions itself (which is not trivial [63]).

Now that we have shown that GIGA+ should stop splitting, the natural question is: when should the system stop splitting? To answer this we evaluate the effect of incremental scale-out, where a directory starts small and the grows big, on the system throughput.

Figure 5 shows the first 8 seconds of the *concurrent creates* workload. The primary result in this figure is the near linear create rate seen after the first few seconds, which is also near linear in the number of servers. 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 a 2-server configuration, the directory starts on a single server and splits when it has more than 8,000 entries in the partition. When the server are busy splitting the partition, at 0.8 seconds, the throughput drops to half for a very 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. There are two reasons for this behavior: partitions overflow at about the same time and servers split without any co-ordination. Just before the 3-second mark, the directory has four partitions on four servers. Because of the uniform distribution of the hash function, all four partitions get filled up to reach the per-partition split threshold at about the same time. And because servers split without co-ordination, all four servers are busy splitting their partitions to four more servers. After the split spreads the directory partitions on eight servers, the aggregate throughput achieves the desired linear scale. Thus figure 5 shows that GIGA+ achieves linear scalability after the directory partitions are distributed over all servers.

Lesson #1: To avoid the high overhead of continuous splitting in a distributed scenario,

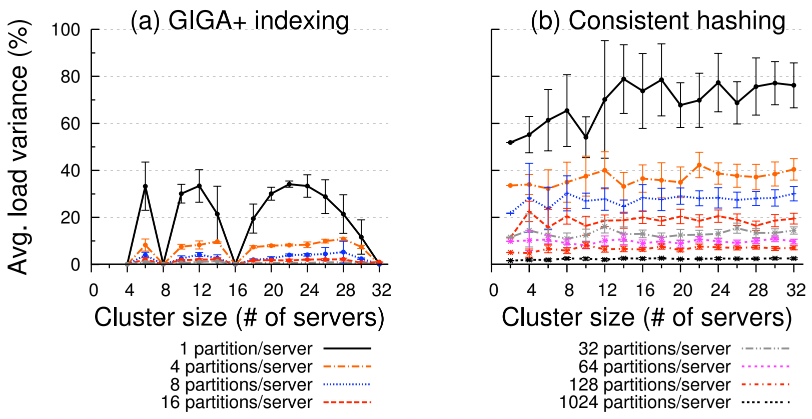


Figure 6. Load-balancing in Giga+. *These graphs show the quality of load balancing measured as the mean load variance across the entire cluster (with 95% CI error bars). Like virtual servers in consistent hashing,* GIGA+ *also benefits from using multiple hash partitions per server.* GIGA+ *needs two orders of magnitude fewer partitions per server, and still achieves a more uniform load distribution than consistent hashing.*

GIGA+ stops splitting after all servers have at least one directory partition, and lets a server's local file system (or GIGA+ server process) deal with out-of-core management of very large partitions.

4.3 Load Balancing Efficiency

Figure 5 showed the case where the number of servers is a power-of-two; this is a special case because it naturally load-balances: the only partition on each server is responsible for the hash-range of same size: 2^N-th part of the total hash-range $(0, 1]$. However, in violation of Lesson #1, we discovered that splitting to create more than one partition per server has an important benefit: it significantly improves the load distribution when the number of servers is not a power-of-two. In Figure 6(a)), load balancing efficiency is measured as the skew (load variance) in the load handled by each server compared to the average load for all numbers of servers from 1 to 32. 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% CI error bars) over all the servers.

Figure 6(a) shows the expected periodic pattern where the system is uniformly load-balanced when the number of servers is a power-of-two. This is because GIGA+ follows a binary splitting policy that divides a partition into two, and maps them on to different servers. When the number of servers is not a power-of-two, load imbalance happens when different servers are each holding a single partition with different sizes of the hash-range (some are twice the size of others).

Figure 6(a) also shows the load imbalance when GIGA+ continues splitting past one partition per server to 4, 8, 16 and 32 partitions per server. With more partitions, each will represent a smaller portion of the hash-range and the sum of these many smaller partitions will be less variable than a single large partition. In fact, having multiple partitions per server is analogous to the concept of virtual nodes in consistent hashing where each node is assigned more than one hash-range. Figure 6(b) shows the load balancing effectiveness of consistent hashing using virtual servers as used in systems such as Amazon's Dynamo [15]. Because consistent hashing partitions the hash-space in random ranges, it has a higher load variance when using a single hash-range partition

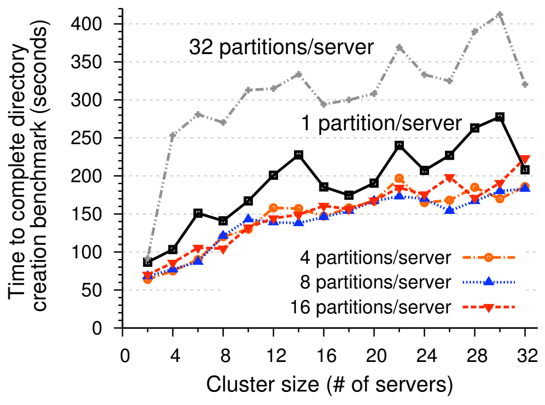


Figure 7. Cost of splitting partitions. *Using 4, 8, or 16 partitions per server improves the performance of* GIGA+ *directories layered on ext3 relative to 1 partition per server (better load-balancing) or 32 partitions per server (when the cost of more splitting dominates the benefit of load-balancing).*

per server [9] – almost two times worse than GIGA+ in the worst case for both. Increasing the number of hash-range partitions significantly improves the 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 and splitting that lasts longer in the directory building process. Assuming consistent hashing and GIGA+ scale equally well, this overhead makes consistent hashing less attractive for scenarios that need incremental growth.

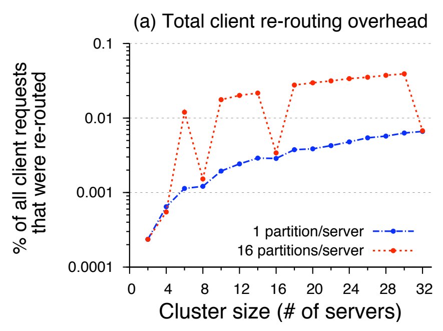
Since having multiple partitions per server significantly improves load-balancing, do we always continue to split to create a large number of partitions? Figure 7 shows the *concurrent create* benchmark time for GIGA+ as a function of the number of servers for 1, 4, 8, 16 and 32 partitions per server. 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 6(a) that the load-balancing efficiency from using 32 partitions per server is only 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 overall performance is higher with only a few partitions per server.

4.4 Cost of Weak Mapping Consistency

Figure 8(a) measures the overhead incurred by allowing client bitmaps to become stale. We measure the percentage of all client requests that were re-routed when running the *concurrent create* benchmark on different cluster sizes. The graph on the left shows that, in absolute terms, fewer than 0.05% of the requests are addressed incorrectly. The number of addressing errors increases proportionally with the number of partitions per server because periodic splitting keeps creating new partitions. In the case when the number of servers is a power-of-two, as described in the previous section, after each server has one partition, subsequent splits are created locally on the same server causing no additional addressing errors.

We take the worst case from Figure 8(a), 30 servers with 16 partitions per server, to study when these addressing errors occur. Figure 8(b) shows the number of errors encountered by each request generated by one of the 260 client threads that creates 50,000 files in a directory striped on



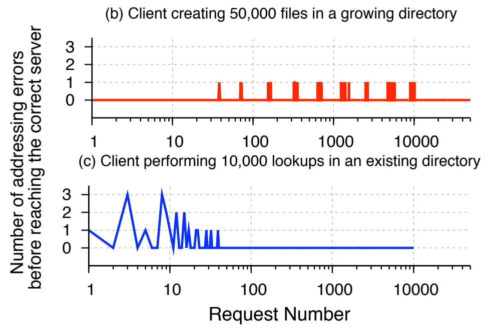


Figure 8. Cost of using inconsistent mapping at the clients. *Using weak consistency for mapping state at the clients has a very negligible overhead on client performance (a). And this overhead – extra message re-addressing hops – occurs for initial requests until the client learns about all the servers (b and c).*

30 servers. We observe that all addressing errors are clustered around the time when new partitions are created, and these errors stop after the number of partitions has stopped growing and the client has acquired an accurate bitmap. And in Figure 8(c), a fresh client that issues 10,000 lookups in the recently created large directory learnt about the entire bitmap in the first 40 requests (after which it does not commit any addressing errors).

Lesson #3: GIGA+ clients incur neglible overhead (in terms of incorrect addressing errors) due to inconsistent bitmaps, and no overhead shortly after the servers stop splitting to create new partitions.

Because the GIGA+ implementation chose to push the out-of-core indexing to the local file systems on each server, we study how the design and implementation of ext3 and ReiserFS affects various GIGA+ indexing primitives, particularly partition splits. GIGA+ maps logical partitions to directories on the server's local file system.

Although different local file system implementations can be expected to have different performance, especially for emerging workloads like ours, we were surprised to see how much difference occurs when GIGA+ is layered over these local file systems. Figure 9 shows create rates over time for four different configurations: Ext3 and ReiserFS, under GIGA+ with 16 servers, each with 1 partition per server and 16 partitions per server. Linux ext3 directories use h-trees [10] (significant improvement over linked lists used in ext2) and ReiserFS uses balanced B-trees [45]. We observed two interesting phenomenon: First, the benchmark running time varies from about 100 seconds to over 600 seconds. And second, which backend file system yields the faster performance is different when there are more partitions on each server.

Comparing the single partition case of GIGA+ over ReiserFS and over ext3 (left column in Figure 9), we observe that the benchmark completion time increases from about 100 seconds using ReiserFS to nearly 170 seconds using ext3; (for comparison using `tmpfs` as the backend, the benchmark completes in 70 seconds.) The backend local file system and real disks have a large impact on file create performance. Ext3 performance periodically drops from 75,000 creates per second to as low as 7,500 file creates per second. As the directory grows, the journal also grows in size which triggers ext3's `kjournald` daemon to flush a part of the journal to disk. Because the directory is growing on all the servers at roughly the same rate, multiple servers flush the journal to disk at about the same time. 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 the machine, and this eliminated most of the throughput variability observed

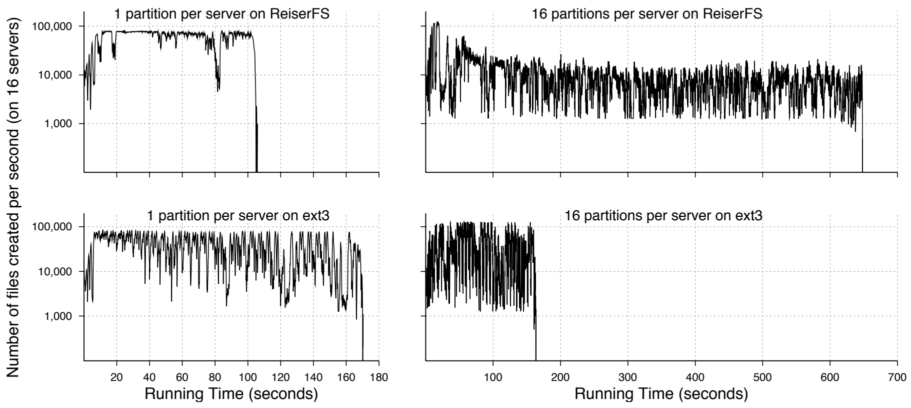


Figure 9. Effect of underlying file systems. *This graph shows the create benchmark behavior when the* Giga+ *directory service is distributed on 16 servers with two local file systems, Linux ext3 at the bottom and ReiserFS on the top. For each file system, we show two different numbers of partitions per server, 1 and 16.*

in ext3. But for ReiserFS, placing the journal on a different disk had little benefit.

4.5 Interaction with backend file systems

The second phenomenon we observe in the right column of Figure 9 is that for 16 partitions per server, ext3 under Giga+ completes the create benchmark more than four times faster that ReiserFS under Giga+. We suspect that this results from the on-disk directory representation. ReiserFS uses a balanced B-tree structure for *all objects* in the file system, which re-balances as the file system grows (changes) over time [45]. When partitions are split more often, as in case of 16 partitions per server, there are more changes in backend file system structure, triggering more re-balancing in ReiserFS and slowing the instantaneous create rate. Nonetheless Figure 3 showed that Giga+ achieved more than 98,000 creates per second using ReiserFS with a single partition.

Lesson #4: Design decisions of the backend file system have subtle but important side-effects on the performance of a distributed directory service, and perhaps backend file systems need to be customized for this workload.

5 Conclusion

In this paper we addressed the emerging need for file system directories that store massive number of files and sustain hundreds of thousands of concurrent mutations. The central principle of Giga+ is to use asynchrony and inconsistency to push the limits of scalability and concurrency of file system directories. We used these principles to build a POSIX-compliant directory implementation that scales linearly and outperforms existing distributed directory techniques on a 32-node configuration. Our analysis also shows that Giga+ achieves better load balancing than consistent hashing and incurs a neglible overhead from keeping inconsistent lookup state at the clients.

Our future plans include understanding the behavior at much larger scales (billions of files and thousands of servers), exploring how Giga+ can be layered on cluster file systems (such as PVFS) in an efficient manner, and extending the POSIX directory API (particularly `readdir()`) to handle large mutating directories.

References

[1] Private Communication with Brent Gorda, LLNL (now at Whamcloud Inc.).

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

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

[4] 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 *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*.

[5] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. A Five-Year Study of File-System Metadata. In *FAST '07 Conference on File and Storage Technologies*.

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

[7] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A Checkpoint Filesystem for Parallel Applications. In *ACM/IEEE Conference on Supercomputing (SC '09)*.

[8] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*.

[9] Byers, J., Considine, J., and Mitzenmacher, M. Simple Load Balancing for Distributed Hash Tables. In *Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*.

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

[11] 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 *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*.

[12] Cloudera. The Small Files Problem (Feb 2009). http://www.cloudera.com/blog/2009/02/02/the-small-files-problem/.

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

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

[15] 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 *21st ACM Symposium on Operating Systems Principles (SOSP '07)*.

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

[17] 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).

[18] Fikes, A. Storage Architecture and Challenges (Jun 2010). Talk at the Google Faculty Summit 2010.

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

[20] Ghemawat, S., Gobioff, H., and Lueng, S.-T. Google File System. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*.

[21] 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 *8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*.

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

[23] Gribble, S., Brewer, E., Hellerstein, J., and Culler, D. Scalable Distributed Data Structures for Internet Service Construction. In *4th Symposium on Operating Systems Design and Implementation (OSDI '00)*.

[24] Haeberlen, A., Mislove, A., and Druschel, P. Glacier: highly durable, decentralized storage despite massive correlated failures. In *2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*.

[25] Hartman, J. H., and Ousterhout, J. K. Serverless Network Filesystem. In *14th ACM Symposium on Operating Systems Principles (SOSP '93)*.

[26] HBase. The Hadoop Database. http://hadoop.apache.org/hbase/.

[27] HDFS. The Hadoop Distributed File System: Architecture and Design.

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

[29] 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 *ACM Symposium on Theory of Computing (STOC '97)*.

[30] Kogge, P. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. DARPA IPTO Report, Released on Sep 28, 2008, Sept. 2008.

[31] Lakshman, A., and Malik, P. Cassandra - A Decentralized Structured Storage System. In *Workshop on Large-Scale Distribued Systems and Middleware (LADIS '09)*.

[32] Lee, E. K., and Thekkath, C. A. Petal: Distributed virtual disks. In *7th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*.

[33] Litwin, W. Linear Hashing: A New Tool for File and Table Addressing. In *6th International Conference on Very Large Data Bases (VLDB '80)*.

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

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

[36] Lustre. Architecture - Clustered Metadata (2010). http://wiki.lustre.org/index.php/Architecture_-_Clustered_Metadata.

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

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

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

[40] mdtest. mdtest: HPC benchmark for metadata performance. http://sourceforge.net/projects/mdtest/.

[41] Muthitacharoen, A., Morris, R., Gil, T., and Chen, B. Ivy: A Read/Write Peer-to-peer File System. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*.

[42] NetApp-Community-Forum. Millions of files in a single directory (Feb 2010). http://communities.netapp.com/thread/7190?tstart=0.

[43] Newman, H. HPCS Mission Partner File I/O Scenarios. Revision 3, Released on Nov 17, 2008, Nov. 2008.

[44] PVFS2. Parallel Virtual File System, Version 2. http://www.pvfs2.org.

[45] Reiser, H. ReiserFS. http://www.namesys.com/, 2004.

[46] Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., and Kubiatowicz, J. Pond: the Oceanstore Prototype. In *FAST '03 Conference on File and Storage Technologies*.

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

[48] 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 2006.

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

[50] Schmuck, F., and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *USENIX FAST 2002*.

[51] Seltzer, M. Beyond Relational Databases. In *Communications of the ACM (Vol. 51, No. 7)* (July 2008).

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

[53] stackoverflow. Millions of small graphics files and how to overcome slow file system access on XP (Oct 2009). http://stackoverflow.com/questions/1638219/.

[54] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM Conference (SIGCOMM '01)*.

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

[56] Thekkath, C. A., Mann, T., and Lee, E. K. Frangipani: A Scalable Distributed File System. In *16th ACM Symposium on Operating Systems Principles (SOSP '97)*.

[57] Tweed, D. One usage of up to a million files/directory (Nov 2008). http://leaf.dragonflybsd.org/mailarchive/kernel/2008-11/msg00070.html.

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

[59] 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 *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*.

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

[61] Welch, B. Integrated System Models for Reliable Petascale Storage Systems. In *Petascale Data Storage Workshop (at Supercomputing '07)*.

[62] 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 *FAST '08 Conference on File and Storage Technologies*.

[63] Wheeler, R. One Billion Files: Scalability Limits in Linux File Systems. In *LinuxCon* (2010).

[64] zfs discuss. Million files in a single directory (Oct 2009). http://mail.opensolaris.org/pipermail/zfs-discuss/2009-October/032540.html.