GIGA+ : Scalable Directories for Shared File Systems

Swapnil V. Patil
Carnegie Mellon University
svp@cs.cmu.edu

Garth A. Gibson
Carnegie Mellon University
garth@cs.cmu.edu

Sam Lang
Argonne National Lab
slang@mcs.anl.gov

Milo Polte
Carnegie Mellon University
milop@cs.cmu.edu

1. INTRODUCTION

There is an increasing use of high-performance computing (HPC) clusters with thousands of compute nodes that, with the advent of multi-core CPUs, will impose a significant challenge for storage systems: The ability to scale to handle I/O generated by applications executing in parallel in tens of thousands of threads. One such challenge is building scalable directories for cluster storage – i.e., directories that can store billions to trillions of entries and handle hundreds of thousands of operations per second.

Today some parallel applications use the file-system like a fast, lightweight "database", resulting in directories containing tens of millions of files [9]. For instance, phone companies may wish to monitor and record information about calls by their subscribers for billing. One such monitoring application creates a file that logs the start and end time of every call [1]. To support large call volumes, telecom companies are deploying "new" infrastructure to support more than hundred thousand calls per second [13]. Typically telecom infrastructure is over-provisioned, so even a system that is running at one-third utilization can easily create more than 30,000 files per second. Another example comes from per-process checkpointing, where every process running on a large HPC cluster generates a periodic checkpoint file. A 25000-node cluster, where each node has 16-32 cores, may nearly simultaneously create more than half a million files, one per core, to contain that core's periodic checkpoint. Since millions of files occur in directories today and we expect larger directories in the future, we decided to push the envelope for our design by setting a goal to scale to billions of files in a directory and to make extensions to trillions of files as simple as possible.

Traditional local file systems with UNIX semantics organize directories in a flat, sequential data-structure. This results in an $O(n)$ lookup cost – for large directories this can run in the order of few minutes. To improve the lookup speeds, various file-systems have used faster indexing structures like B-

trees for $O(\log n)$ lookups and hash-tables for $O(1)$ lookups. Local file systems that use fast indexing structures, such as XFS's use of B-trees [11] or Linux's Ext2/Ext3 hash tables [12], don't scale beyond a single machine, which is insufficient for parallel HPC applications.

Several distributed file systems have been proposed for greater scalability - GPFS uses extendible hashing [10] and Boxwood builds a shared storage abstraction using B-link trees [6]. GPFS uses lock tokens that are used for parallel file data access. For synchronized file metadata access, GPFS elects "metanodes" that control metadata writes. In Boxwood, multiple threads use global locks for synchronizing access to shared data. However, using a global lock service provides significant opportunities for bottlenecks and shared state synchronization overhead, especially at highly concurrent access.

Based on our target environment and application needs, we propose the following design goals for a scalable directory service:

- **Maintain UNIX file-system semantics:** To ease adoption by existing applications, we maintain UNIX file-system semantics (like no duplicates, no range queries and unordered readdir scans) instead of proposing a radically different interface like scientific databases or custom file-system API.

- **High throughput and scalability:** The overall system performance should scale with increasing cluster size. Our goal is to store billions to trillions of entries in a directory and handle hundreds of thousands of operations per second.

- **Incremental growth:** Measurement studies have shown that while most files are stored in a few large directories, vast number of directories are small [7]. So, small directories should not incur storage or performance overhead just because large directories exist.

- **Minimal bottlenecks and shared state synchronization:** Many distributed file systems use centralized servers for various functions like locking, lookups and ease of manageability. To achieve high throughput, we seek to eliminate as much of the bottlenecks and synchronization overhead during highly concurrent access as possible.

- **Burst performance:** Large directories start small and sometimes grow very quickly. These "storms" while building a large directory, i.e., when all clients simultaneously start inserting files into a newly created directory at maximum rate, need to be handled efficiently and not lead to system instability.

The following section describes a design we're investigating to meet these goals.

2. DESIGN OF GIGA+

We are engaged in a research project to map out the trade-offs in a distributed directory service, called GIGA+, that meets the above goals. GIGA+ partitions each directory over a scalable number of servers – i.e., huge directories are striped over large number of servers. Directories are partitioned in a manner that effectively load-balances all servers; GIGA+ achieves this uniform distribution by hashing the name of the directory entries.

2.1 Client Lookup in GIGA+: P2SMap and Incremental Growth

For scalable performance, a good solution needs a mapping technique that allows the clients to lookup the correct partition and the server it maps to without an intermediate lookup service or a consistent client cache. Using an intermediate lookup server can introduce a bottleneck in the system. Directories with billions of entries can grow to 10-100 GB in size; as a result, caching a directory at the client is hard. In addition under high insert rates, a cache consistency protocol can incur a significant overhead in terms of number of messages and cache synchronization latency. Even just caching the mapping i.e., non-leaf nodes of the B-tree, can experience large rates of change in a high throughput system. Thus it is important in GIGA+ that clients be able to derive the correct partition and server for an operation without overloading some specific look-up server or keeping up-to-date caches of all clients.

A novel property of GIGA+ is that each client caches its own partition-to-server map (P2SMap), without using a traditional, synchronous cache consistency protocol. As the directory grows, at high insert rates, the directory is partitioned on more servers. Clients will not immediately know about the partitions created at the server due to an operation sent by another client. As a result, all clients will end up having different, out-of-date copies of the P2SMap. Despite the inconsistent copies of P2SMap, GIGA+ ensures that the clients' requests are forwarded to the correct server.

GIGA+ guarantees correctness by keeping local records of how the directories grow with usage – i.e., each server keeps a split history of all partitions they store. To provide incremental growth, each directory in GIGA+ is represented using an extendible indexing structure like a B-link tree [5] or extendible hashing [4]. In this discussion, we limit ourselves to extendible hashing, which uses a hash-table that grows and shrinks dynamically with usage, although a B-tree approach would be similar. Extendible hashing uses a two-level structure: A header table at the first level and buckets at the second level. The header table is an index that maps a key to the appropriate buckets; in GIGA+ the P2SMap serves as the header table. Similarly, in GIGA+ , buckets contain the contents of the directory and are distributed across the servers; these buckets are referred to as (directory) partitions.

Inserting an entry in a full partition results in splitting the partition across two servers. GIGA+ servers keep a split history for every partition they store, recording all splits they perform. For instance, splitting a partition P_1, at server S_1, creates a new partition P_2 on a another server S_2. The split history for P_1, recorded by S_1, includes information about the new partition resulting from the split: P_2, the server that holds P_2 (i.e. S_2) and other information about P_2, like the range of keys. A client with stale P2SMap sends its request to an "incorrect" server that no longer holds the desired partition. However this "incorrect" server knew about the desired partition in the past; so, the server uses its split history to update the client's P2SMap. Eventually client requests are sent to the "correct" server . In the example above, S_1 may receive operations from a client with a stale P2SMap for keys now stored in P_2, which it cannot service. However, using its split history, S_1 is able to redirect the client to a server more recently responsible for those keys, i.e. S_2 In this worst case, this process could take $O(\log n)$ forwarding hops, but the client updates its P2SMap with each forward, so the number of times this action is performed is bounded. Thus, the split history at the servers allows GIGA+ clients to maintain an inconsistent copies of P2SMap without affecting the correctness of the system.

Another key feature of GIGA+ is the novel representation of P2SMap that uses a deterministic mapping between partitions and the servers. Each huge directory is partitioned over a set of servers known a priori, called the server list. GIGA+ achieves a load-balanced partitioning by using keys obtained by hashing the directory entry names; this key serves as an index to lookup the desired partition. As a result, given the server list and the index, clients in GIGA+ use their P2SMap to locate the partition and the server that holds partition.

The P2SMap is represented as a bitmap, where the status of a bit indicates the presence or absence of a partition as shown in Figure 1. Given an index I, clients check the status of the bit at position I in the bitmap. If the bit is set, the partition exists and we use the server list to find the server and the partition ID on that server. If the bit is not set, either the partition does not exist or the client bitmap is stale and doesn't know about the partition. In both cases, we check the status of the bit at position $\lfloor I/2 \rfloor$; if the bit at this position is not set, we check the bit at position $\lfloor I/4 \rfloor$, etc. Clients repeat this until they encounter a bit that is set and send the request to the appropriate server as shown in Figure 2.

The bitmap representation of P2SMap has several significant advantages. First, the size of P2SMap is small enough (few kilobytes) to reside in-memory at most times. A directory with billion entries, which is split such that each partition holds 10000 entries, needs 100000 buckets – the P2SMap for this directory is a little more than 12 kilobytes. Second, bitmap representation enforces that partitions on a given server are stored in an ordered table. Since, each server can
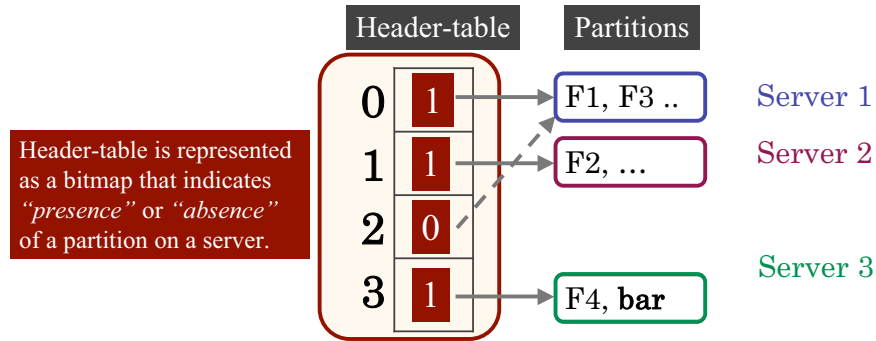
Figure 1: This figure shows how an extendible hash-table [Fagin79] is used to store a directory, which is split into partitions (which store the dir entries) that are distributed across many servers. The header-table represents the mapping information used by clients lookup the correct partition and its server. Instead of keeping the header-table, GIGA+ uses a bitmap, which indicates the "presence" or "absence" of a partition on the server. This provides a deterministic mapping of partitions to the correct server.

potentially store many partitions for a directory (and many such directories), the lookup for the right partitions is fast.

2.2 Optimizations: Two-Level Metadata and Insert Storms

GIGA+ uses a few optimizations to meet its design goals. Large insert rates can cause bottlenecks when updating the metadata at the metadata servers (MDS). GIGA+ avoids this by using a two-level metadata hierarchy that updates the remote MDSs very rarely, by keeping the most frequently updated metadata on the local server. For example, infrequently updated metadata, such as the owner and creation time of a directory may be stored at a centralized server without introducing a bottleneck, whereas highly dynamic attributes such as access and modification time are allowed to diverge across all servers managing the directory. It is the responsibility of clients to achieve the consistency they desire for these non-centralized attributes, e.g. by polling. In order to tolerate and recover from failures, GIGA+ keeps a write-ahead log of server operations and uses active-passive synchronous backups.

While GIGA+ should perform well in steady state, certain workloads can generate bursts of high traffic that result in packet loss and increased latency. GIGA+ provides server resources proportional to the directory size – i.e., small directories are striped across small number of servers. When many clients start inserting into a small directory at their maximum rate, the servers holding the directory stripes will be overloaded. For example, scientific workloads periodically checkpoint by barrier syncing all clients, and causing each core to create separate checkpoint files in a newly-created directory. The volume of inserts operations sent to the single server storing the directory's initial partition may overwhelm the network switches, saturate buffers at the server, and force the directory to undergo many partition splits to accommodate the new files, causing general instability. A similar phenomenon was observed in the Sprite file-system when a server recovering from a crash experienced a storm of recovery messages from clients [2]. We call this phenomenon an *insert storm*.

Similar to the approach in the Sprite file-system, in GIGA+ , if a server is overloaded, i.e., doesn't have resources to service a request, it responds with a negative acknowledgment (NACK) that tells the clients to back-off and engage in flow control. Without these NACK messages, clients can falsely assume that the server is unavailable and either stop sending messages or send them continuously at a rapid rate. However, NACKing all requests at the server may be detrimental to high priority control messages like "heart-beats" to detect server availability. We plan for servers to maintain a separate channel reserved for high priority messages, a technique used in some cluster storage systems [3].

However, using NACKs doesn't resolve our initial concern: GIGA+ provides server resources only in proportion to the current size of the directory, which may be insufficient for quickly and stably handling a storm. Ideally, to service a storm, the system should preemptively allocate extra servers to handle the expected final size of the directory. In order to estimate the final size of the directory, we need to aggregate statistics about the pending requests in the system.

In GIGA+ , we propose to use intermediate proxies that help both in aggregating statistics and in diverting bursty traffic to non-overloaded servers in the face of an insert storm. If a server is overloaded and clients begin receiving NACKs, each client sends their requests to a randomly selected proxy server that buffers and aggregates the client traffic rather than directly to its destination. These proxy servers control the traffic sent to the overloaded servers by using flow control mechanisms. In addition, aggregating all client requests at the proxy servers allows the collection statistics about these requests.

Proxy servers use these statistics of client requests to control the behavior of the system under a storm using the high priority channel. If the statistics indicate a storm of file creates in the same directory, GIGA+ would preemptively split the partition to accommodate the high insert rates. By splitting a directory's partitions when they are mostly empty, rather than waiting for them to be full, we seek to
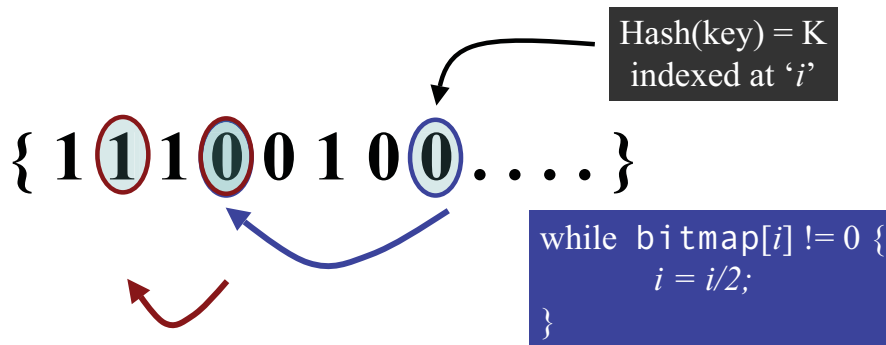
Figure 2: This figure shows how we use the BITMAP representation to lookup the presence or absence of a partition on any server. A bit-value of "1" indicates the presence of a partition on a server, and value "0" indicates the absence of the partition on a server. If the bit-value of "0", GIGA+ indexing techniques halves the index and checks the status of the parent partition..

eliminate most of the traffic needed to share the contents of the partition. This helps GIGA+ service the storm of create requests by pre-allocating extra servers for the directory, thus allocating server resources in proportion to the final size rather than the current size. Even though this optimization introduces another level of indirection to traffic, it avoids the bottleneck of a single server during an insert storm and uses pre-splitting to reduce the expense of many incremental growths from a small directory into a large one. As a result, GIGA+ can handle bursty request traffic by minimizing the overhead caused by dropping requests and increased latency.

3. STATUS

Currently, we are implementing GIGA+ in PVFS (Parallel Virtual File System), an open-source cluster file system [8], used in production at various national labs. PVFS stores directories on a single server, which limits the scalability and throughout of operations on a single directory. Our GIGA+ prototype extends PVFS by striping large directories over multiple servers.

4. REFERENCES

[1] Private Communication with Garth A. Gibson, Panasas Inc.

[2] M. Baker and J. K. Ousterhout. Availability in the Sprite Distributed File System. *Operating Systems Review*, 25(2), Apr. 1991.

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

[4] 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), Sept. 1979.

[5] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6(4), Dec. 1981.

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

[7] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. of 10th ACM Symposium on Operating Systems Principles (SOSP '85)*, Orcas Island WA, Dec. 1985.

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

[9] 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. http://institutes.lanl.gov/hec-fsio/docs/HECIWG-FSIO-FY06-Workshop-Document-FINAL6.pdf, 2006.

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

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

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

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