Swapnil Patil (Carnegie Mellon University)
Ph.D. Advisor: Prof. Garth Gibson
swapnil.patil @ cs.cmu.edu

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

Submitted to the *ACM Student Research Competition Grand Finals 2011.*

Introduction

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, object-based architectures and distributed locking. [16–18, 30, 41, 47]. Few file systems scale metadata performance by using a coarse-grained distribution of metadata over multiple servers [12, 36, 41, 49]. But most file systems cannot scale access to a *large number of files* [14, 52]. 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 [14, 34, 39].

One such motivating application is the most important I/O workload in today's supercomputers: 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]. 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 [34] will become much bigger in the impending Exascale-era, when applications may run on clusters with up to billions of CPU cores [23].

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 [48], 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, 42, 46].

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 create 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 [13, 27, 41], 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, 10], GIGA+ represents small directories in one partition so most directories will be essentially like traditional directories.

We have built a skeleton cluster file system with GIGA+ directories that layers on existing lower layer file systems using FUSE [15]. It uses the traditional UNIX VFS interface and provides 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 [49] and the HBase distributed key-value store [19], and exceeding peta-scale scalability requirements [34]. 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 [22, 45].

Related Work

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 [33, 44, 53] and even distributed file systems that run on the largest clusters, including HDFS [43], GoogleFS [16], PanFS [51] and PVFS [36], 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 [8] to provide a distributed file system metadata service [14].

Although there are file systems that distribute the directory tree over different servers, such as Farsite [12] and PVFS [36], to our knowledge, only three file systems now (or soon will) distribute single large directories: IBM's GPFS [41], Oracle's Lustre [28], and UCSC's Ceph [49]. GIGA+ has been influenced by the scalability and concurrency limitations of these prior techniques.

GPFS is a shared-disk file system that uses a distributed implementation of Fagin's extendible hashing for its directories [13, 41]. 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) [13]. 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 [41]. These mechanisms allow concurrent readers to cache the directory blocks using shared reader locks and deliver high read and lookup performance. But the same locking and consistency protocols hinder the scale and performance of concurrent writers adding files to a directory [1, 20]; that is, GPFS enforces strong consistency of the mapping information by updating a directory's on-disk i-node every time the directory adds more blocks. This effect is further exacerbated due to the lock manager contention and messaging overhead. 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 [28, 29]. 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 [49, 50]. 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 [25]. Its distributed variant, called LH* [26], stores buckets on multiple servers and uses a central split coordinator that advances permission to split a partition to the next server. After every split, LH* does not update a client's mapping state synchronously – a property use by GIGA+ too.

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 [26, 27]; GIGA+ allows any server to split a bucket at any time without any coordination. LH* offers a complex pre-split optimization for higher concurrency [27], 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 is another popular distributed hashing scheme that divides the hash-space into randomly sized ranges distributed over many nodes [22, 45] and efficiently manages node membership changes by splitting or joining hash-ranges of adjacent nodes only. The latter property makes it popular for wide-area (peer-to-peer) storage systems that have high rates of membership churn [9, 32, 37, 40]. Cluster systems, even though they have much lower churn than Internet-wide systems, have also used consistent hashing for data partitioning [11, 24] but discovered that the resulting data distribution has a high load variance, even after using "virtual servers" to map multiple randomly sized hash-ranges to each node [11]. GIGA+ uses threshold-based binary splitting that provides better load distribution even for small clusters.

GIGA+ indexing approach

GIGA+ uses a combination of three ideas to push the limits of scalability and concurrency of file system directories: scale-out data partitioning on servers without any system-wide synchronization, allowing the use of out-of-date partition-to-server mapping state, and facilitating online server addition with minimal data migration overhead. The following subsections presents each of these ideas in details.

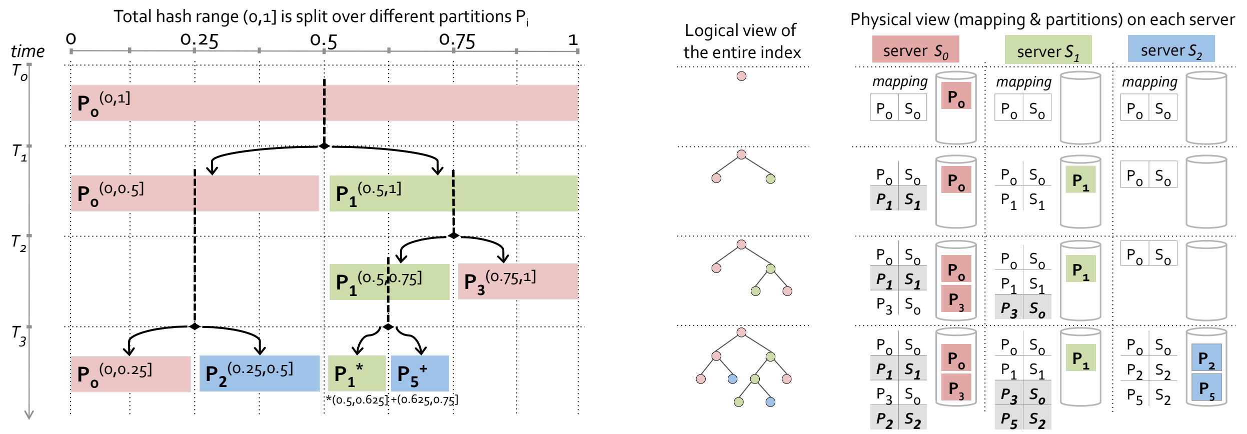

Figure 1 – Concurrent and unsynchronized data partitioning in GIGA+. The hash-space $(0,1]$ is divided into multiple partitions (P_i) that are distributed over many servers (different colors). An index starts small (on one server) and scales out incrementally by enabling each server to independently split its partitions *without global co-ordination*. Each server has a *local, partial view* of the entire index which includes the partitions they store and the history of splits of these partitions (shaded entry in the mapping table).

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 but is not specific to it. GIGA+ relies only on one property of the selected function: for any distribution of unique filenames, the hash values of these filenames must be uniformly distributed in the hash space [38].

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 colors. $P_i^{(x,y]}$ denotes the hash-space range $(x,y]$ held by a partition with the unique identifier i. 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. GIGA+ computes a split's target partition identifier using well-known radix-based techniques.

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

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

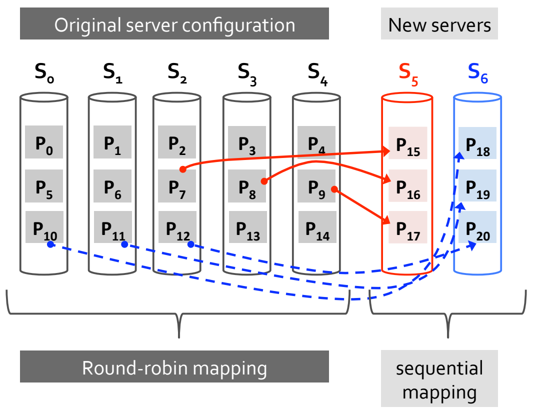

Figure 2 – GIGA+ server additions. By changing the partition-to-server mapping from round-robin on the original server set to sequential on the newly added servers, GIGA+ can minimize the amount of data migrated (shown by arrows indicating splits).

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. 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) suffers addressing errors even when the index stops mutating [27].

On-line server additions

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 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 [21], 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.

Experimental Evaluation

We built a user-level GIGA+ distributed directory prototype using the FUSE API [15]. It has three components: unmodified applications running on clients, user-level GIGA+ indexing processes (of the 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 to the client-side GIGA+ indexing module which implements the indexing technique to send client requests to the appropriate server. The GIGA+ server module's primary purpose is to manage 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

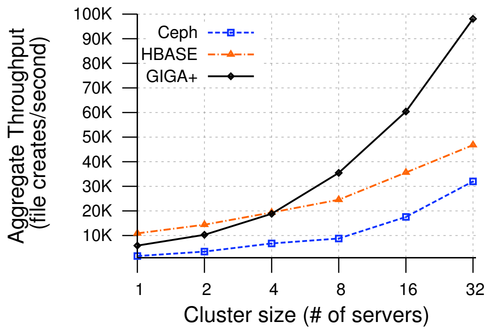

Figure 3 – Scalability of GIGA+ FS directories. GIGA+ directories deliver a peak throughput of roughly 98,000 file creates per second. The behavior of underlying local file system (ReiserFS) limits GIGA+'s ability to achieve ideal linear scalability.

to directory objects within the backend file system. The GIGA+ server also handles splits by locally locking the particular directory partition, scanning its entries to build two sub-partitions, and then migrating ownership of one partition to another server (before releasing the lock).

For experiments, we used 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 ReiserFS on one 7200rpm SATA disk. 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. To create a directory workload, we used the synthetic `mdtest` benchmark [31] (used by parallel file system vendors and users [20, 49]).

Figure 3 plots aggregate operation throughput, in file creates per second, averaged over the complete concurrent file create benchmark run as a function of the number of servers (on a log-scale X-axis). In this experiment, the `mdtest` application creates a large number of files, in proportion to the number of servers used, concurrently in a single directory: 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. 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 [34].

Figure 3 compares GIGA+ with the scalability of the HBase distributed key-value store and the Ceph file system. 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[14]. We setup HBase on a 32-node HDFS configuration with a single copy (no replication) and tuned it to achieve the best small-file performance possible; This 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. Figure 3 also shows the Ceph results from their original paper [49].

GIGA+ achieves a well-balanced data distribution with 10-100 times fewer partitions (and splitting) than the popular consistent hashing technique.
GIGA+ clients incur a neglible overhead when using stale, inconsistent mapping state, i.e. less than 0.1% requests need to be re-routed to the correct server, and this re-routing overhead is eliminated after the directory stops mutating.
GIGA+ performance is affected by the design decisions (such as on-disk layouts and journaling policies) made by different underlying local file systems.

Table 1 – Summary of the analysis of GIGA+ design choices

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.

Table 1 summarizes the results from our detailed analysis of the GIGA+ including load balancing, weak index consistency and choice of the lower-layer local file system used for out-of-core indexing. [1]

Contributions

In this research, we address the emerging requirement for 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.

Our contributions in scaling file system directories are beginning to have a real-world impact: OrangeFS is currently integrating a GIGA+ based distributed directory implementation into a system based on PVFS [2, 35] and (in March 2011) Exascale-era workgroups have chosen GIGA+ as one of the few recent research results that need to be converted to real production systems to meet the scalability challenges in the next decade.

[1]Full paper on GIGA+ appeared in USENIX FAST 2011.

References

[1] Personal communication: F.Schmuck and R.Haskin, IBM, 2009.

[2] Private Communication: Walt Ligon, OrangeFS, Nov. 2010.

[3] A. Abouzeid, et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB 2009*. 2009.

[4] N. Agrawal, et al. A Five-Year Study of File-System Metadata. In *USENIX FAST 2007*.

[5] R. Agrawal, et al. The Claremont report on database research. *ACM SIGMOD Record*, 37(3), Sep. 2008.

[6] D. Beaver, et al. Finding a Needle in Haystack: Facebook's Photo Storage. In *USENIX OSDI 2010*.

[7] J. Bent, et al. PLFS: A Checkpoint Filesystem for Parallel Applications. In *ACM/IEEE Supercomputing (SC) 2009*.

[8] F. Chang, et al. Bigtable: A Distributed Storage System for Structured Data. In *USENIX OSDI 2006*.

[9] F. Dabek, et al. Wide-area cooperative storage with CFS. In *ACM SOSP 2001*.

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

[11] G. DeCandia, et al. Dynamo: Amazon's Highly Available Key-Value Store. In *ACM SOSP 2007*.

[12] J. R. Douceur, J. Howell. Distributed Directory Service in the Farsite File System. In *USENIX OSDI 2006*.

[13] R. Fagin, et al. Extendible Hashing – A Fast Access Method for Dynamic Files. *ACM TODS*, 4(3), Sep. 1979.

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

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

[16] S. Ghemawat, H. Gobioff, S.-T. Lueng. Google File System. In *ACM SOSP 2003*.

[17] G. A. Gibson, et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS 1998*.

[18] J. H. Hartman, J. K. Ousterhout. The Zebra Striped Network File System. In *ACM SOSP 1993*.

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

[20] R. Hedges, et al. 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`.

[21] P. Hunt, et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Conference '10*. 2010.

[22] D. Karger, et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM STOC 1997*.

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

[24] A. Lakshman, P. Malik. Cassandra - A Decentralized Structured Storage System. In *LADIS Workshop 2009*.

[25] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *VLDB 1980*.

[26] W. Litwin, M.-A. Neimat, D. A. Schneider. LH* - Linear Hashing for Distributed Files. In *ACM SIGMOD 1993*.

[27] W. Litwin, M.-A. Neimat, D. A. Schneider. LH* - A Scalable, Distributed Data Structure. *ACM TODS*, 21(4), Dec. 1996.

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

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

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

[31] MDTEST. mdtest: HPC benchmark for metadata performance. `http://sourceforge.net/projects/mdtest/`.

[32] A. Muthitacharoen, et al. Ivy: A Read/Write Peer-to-peer File System. In *USENIX OSDI 2002*.

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

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

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

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

[37] S. Rhea, et al. Pond: the Oceanstore Prototype. In *USENIX FAST 2003*.

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

[39] R. Ross, et al. High end computing revitalization task force (HECRTF), inter agency working group (HECIWG) file systems and I/O research guidance workshop 2006.

[40] A. Rowstron, P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *ACM SOSP 2001*.

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

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

[43] K. Shvachko, et al. The Hadoop Distributed File System. In *MSST 2010*. 2010.

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

[45] I. Stoica, et al. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*.

[46] M. Stonebraker, et al. C-Store: A Column-Oriented DBMS. In *VLDB 2005*.

[47] C. A. Thekkath, T. Mann, E. K. Lee. Frangipani: A Scalable Distributed File System. In *ACM SOSP 1997*.

[48] 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.

[49] S. A. Weil, et al. Ceph: A Scalable, High-Performance Distributed File System. In *USENIX OSDI 2006*.

[50] S. A. Weil, et al. Dynamic Metadata Management for Petabyte-Scale File Systems. In *ACM/IEEE Supercomputing (SC) 2004*.

[51] B. Welch, et al. Scalable Performance of the Panasas Parallel File System. In *USENIX FAST 2008*.

[52] R. Wheeler. One Billion Files: Scalability Limits in Linux File Systems. In *LinuxCon*. 2010.

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