Scale and Concurrency of Massive File System Directories

Swapnil Patil

CMU-CS-13-113

May 2013

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth Gibson, Chair
William J. Bolosky (Microsoft Research)
Christos Faloutsos
Gregory R. Ganger
Mahadev Satyanarayanan

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the Department of Energy (DOE) under grant number DEFC0206ER25767, by the Los Alamos National Laboratory (LANL) under contract number 1535931, by the National Science Foundation (NSF) under grant numbers CCF1019104 and CCR-0326453, by the UIUC (NSF) award 0430781, and by generous research awards from Intel and Seagate. This material is also supported in part by the NSF awards CNS1042537 and CNS1042543 (PRObE http://www.nmc-probe.org/). We also thank the members and companies of the PDL Consortium for their interest, insights, feedback, and support.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any corporations or or the U.S. Government.

Keywords: Scalable Directory Indexing, Load Balancing, Distributed Hashing, High-speed File Ingest, Cluster File Systems

For my parents.
For their unwavering support.

Abstract

File systems store data in files and organize these files in directories. Over decades, file systems have evolved to handle increasingly large files: they distribute files across a cluster of machines, they parallelize access to these files, they decouple data access from metadata access, and hence they provide scalable file access for high-performance applications. Sadly, most cluster-wide file systems lack any sophisticated support for large directories. In fact, most cluster file systems continue to use directories that were designed for humans, not for large-scale applications. The former use-case typically involves hundreds of files and infrequent concurrent mutations in each directory, while the latter use-case consists of tens of thousands of concurrent threads that simultaneously create large numbers of small files in a single directory at very high speeds. As a result, most cluster file systems exhibit very poor file create rate in a directory either due to limited scalability from using a single centralized directory server or due to reduced concurrency from using a system-wide synchronization mechanism.

This dissertation proposes a directory architecture called GIGA+ that enables a directory in a cluster file system to store millions of files and sustain hundreds of thousands of concurrent file creations every second. GIGA+ makes two contributions: a concurrent

indexing technique to *scale out* a growing directory on many servers and an efficient layered design to *scale up* performance. GIGA+ uses a hash-based, incremental partitioning algorithm that enables highly concurrent directory indexing through asynchrony and eventual consistency of the internal indexing state (while providing strong consistency guarantees to the application data). This dissertation analyzes several trade-offs between data migration overhead, load balancing effectiveness, directory scan performance, and entropy of indexing state made by the GIGA+ design, and compares them with policies used in other systems. GIGA+ also demonstrates a modular implementation that separates directory distribution from directory representation. It layers a client-server middleware, which spreads work among many GIGA+ servers, on top of a backend storage system, which manages on-disk directory representation. This dissertation studies how system behavior is tightly dependent on both the indexing scheme and the on-disk implementations, and evaluates how the system performs for different backend configurations including local and shared-disk stores. The GIGA+ prototype delivers highly scalable directory performance (that exceeds the most demanding Petascale-era requirements), provides the traditional UNIX file system interface (that can run applications without any modifications) and offers a new functionality layered on existing cluster file systems (that lack support for distributed directories).

Acknowledgements

The Ph.D. process is a great experience, and most people are happy to have gone through it once — and only once. But I would jump into it again — really, all over again, another Ph.D. in CMU CSD in Pittsburgh — in a jiffy! And the only reason I say this is because of all the people who made this a great journey.

I would not be here without the exemplary guidance of my advisor Garth Gibson. His enthusiasm and energy both in research and in life kept me motivated all these years. Garth let me poke into interesting research problems (not related to my dissertation) and learn how lessons from one research are often applicable in different contexts. Of course, when I drifted too far from my dissertation, he would nudge my focus back to the thing at hand. Garth always emphasized on distinguishing a research contribution from an implementation artifact. Our meetings would often end with me thinking "Why did I not think of this?". His ability to distill complex thoughts into simple ideas was incredible. For any problem, related to research or job search or life in general, Garth often came up with brilliant and practical advice that gave me a better perspective on the problem at hand. I thank Garth for his advice for all these years.

I would also like to thank members of my thesis committee for helping me improve this dissertation. Greg Ganger was always fun to interact with. He would question ideas and results to ensure that I have covered all bases and thought through everything. His advice on non-research aspects of graduate school was always filled with "it depends" trade-offs that made my decision making much easier. He is also responsible for making PDL a great place to collaborate with students and industry partners. Christos Faloutsos helped me sharpen the database indexing related aspects of my dissertation. His boundless enthusiasm and optimism always meant that our meetings ended on a constructive and instructive note. Bill Bolosky graciously accepted to serve as an external member on my committee. I am grateful to Bill for traveling to Pittsburgh for both my thesis proposal and my thesis oral. He strongly encouraged the idea of using asynchrony and inconsistency of GIGA+ while giving up other aspects of the design. His meticulous review of the dissertation draft was immensely helpful. M. Satyanarayanan improved my dissertation further by questioning about high-performance computing systems and clarifying their differences from enterprise distributed file systems. He forced me think hard about the broad applicability and generality of my research ideas.

The computer science department at Carnegie Mellon is a great place. I was constantly motivated by the enthusiasm of other faculty members. In particular, Dave Andersen, Hui Zhang, Peter Steenkiste and Srini Seshan always shared great advice. I thank Dave Andersen for teaching me the nitty-gritty details of systems research during my early research projects at CMU. The administrative staff in the department is another reason that made graduate school a smooth ride. Deb Cavlovich is a magician who made life incredibly easy for every student in the department. She patiently handled all university-

wide requirements for all the lazy students (like me) who would totally forget about administrative deadlines. Angela Miller helped with numerous logistics for research travel and meetings. I enjoyed our conversations about the state of Steelers football. I would also like to thank Joan Digney for creating posters that I needed for numerous workshops, conferences and PDL events.

My research greatly benefited from collaboration with several students and external researchers. Kai Ren worked together on improving GIGA+ through his ideas about metadata-optimized stores. His tenacious hacking skills helped us get the right graphs before numerous paper deadlines. Anthony Chivetta worked with me on trying to implement GIGA+ on PVFS. Despite juggling a hectic undergraduate workload and extracurricular responsibilities, Anthony was brilliant at digging deep into PVFS to find and fix problems when merged with GIGA+. I also thank several other students who worked on different GIGA+ prototypes: Sanket Hase, Aditya Jayaraman, Vinay Perneti, Sundararaman Sridharan and Kartik Kulkarni. Several external collaborators supported the idea of GIGA+ for scalable directories. Gary Grider, John Bent and James Nunez from Los Alamos National Laboratory greatly encouraged us to solve this rather obscure problem (when I first started working on it). Rob Ross, Sam Lang and Phil Cars from Argonne National Laboratory helped us with PVFS cluster file system on numerous occasions. Chuck Cranor, Mitch Franzos, Zis Economou and Mike Stroucken ensured that cluster computing resources were available for experiments.

I also thank my fellow students, Milo Polte, Wittawat Tantisiriroj, Kai Ren, Lin Xiao, Jiri Simsa and Vijay Vasudevan, for their enthusiasm and criticism about life in graduate school. They would always be a stone's throw away when I needed to bounce off

some research ideas. And they would never shy away from stating their opinion about my research. They all added their own flavor to life outside of work. My hiking trip to Banff National Park and Glacier National Park with Milo was exhilarating and excruciating. My late night paper writing sessions with Wittawat were more fun when he shared his (crazy) fun-filled stories with me. Lin was my office mate, and she was the most optimistic person I have known in graduate school. Her boundless enthusiasm for trying out new things (mostly restaurants) was very refreshing. Jiri and I shared a secret communication code, filled with sarcasm and nastiness, that made our interactions fun (and puzzling to others). Vijay shared my enthusiasm for espressos and cappuccinos, and was always eager to go try out a new coffee shops in town.

Vyas Sekar, Hetu Kamichetty, Gaurav Veda, Varun Gupta, Amar Phanishayee, Vivek Seshadri, Ravi Krishnaswamy, Harsha Simhadri, Pranjal Awasthi and Srivatsan Narayanan all had a vigorous approach to work and life; they motivated me to work hard and play hard. My early officemates, Dan Wendlandt and Rob Shields, and neighbors, Mike Dinitz and Mike Tschantz, made transition to graduate school much more fun than I expected it to be. Our heated political debates, drinking games and foosball playoffs made our Doherty offices look cool. Niraj Tolia, Julio Lopez and Mike Abd-El-Malek always had time to share their wisdom and advice about graduate school and the life after.

Outside of work, Apurva Samudra and Suyash Shringarpure were successful in providing multiple ways to procrastinate from work. I thank them for subjecting me with their "thought experiments", introducing me to British television, joining me to watch midnight movie premieres, and many other fun experiences. I also thank Apurva for helping me start the quiz club for trivia enthusiasts. For several years it was a fantas-

tic (and nerdy) way to spend Friday evenings; Keshav Seshadri and Supreeth Achar were particularly great at making it happen and continuing it even after I graduated.

None of this would have been possible without the support of my family: my wife and my in-laws, my sister and my brother-in-law, and my parents. My wife Shilpa helped me see this to the end — she ran with me in the crucial last 3 miles of a marathon when one needs a new boost of energy to finish. She was especially cool when I spent the weekends working on the dissertation in a coffee shop. My sister Shwetal was a great listener who tolerated our phone conversations when I was distrated by another graph, a new bug, or another paper deadline — and she never kept the phone down before I did! And my biggest supporters, my parents, endured the countless "Oh, Swapnil is still studying!" comments and "Isn't he done yet?" questions with a proud smile on their face. My mom pretended (very well) that she understood all my boring graduate school mumbo-jumbo on the phone even when she was only doing that to make sure that I stay focused and determined (and that I ate my fruits and veggies!). My dad asked the "hard" questions about my thesis, my research and my meetings. I remember his first question after a week or two in the Ph.D. program: "So have you started working on your thesis yet?". It took my years before I could answer that question with a "Yes", but he will certainly be proud to hear my answer now: "BTW, the dissertation is now done." :-)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Everything builds from files as a base. ... But, file systems have no metadata beyond a hierarchical directory structure and file names. ... Lastly, most file systems can manage millions of files, but by the time a file system can deal with billions of files it has become a database system.
>
> -- Jim Gray in *"Scientific Data Management in the Coming Decade"* [Gray 2005]

This dissertation pertains to file system support for high-performance applications, in both scientific computing and Internet-scale computing, that produce and consume millions to billions of small files at very fast speeds. These applications run on thousands of machines, and rely on a cluster file system to store data on thousands of storage nodes. Modern cluster file systems use data servers to manage data blocks containing file contents and metadata servers to manage metadata including directories, file attributes and data block locations [Ghemawat 2003, Welch 2008, Lustre 2010b, Schmuck

2002, Weil 2006a]. These file systems provide high-performance I/O bandwidth on the data path through concurrent access to large files using techniques such as data chunking and striping [Ghemawat 2003, Hartman 1993, Anderson 1995], object-based abstractions [Gibson 1998, Lustre 2010b, Welch 2008] and distributed locking [Schmuck 2002, Thekkath 1997]. Most cluster file systems do not provide parallel access on the metadata path [Ross 2006, Newman 2008, Raicu 2011] — they serialize and centralize access to the file system namespace and directories.

This dissertation presents an approach for cluster file systems to provide a scalable and parallel directory subsystem for applications that generate metadata-intensive workloads such as storing millions to billions of files in a single directory and performing hundreds of thousands of concurrent accesses, particularly mutations like file creates, in a single directory. Unfortunately current directories are ill-equipped to meet these requirements for two reasons.

The first reason is that the design of file system directories is governed by a fifty-year old principle in which directories are a means to achieve hierarchical structure and organization of files for human usage [Daley 1965, Seltzer 2009]. This use-case as an organizing principle for ease of human use, and not program access, has two characteristics — low degree of concurrent accesses, particularly file creates, to each directory and small number of files, typically tens to hundreds, in each directory. The second reason is that cluster file systems are built to handle large files and not specifically large directories, and most most use a single metadata server to store directories [Ghemawat

2003, Shvachko 2010, Lustre 2010b].[1] These centralized metadata servers rely either on existing local file system directories [Cao 2007, Mathur 2007, Reiser 2004] or customized directory implementations [Zhen 2011, Dilger 2012] — both single-site approaches, however, do not provide the scalability required by applications with massive numbers of files and fast concurrent accesses in a single directory. This dissertation aims to push the limits of scale and concurrency of directory subsystems for cluster file systems.

1.1 Challenges

This dissertation is motivated by two new trends in technology and application workloads that call for a scaling the metadata path in modern cluster file systems.

Trend #1: Massive application-level parallelism

> The last decade has seen tremendous innovation in large-scale parallelism. Initial efforts focused on increasing the number of cores on each CPU and increasing the number of CPUs in each physical machine. These efforts then focused on increasing the size of compute clusters: clusters today comprise of hundreds to thousands of machines each with one to eight cores. In fact, this number is growing rapidly [Top500 2012]. Some workgroups are predicting that Exascale-era clusters may have as many as one billion CPU cores [Kogge 2008].
>
> Furthermore, these clusters are now easily accessible to everyone. Until a few years ago, clusters of hundreds to thousands of machines were available only to a "niche"

[1] Some file systems, like Panasas PanFS [Welch 2008] and PVFS [PVFS2 2010], use multiple metadata servers to distribute the file system namespace. But they continue to store individual directories on one metadata server.

set of users, such as supercomputing sites and large organizations. Today users can easily run their applications on a large collection of resources that can be leased from cloud computing providers like Amazon [AWS 2012] and Joyent [Joyent 2012]. Availability of such massive computational resources has led to rapid innovation in applications that run at a large-scale; thus, increasing the burden on the underlying cluster-wide storage systems.

Despite a plethora of scalable storage systems such as distributed databases and key-value stores, cluster file systems continue to be the dominant interface for applications to store and access data in a large cluster. In fact, most distributed databases and key-value stores are built as an additional layer on top of a distributed cluster-wide file system. For example, Google's BigTable and Spanner data stores rely on the Google file system and, its successor, the Colossus file system [Chang 2006, Corbett 2012, Ghemawat 2003, Fikes 2010]. Thus cluster file systems need to keep evolving to provide scalable performance for a wide range of application workloads.

Trend #2: Metadata-intensive small-file workloads

Large-scale distributed applications, such as scientific computation and batch processing frameworks, typically generate workloads that are dominated by accesses to large files [Dayal 2008]. Thus, as massive systems have evolved, data parallelism has become a norm and predictably manageable. The next, and currently not well supported, bottleneck will be metadata parallelism. There is a growing set of applications in both scientific computing and Internet-scale computing that generate one of the most difficult workloads for file systems to handle efficiently: metadata-

intensive I/O accesses. In fact, the amount of metadata and small objects in today's data-sets is growing rapidly; Amazon's S3 storage service, for example, has grown to store more than a trillion objects in the five years since it began service [Barr 2012].

This new workload is generated by applications that use the storage system as a fast, lightweight data store by (simultaneously) creating large number of files at high speeds. Cluster file systems in HPC, for example, are struggling with applications that want to create millions of files rapidly in a single directory in bursts [Ross 2006, Newman 2008, Raicu 2011]. These concurrent workloads have several characteristics: they are dominated by small-sized accesses, typically few kilobytes to megabytes; they happen on the metadata and the data path; and, they are generated in several ways, including long periods of high file create rate, or short bursts of even higher file create rates, or a mix of the two. Examples of such file system operations include creating new files in one or more directories, writing small amount of data to files immediately after creation, and accessing or updating file attributes.

The combination of these trends — *large-scale, parallel metadata-intensive workloads* — is the main motivation of this dissertation. The next section gives examples of applications that generate such workloads that stress a cluster file system's metadata service and result in surprisingly low application performance.

1.2 Problem

This dissertation focuses on metadata-intensive metadata workloads, particularly those that create billions of files in a single directory at very high speeds in parallel.

Most file systems today cannot handle billions of files efficiently. Performance of local Linux file systems degrades when scaled up from a few million to a billion files [Wheeler 2010] and the stability of cluster file system is questionable after a few tens of millions of files [Fikes 2010]. File system directories are even worse when users put large numbers of files in a single directory. Local file systems are still unable to handle little more than tens of thousands of files in each directory [ZFS-discuss 2009, NetApp 2010, StackOverflow 2009] and even cluster file systems that run on the largest clusters, including Lustre [Lustre 2010b], HDFS [Shvachko 2010] and GoogleFS [Ghemawat 2003] are limited by the speed of the single metadata server that manages an entire directory.

There are several real-world applications that use directories at such a large scale. One example, and the motivating use-case for this work, is checkpoint restart in supercomputing where many parallel applications running on, for instance, ORNL's CrayXT5 cluster (with 18,688 nodes of twelve cores each) periodically write application state into the cluster file system using a file per process, where all files are stored in the same directory [Ross 2006, Bent 2009]. 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 [Bent 2009]. This dissertation grew out of a challenging requirement in 2006 of supporting 40,000 file creates per second in a single

directory [Newman 2008]; this requirement will become much bigger in the impending Exascale-era, when applications may run on billion-core clusters [Kogge 2008].

Supercomputing checkpoint-restart, although important, may not be a sufficient and large reason for overhauling the current file system directory designs. Yet there are other applications, such as gene sequencing [Yaschenko 2011], image processing [Tweed 2008]. phone logs for accounting and billing [Verizon 2006], and photo storage [Beaver 2010], that essentially want to store an unbounded number of files that are logically part of one directory. Some image processing applications store information associated with all frames in separate files in one directory associated with that image [Tweed 2008]. Computational genomics applications often create a small file for every possible gene sequence during a micro-array sequencing experiment [Yaschenko 2011, Goldstein 2012]. Astrophysicists sometimes store zero-byte files, with all application-defined attributes embedded in the filename, associated with every luminous object found in a telescopic images of the sky that are used in long running simulations [Kantor 2010]. Even applications used by Internet-scale services are beginning to see such challenging use-cases. Batch processing frameworks, like Hadoop [Hadoop 2011], routinely produce large number of KB-sized files (with intermediate results) to be stored and accessed using a cluster file system [White 2009].

Because the file system metadata service suffers from performance bottlenecks, authors of applications that generate such workloads have used several ad hoc approaches to meet their goals.

One approach, and a rather extreme one, is to force applications to avoid generating access patterns that stress the metadata service. Instead application developers are ex-

pected to program to the underlying file system's strength. A popular example of this approach is the Google file system; its authors claimed that *"When [we are] regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it ..."* [Ghemawat 2003]. For high-performance, the Google file system developers encourage application writers to perform large sequential writes through a custom-built atomic append primitive and a non-POSIX programming interface [Ghemawat 2003].

Another approach is to move the burden of supporting emerging workloads from the file system to the application. For example, web browsers manage a large logical directory by creating many small, intermediate sub-directories with files hashed into one of these sub-directories; but each browser re-implements this same functionality in their own application logic. Another example is the way Hadoop distributed file system (HDFS) handles its small files problem of managing a large number of tiny files with intermediate results. It forces applications to use a custom library that bundles many small files into few large files using custom formats such as sequence files and Hadoop Archives format [White 2009, Kerzner 2009].

The final approach is to build custom storage APIs that are designed to handle new, anticipated workloads. Such specialized interfaces and semantics are starting to emerge in both Internet-scale computing and scientific computing. Facebook's Haystack storage system uses a custom, metadata-optimized storage format to store information required to speed-up access to a user's photos [Beaver 2010]. Google's Colossus file system stores all the file system metadata in a tabular format in the Bigtable distributed data store [Chang 2006, Fikes 2010]. LANL's MDHIM system is targeting a user-level library

that stores file system metadata in an indexed on-disk format, such as ISAMs, to provide high-throughput operations for scientific data management [Nunez 2012].

The aforementioned approaches have been successful in their respective ways, but they have several drawbacks. They require an undesirable re-write of legacy applications to use new interfaces or custom semantics. They bind an application to a system that lacks a general-purpose design, and may struggle to adapt to new workloads. In fact, these ad hoc approaches are ossifying, not innovating, the design of general-purpose file systems by not adding support for emerging workloads.

Both vendors and users of cluster file systems will benefit from a general-purpose scalable metadata subsystem. By extending existing file system implementations with new features like scalable directories, vendors can support a broader range of application workloads and users can run applications without complex porting or rewriting. Furthermore, such a workload-agnostic metadata service can make more informed decisions to adapt to challenges at large scales such as load balancing and work distribution.

Given the challenges enumerated this far, this dissertation explores ways to build a scalable metadata service that has following objectives:

- It should inherit data-path scaling features of the existing cluster file systems while adding new support that distributes namespace management and parallelizes directory indexing.

- It should work with legacy unmodified applications through the traditional UNIX file system interface and POSIX-like semantics offered by many cluster file systems.

- It should strive to meet the requirements of future metadata-intensive applications that are predicted to run at a scale that is ten to hundred times bigger than today.

1.3 Thesis contributions and roadmap

The thesis of this dissertation is that:

> *Cluster file systems lack high-performance directory implementations that can meet high-speed ingest requirements of data-intensive applications. File system directories can evolve to be scalable and parallel: each directory can store hundreds of millions of files and sustain hundreds of thousands of concurrent operations, particularly mutations like file creates, per second. These high-performance directories enable existing modern cluster file systems to scale metadata access while providing the traditional UNIX file system API and POSIX-like semantics.*

This section describes the scope of this thesis statement, the contributions made to support this statement, and the roadmap for this dissertation.

1.3.1 Goals and non-goals

Given the challenges enumerated thus far, this dissertation explores ways to build a scalable metadata service that has following objectives:

Application visible semantics —

Most high-performance applications use cluster file systems that offer the traditional UNIX file system interface and POSIX semantics. To ensure that legacy applications work out-of-the-box (without any modifications), this dissertation focuses

on POSIX semantics. However, there are several other popular file system semantics such as Microsoft's SMB and CIFS standards [SMB 2013, CIFS 2013]. Although most techniques described in this dissertation should work with non-POSIX standards, they may result in less than optimal performance.

One example is directory scans using the *readdir()* method. SMB standard offers an API for range scans of a directory based on the name of the file [SMB 2013]. But POSIX neither offers such an API nor expects the file system to return entries in an ordered manner. To efficiently support range scans, an SMB-compliant system would benefit more from using an indexing structure, such as B-trees, that preserves key locality than from using a structure, such as hash-tables, that do not offer key locality. In the latter indexing structure, the system may have to scan more data than necessary, and that may result in slower scan performance.

Consistency semantics and mechanisms —

Different file systems offer different data consistency guarantees, and applications use a file system the provides the desired consistency properties. Most POSIX-compliant file systems offer strong data consistenty guarantees to the application. However these guarantees may not apply to the internal state of the file system. This is often referred to as *external consistency* and *internal consistency*.

This dissertation also uses these consistency models: applications are guaranteed a strong data consistency but the directory's internal state, such as indexing and mapping state, may be temporarily inconsistent. Thus applications will not experience any semantic side-effects but they may experience performance-related side-effects. In particular, application requests that are serviced when the file system

has temporary internal inconsistencies may experience slower response times or reduced throughput. And this performance degradation is often dependent on the mechanisms and policies used by the file system to fix these internal inconsistencies.

Fault-tolerance and configuration management —

Most cluster file systems have sophisticated techniques for handling failures and configuration management. Techniques such as data and process replication, data encoding, and primary-backup failover are common among many real-world file systems [Lustre 2010b, Welch 2008, Schmuck 2002, HDFS 2010, Ghemawat 2003]. Some file system also rely on tools and techniques that manage server membership and configuration changes [Burrows 2006, Hunt 2010, Welch 2007].

This dissertation does not propose any new techniques to address fault tolerance and configuration management. This decision was driven by two reasons. The first reason is that scalable directory and metadata support is envisioned as an extension to cluster file systems that only provide scalable and reliable data access. Our goal is to build a system that reuses and relies on the cluster file system's support for fault tolerance. The second reason is that there has been decades of research on techniques to improve fault tolerance and configuration management. When a cluster file system's mechanisms are insufficient, our goal is to rely on well-studied techniques to meet the desired availability, reliability and manageability requirements.

1.3.2 Contributions

To support the thesis statement, this dissertation describes the design, implementation and evaluation of a scalable directory service called GIGA+. The research contributions made by GIGA+ are as follows:

- *GIGA+ uses a distributed directory index that improves concurrency by trading space efficiency: it uses asynchrony and inconsistency for scale-out growth and non-blocking concurrency at the cost of less-than-optimal size of index representation.*
  GIGA+ indexing uses hashing to divide a large directory into partitions that are distributed on multiple servers in a "shared nothing" manner such that GIGA+ clients and servers have only a partial view, which is both deterministic and self-computable, of partitions in the system. This enables two novel design decisions. First, GIGA+ servers make independent decisions about when to split a growing directory, and then split across servers without any system-wide co-ordination, synchronization or serialization. Second, as the servers scale-out a directory in parallel, GIGA+ tolerates the client-side indexing state, such as partition to server mapping, to become stale and inconsistent; GIGA+ servers update a client's state in an on-demand lazy manner at the cost bounded number of addressing errors.

- *GIGA+ uses a layered implementation that decouples inter-server distribution and parallelization from intra-server on-store representation.*
  GIGA+ layers its distributed indexing on existing backend storage systems such that it uses GIGA+ servers only to manage access to hash partitions and it relies on the backend storage system to store partitions efficiently. This modularization

has several benefits. First, the distributed indexing scheme can spread directory entries on multiple servers only until it can benefit from the decentralization (such as good load-balancing and more parallelization). Second, backend storage systems are better equipped at out-of-core indexing of hash partitions on persistent media; consequently even when GIGA+ stops cross-server indexing, the backend stores can handle growing hash partitions using the optimized implementations. Third, layering also allows for GIGA+ to focus on optimizing directory performance while re-using features that are well-supported by a backend storage system; if the backend store is a cluster file system, GIGA+ can re-use features such as scalable data access, fault tolerance and system configuration that are found in most cluster file systems. The final benefit of layering is the ease of development and deployment of new functionality as extensions to existing systems.

- *GIGA+ proposes a specialized directory representation that is efficient at both representing directory entries on each server and distributing entries across servers.*

  Experimental analysis of the GIGA+ prototype layered on local file systems demonstrates how the interaction between lower-level implementations and GIGA+ system behavior has significant implications of scale and performance. This dissertation demonstrates that representing traditional directory entries as symbolic links allows cluster file systems to support a scalable metadata path by re-using their high-performance data path.

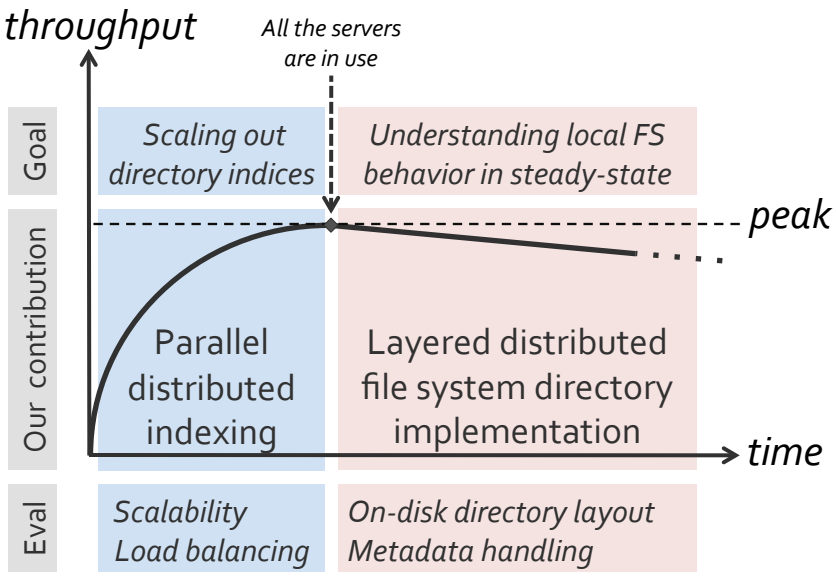

Figure 1.1 — Outline of the research contributions in this dissertation.
Scalability of the GIGA+ user-level layered file system directories, presented in Chapter 3, depends on how the higher-layer scales out a directory index on available servers and how each server represents directory entries on an on-disk backend storage system. This dissertation studies the trade-offs made by a parallel distributed indexing to enable highly concurrent accesses, particularly large numbers of simultaneous file inserts, in a single directory (Chapter 4 and 5). This dissertation also analyzes how the interaction between the lower-layer backend stores and higher-layer indexing scheme affects the scale and performance of the system (Chapter 6).

1.3.3 Dissertation roadmap

Figure 1.1 illustrates the research contributions and a roadmap of this dissertation. It shows a canonical graph of how system performance (throughput on Y-axis) changes over the time (on X-axis). The blue area shows the state of the system when GIGA+ is indexing a large directory on the available servers to provide scale-out growth. The red area shows how the interaction between the indexing technique and the backed storage

system affects the performance of GIGA+ after it has used all available server resources. This dissertation studies these aspects of GIGA+ design in the following chapters.

Chapter 2 presents a discussion of the background and related work in the context of file system metadata management and directory service. It presents a taxonomy of metadata management techniques used in modern file systems, including local file systems, distributed file systems, and high-performance cluster file systems. This chapter also describes the design of distributed data-structures, particularly hash-table based indices, and the tradeoffs to make these structures scalable and concurrent in distributed storage systems.

Chapter 3 gives an overview of the design and implementation of the GIGA+ file system directory prototype. The key idea of the GIGA+ prototype is that it decouples the logical, user-visible namespace from the physical, on-store namespace. This decoupling is achieved by (1) using an indexing technique that provides a decentralized and parallel directory path between GIGA+ clients and servers, and (2) using a layered implementation that allows GIGA+ servers to offload physical storage and on-disk representation of directory contents to a backend storage system. This chapter demonstrates the GIGA+ prototype scale and performance for configurations up to 64 servers.

The next three chapters focus on the indexing technique (discussed in Chapter 4 and 5) and the layering on backend storage systems (discussed in Chapter 6).

Chapter 4 and chapter 5 describe the distributed hash-table used in GIGA+ for indexing directories. This indexing scheme has two design principles — asynchronous server-side growth and inconsistent indexing state — used in GIGA+ both when the number

of servers is steady and when servers are being added to the system; these two design principles are discussed in Chapter 4 and Chapter 5 respectively. These chapters also present experimental and analytical evidence about various tradeoffs made by the design of GIGA+ . This analysis answers several questions: how do servers split a directory to harness the available parallelism in the system? how well-balanced is the load-distribution after GIGA+ indexing distributes directories over many servers? what is the cost-benefit of using inconsistent indexing state on GIGA+ clients and servers? under what circumstances is the GIGA+ directory distribution better or worse than existing techniques such as database indexing, consistent hashing and other relevant related work?

Chapter 6 studies the effect of layering GIGA+ indexing on different types of backend storage systems including local file systems and shared storage systems (emulated with NFS-mounted shared volumes). This chapter demonstrates how several aspects of GIGA+ indexing, such as hash partitions and cross-server splits, are dependent on the support available in the backend storage system. This chapter also shows how on-disk representation of file system directories provides highly variable performance, and shows how using a optimized data-structures are more suitable for directory-intensive workloads.

This dissertation concludes with Chapter 7 that summarizes the lessons learnt in this dissertation and highlights several areas that call for more future work.

Chapter 2

Background and related Work

This chapter presents a background on file system directories, and how prior work on distributed indexing has motivated the design of GIGA+ directories. Given the vast body of related work in out-of-core indexing, this chapter covers techniques with a decentralized design or a distributed implementation, and compares them with the GIGA+ design.

Section 2.1 presents an overview of traditional directories in local file systems, and Section 2.2 summarizes how networked file systems, both in enterprise environments and high-performance environments, manage file system metadata such as namespaces and directories. Section 2.3 describes hash-based data-structures and their distributed variants used in scalable storage systems. Although this dissertation is about file systems, Section 2.4 discusses why users of large directories find it hard to use database systems such as relational databases and key-value data stores.

2.1 Overview of file system directories

The original UNIX and BSD file systems were designed with the principle that everything, including a directory, is a file. Directories are special files whose contents are a tuple, called the directory entry, of file name and the i-node number of that file. Figure 2.1 shows how the file system directory tree (or the namespace) and individual directories are structured in traditional UNIX local file systems. In this figure, directory */b* has several directory entries, including a sub-directory *D1* and several files *F1* to *F4*. Each entry in the file corresponding to directory */b* is represented as a *{name, i-node number}* tuple. File *F1*, for example, has an i-node number 12345 which stores the file's attributes, such as size, permission, and owner, and pointers to disk blocks of file data.

Local file systems use either hash-tables or B-trees (or a combination of the two) for directory indexing. For example, SGI XFS directories use B-trees [Sweeney 1996], ReiserFS directories use a special tree-like index [Reiser 2004], Linux Ext file systems directories use hashed trees called H-trees [Cao 2007, Phillips 2001, Mathur 2007], and Oracle ZFS and Redhat GFS directories use extendible hashing [ZFS 2007, Soltis 1996].

2.2 Namespaces and directories in networked file systems

Networked file systems can be categorized as enterprise distributed file systems or cluster file systems. The former includes systems, such as NFS-based aggregation [Sandberg 1985], AFS [AFS 2013], and FARSITE [Adya 2002], designed for remote file access for workloads, such as user home directories and volumes, that have a very low degree of shared concurrent accesses compared to data-intensive applications. The latter includes sys-

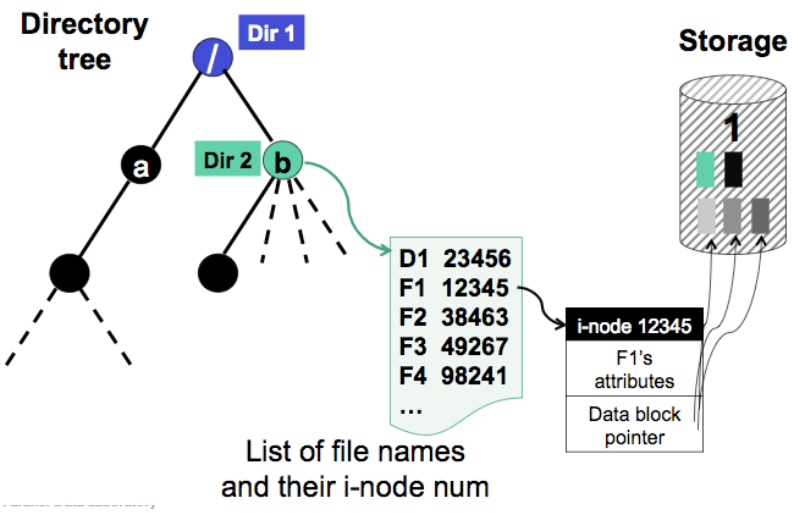

Figure 2.1 — Directory namespace in modern file systems.
This figure shows how directories are stored as regular files of tuples of directory entry name and i-node number.

tems, such as parallel file systems in scientific computing and Internet services, that deal with massive files and large numbers of concurrent accesses to a single file or directory.

Another difference between these two types of networked file systems is the way they store metadata. Typically, classic distributed file systems store the namespace of a user-visible mountable volume on a remote server that uses the i-node number of directory entries (like traditional directories in Figure 2.1) on the same server as the file associated with that entry. Cluster file systems, however, store all file system metadata including the namespace and directories on a metadata server. The i-node numbers in these directory entries are replaced by addresses of the location of data blocks stored on separate servers called data servers.

This section uses the following taxonomy to present a survey of how networked file systems manage the hierarchical namespace and individual directories on one or more servers.

2.2.1 Single-server for both namespace and directories

The first, and the most common category, consists of cluster file systems that use a single metadata server to manage and store the entire file system namespace including directories. Several recent file systems, such as Google file system [Ghemawat 2003], Hadoop distributed file system [Shvachko 2010] and Lustre file system [Lustre 2010b], use this approach. The main benefit of this approach is that it simplifies both administration and implementation of the metadata subsystem.

A single metadata server, however, has limited performance that may not scale as metadata-intensive workloads get larger in size. For example, early versions of the Google file system could handle only about 50 million files because their in-memory metadata management was insufficient to store metadata associated with more files [Fikes 2010]. One way to improve performance is to optimize metadata management on that single server. Lustre file system, for example, improved the throughput of their centralized metadata service using fine-grained locking of subtrees to enable highly concurrent namespace operations [Dilger 2012]. While this approach improves performance, it does not scale beyond the performance of one single metadata server.

2.2.2 *Multi-server namespace, single-server directories*

The next category includes file systems that distribute the namespace over multiple servers, but each directory is managed by only one server. This approach is common among both enterprise distributed file systems and cluster file systems; examples include PVFS [PVFS2 2010], Panasas's PanFS [Welch 2008], and an early version of FARSITE [Adya 2002].

A classic technique, used by Sprite [Welch 1992] and Panasas's PanFS [Welch 2008], is to rely on static name-based subtree partitioning where each subtree is assigned to a different server. This approach derived from the function of a mount table in client nodes is similar to NFS or CIFS based file aggregation techniques used in many enterprise systems [Sandberg 1985, CIFS 2013, NFSv4 2013]. This approach requires careful system administration to monitor load and size of different subtrees, and, if needed, migrate the hot-spot subtrees using manual configuration or automated re-balancing daemons.

One approach of re-distribution is adopted by the FARSITE file system that dynamically migrates heavily accessed subtrees using a hierarchical identifier structure used by the file system namespace [Douceur 2006]. Microsoft's FARSITE, a file system for an untrustworthy network of desktops, proposed a distributed directory service using file identifier-based metadata partitioning and fine-grained distributed locking to allow multiple writers to have concurrent access to an object (like a shared directory) [Douceur 2006, Adya 2002]. But FARSITE was built for desktop applications, not for data-intensive services; if used for the latter use-case, its distributed locking semantics will severely limit the scalability under highly concurrent workloads.

To alleviate load imbalance issues in static namespace partitioning, some file systems use multiple metadata servers. Examples of this approach include Lustre (particularly their proposed clustered MDS service) [Lustre 2010a, 2009], Intermezzo [Braam 1999] and Vesta [Corbett 1996]. They hash an object's pathname or identifier to control its placement on one of the metadata servers. If the hash function provides a statistically uniform distribution over a large key-space, this approach alleviates the server's responsibility for well-distributed file placement, and allows clients to directly hash and address the request to the appropriate server. The cost of this "implicit" load balancing is the loss of namespace locality: it is entirely possible that different sub-directories in a subtree and different files in a directory may all be assigned to different servers causing pathname lookups to address multiple servers. To improve pathname lookup performance, file systems have adopted several optimizations including namespace caches that store the prefix of recently accessed pathnames [Welch 1992], nonuniform randomization hash functions that adapt to changes in server load [Wu 2004], or lazy pathname traversals only during certain operations (such as changes to access permissions) [Brandt 2006]. Furthermore, directories and namespace do not adapt to any changes if certain parts of the namespace grow larger than others, resulting in a potentially imbalanced distribution.

GIGA+ uses a hashing based technique to distribute both the namespace and the files in a directory on multiple servers, and uses a similar optimization to speed up pathname lookups. This choice was driven by two factors: metadata-intensive workloads running at large scales suffer drastically in presence of a load imbalance, and the massive amount

of data managed by storage servers makes it undesirable to use data redistribution or migration in case of a hot-spot in the system.

2.2.3 *Multi-server namespace, multi-server directories*

The final category includes file systems that distribute both their namespace and large directories on multiple servers. The pioneering work in distributed directories is IBM's GPFS [Schmuck 2002] which is discussed in depth in the next section. OrangeFS [OrangeFS 2010, Yang 2011] and Ceph [Weil 2006a] also provide support to distribute large directories, while Lustre's clustered metadata service plans to provide this support in future releases of the system [Lustre 2010a, 2009].

OrangeFS, a commercially supported distribution of the PVFS cluster file system, has implemented a basic version of GIGA+ distributed directories in their recent release [Yang 2011, OrangeFS 2010]. However, the OrangeFS implementation uses a simplified version of GIGA+ that splits a directory *only once* and *over all available servers* when the directory grows beyond a certain size. This simplification allowed OrangeFS developers to quickly add the distributed directory feature in a beta version of their product; they plan to implement the incremental growth technique adopted in GIGA+ in future releases of OrangeFS [Ligon 2010].

Ceph is an object-based cluster file system that uses dynamic sub-tree partitioning of the namespace and hashes individual directories when they get too big or get too many accesses [Weil 2006a, 2004, 2006b]. While the experimental version of Ceph (from 2006) shows promising directory scalability, the recent versions of Ceph directory clustering have been described as "less stable" [ceph users 2013] and "buggy" [ceph devel 2013]

(and are often disabled by users). Compared to Ceph, GIGA+ facilitates dynamic server addition achieving balanced server load with minimal migration. It is unclear from the current Ceph documentation if (and how) Ceph directories are re-balanced after server additions.

Lustre is an object-based cluster file system that has clustered metadata for high availability, but stores all metadata centrally on a single server [Lustre 2010b]. However, Lustre has proposed a clustered metadata service that will split a directory using a hash of the directory entries only once over all available metadata servers when it exceeds a threshold size [Lustre 2010a, 2009] (this strategy is similar to OrangeFS's simple modification to GIGA+). 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. Experiments, conducted by NCAR in 2005, using the single server directory service, showed that Lustre scaled to about a few thousand creates/second [Cope 2005]. Preliminary experiments from 2005 suggest that an early version of Lustre's scalable metadata service can scale-up file creates per second [Studham 2005]. But Lustre has yet to release a stable, supported version of the proposed cluster metadata service [Dilger 2012, Zhen 2011]. Nevertheless, GIGA+ avoided using a split-once strategy because it stands the risk of splitting a rather small directory across all the servers, which may result in sub-optimal performance of directory scans. This strategy is also hard to optimize for new server addition without extensive metadata migration.

2.3 Distributed data-structures for indexing

Historically, variants of hash-tables and B-trees were designed for high-performance out-of-core indexing on a single machine, particularly for databases whose data did not fit in memory. As systems grew from running on one machine to running on many machines, these indexing techniques had to be decentralized: data was indexed on multiple machines such that B-tree leaves or hash-table buckets were stored separately from the pointers to these leaves or buckets. These distributed implementations had to deal with challenges related to performance, fault-tolerance, correctness and load-balancing. And all these issues are even more difficult when dealing with highly concurrent accesses to the same index. This section focuses on how distributed data-structures and their implementations handle these challenges.

One of the earliest design decisions that GIGA+ made was to use a hash-table instead of a B-tree. This decision was based on two factors — load balancing and range query support — that differentiate these data-structures and their relative importance to parallel applications that use cluster file systems.

At massive scales used in modern cluster computing, load-balancing is very important for overall system performance. Hash-tables implicitly provide good statistical distribution of keys into many buckets. Although B-trees have been used in many scalable distributed data stores [Chang 2006, HBase 2010, MacCormick 2004], the storage system has to provide explicit mechanisms to ensure that the leaves of a B-tree are split and evenly distributed across the cluster. The implicit uniform distribution properties of a hash functions alleviates this implementation burden on a cluster storage system.

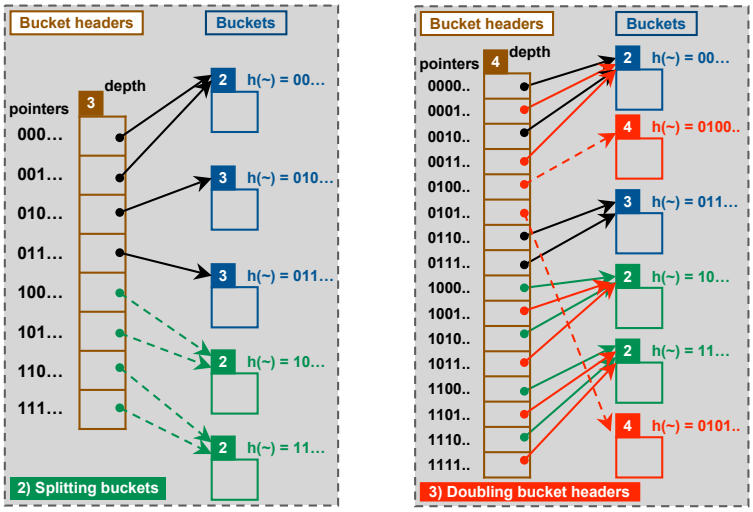

Figure 2.2 — Fagin's extendible hashing [Fagin 1979].
Two-level structure comprising of bucket pointers and bucket. Splitting overflow buckets doubles the header table of bucket pointers with new pointers pointing to buckets created from splits.

On the other hand, B-trees are better than hash-tables at performing high-performance range queries and scans. This property makes B-trees essential for databases that need to support high-performance scans that return data ordered on the key of a table. Most POSIX file systems, however, do not provide in-order range query semantics.[1] Current POSIX semantics for directory scans assume that *readdir*() scans will return unordered results and the application (such as ls) sorts the results in the desired order.

This section describes the details of three hashing data structures — extendible hashing, linear hashing and consistent hashing — that are widely used in distributed systems.

2.3.1 *Extendible hashing and IBM GPFS*

Extendible hashing is a dynamically growing hash table proposed by Ronald Fagin more than three decades ago [Fagin 1979]. Fagin's extendible hashing dynamically doubles the size of the table containing pointers to pairs of post-split links to the original bucket, and expands only the overflowing bucket (by restricting implementations to a specific family of hash functions) [Fagin 1979]. Figure 3.1 shows the two levels used by extendible hashing: at the bottom level, there are buckets that store the keys (or directory entries), and the top level comprises of a table of pointers to these buckets. The expansion algorithm doubles the pointer table in one step, with two bucket pointers pointing to each bucket, so looking up an entry through either pointer will find the entry when the shared bucket is scanned (linearly). It then splits the overflow bucket by creating a new bucket, transferring half the keys from the old bucket and updating one of the bucket pointers to point to this new bucket.

Concurrent access, particularly updates, to an extendible hash-table is limited by the speed and performance of the central node that serializes all updates to the bucket pointer table. Much prior research has focused on improving Fagin's original design using fine-grained locking mechanisms for highly concurrent updates [Ellis 1983, 1985, Kumar 1990]. Extendible hashing has been used in local file systems, including Oracle's ZFS [ZFS 2007], and distributed file systems, including Redhat's Global File System 2 (GFS2) [Soltis 1996, Whitehouse 2007] and IBM's GPFS [Schmuck 2002].

[1]NTFS is an exception here: it supports range scans for file system directories.

IBM's General Parallel File System (GPFS) uses Fagin's extendible hashing for its distributed directory implementation [Schmuck 2002]. GPFS is a shared-disk, high-performance file system that enables highly concurrent read and write access to blocks of data that are striped on many disks in the system. GPFS is one of the most widely deployed parallel file system in high-performance computing environments [Top500 2012]. Like traditional UNIX file systems, GPFS represents directories as a special file and stripes blocks of this file on multiple disks. Using extendible hashing for directory indexing, GPFS represents each hash-table bucket as a disk block and the pointer table as the block pointers in the directory's i-node. When a 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 uses a symmetric architecture where a cluster node can be a client or a server, and can understand the underlying block layout. This enables GPFS to use client cache consistency and distributed locking to facilitate concurrent access to blocks of a file including regular files and directories [Schmuck 2002]. Concurrent readers can acquire a shared reader lock for a directory from the lock manager and then cache these blocks for subsequent use. All directory reads are served from the cached blocks; thus, delivering high throughput for read-intensive workloads. Writers, however, need to acquire a exclusive write lock from the lock manager before updating the directory's blocks stored on shared-disk storage.

This combination of cache consistency and distributed locking imposes significant bottlenecks on concurrent write workloads such as application threads that are simultaneously creating files in one directory. Early versions of GPFS used whole directory

locking. When two threads create files in the same directory, the process of releasing (or acquiring) locks forces directory blocks to be flushed to disk (or read from disk). This disk I/O happens for both blocks containing directory entries and i-nodes containing block pointers. Experiments performed using these early versions of GPFS, by NCAR in 2005, show that read-only lookups scale very well but concurrent creates are limited due to the high overhead from lock contention and resulting disk I/O [Cope 2005].

Newer releases of GPFS have several optimizations to alleviate consistency and serialization bottlenecks during highly concurrent directory accesses. GPFS v3.2.1 and onwards use fine-grained directory locking (FGDL) that allows nodes to lock individual directory blocks instead of locking the entire directory [GPFS 2008, Schmuck 2010]. In addition, modifications to locking and cache consistency protocol reduce lock contention and disk I/O: once a writer acquires a exclusive lock to insert entries in a directory block, all other writers wanting to insert entries in that same block send their insert requests directly to the current lock holder. This reduces the disk I/O overhead by avoiding block reads and writes through the shared disk subsystem [Schmuck 2010].

Compared to early versions, these optimizations have improved GPFS performance for concurrent directory insertions [Frings 2011, Artiaga 2010]. GPFS, however, suffers from synchronously writing the directory's i-node, which contains the extendible hashing mapping state, and invalidating client caches to provide strong consistency guarantees [Schmuck 2010]. This simplifies fault tolerance and recovery, but lowers performance of file creates. GPFS users continue to report unsatisfactory file create rate in a single directory [Frings 2011, Calderon 2010, Artiaga 2010], compared to both other parallel file systems [Hedges 2010, Alam 2011] and other ad-hoc approaches [LRZ 2013].

Drawing from GPFS's lessons, GIGA+ takes a different approach: it strongly embraces asynchrony and inconsistency in all aspects of design. Directories in GIGA+ expand on many servers in parallel without system-wide serialization or synchronization. Mapping state, particularly at clients, is allowed to be inconsistent and out-of-date; servers can update this state eventually in an on-demand manner. GIGA+ also facilitates server additions in a non-blocking manner to minimize data migration and service disruption in an existing service.

*2.3.2 Linear hashing and LH**

Linear hashing is another re-sizable hash-table that grows by splitting its hash buckets in a linear order using a pointer to the *next* bucket to split [Litwin 1980]. Unlike extendible hashing, which shares a pointer table with clients, linear hashing only shares state about the *next* bucket to split. Linear hashing is used in many modern single-node databases including Postgre, MySQL and BerkeleyDB [Geschwinde 2002, MySQL 2013, Olson 1999], but not found in any (well-known) file system.

Linear hashing has a distributed variant, called LH* [Litwin 1993], that stores buckets on multiple servers and uses a central split coordinator that advances permission to split a partition to the next server. This split co-ordinator maintains a globally consistent and shared pointer, called split pointer, to the next bucket to split.

An attractive property of LH* is that it does not update a client's mapping state synchronously after every new split. Clients continue to use their stale mapping state which is updated when a server is addressed incorrectly (using the old mapping information).

GIGA+ uses similar mechanism that allows client caching of "location" information to become state and correct it on addressing an incorrect server.

A problem is that LH* can split only one bucket at a time and the next split operation has to wait until the previous one is completed [Litwin 1993, 1996]. This happens because the co-ordinator serializes all split operations to maintain a single true value of the split pointer and because LH* enforces a round-robin order of splitting partitions. An overflow bucket will often not split until its turn arrives, leading to a transient load imbalance.

Authors of LH* have proposed two optimizations that, in theory, alleviate these scalability bottlenecks [Litwin 1996]. The first optimization eliminates the need for a central split co-ordinator by using a *split token* that is passed among servers to decide which bucket to split. Unlike LH* with a split co-ordinator, which splits the next bucket if *any bucket* overflows, LH* without an co-ordinator can split the next bucket only if the *token-holder bucket* overflows. Assuming uniform distribution of hashing, this token-holder bucket will overflow at about the same rate as other buckets. This leads to cascading splits where many buckets split as soon as they receive the token, and resulting in long periods when many servers are busy splitting followed by long periods of no splitting.

The second optimization relaxes another LH* assumption, that a split starts only when the previous one terminates, using a client-shared variable that tracks the last committed (or completed) split operation [Litwin 1996]. But these pre-splits are made visible to the clients only when the split co-ordinator moves the committed pointer to the appropriate bucket. This optimization is further generalized to allow any bucket to start and finish a split at any time. But it still has several problems: split order remains serialized with a round-robin ordering, messaging traffic overhead grows for each op-

eration and implementation increases in complexity. LH* authors also claim that these optimizations need significant changes to the load control and client addressing mechanisms, and may lead to unbounded number of addressing errors [Litwin 1996].

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 [Litwin 1993, 1996]; GIGA+ allows any server to split a bucket at any time without any coordination. LH* offers a complex partition pre-split optimization for higher concurrency [Litwin 1996], 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.

2.3.3 Consistent hashing

Consistent hashing divides the hash-space into randomly sized ranges that are distributed on server nodes [Karger 1997]. A hash-space range is assigned to a server, typically, based on the hash of a server's identifier such as name or IP address [Stoica 2001]. Figure 2.3(A) illustrates how a 3-server configuration uses consistent hashing for data partitioning; server A, B and C each have three ranges indicated by different colors. A request for key X is addressed to the server (A) that holds the range of hash-space associated with the hash of the key $hash(X)$.

The main advantage of consistent hashing is how it re-distributes hash-space range when servers are added or removed from an online system. Consistent hashing is efficient at managing membership changes because addition or deletion of servers typically results in split or join of hash-ranges only for servers adjacent in the hash space. Figure 2.3(B) and (C) shows how consistent hashing re-distributes the hash-space range

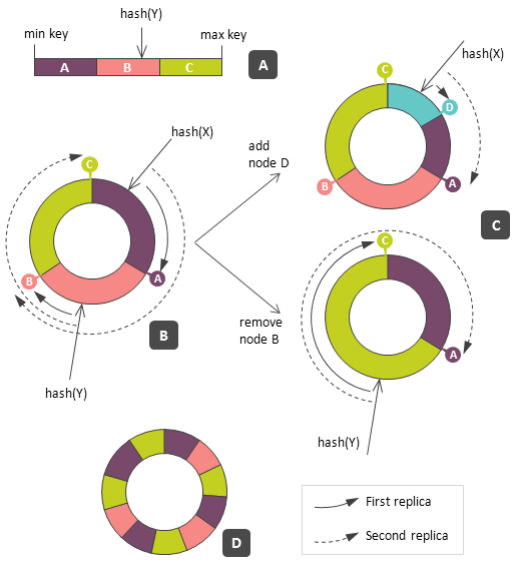

Figure 2.3 — Consistent hashing

(A) This figure illustrates a hash-space that is divided into three ranges assigned to three different servers (A, B and C) identified through different colors. (B) In this illustration, the neighboring servers also server as backups of the ranges held by other servers (C) When a new server D is added to the system, it is a given a random part of the hash-space range held by another server A. This random hash-space division leads to imbalanced assignment on each server. Similar imbalance may result if a server leaves the system; in this example, if server B leaves the systems, adjoining server C takes over its hash-space range and now controls two-third the range of the entire two-server configuration. (D) Hash-space distribution skew is alleviated by dividing the hash-space into more parts than the number of servers, and assigning multiple ranges to each server. (This figure was found on the Internet [Katsov 2012].)

when a new server D is added or when an existing server B is removed. It shows that new server D is randomly assigned a range in the hash-space based on where hash to server D's identifier lies in the hash-space. Server D then is responsible for a part of the hash-space range held by an adjacent server (A). Similarly when an existing server (B) is removed, another adjacent server (C) is responsible for the range held by the removed server. This property of handing server membership changes makes consistent hashing popular for wide-area peer-to-peer storage systems [Dabek 2001, Rowstron 2001, Muthitacharoen 2002, Rhea 2003]: these systems experience a much higher rate of change in server membership than cluster systems, and distributed hash-tables have used several optimizations to handle this churn [Rhea 2004, 2005]. Numerous cluster systems have also used consistent hashing for data partitioning [DeCandia 2007, Voldemort 2010, Basho 2013, Lakshman 2009], but have faced load-balancing issues resulting from random sized hash-space division [DeCandia 2007].

Figure 2.3(C) shows the imbalance in the hash-space range held by each server when a server is added or removed. This imbalance is worse in large-scale systems. Amazon's Dynamo uses consistent hashing to partition the key-values stored in a cluster wide storage system [DeCandia 2007]. Developers of Dynamo observed that consistent hashing's data distribution resulted in high load variance, even after using the *virtual servers* optimization [DeCandia 2007]. Virtual servers is a mechanism that divides the hash-space into more partitions than number of servers allowing each server to map multiple ranges to a single server [Stoica 2001, DeCandia 2007]. Figure 2.3(D) shows an example of how virtual servers divides the hash-space such each each server (indicated by a different color) has four hash-space ranges. Each server now has to maintain more mapping state

about the other hash ranges, and a newly added server needs to get its hash-range from more than one server. In contrast, GIGA+ uses threshold-based equal-sized splitting that provides better load distribution that consistent hashing.

Another problem with consistent hashing is that it implicitly assumes a very large data-set: this allows aggressive partitioning on many nodes *to begin with* and alleviates load imbalance in the distribution. This property, however, is not the best fit for workloads, such as file system directories, that have mix of mostly small data-sets and a few large data-sets. In such cases, multi-node partitioning is required only when data-sets grow large incrementally — an important design criteria for GIGA+ large directories.

2.4 Storage systems without a file system API

The primary use-case for scalable directories is small-file intensive workloads; this raises the question, *why not use a database, relational or key-value NoSQL, instead?* The main reason is that traditional relational databases lack support for running at scale with high parallelism, and modern key-value data stores lack a common general-purpose programming interface and semantics.

Traditional "one size fits all" relational database systems (RDBMS) do not meet the scalability and performance requirements of parallel data-intensive applications [Agrawal 2008, Seltzer 2008, Stonebraker 2007b]. RDBMS designs were conceived four decades ago, when online transaction processing workloads dominated the database market and when application logic also included data management logic, including physical layout and access methods. Over time RDBMS vendors have added numerous, tightly-coupled subsystems, such as lock hierarchies, buffer managements, query optimizers and concurrency

control, with a promise of better performance for diverse set of workloads. But these monolithic databases often lack the desired scale and performance, and they are also deemed too heavy-weight a solution for applications that require simple non-transactional semantics such as simple key-value lookups and write-once-read-many accesses [Stonebraker 2007b,c,a]. For instance, a recent study has shown that stripped down databases, without any locking, latching and other concurrency control features, can be about 20 times faster than the original system [Harizopoulos 2008].

To alleviate these concerns, new key-value data stores were designed *from scratch* using only those database semantics and functions that were required by target applications [Chang 2006, DeCandia 2007, Stonebraker 2009]. These data stores scale out, typically, by supporting only a small subset of an RDBMS's transactional ACID semantics [Seltzer 2008]. Different stores relax different semantics and offer different properties. Some stores, for example, limit atomicity to per-object or per-row mutations [Chang 2006, DeCandia 2007], while others relax consistency through eventual application-level inconsistency resolution or weak integrity constraints [DeCandia 2007, Stonebraker 2009], Moreover, an application programmed to use one data store may not be able to use another data store without significant re-implementation that uses a different interface. This is a significant shortcoming compared to using a file system.

Most general-purpose file systems, particularly cluster file systems used in supercomputing, offer the traditional UNIX file system interface and POSIX-like semantics. [2] This makes applications portable: they can run on any file system that supports the

[2]Several scalable file systems, such as the Google file system [Ghemawat 2003] and HDFS [Shvachko 2010], offer customized, non-POSIX interface and limited semantics targeted at only a handful of workloads and access patterns. Such purpose-built file systems are rare in supercomputing environments where

UNIX interface and POSIX semantics. Furthermore, most modern key-value data stores, such as Google's Bigtable [Chang 2006], Amazon's Dynamo [DeCandia 2007], and their open-source reincarnation [HBase 2010, Lakshman 2009], are built as user-level systems layered on existing file systems; their main purpose is to serve as middleware to support workloads, particularly small file workloads, that are not supported by the lower-layer file system. This dissertation takes a different approach: it pushes these extensions that support new workloads, such as highly concurrent directory accesses, inside the cluster file system. This approach benefits both cluster file systems and applications: a file system can support new features without changing the interface, thus making it compatible with legacy applications that can run without any modifications.

2.5 Summary

Table 2.1 summarizes how GIGA+ compares with the other distributed hash-based indexing techniques. The main facets of GIGA+ that differentiate it from prior work is the combination of unsynchronized concurrent partition splitting on servers and inconsistent mapping state at clients. GIGA+ also enables online server addition with minimal disruption, lazy configuration update and load re-balancing. Finally, GIGA+ decouples distributed indexing from on-store representation that facilitates easy deployment on existing cluster file systems

applications generate a wide-range of workloads that are often non known a priori; such applications are best served by a general-purpose file system. Although Microsoft NTFS is a very popular general-purpose file system running on most (Windows-based) computers, UNIX-based file systems with POSIX semantics remains the de facto choice for high-performance computing.

	Extendible hashing [Fagin 1979]	Linear hashing, LH* [Litwin 1980, 1996]	Consistent hashing [Karger 1997]	B-link trees [Lehman 1981]
Incremental, load balanced growth	Yes	Yes	No	Yes
Unsynchronized, concurrent splitting	No	No	Yes	No
Uses globally shared, consistent state	Yes	Yes	No	Yes
Allows inconsistent lookup at clients	No	Yes	Yes	No
Complexity of adding new servers	Unknown	Unknown	Easy	Hard
Distributed implementations	IBM GPFS	Unknown	Dynamo	Boxwood

Table 2.1 — Comparison with related work.

Chapter 3

GIGA+ file system directory service

Most modern cluster file systems, discussed in the previous chapter, have directory subsystems that either have no decentralized indices or have decentralized indices that require strong synchronization and consistency. Past user experience and experimental evidence has shown that these file system directories will not meet the growing scaling requirements of metadata-intensive workloads of massively parallel applications of the future [Ross 2006, Newman 2008, Raicu 2011, Kogge 2008].

This chapter introduces a new file system directory, called GIGA+ , that is designed *to push the scalability and concurrency of directory accesses, particularly when creating large numbers of files at high speed in a single directory*. GIGA+ uses a novel parallel indexing technique that is layered on existing unmodified storage systems. The rest of the chapter describes the GIGA+ architecture and its key components. This chapter also presents the scale and performance of the GIGA+ file system directory prototype for various configurations and workloads.

3.1 Overview of GIGA+ file system directories

GIGA+ file system directories are designed as a part of a user-level file system layered on top of existing, unmodified file systems. An alternative to this layered approach would be to modify an existing cluster file system to intrinsically support a distributed directory service.[1] However layering allows GIGA+ to provide an *incremental extension* to an existing cluster file system, and it has two key benefits.

The first benefit is the ease of development and deployment which has been well studied in prior work [Zadok 2006]. The ease of deployment is particularly important for GIGA+ because most target applications (that need scalable directories) run in environments where it is both unreasonable and infeasible to replace existing file systems easily. For such deployments, a "middleware" layer that sits between unmodified applications and existing cluster file systems is an attractive approach.

The second benefit is that layering allows GIGA+ to re-use, rather than re-invent, sophisticated techniques of real-world production cluster file systems. Building on decades of research in distributed systems, these systems have numerous features for high performance, better dependability, and simpler manageability. Most cluster file systems provide highly parallel techniques to read and write large amounts of data at high speeds [Ghemawat 2003, Hartman 1993, Anderson 1995, Gibson 1998, Lustre 2010b, Welch 2008, Schmuck 2002, Thekkath 1997]. These file systems also have sophisticated fault tolerance mechanisms to tolerate various types of failures transparently without affecting the active users [Thekkath 1997, Ghemawat 2003, Welch 2008]. Large-scale file systems also use

[1] GIGA+ techniques have also been implemented inside some releases of the OrangeFS cluster file system [OrangeFS 2010, Yang 2011].

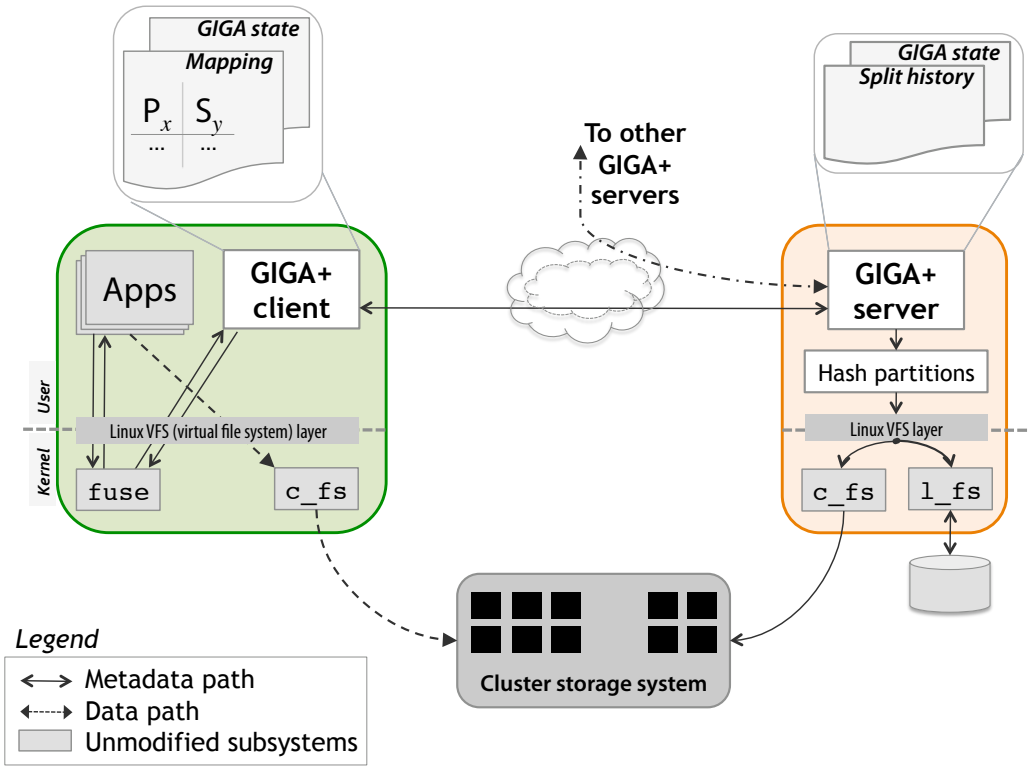

Figure 3.1 — GIGA+ system architecture.
GIGA+ provides a user-level distributed file system directory service layered on an unmodified file system.

system management tools for node membership and global configuration [Hunt 2010, Welch 2007, Burrows 2006]. GIGA+ is not intended to replace membership or fault tolerance or data access mechanisms; it avoids this where possible and re-uses them where needed.

Figure 3.1 shows the architecture of GIGA+ file system directories. It uses a client-server architecture and has three components: unmodified applications running on clients, the GIGA+ distributed indexing modules running on clients and servers, and a backend

persistent store accessed by the server (and sometimes by the client). Figure 3.1 shows two types of backends, a local file system (denoted by l_fs) and a cluster file system (denoted by c_fs).

By default, applications use the traditional UNIX VFS API, such as *open()*, *creat()* and *close()* calls, to interact with GIGA+ layer. GIGA+ is a mounted as a file system using the FUSE (File System in User-Space) toolkit [FUSE 2010]. Figure 3.1 shows how the FUSE kernel module intercepts VFS calls and redirects them to the user-level GIGA+ client process that sends the operation to an appropriate GIGA+ server process. The GIGA+ prototype uses a pathname-based high-level API offered by FUSE.[2]

GIGA+ uses a novel high-performance distributed indexing technique that algorithmically determines how large directories are divided over many servers and how clients address file system operations to the correct servers. GIGA+ clients send all metadata operations, such as *open()*, *create()*, *mkdir()* and *readdir()*, to a GIGA+ server whose address is determined by the indexing algorithm. The GIGA+ indexing algorithm also determines how a server services a client's requests. A GIGA+ server can service a request on its own or by collaborating with other servers.

In GIGA+ , the role of a server is only to manage a file system object like a directory or a file. The server does not store the object itself; it only manages access to the object. The GIGA+ server invokes the backend storage system to store the object persistently. In other words, a GIGA+ server can either either create (or delete) an object in the backend store or access an existing object from the backend store; depending on the choice of

[2]FUSE also offers a low-level API that uses i-node numbers. But it has a higher complexity because the user-level code needs to handle caching and i-node number management. The high-level API has an internal name lookup cache that simplifies implementation.

backend store, GIGA+ servers may or may not read or write data from the file. For example, GIGA+ servers can create a directory entry for a new file, change its attributes over the time, delete the directory entry for a new file, or return a file descriptor for an application to read or write.

The main idea of the layered GIGA+ design is to decouple directory indexing from backend storage and representation. GIGA+ manages two namespaces: a logical namespace used by applications and a physical namespace used on backend stores. Consequently, GIGA+ clients and servers are responsible for translating the application's logical namespace operation into the backend store's physical namespace operation. For example, if an application accesses a file */big-dir/f1.log*, GIGA+ may may physically store in a backend cluster file system at the pathname */mnt/clusterfs-store/d123/f567.log*. GIGA+ clients and servers work together to perform this application's namespace translation during the file system metadata operations.

Through this decoupling, GIGA+ may have different code-paths for accessing data and metadata depending on the choice of backend store. If the backend store is a local system, like a Linux file system, GIGA+ servers are responsible for both translating the logical namespace operations and performing the data operations using the physical namespace. But if the backend is a cluster storage system, such as a parallel file system, GIGA+ servers are needed only to translate the logical namespace to physical namespace. The result of this translation, which may be a symbolic link or a file handle, is passed to the GIGA+ client who can directly operate on the physical namespace through the cluster file system's module. Figure 3.1 shows how a cluster file system based backend (c_fs) enables GIGA+ clients to access the data through the physical namespace of a backend store

without going through the GIGA+ server. In contrast, a local file system (l_fs) forces both the data path and metadata path operations to go through the GIGA+ server.

By decoupling data path operations from metadata path operations, GIGA+ can inherit the scalable techniques for reading and writing file data that are already present in the underlying cluster file systems. Thus, the GIGA+ user-level file system serves as a middleware that allows an existing cluster file system, without a scalable metadata path, to provide high-performance metadata operations through distributed directory indexing.

3.2 Key components of the GIGA+ directory service

This section describes how the key GIGA+ components — indexing technique, GIGA+ clients and GIGA+ servers — interact with each other.

3.2.1 GIGA+ distributed indexing technique

GIGA+ uses dynamically re-sizable hash-based indexing that splits a directory into partitions and distributes these partitions on many servers. A directory in GIGA+ is created on a single server; as this directory grows in size, the GIGA+ server divides it into multiple partitions and places these partitions on different servers. Each partition has a range of the hash-space associated with it, and a partition holds only those directory entries that hash into its range. To access a file in a directory, GIGA+ clients hash the file name and send the request to the server that holds the partition responsible for the hash-space range corresponding to the hash of the file name. Two aspects of GIGA+ differentiate it from prior work on adaptive hash tables such as extendible hashing [Fagin 1979] and linear hashing [Litwin 1980]. First is how GIGA+ servers split and distribute partitions, and

second is how GIGA+ maintains partition-to-server mapping that allows clients to send requests to correct servers.

GIGA+ servers split partitions independently without any system-wide coordination. Each server manages only its own partitions, and makes independent decisions about when to split its overflow partitions. This enables GIGA+ servers to expand a growing directory in parallel without any total ordering of splits of overflow partitions and without any distributed locks during cross-server splits. This uncoordinated growth enables highly concurrent mutations of the GIGA+ directory index, but it also incurs two side-effects.

The first side-effect is that each GIGA+ server has only a partial view of the directory (i.e., a server only knows about the partitions it stores). To increase the set of known partitions, each GIGA+ server also maintains the history of every split operation performed on each of its partitions. The second side-effect is that the partition-to-server mapping changes frequently when servers are simultaneously splitting old partitions to create new partitions. To keep this mapping updated, every split will require the servers to notify all other servers and all clients (potentially thousands of them!) to update their mappings consistently. GIGA+ chose to avoid such a strong consistency for indexing state. (However, GIGA+ provides strong consistency for the application data.)

GIGA+ allows partition-to-server mapping state to get stale and out-of-date. As a result, a client's mapping state (i.e., client's view of the directory) may be different from the server's view. This causes a client to send its request to an incorrect GIGA+ server that no longer holds the partition with the desired hash-space range because that hash-space range was assigned to a different partition during a prior split. This incorrectly

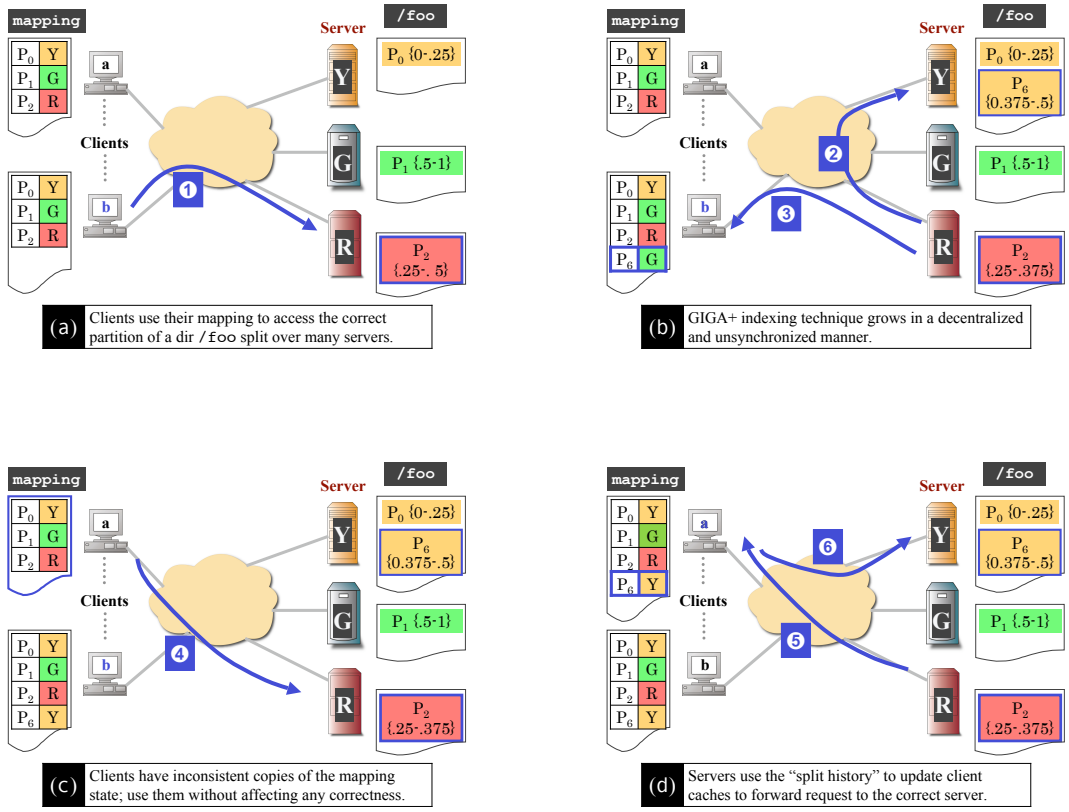

Figure 3.2 — Using GIGA+ indexing for inserts and lookups.

Directory /foo is divided into partitions that are distributed on three servers, such that each partition holds a particular range in the hash-space (denoted by $\{x - y\}$). (1) Client b inserts a file test.log in the directory by hashing the filename to find the appropriate partition using their partition-to-server mapping. Assuming $hash(\text{"test.log"}) = 0.4321$, the filename gets hashed to partition P_2 and the request is sent to server R. (2) Server R receives the insert request and finds that partition P_2 is full. Using the GIGA+ split mechanism (described in Chapter 4), it splits P_2 to create a new partition P_6, with half the hash-space range, on server Y. This example assumes that file test.log is also moved to P_6. (3) Once the split is complete, server R sends a response to client b who updates its partition-to-server map to indicate the presence of P_6 on server Y. (4) Other clients are unaware of this split and their mapping becomes inconsistent, but GIGA+ continues to use this stale mapping state. Client a issues a lookup on test.log and its out-of-date mapping indicates (incorrectly) that the entry is located on P_2 on server R. (5) The "incorrect" server R receives client a's request and detects from its split history that the desired hash-space range been moved to another partition P_6 on server Y. Server R uses the split history of P_2 to update client's stale mapping. (6) Client a then sends its request to the "correct" server.

addressed server uses the split history of the incorrectly addressed partition to provide a *lookup hint* to the client. On receiving this hint, a client updates its stale mapping state and sends the request to another server.

Figure 3.2 presents an example of insert and lookup operations on a GIGA+ distributed directory. The example in this figure shows a directory */foo* that is managed by three servers, *Y*, *G*, and *R*. Figure 3.2 uses the notation $P_i\{x-y\}$ to denote the range of the hash-space $\{x-y\}$ held by a partition P_i. As the directory grows in size, partitions get filled up and GIGA+ servers *split* them into two by transferring the second half of the hash-space range to the new partition. Chapter 4 and 5 describe the details of how GIGA+ servers split partitions in parallel and how GIGA+ clients can lookup entries without full knowledge of server-side expansion.

3.2.2 Clients in GIGA+

GIGA+ clients are stateless: they do not cache any file system data or metadata, but only keep (soft) state associated with directory indexing. Because GIGA+ is targeting concurrently shared directories with up to billions of files, caching such directories at each client is impractical: the directories are too large and the rate of change too high. GIGA+ clients do not cache directories and send all directory operations to a server. Directory caching only for small rarely changing directories is an obvious extension employed, for example, by PanFS [Welch 2008], but it may require careful engineering to handle cases of write sharing. The current GIGA+ prototype chose to avoid the significant complexity of implementing cache consistency protocols, and instead incurs the cost of one client to

server message for small directories.[3] In fact, production cluster file systems, like PVFS, also avoid data caching on the client because they do not want to incur the complex implementation and unpredictable performance of a distributed cache consistency mechanisms [PVFS2 2010].

GIGA+ clients only keep indexing state, such as partition-to-server mapping, number of partitions and list of servers, associated with directories that are presently being accessed. A GIGA+ client uses this indexing state to determine the server that holds the partition associated with the application's file request. Furthermore, all indexing state can be stale and out-of-date; GIGA+ indexing can tolerate inconsistent mapping state and resolve the inconsistency in a lazy manner (described in details in Chapter 5).

This soft state also makes it easy to handle GIGA+ client failures. If a client reboots, it can only lose operations in flight and the indexing state with respect to GIGA+ directories. GIGA+ relies on 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 because a GIGA+ client with no state is treated as a client with stale and inconsistent state which is eventually updated by GIGA+ servers.

[3]In many HPC deployments, clients are considered "ephemeral" where they may be middleware libraries or MPI programs that may come or go more seamlessly than server-side processes.

3.2.3 *Role of GIGA+ servers*

The primary purpose of a GIGA+ server is to *manage* partitions by synchronizing and serializing interactions between clients and partitions. Note that GIGA+ does not use any system-wide, multi-server synchronization or serialization; GIGA+ servers only ensure correct and safe access by thousands of clients to *a given partition*. A GIGA+ server does not store partitions, but it manages access to them. GIGA+ servers map logical hash partitions to be stored in an underlying backend storage system. Partitions can be stored in a backend store that is either local to a server, such as Linux file systems or a key-value store, or shared across and accessible from all servers, such as a cluster file system or federated NFS servers. GIGA+ servers rely on the backend store to control how a partition is represented on persistent media. A store can represent a partition as a regular directory or as flat files or as a specialized data-structure. This approach allows GIGA+ to decouple indexing from on-disk representation.

GIGA+ servers are also responsible for splitting overflow partitions using a multi-step process. First, the GIGA+ server takes a local lock on the partition being split. The server then reads the partition from the backend store, scans all entries to find entries that will migrate to the new partition, and creates the new partition with the appropriate entries. Next, the server initiates a cross-server migration by sending the new partition to another server that will be responsible for it. Once the migration completes successfully, both servers, the split initiator and the split recipient, update their mapping state. Finally, the initiator server releases the lock on the overflow partition. The cur-

rent GIGA+ prototype blocks inserts into the partition undergoing split; it led to a much simpler implementation than using a sophisticated fine-grained locking scheme.

One challenge with splits is the overhead of migrating data entries. This cost is dependent of the choice of the underlying backend store. If a GIGA+ server uses a local backed store, splits need to move both the directory entries and the file data associated with those entries. This makes splits much more expensive than a backend that is shared among servers. For shared storage backends, such as a cluster file system, splits only move the directory entries without moving the file data on the data servers. These split-related tradeoff are discussed in details in Chapter 6.

Another challenge in cross-server splits is failure-free operation of splits. Ideally, splits would require support for distributed transactions for atomicity in face of failures [Gray 1992]. The current GIGA+ prototype does not implement this complex mechanism and relies on reusing the backend cluster storage system's tools for reliable cross-server migration during splits [Sinnamohideen 2010]. More generally, much of the failure handling in GIGA+ is dependent on the functionality of the underlying backend stores, particularly cluster file systems.

Modern cluster file systems scale to sizes that make fault tolerance mandatory and sophisticated [Ghemawat 2003, Welch 2007, Braam 2007]. With GIGA+ integrated in a cluster file system, fault tolerance for data and services is already present, and GIGA+ does not add major challenges. Reliability issues, such as on-disk representation and disk failure tolerance, are a property of the existing cluster file system's directory service, which is likely to be based on replication even when large data files are RAID encoded [Welch 2008]. Moreover, if partition splits are done under a lock over the entire partition, which

is the current state of GIGA+ prototype, the implementation can use a non-overwrite strategy with a simple atomic update of which copy is live. As a result, recovery becomes garbage collection of spurious copies triggered by the failover service when it launches a new server process or promotes a passive backup to be the active server [Burrows 2006, Hunt 2010, Welch 2007].

While the GIGA+ design assumes that is integrated into a full featured cluster file system, it is possible to layer GIGA+ as an interposition layer over and independent of a cluster file system, which itself is usually layered over multiple independent local file systems [Ghemawat 2003, Shvachko 2010, Welch 2008, PVFS2 2010]. Such a layered GIGA+ prototype would not be able to reuse the fault tolerance services of the underlying cluster file system, leading to an extra layer of fault tolerance. The primary function of this additional layer of fault tolerance is replication of the GIGA+ server's write-ahead logging for changes it is making in the underlying cluster file system, detection of server failure, election and promotion of backup server processes to be primaries, and re-processing of the replicated write-ahead log. Even the replication of the write-ahead log may be unnecessary if the log is stored in the underlying cluster file system, although such logs are often stored outside of cluster file systems to improve the atomicity properties writing to them [Chang 2006, HBase 2010].

To ensure load balancing during server failure recovery, the layered GIGA+ server processes could employ the well-known chained-declustering replication mechanism to shift work among server processes [Hsaio 1990], which has been used in other distributed storage systems [Lee 1996, Thekkath 1997]. 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 [Hsaio 1990]. Some systems have used chained declustering to spread "hot" partition reads across its two servers [Lee 1996, Thekkath 1997], but hashing in GIGA+ 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 [Burrows 2006, Welch 2007]. 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 [Ghemawat 2003].

3.3 Evaluation of scale and performance

This section presents the experimental evaluation conducted to measure the scale and performance of the GIGA+ file system directory prototype; it begins by describing the experimental setup, including details about the cluster resources and configurations of different storage backends used by GIGA+ servers, and then reports the measured performance of the GIGA+ prototype for various workloads. Detailed analysis of the design trade-offs made in GIGA+ are presented later in Chapters 4-6.

3.3.1 Experimental setup

All experiments are performed on a 128-node compute cluster, called Marmot, available through the PRObE initiative [PROBE 2012]. The cluster comprised of machines running the Linux 2.6.32-24-generic kernel (Ubuntu release) on hardware described in Table 3.1. Each machine in this test cluster has one disk that is managed by a Linux file system.[4]

For experimentation, GIGA+ prototype was layered on different types of backend stores:

Linux local file systems — Most large clusters use local file systems to manage on-disk storage, even when there is a higher-layer distributed storage system that manages data access, data placement, and data migration. Many of the largest file systems in the world rely on Linux local file systems such as Ext3/4, Reiser and XFS. For example, the Google file system [Ghemawat 2003] and its open-source reincarnation Hadoop distributed file

[4]Real world cluster systems use more than one disk per machine [Dean 2009].

CPU	Dual Socket, Single Core AMD Opteron
Memory	16 GB (8GB per core)
Disks	WD Black SATA 7200 RPM 2TB disk
Network	Gigabit Ethernet connected to Black Diamond 6808 switch

Table 3.1 — Specifications of the cluster used for experimental analysis. Experiments were performed in a dedicated setup where no competing services were using the machines. This cluster uses the Emulab tools for machine setup [White 2002, PROBE 2012].

system (HDFS) [Shvachko 2010] rely on multiple disks on each server that are managed by local file systems. The Lustre file system was layered on nodes running Linux Ext4 but is moving to run on ZFS [Dilger 2012]. Some cluster file systems, like Panasas's PanFS [Welch 2008] and Ceph [Weil 2006a], build on specialized object-based local file systems.

Networked file systems — Networked file systems comprise of a client-server model where applications make their file I/O requests through a file system client which communicates to the remote servers. Popular networked file systems often provide a kernel-mode client that can rely on functionality, such as auto-mounting and VFS layer indirection, provided by UNIX file systems [Sandberg 1985]. Some file systems, such as PVFS [PVFS2 2010], Ceph [Weil 2006a], and HDFS [Shvachko 2010], also include a library interface to link directly against applications.

Database-like local stores — The last few years have seen an emergence of purpose-built database-like storage systems that store and access data using data structures optimized for high-throughput read and write performance. These systems typically offer simple APIs like *put()* and *get()* operations and lightweight semantics required by the anticipated workloads. Examples of such stores includes LevelDB [LevelDB 2012], which uses

LSM-trees [O'Neil 1996], TokuDB [Tokutek 2010], which uses fractal trees [Bender 2007], and Acunu [Acunu 2011], which uses stratified B-trees [Twigg 2011].

Experiments in this chapter use a synthetic workload generator tool, called *mdtest* [MDTest 2010], that is used by many parallel file system vendors and users [Hedges 2010, Alam 2011].[5] This tool generates several types of workloads that perform *weak scaling* where each server in the system performs a fixed amount of work and as the system grows bigger the amount of work it does also increases proportionally. In the context of GIGA+ directories, an N-server GIGA+ configuration handles a directory with N million files; N varies from 1 server to 64 servers. Thus, a single server manages 1 million files, a 2 million file directory is created on 2 servers, a 4 million file directory on 4 servers, up to a 64 million file directory on 64 servers. GIGA+ uses eight application threads on a single GIGA+ client to create files on one GIGA+ server. That is, for an N-server setup, an experiment uses N GIGA+ clients each running 8 application threads that perform directory operations such as file creates and lookups.

The primary workload is a create-intensive workload that creates large numbers of files concurrently in a single directory. This is the most important workload and use-case for this research. The second workload is a lookup-intensive workload that performs a *stat()* on random files in a large directory distributed on multiple GIGA+ servers. Several other workloads, such as directory scans and mixed workload with creates and lookups, are also used to evaluate GIGA+ performance; results from such workloads are presented in subsequent chapters of this dissertation.

[5]Earlier version of GIGA+ also used another benchmark program called *metarates* that provides similar functionality using MPI-IO based workload generation [Metarates 2010]

In all experiments in this dissertation, the workload generator tool generates filenames that are random. However, the nature of filenames does not matter because GIGA+ hashes the filename for all operations. All workloads are generated on an empty file system, unless specified. Finally, this experimental evaluation has not tuned the workload for optimal efficiency because the focus of this evaluation is to understand the behavior of the GIGA+ servers and clients. If an experiment changes these assumptions, it is described explicitly in the setup of that experiment.

3.3.2 Scale and performance

The first experiment measures the baseline performance of various GIGA+ configurations and file systems by creating one million files in a single directory using the *mdtest* tool. Table 3.2 compares the number of files created per second in a single directory for several local and networked file systems. This table shows file systems that were readily available and easily accessible for this experiment.

The GIGA+ setup uses two machines, one client and one server. A directory stored on the GIGA+ server does not split because there were no other servers. The *mdtest* tool creates a million files using two methods: first, all file creates went through the VFS API, and second, all file creates went through a non-POSIX library API that was created by directly linking GIGA+ into the application. The library approach allows *mdtest* to use custom object creation calls (called *giga_creat()*) avoiding system call and FUSE overhead [Rajgarhia 2010] in order to compare to *mdtest* directly in the local file system.

For the local file systems, *mdtest* creates files using the UNIX file system API directly in the file system. As expected, both ReiserFS and Linux ext3 deliver high directory in-

	Storage systems that store a directory on one server	*Files created per second in one directory*
Local storage systems	Linux Ext3	16,470
	Linux ReiserFS	20,705
	Linux XFS	1,275
Networked file systems	NFSv3 client-server	521
	Hadoop distributed file system	4,290
	PVFS cluster file system	1,064
GIGA+ client-server setup (one partition on server)	ReiserFS backend (via VFS/FUSE)	5,977
	ReiserFS backend (via library API)	17,902
	LevelDB backend (via VFS/FUSE)	3,760

Table 3.2 — Single node file create rate on different backends.
An average of five runs of running the *concurrent create* workload to create a 1-million file directory from scratch. The standard deviation across five runs of each experiment was too negligible (less than 1%) to be reported.

sert rates.[6] All file systems were configured with commonly recommended parameters for metadata-intensive workloads such as enabling the *-noatime* and *-nodiratime* options. Linux Ext3 used write-back journaling and the *dir_index* option to enable hashed-tree indexing [Cao 2007], and ReiserFS was configured with the *-notail* option, a small-file optimization that packs the data inside an i-node for high performance [Reiser 2004].

Table 3.2 shows that GIGA+ with *mdtest* running on a remote machine using the library interface at the client and ReiserFS backend on the server creates 17,902 files per second —this is about 80% the rate of local ReiserFS configuration in which *mdtest* creates 20,705 files per second in a local directory in ReiserFS. Using the library API in the

[6] XFS was extremely slow during the create-intensive workload. Although XFS provides great performance for reading and writing large files, other metadata-intensive experiments have experienced similar metadata performance issues with XFS [Wheeler 2010].

GIGA+ client-server setup requires one RPC message for each file create. This comparison shows the RPC messaging penalty incurred by GIGA+ client-server setup. The goal of creating a library version was only to compare GIGA+ efficiency to local file systems; all other experiments are performed using the VFS/FUSE interface.

However, creating a single file in GIGA+ through the VFS interface requires three RPC messages. These RPCs are a result of how FUSE creates files: each create first performs a *getattr()* to check if a file exists, followed by the actual *creat()* call, and finally another *getattr()* after creation to load the created file's attributes.[7] Each of these calls results in a RPC message from the GIGA+ client to the server. As expected, Table 3.2 shows that using the VFS/FUSE interface causes GIGA+ to run three times slower that the library API.

Table 3.2 compares the single GIGA+ server setup with networked file systems that store a directory on a single machine. It shows the file create rate for the open-source PVFS cluster file system, an NFSv3 filer, and the Hadoop distribute file system (HDFS). Both HDFS and PVFS are cluster-scale systems with support for reliability and fault tolerance; HDFS uses a write-ahead log and replication in software, while PVFS relies on the storage nodes to provide reliability using RAID. Because the current GIGA+ prototype has not implemented reliability and fault tolerance mechanisms, it would have been unfair to compare it with HDFS and PVFS with their fault tolerance mechanisms turned on. For a more enlightening comparison, HDFS and PVFS were configured to be functionally equivalent to the GIGA+ prototype. Specifically, in this experiment, the write-ahead log and replication was disabled in HDFS. PVFS was also configured to use its default mode

[7] Several other file systems have also observed that aggressive attribute checking in FUSE affects the system performance [Weil 2006a].

which has no redundancy unless a RAID controller is added. Because the NFSv3 filer was used in production, this experiment was not able to disable its RAID redundancy and hence is slower than it might otherwise be. Table 3.2 also shows that GIGA+ directories using the VFS/FUSE interface outperforms all three networked file systems; however, GIGA+ is a skeletal prototype that has a much simpler code path and worse reliability support than the production networked file systems compared in this table.

The next experiment uses GIGA+ configurations from Table 3.2 to evaluate the scalability of distributed directories when GIGA+ is configured to use LevelDB and ReiserFS. The main difference between these backends is the way hash partitions are stored in persistent store. LevelDB stores partitions in a single LevelDB table of key-value pairs and ReiserFS stores partitions as file system directories. Chapter 6 discusses how different backends interact with the GIGA+ directory indexing technique.

Figure 3.3 reports the scalability of GIGA+ with LevelDB based backend store. It plots the increase in average steady-state throughput (on Y-axis) as the number GIGA+ servers doubles (on X-axis). Figure 3.3 has two graphs: the left graph shows the scalability of file creations in a single directory and the right graph shows the scalability of file lookup operations in a single directory. The file creation performance was measured by creating large number of files, one million files on each GIGA+ server, in an empty directory using the weak scaling setup methodology: an N-server configuration stores a directory with N million files created concurrently by a total of N remote client machines (each with eight workload generating *mdtest* threads). The lookup performance was measured in a similar manner: N clients issued *stat()* requests for 25% randomly chosen files in a large directory that contains N million files in a directory on N servers. For the lookup exper-

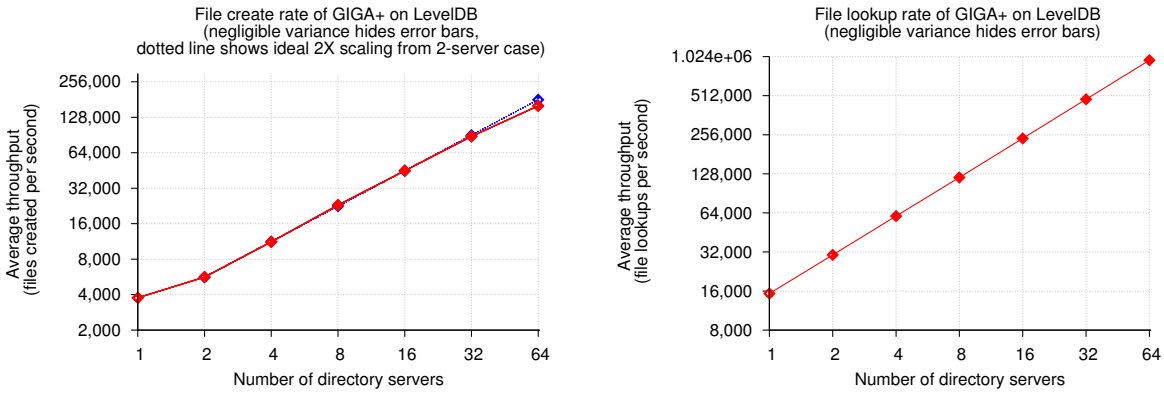

Figure 3.3 — Scale and performance of file create and lookup rate of GIGA+ with LevelDB backend store.

This figure shows how GIGA+ with LevelDB scales the file create rate (left graph) and file lookup rate (right graph) with different number of servers. This configuration delivers a steady-state throughput of roughly 160,000 file creates per second for a directory with 64 million files striped on 64 servers — surpassing the most stringent file creation requirements in high-performance computing [Newman 2008].

iment, the file system was mounted and re-mounted after the directory was created and before lookups were issued.

Figure 3.3 shows that GIGA+ on LevelDB backend scales linearly up to the size of 64-server configuration, and can sustain an average file creation rate of 160,000 file creates per second in a single directory —this exceeds some of the most rigorous scalability demands in supercomputing [Newman 2008]. The right graph in the figure shows the scalable lookup performance of GIGA+ directories; GIGA+ can sustain a little less than 1,000,000 file lookups per second in a directory with 64 million files stored on a 64-server configuration. 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.

Figure 3.4 shows the scalability of GIGA+ layered on a ReiserFS backend and compares them with other systems with support for scalable metadata. The experiments in this figure are performed on a different cluster than the one used in Figure 3.3. This cluster has 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 regular directories in a ReiserFS on one 7200rpm SATA disk. The experimental methodology is same as the previous scaling experiment except that it is at a smaller scale: each server has 400,00 files (an N-server configuration has $0.4N$ million files) and a maximum of 32 servers. Despite the minor differences in the clusters and experiments, Figure 3.4 shows that GIGA+ with ReiserFS backend provides scalability for up to 32 servers; it delivers a peak throughput of about 100,000 file creates per second. The difference in scalability between the LevelDB based

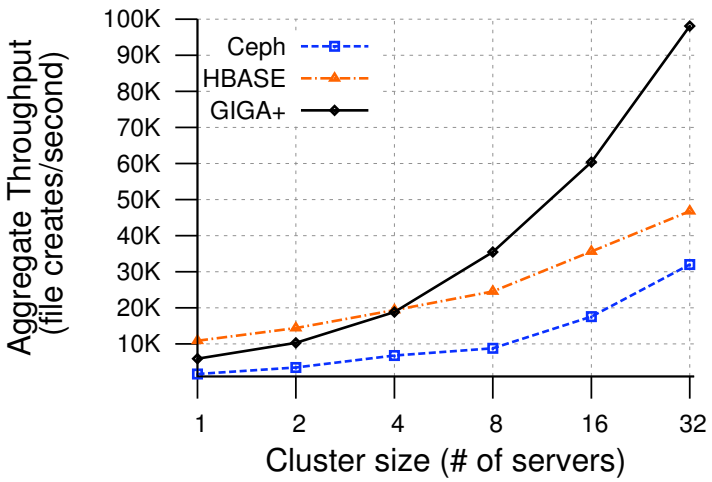

Figure 3.4 — Scale and performance of file create rate of GIGA+ layered on ReiserFS backend.

This graph shows the scalability of file create rate in large directory striped on varying number of servers. For the 32-server configuration, this GIGA+ prototype delivers a little less than 100,000 file creates per second when creating a directory with about 13 million files. This figure also compares GIGA+ with a modified version of HBase [HBase 2010] and experimental performance reported by the Ceph file system [Weil 2006a].

backend and ReiserFS based backend are explored in details in Chapter 6 after the reader has learnt the trade-offs made by the GIGA+ indexing technique.

Figure 3.4 also compares GIGA+ with the scalability of the Ceph file system [Weil 2006a] and the Apache HBase distributed key-value store [HBase 2010]. For Ceph, Figure 3.4 reuses numbers from their original paper that reported experiments from a cluster that used dual-core 2.4GHz machines with IDE drives, with equal numbered separate nodes as workload generating clients, metadata servers and disk servers with object stores layered on Linux ext3 [Weil 2006a].

HBase is used to emulate Google's Colossus file system that stores file system metadata in BigTable instead of internally on single master node [Fikes 2010]. Figure 3.4 shows performance for an HBase configuration on a 32-node HDFS configuration with a single copy (no replication) and two parameters disabled: blocking while the HBase servers are doing compactions and write-ahead logging for inserts (a common practice to speed up inserting data in HBase). This configuration allowed HBase to deliver better performance than GIGA+ for the single server configuration because the HBase tables are striped over all 32-nodes in the HDFS cluster. But configurations with many HBase servers scale more poorly than GIGA+ (which has a much simpler code path).

3.4 Summary

This chapter presented a high-level overview of the GIGA+ distributed indexing scheme, and the roles of GIGA+ client and server of a distributed file system directory service that is designed to be layered on existing backend storage systems. This design is driven by two goals set at the start of this research: (1) to extend existing cluster file systems to be

able to support scalable directory operations, and (2) to push the limits of scalability for concurrent accesses, particularly file inserts, in a single directory.

Experimental evaluation of the GIGA+ directory prototype shows that the throughput of file creates and lookups scales for configurations up to 64 servers and that the achieved throughput exceeds the challenging requirements, particularly for file create rate in a directory, of the high-performance computing community that motivated this research.

So what makes GIGA+ scale? The next few chapters answer this question by describing the design decisions and analyzing the tradeoffs made by the GIGA+ directory service to push for higher scalability and greater concurrency.

- Chapter 4 describes how GIGA+ servers scale-out a rapidly growing directory across many servers in a highly concurrent and load-balanced manner.
- Chapter 5 shows how GIGA+ clients lookup partitions in GIGA+ servers in the face of uncoordinated server-side directory expansion without using strong consistency of the indexing state.
- Chapter 6 studies the interaction of the indexing technique with the underlying backend store to understand how GIGA+ can efficiently leverage the strengths of the backend store.

Chapter 4

Asynchronous scale-out growth

GIGA+ is a concurrent indexing technique that divides a large directory into partitions and distributes these partitions across multiple servers. The design of GIGA+ has two distinguishing tenets — (1) splitting and distributing directories on servers asynchronously, and (2) allowing inconsistent and out-of-date partition-to-server mapping information. They are discussed over the next two chapters.

This chapter pertains to the first tenet: *asynchronous growth*. It describes how the index grows incrementally with the size of a directory and scales out on many servers concurrently without any global, system-wide synchronization and serialization. GIGA+ distributes large directories on servers such that each server only has a partial view of the directory index; this allows GIGA+ servers to make independent and concurrent decisions to expand the index. Furthermore, GIGA+ preserves this non-blocking property when new servers are added to the system: it spreads the existing load on new servers in a manner that minimizes data migration and achieves well-balanced load distribution.

This chapter also analyzes the cost-benefit of splitting large directories and the trade-offs of splitting directories at different rates. Finally, this chapter concludes with how GIGA+ mitigates imbalance for small directory workloads.

4.1 Incremental, hash-based partitioning

All directories are managed by GIGA+ servers: a directory, d, is created by the server that handles the *mkdir(d)* operation and all subsequent accesses to d are sent to the GIGA+ servers responsible for the directory's partition. A successful *mkdir()* operation creates an empty directory with a single partition that stores all entries created in that directory. In GIGA+ , when a directory is created and is small in size, it is represented by a single partition that is managed by one server. When a directory grows big, GIGA+ splits the directory into multiple partitions and distributes them on more servers.

This incremental growth property, i.e. the ability to divide a directory in proportion to its size, is crucial for small directory performance. Most file system directories start small and remain small [Dayal 2008, Agrawal 2007]. Figure 4.1 shows the distribution of directory size (measured in number of entries) found in large-scale file system deployments in a prior study [Dayal 2008]. This figure shows that 99.99% directories contain fewer than 8,000 files. GIGA+ uses this number to define small directories and the split threshold: a directory with fewer than 8,000 entries is a small directory with one partition, and this partition can be split into more partitions once it has more than 8,000 entries.

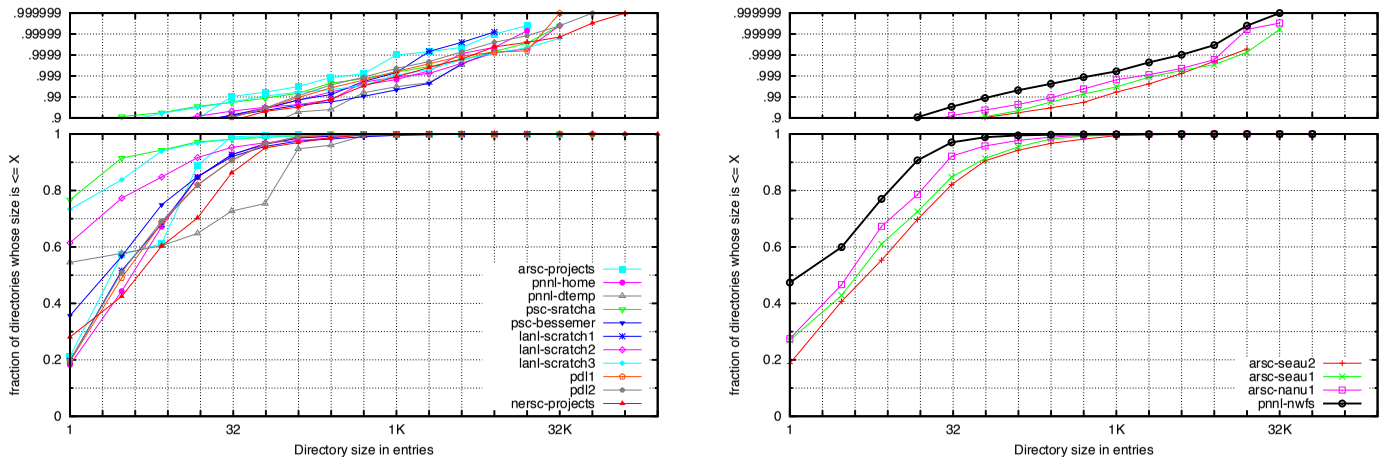

Figure 4.1 — Distribution of number of entries in directories in real-world file systems.
This figure shows a cumulative distribution function (CDF) of directories whose size (measured in number of entries) is less than a given size in various file systems deployments in a prior study [Dayal 2008]. The legend on these graphs corresponds to various file systems analyzed in this study [Dayal 2008]; the observed systems includes non-archival file systems comprising mainly scratch, project and home volumes from high-performance computing sites, and two volumes from departmental file servers at CMU. More details about these file systems can be found in the original study [Dayal 2008]. The X axis is log scale of base 2. The Y-axis is split in two sections: a linear scale from 0 percentile to 100 percentile at the bottom and a log scale from 90 to 99.9999 percentile at the top. The median across all file systems ranges from 0 to 8 entries in a directory, i.e. 50% directories are 0 to 8 entries in size. 90% directories have 0 to 128 entries and 99.99% directories have fewer than 8,000 entries. **GIGA+ classifies any directory fewer than 8,000 entries as a small directory and uses this as the maximum size of a directory partition (all overflow partitions are eligible to be split into smaller partitions).**

Since only a few directories grow to really large sizes, incremental growth allows GIGA+ to avoid degrading performance of small directories. Striping small directories across multiple servers will lead to inefficient resource utilization, particularly for directory scans. A *readdir*() on a small distributed directory will force all servers to scan their respective portions of the directory. These small scans will be dominated by disk seek latency instead of data transfer latency. Perhaps a bigger benefit of incremental growth is that it allows GIGA+ to handle adding new servers with minimal data re-distribution. We discuss both scanning efficiency and adding servers in Section 4.3 and Section 4.5 respectively.

GIGA+ uses hash-based partitioning where each directory partition corresponds to a range in the hash-space. When a directory is created, it has a single partition that holds the entire hash-space range. GIGA+ prototype uses the cryptographic 128-bit MD5 hash function [Rivest 1992]. Thus the single partition holds the entire hash-space range from 0 to $2^{128} - 1$; which is denoted in the rest of this dissertation as $(0, 2^{128} - 1]$. GIGA+ does not require the cryptographic properties of MD5; in fact, it can use any hash function that produces hash values that are uniformly distributed in the hash space for any distribution of unique keys. This property of a hash function is important to allow GIGA+ to load balance keys over partitions. If a large directory is divided in N partitions, each with $1/N$-th the hash-space range, then the hash function should statistically distribute the same number of keys in each partition.

Each filename (contained in a directory entry) is hashed and then mapped to a partition that holds the necessary hash-space range. For a small directory with a single partition, all files are hashed and inserted to that partition. When a partition overflows (after

it has more than 8,000 entries), the GIGA+ server divides the overflow partition into a new partition by assigning the second half of the overflow partition's hash-space range to the newly created partition. Both partitions now have a hash-space range of the same size: the original partition holds the range $(0, 2^{64} - 1]$ and the new partition holds the range $(2^{64}, 2^{128} - 1]$. During splits, GIGA+ migrates approximately half the number of entries (that have hash values in the second half of the hash-space range) from the overflow partition to the newly created partition. As more entries are added to this directory, the two partitions split further to create more new partitions (and half the hash-space range of original partitions). In GIGA+ , the number of partitions for a directory is proportional to the size of that directory.

GIGA+ distributes partitions on multiple servers using a deterministic round-robin mapping of partitions to servers. This mapping relies on a list of servers known a priori. Servers on this list are ordered and the list is known to all GIGA+ servers that manage directories. GIGA+ relies on a configuration management system, such as Apache ZooKeeper [ZooKeeper 2011], to manage server membership (discussed later in Section 4.5).

Thus, splitting allows GIGA+ to distribute a growing directory on many servers. By spreading the partitions on many servers, GIGA+ can enable parallel access to large directories.

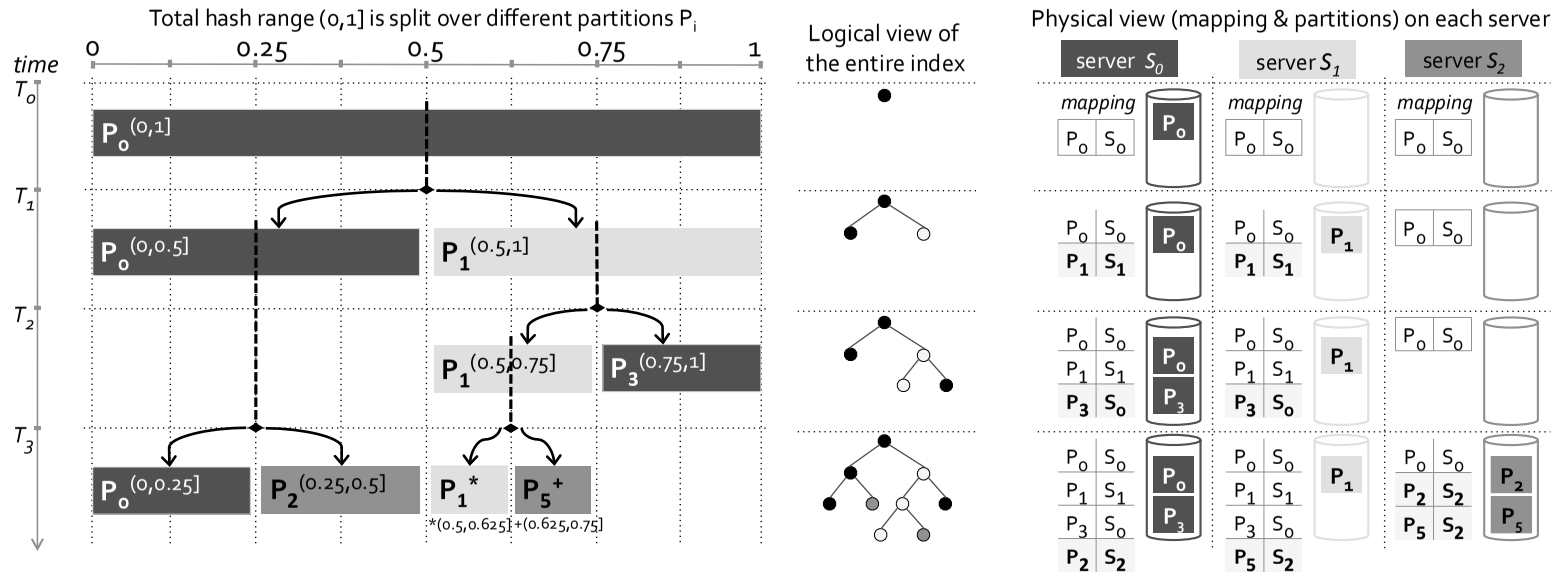

Figure 4.2 — Concurrent and unsynchronized data partitioning in GIGA+.
The hash-space $(0, 1]$ is divided into multiple partitions (P_i) that are distributed over many servers (different shades of gray). Each server has a *local, partial view* of the entire index and can independently split its partitions *without global co-ordination*. In addition to enabling highly concurrent growth, an index starts small (on one server) and scales out incrementally.

4.2 Concurrent, unsynchronized splitting

Figure 4.2 shows how a directory is divided and distributed using GIGA+ indexing. 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. This example and the remainder of this dissertation, assumes that hash values are fractional numbers in the range $(0, 1]$; for simplicity, the hash value zero is not used. GIGA+ uses the identifier i to map P_i to an appropriate server S_i in a round-robin layout, i.e. server S_i is computed as "i MODULO $num\_servers$". Partitions and servers uses a color coding: a partition with a certain shade of gray resides on the server with the same shade of gray. 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 overflow partitions onto more servers.

A key goal for GIGA+ is for each server to split independently, without system-wide serialization or synchronization. Servers make local decisions to split a partition and can split together at the same time. For example, in Figure 4.2, at time T_3, servers S_0 and S_1 split partitions P_0 and P_1 simultaneously and independently without any inter-server coordination.

The side-effect of uncoordinated growth is that other 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. Figure 4.2 shows the partition-to-server mapping table as example of the server's view. For each GIGA+ server, this view consists of the partitions that a server manages and the "child" partition (on different servers) created by splitting these partitions. In Figure 4.2, at time T_3, server S_1 manages partition P_1 with hash-space range $(0.5, 0.625]$, 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 unaware about partitions created by splits that happened on other servers and that 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 a directory partitioning, which is the transitive closure over the local mappings on each server. This transitivity, enabled by split histories, is useful for three reasons.

First, the full index is 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. Second, split histories enable GIGA+ servers to correct significantly inconsistent, out-of-date mapping state at the clients that send operations to "incorrect" servers. Finally, split histories may also help recreate the data-structures used to maintain the GIGA+ index. For instance, partition-to-server mapping cached by servers that get lost due to server reboots, may be reconstructed by traversing the split histories of a partition to learn about other partitions and their servers.

4.3 Trade-offs of *how* to split directories

The previous sections described how cross-server partition splitting enables GIGA+ to provide parallel access to large directories. This section analyzes the different ways for GIGA+ to achieve this parallelism. GIGA+ expands a directory incrementally by splitting an overflow partition into a new partition, with half the hash-space range, managed on another server. This allows GIGA+ to spread a directory on only one new server at a time, and it raises a question: *if GIGA+ wants to harness all the available parallelism, why not split the partition into N new partitions and distribute them on on N available servers?*

This section studies the cost-benefit trade-off of splitting partitions using two policies. The first policy, called "incremental splits", is used in GIGA+ to split large directories such that it uses one additional server with each partition split. The second policy, called "split once", is to split a large directory *once*, when its first partition overflows, into N partitions distributed on N servers (creating one partition per each server).

4.3.1 Benefits of splitting once on all servers

The benefit of splitting partitions on multiple servers is the higher throughput (operations per second) from using more servers. This section presents experimental evidence of the how the throughput improves using the two split policies.

Experimental methodology — Experiments in this section report how the system throughput scales as a growing directory is striped on increasing number of GIGA+ servers. The workload in this experiment creates an empty directory and populates it with large number of files proportional to the number of servers (in a weak scaling manner). An N-

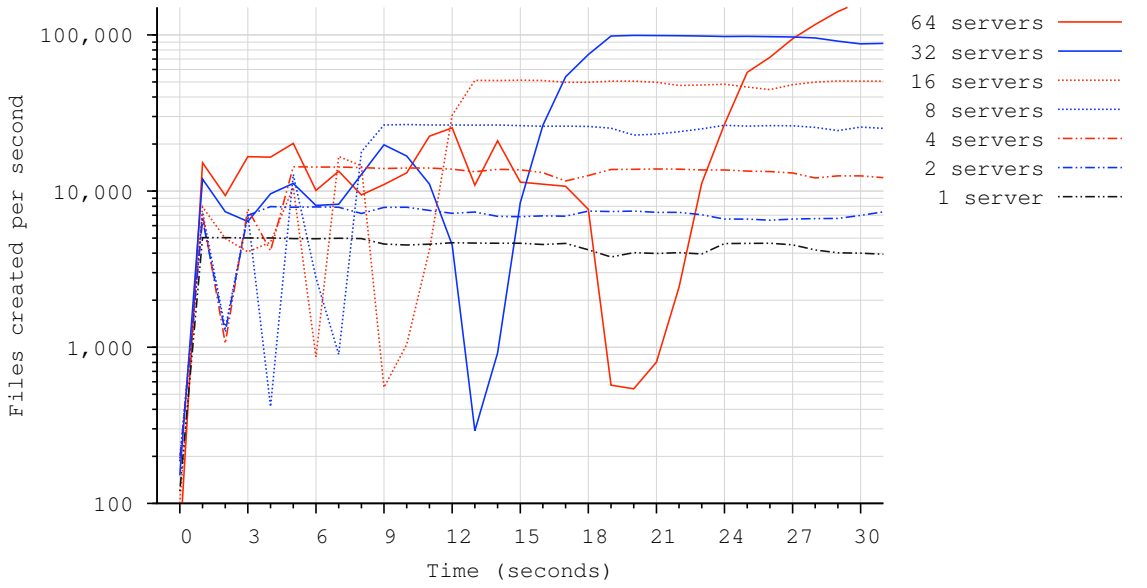

Figure 4.3 — Scale-out growth using "incremental split" policy
This graph shows instantaneous file creation rate (Y-axis) during a 30-second period (X-axis) from the beginning of an experiment that creates large number of files in a newly created directory that is striped on varying number of GIGA+ servers (shown in the legend of the graph). During this incremental growth scale-out phase, massive drops in aggregate create rate correspond to inter-server splits of overflow partitions; after the splits are completed, the throughput doubles as the number of servers is doubled.

server configuration will store a directory that with N million files created concurrently by a total of N remote client machines (each with eight workload generating threads). In other words, a single-server configuration has 1 million files created by 8 threads on a single client, a two-server configuration has 2 million files created by two clients (with 16 total threads), and so on. Experiments were done on the cluster described in Table 3.1. Each GIGA+ server stores its partitions in a LevelDB-based backend stored in an on-disk Linux Ext3 file system managing a single 7200RPM disk.

Observations — Figure 4.3 reports the instantaneous throughput, measured as file creates per second, on a log-scale Y-axis during the first 30 seconds of the workload. The throughput is studied for GIGA+ configurations of 1, 2, 4, 8, 16, 32 and 64 servers. This figure highlights the effect of using "incremental split" policy during the 30-second initial scale-out growth phase.

In Figure 4.3, the throughput of the single server remains flat, as expected, at roughly 7,500 file creates per second due to the absence of any other server. In the 2-server case, the directory starts on a single server and splits when it has more than 8,000 entries in the partition. When the servers are busy splitting, at the 1-second mark, throughput drops significantly for a short time. Throughput degrades even more during the scale-out phase as the number of GIGA+ servers goes up. For instance, in the 8-server case, the aggregate throughput drops from roughly 14,000 file creates/second at the 4-second mark to as low as 1,000 creates/second before growing to the about 28,000 creates/second. This happens because all servers are busy splitting at the same time. Because GIGA+ splits partitions in a binary manner, each of N-server configurations in Figure 4.3 has equal-sized range in the hash-space. And the uniform distribution property of the hash function ensures that all partitions fill up at the same rate and overflow at roughly the same time. This causes all servers (where these partitions reside) to split without any co-ordination at the same time. And after the split spreads the partitions on more servers, the aggregate throughput achieves the desired linear speed-up in performance.

To avoid these significant periods of throughput degradation when all servers are busy splitting partitions simultaneously, GIGA+ based systems could implement one of the following optimizations. The first optimization would be to stagger splits by adding a

random back-off delay before each server decides to split its overflow partitions. Depending on the choice of back-off periods, it is conceivable that a much smaller set of servers may split at about the same time. The second optimization would be to use a more fine-grain locking. The current GIGA+ implementation locks the entire partition before splitting; a more sophisticated implementation to use multiple, fine-grained locks to scan and migrate keys in the overflow buckets. However, the latter technique is significantly more complex than the former technique. The outcome of both these optimizations would be to alleviate the duration of splits and throughput degradation during splits by splitting different partitions at different times. However, once all servers are split, the aggregate throughput achieves the same linear scale-up as before.

Another observation in Figure 4.3 is the high variance during the 30-second incremental growth phase. This variance is associated with the on-disk representation of GIGA+ hash partitions. In this experiment, the GIGA+ servers store the hash partitions in a LevelDB-based store. This causes the splits, which involve reading from disk, scanning entries, writing appropriate entries to a different partition, and writing to disk, to be subjected the complex interactions of partition representation and on-disk I/O behavior. Although Chapter 6 studies the variability from this interactions in details, Figure 4.4 takes a short detour to show what happens if disk I/O is eliminated. Figure 4.4 shows the behavior for a similar experiment when GIGA+ servers store all partitions in-memory on a Linux tmpfs backend. As expected, an in-memory store speeds up the experiment; Figure 4.4 shows the first 8-second period of the incremental scale-out phase. The main point of this figure is to illustrate the incremental growth behavior more clearly than Figure 4.3: splits cause the throughput to drop before the throughput doubles after invoking twice

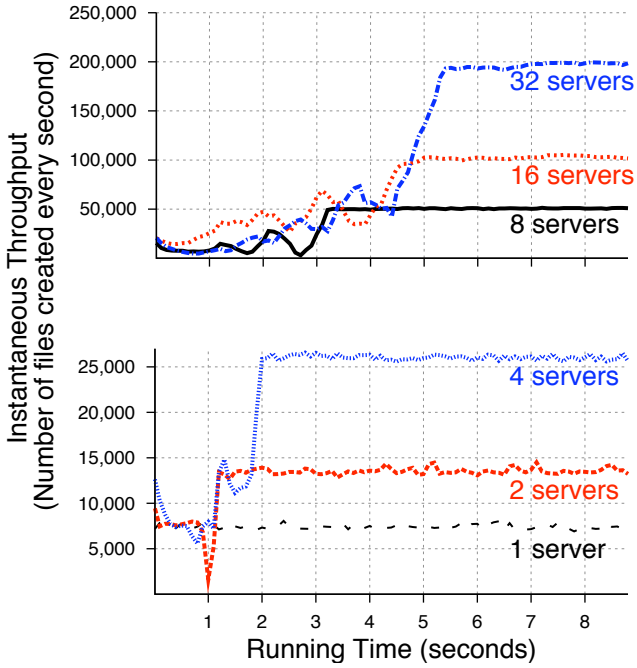

Figure 4.4 — Scale-out growth using "incremental split" policy on a in-memory Linux tmpfs **backend store**

This graph shows the scale-out growth similar to Figure 4.3 with the only difference that GIGA+ servers store all partitions in memory. Compared to Figure 4.3, which used on-disk partition representation, this figure shows much less variance in the observed throughput on the Y-axis. Detailed analysis of variability from using on-disk representations is discussed later in Chapter 6.

the number of servers. The rest of the experiments use on-disk backend stores (unless explicitly specified).

An alternative to this "incremental splits" policy is the "split once" policy that splits a large directory only once and splits it over all available servers. Figure 4.5 compares these two split policies for a 32-server configuration that performs the same experiment as described before in Figure 4.3. As expected, Figure 4.5 shows that the "split once" pol-

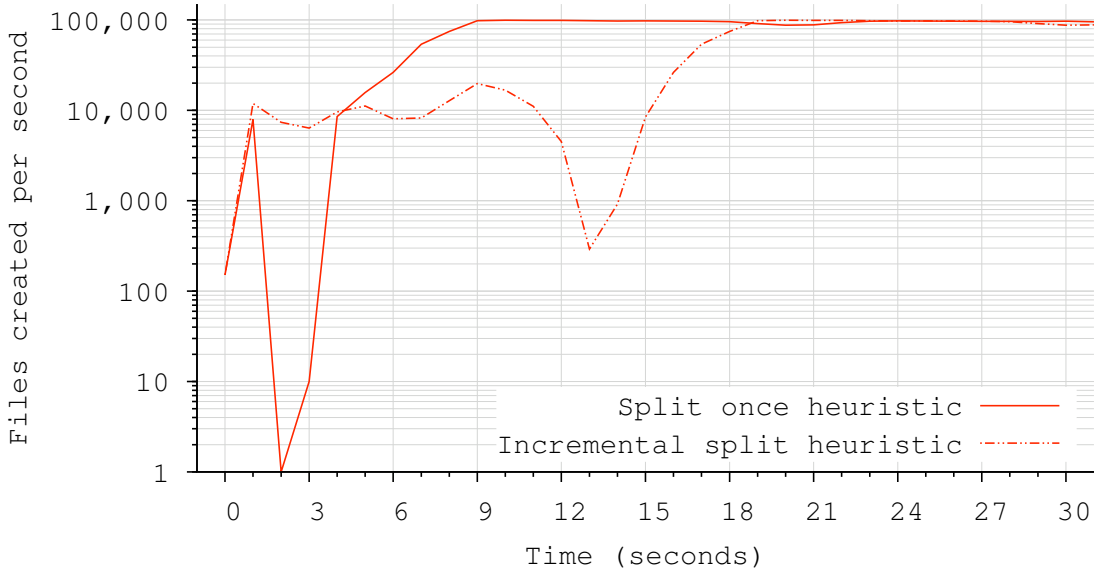

Figure 4.5 — Comparing incremental splitting and "split once" policy.
When a directory exceeds the threshold size, the "split once" policy splits it once and on all the servers; this figure uses a 32-server configuration to compare the incremental scale-out phase using both the "split once" policy and the incremental splitting policy.

icy is able to harness the maximum parallelism in the system by expanding the directory on all the servers instantly. The "split once" policy splits at the 2-second mark to distribute the directory on all 32 servers. Compared to the binary splits in the "incremental split" policy, this split takes longer and causes a higher throughput degradation because spreading partitions on 32 servers involves more complexity than spreading it on 2 servers. However, after its first and only split, the "split once" policy attains the maximum throughput after the 7-second mark. In comparison, the "incremental split" policy needs almost 19 seconds to achieve maximum system throughput.

4.3.2 *Cost of splitting once on all servers*

So why use incremental splitting in GIGA+ ? Is there a reason to not use the "split once" policy to spread directories on all servers instantly? To answer these questions, this section studies the directory scan performance, particularly for small directories which constitute more than 99.99% of all file system directories [Dayal 2008]. This analysis uses the default GIGA+ split threshold of 8,000 entries. Using the "split-once" policy on a 64-server setup will cause a directory with 8,000 entries to split into 64 partitions, one on each server, with 125 entries per partition. Using the GIGA+ "incremental splits" policy, this directory with 8,000 entries will split in two partitions with 4,000 entries each on two separate servers. To understand the difference in these two split policies, this section measures the performance and efficiency of the *readdir*() operation. Scans on directories with "split-once" policy are referred to as *split-once scans* and scans on directories with "incremental split" policy are referred to as *incremental split scans*.

Implementing *readdir*() **in GIGA+** — Applications that need all the contents of a directory, such as ls and find, typically use the *readdir*() library call.[1] By default, the *readdir*() library call uses a 32 KB buffer that is passed to the the *getdents()* system call that returns the directory entries in this buffer. If all directory entries are fit in the buffer, the call returns to the application. For directories with millions of files, *readdir*() needs to make multiple *getdents()* system calls that fetch 32 KB worth of entries repeatedly until all entries are retrieved. Applications seeking high-performance scans often use bigger *readdir*() buffer sizes that minimize the number of *getdents()* system calls [Congleton

[1] In a UNIX-based system this library is typically *libc* or a similar variant.

2011].[2] For distributed directories, scans need to get entries from all partitions stored on multiple GIGA+ servers.

The FUSE-based GIGA+ prototype intercepts the *readdir*() operation and forwards it to the user-space GIGA+ client. This client determines the partitions associated with the directory being scanned. If the directory is spread on N servers, the GIGA+ client issues an RPC_READDIR message to all N GIGA+ servers in parallel. An alternative to this approach would be to send RPCs to a smaller set of servers, i.e. fewer than the total N directory servers, depending on dynamic factors such as server load and client buffering. However, in comparison, sending the RPC message to all N servers concurrently may yield lower latency *readdir*() operations and, more importantly, a simpler implementation.

Each RPC_READDIR message contains a 1 MB buffer that is used by the server to return directory entries. GIGA+ chose to use a 1 MB buffer because it is bigger than the default *readdir*() buffer size (32 KB), a practice recommended for faster scan performance in both local file systems and cluster file systems [Congleton 2011, Dilger 2012]. If a GIGA+ server has more than 1 MB entries, it responds with a cursor of how many entries were returned successfully in one message. The client uses the cursor to request to the next batch of 1 MB directory entries. This process completes after the server acknowledges that it has sent all directory entries (for all partitions) that it stores for the directory in question.

File system directory scans, however, do not guarantee the accuracy of returned results, particularly for mutating directories; in fact, POSIX semantics for *readdir*() also suf-

[2] Some users choose to improve scan performance by directly calling *getdents()* and performing application-specific management of buffers filled with directory entries [Congleton 2011].

fer from the same problem [Drepper 2007]. The POSIX semantics state that *readdir*() must return all directory entries (in an unordered manner), which exist when the call was issued, once and only once. This does not guarantee the correctness of results when directory entries are created and deleted simultaneously during the *readdir*() operation [Whitehouse 2007]. An entry that is created when *readdir*() is being processed may not be listed in the results and, similarly, a deleted entry may still be listed in the results.

On the server side, for each RPC_READDIR message, the GIGA+ server fetches 1 MB of directory entries from persistent storage. The current implementation of GIGA+ server does not explicitly prefetch or read-ahead the entries from the on-disk store; it assumes that the backend file system may already perform such optimizations. On the client side, the GIGA+ client dynamically allocates memory as entries are returned in the RPC_READDIR response messages. GIGA+ clients do not keep any state such as entry names or partition identifiers. Once all entries are returned, the GIGA+ client returns these entries to the *readdir*() caller. This design imposes a high memory pressure at the client because a directory with billions of files may not fit in memory unless the GIGA+ client explicitly performs buffer management for *readdir*() operations. However the current GIGA+ implementation chose to avoid complex client-side memory management to preserve the simplicity of the implementation and interface of FUSE's *readdir*() implementation [FUSE 2010].

Methodology — Scan performance of GIGA+ directories is analyzed by studying three different parameters that capture the effect of disk I/O, network messages and server load. The three parameters used to understand *readdir*() performance for "incremental split" and "split once" policies are:

1. Buffer size in the RPC_READDIR message.

2. Size of the file system, measured in terms of number of directory entries (or files) in all the directories in the file system.

3. Idleness of the servers used by the directory subsystem

To understand the effect of these parameters, this section uses a metric called *scan efficiency* to measure the effectiveness of both split policies. Scan efficiency is defined as the average time to scan each entry returned in the scan results; for example, if a scan returns 1,000 entries in 5 seconds, the scan efficiency is 5 ms/entry. Thus, for scan efficiency, the lower number is better than higher number. Because *readdir*() scans are performed in parallel on all partitions of a directory, the scan efficiency is a per-server metric, i.e. scan efficiency reflects the time to scan the partition on *one* server plus the time to get the results to the client and consolidate them. In case of directories that span multiple servers, the reported scan efficiency is the average of each server. In most cases, the variance of scan efficiency across multiple servers is neglible (less than 2%); hence, the measurements in this section do not report variance unless it is significant.

Parameter #1: Buffer size in RPC_READDIR message

The buffer size used in an RPC_READDIR message determines how many client-server messages are required to complete a *readdir*() request. A large buffer size reduces the number of messages at the cost of higher memory pressure and bigger network message sizes, while a smaller buffer size may need higher number of messages. For a 1 MB buffer, *readdir*() completion time for incremental split scans was higher than split once

Phase 1	Insert 8,000 files in an empty directory (dir1)
Phase 2	Scan directory dir1 created in Phase #1
Phase 3	Insert 64 million files in another empty directory (dir2)
Phase 4	Scan directory dir1 created in Phase #1

Table 4.1 — Four phases of the multi-phase scan experiment. This experiment is performed a 64-server setup that starts with a new, empty file system.

scans. This happens because a 1 MB buffer cannot fit all the entries that the server needs to send back to the client. Clients need to send two RPC_READDIR requests for the incremental split scans case, while the split-once scans require a single RPC_READDIR request. By doubling the buffer size to 2 MB, both the scan use-cases complete their *readdir*() operations in about the same time.

However, *readdir*() completion time alone is insufficient to capture the server-side efficiency of directory scans. Scan efficiency is measured using a four-phase experiment, described in Table 4.1, on a 64-server configuration. This experiment creates files in phase 1 and 3, and scans the small directory in phase 2 and 4. Phase 1 starts with an empty directory and creates 8,000 files, just enough to trigger a split that divides the directory either in 64 partitions (of 125 entries each) on 64 servers for the "split once" policy or in 2 partitions (of 4,000 entries each) on 2 servers for the "incremental split" policy. Phase 3 creates 64 million files in another new directory; irrespective of the policy, this directory is large enough to split in 64 partitions of 1,000,000 entries each that are spread on 64 servers. The two measurement phases, phase 2 and phase 4, perform a similar operation: scan the small directory of 8,000 entries create in phase 1. The key idea is that Phase 2 scans a small directory in a file system containing only one small

	Split-once policy	Incremental split policy
Scan efficiency in phase 2	2.96 ms/entry	0.10 ms/entry
Scan efficiency in phase 4	6.41 ms/entry	0.17 ms/entry

Table 4.2 — Average scan efficiency of servers during multi-phase scans in Table 4.1

Scan efficiency variance across 64 servers was neglible (>5%) and is omitted.

directory, while Phase 4 scans a small directory in a file system containing one small and one big directory. To eliminate any effects of caching, the file system was unmounted and re-mounted after every phase.

Table 4.2 reports the scan efficiency of scan operations performed in the two measurement phases (phase 2 and phase 4). Recall that scan efficiency is defined such that a smaller number indicates better performance. Table 4.2 shows that scan efficiency of a small directory using "split-once" policy is 30 times worse than the efficiency from using "incremental splits" policy. This difference in scan efficiency grows to about 37 times, when the file system on each server grows by 1,000,000 entries. In other words, with more entries in a file system, scan efficiency of small directories using "split-once" policy is orders of magnitude worse than using "incremental splits" policy. This observation can be explained by the disk efficiency during scans. The cost of small scans is dominated by seek and rotation time, which has much lesser effect on large scans (4,000 entries) that are dominated by data transfer time. This is consistent with current disk performance where it is generally recommended that scans of 1-4MB, which is the size of a track, are much more efficient than smaller scans [Anderson 2003, Schlosser 2005, Schindler 2011].

Each directory entry in GIGA+ is about 256 bytes, and a large scan is about the size of a track on modern disks while a small scan (of 125 entries) is only about 32 KB in size.

Parameter #2: Total number of entries in the file system

Results from the previous experiment in Table 4.2 indicate that scans become inefficient as the number of entries in the file system grows over time. This phenomenon is explored further by varying the number of entries in a file system.

Figure 4.6 shows how the scan efficiency varies with different file system sizes (shown on X-axis). It denotes the number of entries on *each* server using a pair (x, y) where x is the number of entries in small directories and y is the number of entries in large directories. For example, the configuration $(4M, 10M)$ indicates a file system where each server has four million entries in small directories and ten million entries in large directories, i.e. on a 64-server setup, this configuration has 256 million entries in small directories and 640 million entries in big directories.

The Y-axis in Figure 4.6 shows the scan efficiency of *readdir*() on a small directory for both the "split once" policy and the "incremental splits" policy. Figure 4.6(a) shows the scan efficiency of a small directory with 8,000 entries as a function of the file system size. It shows that as the file system grows in size, both split policies yield a more inefficient scan performance. The "incremental splits" policy, however, has 1-2 orders of magnitude better scan efficiency than the "split once" policy.

This analysis raises a question: why not increase the split threshold such that each server will have more entries (which results in more efficient scans)? To answer this question, split threshold is increased by four times, from 8,000 entries to 32,000 entries. The

growth in split threshold changes the notion of a small directory, i.e. if a system uses the "split once" policy, the directory splits over all servers only when it has more than 32,000 entries. Thus, in a 64-server setup, each server has 500 entries after the first split. Similarly, for the "incremental splits" policy, a directory with 32,000 entries has 16,000 entries on each server after the first split.

Figure 4.6(b) shows the scan efficiency of the two split policies with the larger split threshold of 32,000 entries. These results show that scans in "split once" policy are inefficient compared to "incremental splits" policy, and that both scans are more inefficient with the increasing file system sizes. While this is similar to the observations in Figure 4.6(a), scans are two to three times more efficient when the split threshold is larger (i.e., 32,000 entries).

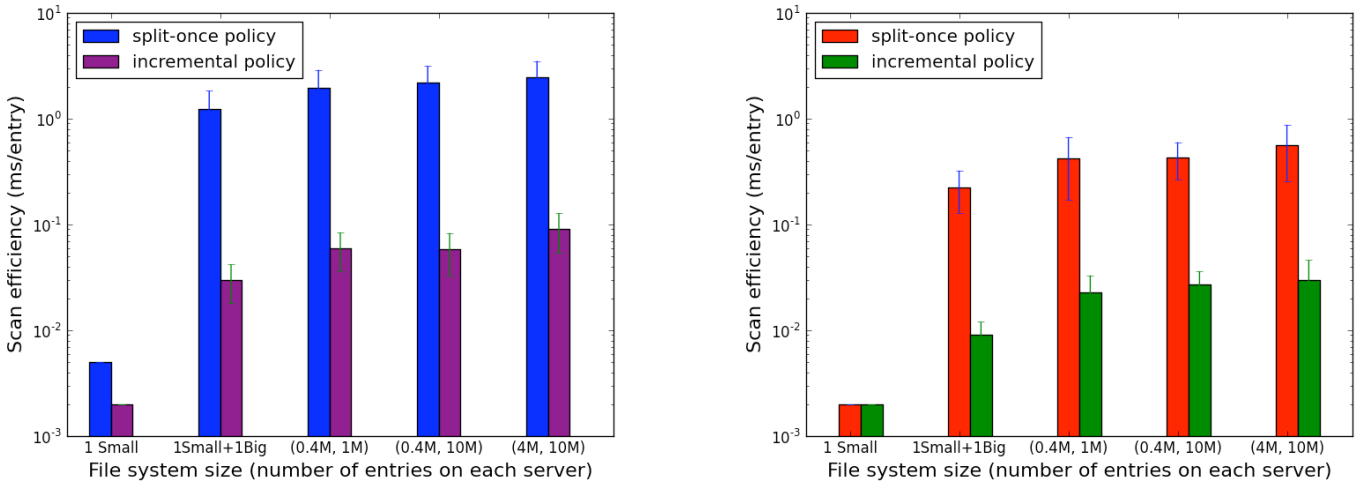

Figure 4.6 — Scan efficiency of *readdir*() on small directories in file systems of different sizes.
This graph shows the scan efficiency of *readdir*() requests for a small directory in a file systems of different sizes (measured as the number of entries). Graph (a) report the scan efficiency for GIGA+ configuration with both the "split-once" and "incremental splits" policy for a split threshold of 8,000 entries. Graph (b) has a similar setup but with a larger split threshold of 32,000 entries.

With a large split threshold, the partitions are also large and scanning these partitions is more efficient than scanning smaller partitions. The disk bandwidth utilization is much higher than when the split threshold was smaller (i.e., 8,000 entries). However, large split threshold will take longer for a large directory to split to all available servers. This causes GIGA+ to harness the system-wide parallelism much slower than when the split threshold was lower.

Parameter #3: Idleness of the GIGA+ directory servers

So far, all experiments show the scan efficiency of an idle system, i.e. a system where there is only one outstanding request — the scan request issued by the client. A more realistic use-case would have different types of directory operations simultaneously. This real-world scenario is emulated by performing directory scans in conjunction with simultaneous update and lookup requests on the GIGA+ server.

This experiment starts with a file system with small directories and big directories. It then randomly picks ten small directories to be scanned. Each directory is scanned once and there is a 30-second pause in between scans of two directories. During this 30-second quiet period, other clients in the system perform a mix of update and lookup operations of all other directories in the system. The mix workload constitutes using a *chmod()* operation and a *stat()* operation on files chosen randomly from directories that are not being scanned.

Figure 4.7 shows the scan efficiency of each of the ten scan requests performed during this experiment. It reports results for two different split thresholds (8,000 entries and 32,000 entries) and compares the "split-once" policy and "incremental splits" policy. For

both policies, Figure 4.7 shows that the first scan request is one to two orders of magnitude more inefficient than all other scan requests. This happens for two reasons that are related with LevelDB based on-disk backend store.

The first reason is that the initial scan request causes the LevelDB store to read data from the disk. This disk read makes the initial scans run much slower, but fetches the scanned data in memory for future use. The second reason is that the intermediate 30-second period of updates causes a lot of activity in LevelDB store. Update requests cause LevelDB to handle modified entries in its buffers and lookup requests force more data from disk, possible evicting other buffered data. Because LevelDB uses a periodic sort-and-merge mechanism before flushing the in-memory buffers to disk, a mixed workload forces LevelDB to reorganize the partitions in an efficient sorted manner. Consequently, all subsequent scans are reading from an almost sorted data structure resulting in similar scan performance. Chapter 6 describes the structure and operations of LevelDB in details.

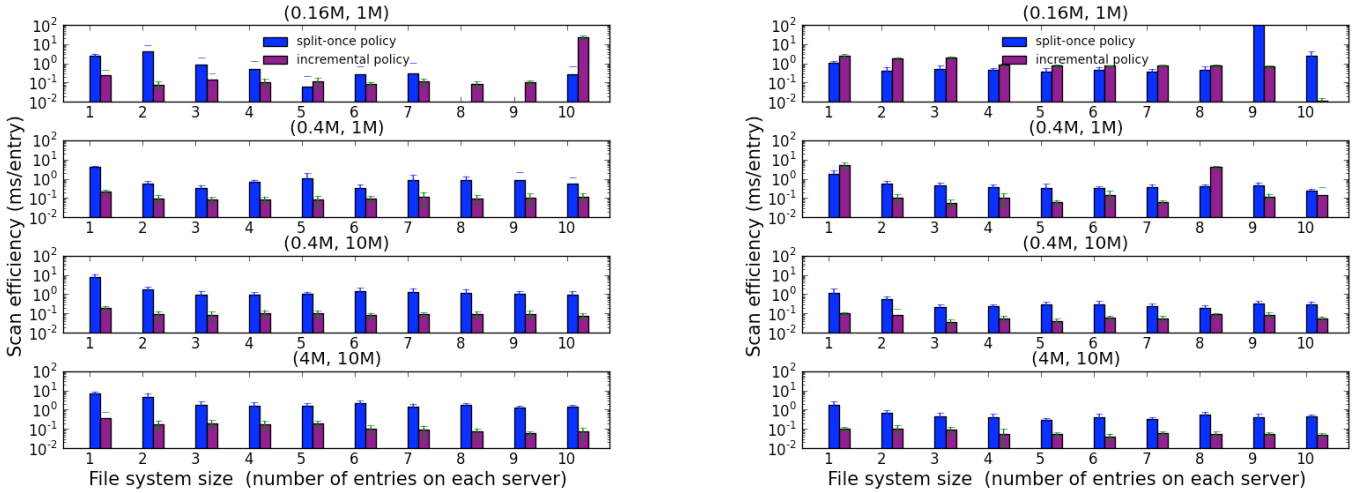

Figure 4.7 — Scan efficiency of a busy system.

This graph shows the scan efficiency of ten *readdir*() requests for ten different small directories (of same size) in an existing large file system. These scans are performed after a 30-second period during which other clients are issuing random lookups (*stat()*) and updates (*chmod()*) to other files in the file system. Graph (a) report the scan efficiency for GIGA+ configuration with both the "split-once" and "incremental splits" policy for a split threshold of 8,000 entries. Graph (b) has a similar setup but with a larger split threshold of 32,000 entries. Both graphs analyze file systems of different sizes.

Key lessons —

- Splitting enables more parallelism: GIGA+ distributes hash partitions of a large growing directory on many servers.
- The rate of splitting partitions, i.e. either splitting a partition into two partitions or N partitions, is governed by various factors including the nature of the workload (i.e., scans and lookups interfere with inserts) and the complexity of splitting (i.e., getting an N-server split right is much harder than a 2-server).[3]

4.4 Understanding the effectiveness and ineffectiveness of splitting

The previous section described how and when splitting a large directory on more servers helps in harnessing available parallelism in the system. This section probes the rationale of splitting further by exploring the question: *under what circumstances does splitting provide little or no benefit, and explores whether GIGA+ can split only when there is benefit in doing so?* The remainder of this section quantifies the benefit, measured as the quality of load balancing, of splitting and devises a strategy to mitigate cases when the cost of splitting outweighs its benefits.

[3] In fact, Lustre has proposed a N-server split once strategy for distributed directories several years ago, but the complexity of that operation has stymied its release in the main distribution [Lustre 2010a, 2009].

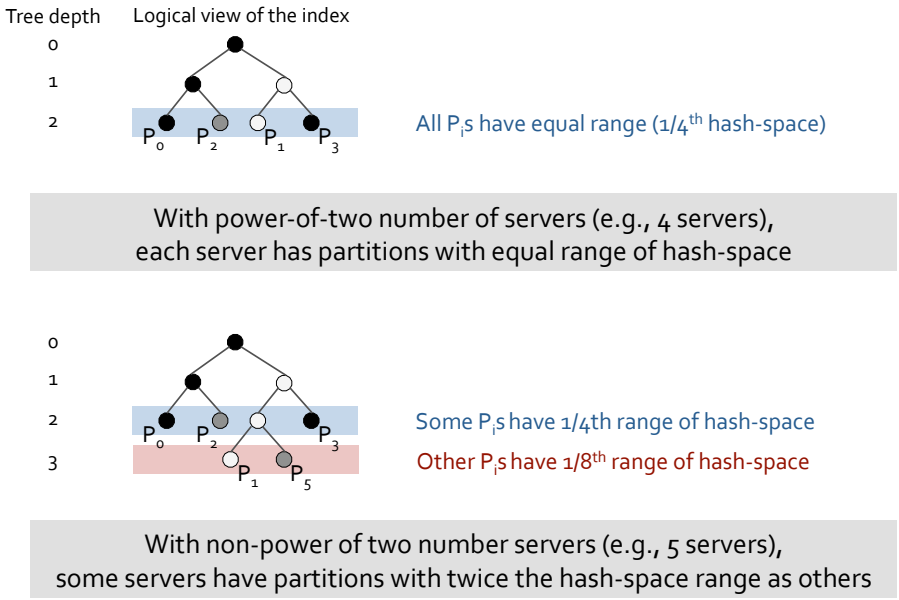

Figure 4.8 — Hash-space distribution in GIGA+ indexing.
The top figure shows an example of power-of-two number of servers where each GIGA+ server is responsible for 1/4th the hash-space range. The bottom figure shows the imbalance among non-power-of-two number of GIGA+ servers caused by different in hash-space range held by partitions: two of the five partitions (and their servers), in this example, have only half the hash-space range as the remaining partitions (and their servers).

4.4.1 Effectiveness of splitting: load-balanced distribution

So far, all analysis has used configurations where the number of GIGA+ servers is a power-of-two. This is a special case because it is naturally load-balanced with only a single partition per server: the partition on each server is responsible for a equal sized hash-range. Figure 4.8 shows an example of a GIGA+ hash tree for a directory that is split into four partitions (top part of the figure) that are stored on four servers. Because each partition has $1/4$-th the hash-space range, servers have a balanced load distribution.

When the number of servers is not a power-of-two, however, there is load imbalance. The bottom half of Figure 4.8 illustrates an example of the hash tree for a large directory that is striped on five servers. In this figure, the partitions on the last level of the tree (level 3) are responsible for half the hash-space range of the partitions on the previous level (level 2) of the tree. This factor of two difference is a result of two-way partition splits that divides the hash-space range among the partitions. As a result, servers have imbalanced distribution of hash-space range.

To better understand the quality of GIGA+ load balancing, Figure 4.9 shows the load imbalance for different configurations of GIGA+ and compares it with consistent hashing [Karger 1997]. The experimental methodology and observations are described below.

Methodology — Figure 4.9 reports results of an analytical simulation that measures the load imbalance for a large directory that is striped on different server configurations. Each configuration contains two parameters, the number of servers in a cluster (shown on X-axis) and number of partitions on each server (shown in the legend).

For each configuration, the Y-axis of Figure 4.9 represents the load imbalance. Load is computed using a Monte Carlo model of hash-space division that decides which partitions should split using a uniform random function distribution. For GIGA+ , when the number of servers N is not a power-of-two, $2^r < N < 2^{r+1}$, then a random set of $N - 2^r$ partitions from tree depth r, each with range size $1/2^r$, will have split into $2(N - 2^r)$ partitions with range size $1/2^{r+1}$. For consistent hashing, the hash-space range held by each partition is determined randomly (typically, based on the hash value of the name of the server [Stoica 2001]).

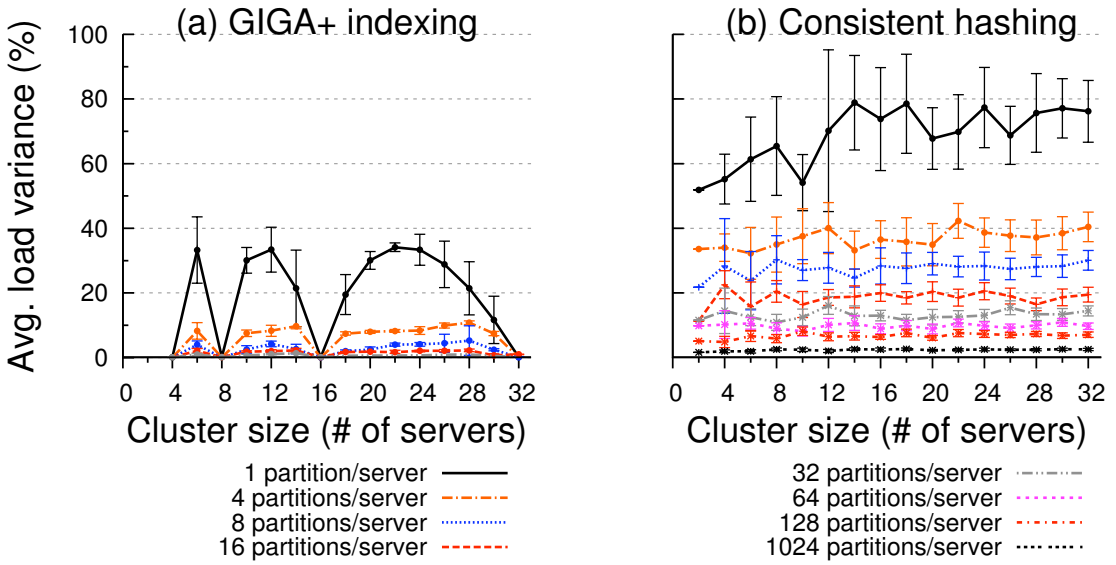

Figure 4.9 — Load-balancing efficiency of GIGA+ and consistent hashing. These graphs show the quality of load balancing measured as the mean load deviation across the entire cluster (with 95% confident interval bars). Like virtual servers in consistent hashing, GIGA+ also benefits from using multiple hash partitions per server. GIGA+ needs one to two orders of magnitude fewer partitions per server to achieve comparable load distribution relative to consistent hashing.

The load imbalance on Y-axis is measured as the average fractional deviation from even load. Here's an example to illustrate load variance computation. 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, the variance of each server measured. Figure 4.9 reports the the average (and 95% confidence interval error bars) over all the servers.

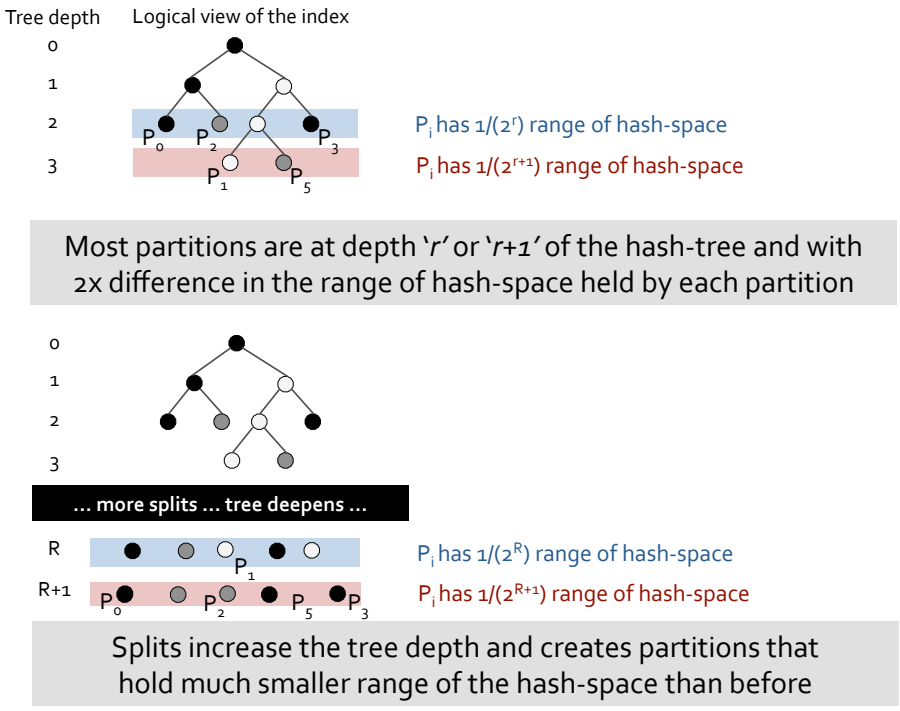

Figure 4.10 — GIGA+ indexing hash-tree before and after splitting.
The top figure shows an example of tree of hash-space range of a large GIGA+ directory. It shows that at most times partitions are either at tree depth r or $r+1$. The bottom figures shows the state of the tree as the GIGA+ directory splits to create more partitions. With more partitions in the server, each partition is responsible for much smaller range of the hash-space.

Observations — GIGA+ configuration in Figure 4.9(a) shows the results of five random selections of $N - 2^r$ partitions that split to the $r+1$ level. Figure 4.9(a) shows the expected periodic pattern where the system is perfectly load-balanced when the number of servers is a power-of-two. But with non-power-of-two number of servers the load imbalance among servers is high because some servers have hash partitions from level r and others have hash partitions from level $r+1$.

However, Figure 4.9 shows that load imbalance reduces if GIGA+ continues to split to create more than one partition on each server. Until now, GIGA+ split a large directory to keep a partition on each server, which is sufficient for a directory to harness all available parallelism in the system. In contrast, Figure 4.9 shows that splitting can also benefit load balancing, when partitions continue to split even after all servers are already in use. Splitting the directory to create more partitions that number of servers causes the hash-space tree, in Figure 4.10, to grow deeper; and as the number of levels of the tree increases, the range of hash-space maintained by each partition decreases. (As described earlier, at tree depth r, each partition hold $1/2^r$-th part of the hash-space.) As a result, when each server has multiple partitions, each partition will manage a smaller portion of the hash-range, and the sum of smaller partitions held on a server will be less variable than a single large partition. This phenomenon explains the observation in case of GIGA+ in Figure 4.9(a) where splitting to create increasing number of partitions per server significantly improves load balance when the number of servers is not a power-of-two. For instance, if GIGA+ splits to create 8 or more partitions on each server, the servers on average have less than 5% load imbalance in the system.

A similar approach of assigning multiple hash-space ranges to each server is also used to alleviate load imbalance in consistent hashing [Karger 1997]. Systems such as Chord DHT [Stoica 2001] and Amazon Dynamo key-value storage system [DeCandia 2007] refer to this approach as *virtual servers*. Consistent hashing associates each partition with a random point in the hash-space $(0, 1]$ and assigns it the range from this point up to the next larger point and wrapping around, if necessary. Figure 4.11 illustrates how consistent hashing's random hash-space distribution works. In this figure, the top two figures

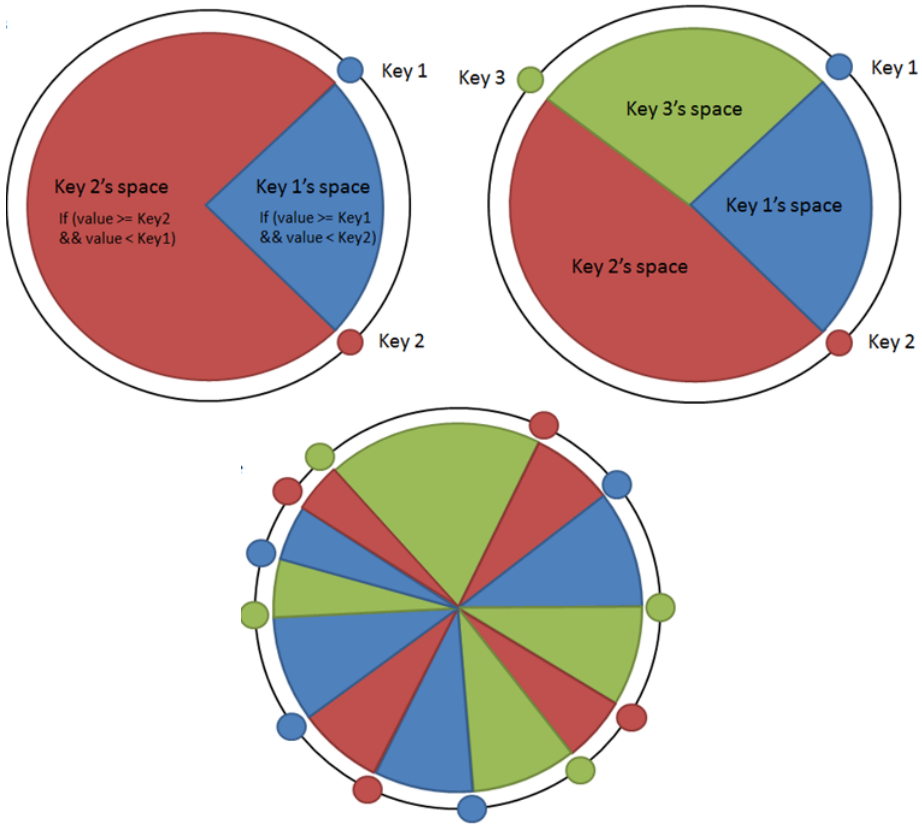

Figure 4.11 — Hash-space distribution in consistent hashing with and without virtual servers.

Each color in this figure indicates the server that holds the corresponding key range. The top-left figure shows a 2-server consistent hashing setup where the hash-space is randomly divided between the *red* and *blue* server. The top-right figure show how the key range is re-distributed in consistent hashing when a third server, *green*, is added and is assigned a random part of the existing range (from *red* server). Given the high variance in the range distribution in the both the top figures, the bottom figure illustrates how consistent hashing divides the key range in many more *virtual partitions* that are distributed among the same set of servers three servers. Consistent hashing literature refers to this distribution of multiple key ranges to the same server as *virtual servers* [Stoica 2001, DeCandia 2007]. (This figure were found on the Internet [Robson 2010].)

show a case where each server, corresponding to a different color, holds uneven parts of the range. Because of the random distribution of the range, both the 2-server and 3-server configuration has uneven ranges associated with them. To reduce this variance in range distribution, the bottom illustration in Figure 4.11 shows how consistent hashing assigns multiple ranges to each of the three server

Figure 4.9(b) shows the load imbalance of using multiple partitions per server (or virtual servers) in consistent hashing by using five samples of a random assignment for each partition and how the sum, for each server, of partition ranges selected this way varies across servers. Because consistent hashing's partitions have more randomness in each partition's hash-range, it has more than twice the load variance than GIGA+ . Figure 4.9(b) shows that increasing the number of hash-range partitions significantly improves load distribution in consistent hashing too. But consistent hashing needs to split much more — the figure shows that consistent hashing needs more than 128 partitions per server to match the load variance that GIGA+ achieves with 8 partitions per server — consistent hashing roughly requires an order of magnitude more partitions.

4.4.2 Ineffectiveness of splitting (and how to avoid it)

Splitting partitions has two benefits: higher concurrency and better load balancing. The natural question to ask is whether GIGA+ should always keep splitting overflow partitions and distributing new partitions on available servers? To answer this question, this section explores the circumstances under which the cost of splitting outweighs its benefits. And in the context of this question, there are generally two strategies found in storage systems.

The first strategy, generally used in classic database indexing, is to never stop splitting, i.e. keep splitting whenever a partition overflows. Classic database indices, such as extendible hashing [Fagin 1979], linear hashing [Litwin 1980], and B-trees [Comer 1979], were developed for out-of-core indexing of records that did not fit in memory of single-node database systems. These indexing schemes, both hash-tables and B-trees, will split a partition (or a leaf page in a B-tree) when its size exceeds the recommended size of optimal storage allocation units, such as pages, disk blocks or large disk extents [Gray 1992]. This implies an unbounded number of partitions per server as the table grows in size.

The second strategy extends the classic "never stop splitting" technique in distributed storage systems such as Google Bigtable [Chang 2006] and Apache HBase [HBase 2010]. Partitions are managed by servers and, on overflow, they are split to be distributed on other servers. However, in this technique, overflow partitions continue to be split and spread on many servers for load-balancing and decentralization.

GIGA+ adopts a strategy similar to other distributed systems in that splitting partitions is used to parallelize access to a large directory by distributing load over all servers. However, the key difference from these other techniques is that GIGA+ can stop splitting partitions after each server has an equal share of work. For a given number of servers, Figure 4.9 demonstrates that GIGA+ can determine the number of partitions per server that provide the desired and tolerable load variance among servers.

When GIGA+ splits the directory into enough partitions and reaches the limit on the number of partitions per server, it stops splitting and distributing partitions. If a partition overflows, GIGA+ servers continue to grow the partition in size and allows the back-

end store to continue its own out-of-core indexing of the on-disk representation of the hash partition. The layered implementation of GIGA+ , described earlier in Chapter 3, decouples the two actions: splitting hash partitions for parallelism and load balancing, and storing hash partitions on the backend storage system. GIGA+ servers handle the inter-server splitting and rely on the backend storage system for intra-server representation. Because most storage systems perform out-of-core indexing, when GIGA+ stops splitting, all overflow partitions continue to be indexed internally by the underlying storage system without involving the GIGA+ servers.

Figure 4.12 compares the effect of two different splitting policies: stop splitting partitions after distributing on all servers (used in GIGA+ design) and keep splitting partitions on overflow continuously (used in most database-like systems). GIGA+ is setup to run using these two policies. Both policies are evaluated using the same methodology: create a large directory on a 16-server configuration that stores all partitions in an in-memory Linux *tmpfs* backend store. These servers are similar to the cluster setup described earlier in Chapter 3. Directory entries are created in a directory which starts empty. Using 128 client threads, this experiment creates a total of 8 million entries in the directory. The *tmpfs* in-memory backend is used to isolate the effect of on-disk representations; the effect of partition splits can now be narrowed to cross-server migration of directory entries. Subsequent experiments will demonstrate the effect of using on-disk backends for storing hash partitions.

Figure 4.12 reports the instantaneous throughput measured as the number of files created in each second (on a log-scale Y-axis) during the time it takes for the experiment to complete, i.e. time required for the directory to contain 8 million files distributed on

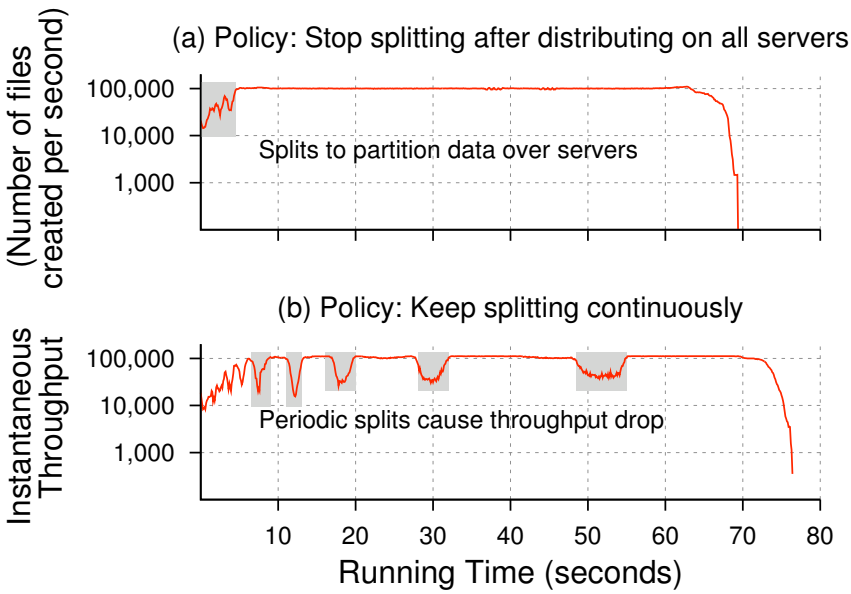

Figure 4.12 — Comparing policies to split partitions (in an in-memory setup). This figure compares the system throughput of 16-server GIGA+ setup with in-memory backends that differ only in their splitting policies. The top graph shows the GIGA+ policy that stops splitting after all servers are in use (in this case, splitting stops after there is one partition per server). The bottom graph shows the policy of splitting partitions continuously as they overflow (as in classic database indices); this policy is detrimental in a multi-server setup because even splitting in an in-memory system, without any disk I/O bottleneck, causes a 10% slow down in the running time of the experiment.

16 servers. In this figure, the top graph labeled (a) shows a split policy that stops when every GIGA+ server has one partition, causing partitions to ultimately get much bigger than 8,000 entries. The bottom graph, labeled (b), shows the continuous splitting policy used by classic database indices where a split happens whenever a partition has more than 8,000 directory entries. The difference between the graphs is that continuous splitting of overflow partitions results in 10% longer completion time than when splitting is stopped after all servers are in use. With continuous splitting, the system also experi-

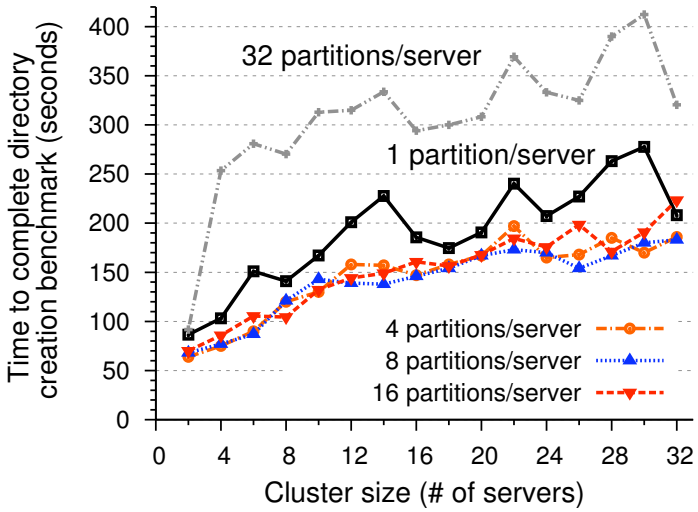

Figure 4.13 — Cost of splitting to create more partitions (in an on-disk system). The cost of splitting partitions, measured on the X-axis as benchmark completion time, for configurations with different number of servers (on Y-axis) for different split policies. The *32 partitions/server* case shows the continuous splitting policy and the *1 partition/server* case shows the GIGA+ policy to stop splitting when all servers are in use. The other intermediate configurations help to highlight the case when the cost of splitting to create extra partitions (from 16 to 32 partitions/server) far outweighs the load balancing benefits from these extra partitions/server as shown in Figure 4.9.

ences 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.[4]

[4]This behavior of all partitions splitting simultaneously can be fixed by engineering techniques that cause each server to randomly delay splitting for small random period of time. However, in our experience, this had minimal effect in improving the completion time of the experiment. But staggering splits may be useful for other applications that require some "soft real-time" performance guarantees.

This cost of continuous splitting — the 10% increase in benchmark completion time — in an in-memory setup gets significantly worse when hash partitions are stored in an on-disk backend store. Figure 4.13 compares the cost of these two splitting policies on an on-disk backend store used by a varying number of servers. This experiment creates a large directory that starts empty and grows to be big using the weak scaling methodology used earlier (in Sections 4.3 and 3.3). An N-server configuration stores an empty directory that will contain $N/2$ million files created concurrently by a total of $8N$-threads running on N remote client machines, i.e. 1-server configuration has 500,000 files created by 8 application threads running on one client machine, 2-server configuration has 1,000,000 files created by 16 application threads running on two client machines, and so on. Each GIGA+ server stores its hash partitions in an on-disk Linux ext3 file system managing a single 7200RPM disk.

Figure 4.13 shows the time required to complete this directory creation benchmark (on Y-axis) for different cluster configurations (on X-axis). It emulates the two split policies: the *1 partition/server* case shows the GIGA+ policy to stop splitting when all servers are holding one partition of a directory and the *32 partitions/server* case shows the other splitting policy that splits repeatedly 32 times, instead of continuously until end of experiment.[5] This figure also shows the cost of splitting a bounded number of times through the *4, 8, and 16 partitions/server* cases.

In Figure 4.13, for multi-server configurations, the 32 partitions per server case takes more than twice as much time to complete the benchmark than 1, 4, 8, and 16 partitions

[5]This experiment chose to stop splitting after 32 times and not continue splitting continuously because the latter choice took a very long time to run for each run.

per server. To compare this on-disk configuration with the in-memory configuration in Figure 4.12, look at the case of 16 servers on X-axis. Using on-disk partitions, Figure 4.13 shows that the continuous splitting policy (emulated by 32 partitions/server) is almost 50% slower than GIGA+ splitting policy (indicated by 1 partition/server); the same policy was only 10% slower in the in-memory configuration of Figure 4.12.

In GIGA+ , the cost of splitting, measured as benchmark completion time in Figure 4.13, is related to the benefit of splitting, measured as load-balancing efficiency in Figure 4.9. Recall from Figure 4.9(a) that the load-balancing efficiency from using 32 partitions per server is only about 1% better than using 16 partitions per server. However, splitting to create twice as many partitions makes the system slower by a factor of two: Figure 4.13 shows that the 32 partitions/server case takes more than twice the amount of time to complete the experiment than the 16 partitions/server case.

The final observation in Figure 4.13 is that splitting to create more partitions per server *does not* result in longer benchmark completion time. In fact, experiments with *4, 8, and 16 partitions/server* complete about 10-30% faster in all cluster configurations than the experiments with *1 partition/server*. This is counter intuitive because splitting to create more partitions per server results in additional disk I/O traffic and data migration traffic. This behavior is explained in Chapter 6 which discusses the effect of different types of backend storage systems and their on-disk representations of hash partitions.

Key lessons —

- Although splitting to create one partition on each server is sufficient to harness the available parallelism in the system, it often results in an imbalanced load distribution.
- To improve load balancing, GIGA+ splits more to increase the number of partitions mapped on each server.
- However, splitting continuously starts having diminishing returns: the cost of additional splitting significantly outweighs the benefits of this splitting. Excessive splitting to incurs high disk I/O and data migration traffic, but these extra partitions on each server provide neglible improvement in load balancing or system throughput.

4.5 Handling new server additions

Large cluster-based systems need to add new server resources to meet growing performance requirements. This section describes how GIGA+ adapts to addition of new servers in a running directory service. [6]

When a new server is added to a system, it is not servicing any requests and is not providing any load or capacity sharing until data is migrated to it. This highlights the key trade-off between additional data migration overhead and transient load imbalance. When new servers are added to an existing configuration, the system is immediately no longer load balanced because the new servers are still idle. The servers need to re-balance

[6] Removing servers through decommissioning, not through failing and replacing, is not as common as adding servers in high-performance computing. This dissertation does not address this problem; GIGA+ assumes that migrating directory state from servers scheduled to be removed can be done out-of-band by data migration and copy tools.

among themselves by migrating directory entries from existing servers. Before this can happens, existing servers need to know the presence of new servers in the system. The rest of this section shows how GIGA+ addresses these two issues.

4.5.1 How does GIGA+ migrate partitions on new servers?

The goal in GIGA+ is to minimize the number of directory entries migrated to the new servers. This corresponds to trying to minimize how hash partitions are re-created and re-mapped to the new server configuration (with additional servers). GIGA+ uses a round-robin partition-to-server mapping shown in Figure 4.2; partitions are mapped to servers using $\{i \text{ MOD } num\_servers\}$ where i is a partition identifier. As described earlier in Section 4.1, round-robin mapping enables faster parallelism when a directory is small and growing by spreading new partitions on more servers. However, for round-robin mapping, a naive server addition scheme would require re-mapping almost all hash partitions (and their directory entries) to new servers whenever new servers are added. Consider a configuration that has N GIGA+ servers with M partitions each. If k new servers are added, then round-robin mapping of $\{i \text{ MOD } num\_servers\}$ will cause $(N-1)M$ partitions to be re-mapped on $N+k$ servers in the system — in other words, all but the first N partitions may end up migrating to different servers.

GIGA+ avoids this extensive re-mapping of hash partitions by changing the partition-to-server layout when new servers are added to the system. To spread existing partitions on newly added servers, GIGA+ does not use round-robin mapping — GIGA+ servers use a *sequential layout* to map partitions on new servers. Figure 4.14 illustrates this in details. This figure shows an example where the original configuration has 5 servers with 3 par-

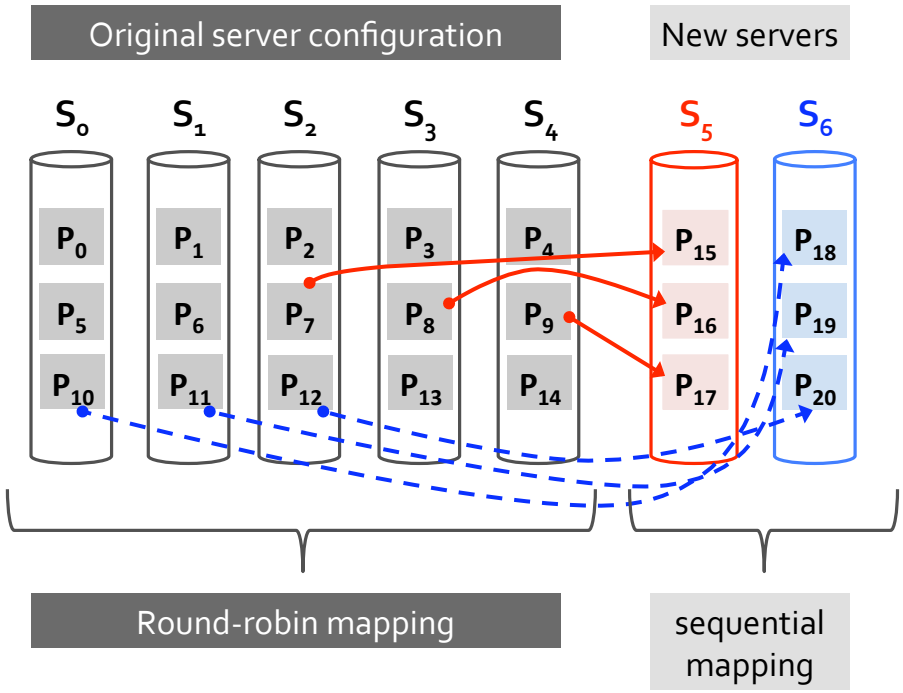

Figure 4.14 — Server additions in GIGA+.
To minimize the amount of data migrated, indicated by the arrows that show splits, GIGA+ changes the partition-to-server mapping from round-robin on the original server set to sequential on the newly added servers.

titions each. Partitions P_0 to P_{14} use a round-robin rule (for P_i, server is $i \bmod N$, where N is number of servers). After the addition of two servers, the six new partitions, P_{15} to P_{20}, are mapped to new servers using the new sequential layout rule: i DIV M, where M is the number of partitions per server (in Figure 4.14 M is set to 3 partitions per server). This mechanism poses two questions.

The first question is why would GIGA+ servers split partitions to new servers? Recall from Section 4.4.2 that GIGA+ stops splitting a directory that has spread on all available servers with the desired load distribution. In such a case, if a large directory continues

to grow larger, GIGA+ hash partitions on each server may overflow but not be split further because any further splitting does not provide any decentralization or parallelism benefits. As a result, GIGA+ relies on the backend storage systems to perform out-of-core indexing of individual partitions that are growing in size. At this point, when the system is saturated and highly utilized, adding new servers is one way to improve throughput. Typically, this is when new servers are added to most systems. Once new GIGA+ servers are available, existing servers can inspect its partitions that are overflowing and are eligible to be split to new servers (as described above).

The second question is how do you choose M, which is the number of partitions on the newly added servers? It is important to note that M may be different for different directories; small directories that have not yet fully striped on the existing set of servers decide M when they have to start using a the newly added servers. For a large directory, M can either be equal to or be greater than the number of partitions in the original set of servers. Figure 4.14 shows the case where the number of partitions of each new server is the same as number of partitions on existing set of servers. Results from earlier in this chapter, in Section 4.4.1, demonstrate that the existing set of servers store multiple partitions on each server depending the desired degree of load distribution. Thus, it is expected that using the same value of M both on existing servers and old servers may not yield any load imbalance. However, Figure 4.10 explained how splitting to create more partitions reduces the hash-space range held by each partition. This requires that the value of new servers have a bigger M because the newer partitions are going to hold a much smaller range in the hash-space.

While round-robin mapping enables a growing directory to stripe on all available servers quickly, sequential mapping for the tail set of partitions does not disturb previously mapped partitions more than is mandatory for load balancing. In fact, this is similar to consistent hashing with virtual servers where each server in consistent hashing holds multiple ranges in the hash-space. A new server that is also assigned multiple ranges that it has to transfer from the old servers responsible for that range. If the new server is assigned N' virtual servers, the random range partitioning in consistent hashing may force this server to contact N' old servers to transfer a part of the range to the new server.

4.5.2 How do existing clients and servers learn about new servers?

Like the asynchronous incremental directory expansion, GIGA+ also uses asynchrony in the way presence of new servers is known to existing servers in the system. New servers are advertised to existing servers in a lazy, on-demand manner. Like many large-scale distributed systems [Ghemawat 2003, Welch 2008], GIGA+ also relies on a distributed configuration management protocol, such as Apache ZooKeeper for HDFS [Hunt 2010], to maintain a globally consistent version of the ordered server list. The arrival of a new server and its order in the global server list is declared by the configuration management protocol which leads 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. Based on the current configuration of the index, i.e. the number of partitions in the system, the existing servers know deterministically which partitions can

be split to create new partitions on new servers using the sequential layout scheme described above. 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. Another strategy to force splits would be based on administrator control driven by a background traversal of a list of partitions whose size exceeds the splitting threshold.

Clients learn about availability of new servers through the update messages triggered after addressing incorrect servers. Once the old servers have completed splitting existing overflow partitions onto new servers, all accesses to the split partitions will trigger an update from the server to the client. For all addressing errors related to a split involving a newly added server, the update message comprises of the new bitmap and the list of newly added servers. The lookup algorithm used at the client uses this information when computing the address of the server responsible to hold a partition with a given identifier. This allows GIGA+ clients to continue normal mode of operations without synchronously learning about the new servers in the system until they are accessed.

4.6 Other issues: mitigating server overloads

Although hashing and splitting in GIGA+ should spread directory contents and traffic over many servers, there are two cases that may create a hot-spot on any server: first is the case of small directories and second is the case of a millions of clients booting up to access one directory. Both cases can be mitigated through randomized server selection.

The incremental growth property of GIGA+ ensures that small directories, with fewer than 8,000 entries, will only have one partition (the zeroth partition P_0) on one server. In order to avoid placing all small directories on one server, GIGA+ chooses a random per-

mutation of the global server list for each directory during *mkdir*() operation. This server list is stored as an extended attribute along with i-node attributes of the directory entry. For example, consider a directory */dir1*. During the *mkdir*() call for *dir1*, the directory entry of *dir1* has an extended attribute that keeps the server list for *dir1*. This server list may be different from another directory, for example, *dir2*. GIGA+ uses the first server in *dir1*'s server list to hold the zeroth partition P_0 created during *mkdir*() . This allows small directories to be distributed evenly among all servers as long as the random permutations of server list are created with a good permutation function.

Similar randomization also helps in mitigating another use-case that may potentially overload the GIGA+ server that holds the zeroth partition for big directories. However, if millions of application threads choose to access one particular (large) directory for the first time, they may all overwhelm the server that holds the zeroth partition. To mitigate this use-case, GIGA+ clients avoid contacting the server with zeroth partition, and instead pick any random server from the directory's server list. If the randomly chosen server is not responsible for the desired partition, it is possible that either the server does not have any partitions or the server has split the partition.

In case the chosen server does not have any partitions for that directory (because the directory has not yet expanded to that server), then the server updates the client to try the parent of the partition it will eventually manage. This process continues until the client reaches the server that holds the appropriate partition. Because GIGA+ splits in a binary manner and maintains a split history, GIGA+ servers can traverse the hash-tree, described earlier in this chapter, to determine the presence or absence of desired partitions.

In the other case, when the chosen server is not responsible for the operation sent by the client (because of a split that is not known to the client), then that server treats the client's mapping state as *inconsistent* and invokes the process of updating the client's state using techniques described in Chapter 5. In a nutshell, the GIGA+ server sends a "hint" to fix the client's inconsistent mapping state and point it to a server that may be aware of the desired partition. The client then retries by sending the message to another server. This process is repeated until the client reaches the correct server.

However, there is one use-case that is not mitigated by random server selection: if a million new clients all access one particular small directory. This use-case of highly popular non-distributed objects may be alleviated through careful use of techniques, such as replicating the popular object and distributing read traffic over multiple replicas or caching the popular object closer to the client, that are beyond the scope of this dissertation.

4.7 Summary

This chapter presented the design, implementation and analysis of a key research contribution in GIGA+ indexing technique — the ability to split overflow partitions on multiple servers without system-wide co-ordination, serialization or blocking operations. GIGA+ uses asynchrony both when the number of servers is known a priori and when new servers are added to the system.

The two main lessons emerged from analysis of trade-offs made by the GIGA+ indexing technique. These lessons are widely applicable to other distributed systems aimed at massive scalability and concurrency.

- While splitting and distributing partitions is necessary to enable parallel access, splitting needs to be stopped when the system is *load balanced* on *all available servers*.
- Further splitting brings no performance benefit — in fact, it may have adverse affect on the overall system performance — until new servers are added and work needs to be transferred to them.

Chapter 5

Bounded inconsistency of indexing state

The previous chapter discussed the first design tenet of GIGA+ about splitting and distributing large directories in an asynchronous manner. This chapter focuses on the second tenet: *tolerating inconsistent state*. The term "inconsistency" generally has two connotations. One that is visible to the applications and other that is internal to the system. The former definition typically pertains to the inconsistency of application data; this data inconsistency is exposed to the application that is often in the best position to resolve these inconsistencies. GIGA+ uses the latter connotation, i.e. it has inconsistencies that are always within the system (and not exposed to the application) and are associated with the indexing state (and not pertaining to the application data).

GIGA+ tolerates inconsistency of the internal indexing-related state and provides strong consistency of the application data. Because GIGA+ is layered on top of backend storage systems, applications and users are offered the same data consistency guarantees as the underlying storage systems. In particular, GIGA+ focuses on providing POSIX-like

file system semantics which guarantee that once a directory entry is created in a directory all subsequent reads (or lookups) will see that directory entry.

This chapter presents how GIGA+ indexing embraces inconsistency by allowing the partition-to-server mapping to be stale and out-of-date. GIGA+ relies on the server-side split histories to resolve these inconsistencies in an lazy manner. Clients that use inconsistent mapping state need extra messages to reach the correct server. This overhead is governed by two factors: frequency of update messages and new information in each update message. This chapter studies the trade-offs that control these two factors.

The GIGA+ approach to use inconsistent updates was motivated by the complementary approach — to keep partition-to-server mapping consistently updated all the time — used in IBM GPFS's distributed directory implementation [Schmuck 2002]. IBM GPFS, as described earlier in Chapter 2, maintains a strongly consistent and synchronized mapping table in the on-disk i-node of the directory that is distributed on a shared disk subsystem [Schmuck 2002]. This property reduces the throughput of a workload that concurrently creates many files in one directory. GPFS users experience a significantly lower file creation rate for such workloads [Artiaga 2010, Hedges 2010, Cope 2005]. This dissertation is motivated by the need to avoid consistency and synchronization bottlenecks that hinder the scalability of high file creation rate of parallel applications.

5.1 Allowing inconsistent partition-to-server mapping

Recall from Chapter 4 that GIGA+ distributes a large directory's partitions such that each server only has a partial view of the index. For each server, this view comprises of the partitions that it manages and the history of how each of these partitions was split by

the server. GIGA+ clients, on the other hand, may or may not have any mapping information. A new client that accesses a large directory for the first time does not know the partition to server mapping, but a client that has been accessing a large directory continuously gradually caches *enough* partition to server mapping. This section describes how GIGA+ enables clients go from having no state to having enough state without disrupting the correctness of operations.

Imagine a client that performs a *stat*() operation to access a file F in a large directory D striped on many servers. Client begins by resolving the D's parent directory entry to get the i-node for D. In addition to the default i-node attributes, this i-node contains extended attributes for the GIGA+ specific directory metadata, including the server S_0 that holds partition P_0 and the server list associated with D. Thus a new client assumes that D has only one partition P_0 and knows only the partition-to-server mapping, $\{P_0 : S_0\}$, about that single partition.

Client uses the hash of the filename, $hash(F)$, to compute the partition identifier that *may* hold the file. Since the client knows only about one partition of the directory, it starts starts by assuming that P_0 holds the entire hash-space range and sends the request to server S_0. Partition P_0 may be the correct partition for file F if $hash(F)$ falls within the current hash-space range of P_0. The client operation, thus, reached the correct server that services it and sends a reply to the client.

However, partition P_0 may no longer be responsible for file F because a previous server-side split, which the client has not learned about, caused P_0 to change its hash-space range. In such a case, the client has addressed an *incorrect* server. This incorrectly addressed server, S_0, detects the addressing error by recomputing the partition identifier

by re-hashing the filename and comparing the partitions (and their hash-space ranges) it stores. (This process is similar to what the GIGA+ clients use send requests to appropriate partitions and servers.)

An addressing error indicates that the incorrectly addressed server was managed a partition that held the file *at some point in the past*. In other words, the partition has split one or more times which migrated the file to a different partition that is responsible for the hash-space range corresponding to the hash value of the filename. The client's partition-to-server mapping state is stale and unaware of the "new" partitions on the server.

The incorrectly addressed server S_0 uses its split history to send a reply that updates the client's indexing state. The client updates its cached version of the indexing state and recomputes the partition identifier to resent the request to a different server. If the newly addressed server S_1 also detects that its partition P_1 is no longer responsible for the file, it sends another update to the client based on its own split history. Client applies the update to its state and retries the request to another server until it finally reaches the correct server.

The drawback of allowing inconsistent indices is that clients may need additional probes before addressing requests to the correct server. The number of addressing errors incurred by any client is governed on two key factors: the new indexing state in an update message sent by a server and the frequency of at which servers send updates to clients. The next two sections, Section 5.2 and 5.3, discuss these in details.

5.2 Effect of new indexing state in update messages

The new indexing state in an update message sent by a server represents a server's guess about about the staleness of a client's indexing state. A GIGA+ server uses the number of partitions to encode the directory's indexing state. The top part of Figure 5.1 shows the hash tree of a GIGA+ directory that has split into several partitions on three servers. When a client first accesses the directory it aware only about the zeroth partition P_0 of the system. The update messages sent to a client with stale mapping allow the client to learn about different parts of the hash tree. These update messages can be in three forms described below.

The first form is that update messages only include information about the first split of partition P_0 which is partition P_1. Subsequent update messages from subsequent addressing errors point to more splits. In other words, the client starts from the root of the tree and performs a depth first traversal of the hash tree. This is labeled as *update form #1* in Figure 5.1(a). Thus, a client with an empty index may send $O(log(N))$ incorrect probes, where N is the total number of partitions in the system, before reaching the correct partition. This is the worst case for GIGA+ where each update message only points to the next partition in the system, i.e. the client traverses the hash tree one-level at a time. Furthermore, when GIGA+ servers have more that one partition per server, this mechanism may cause a client to address the same server multiple times as it searches for newer partitions.

To reduce the number of addressing errors, GIGA+ servers can send the *entire split history* of the incorrectly addressed partition. Unlike the naive technique, this second

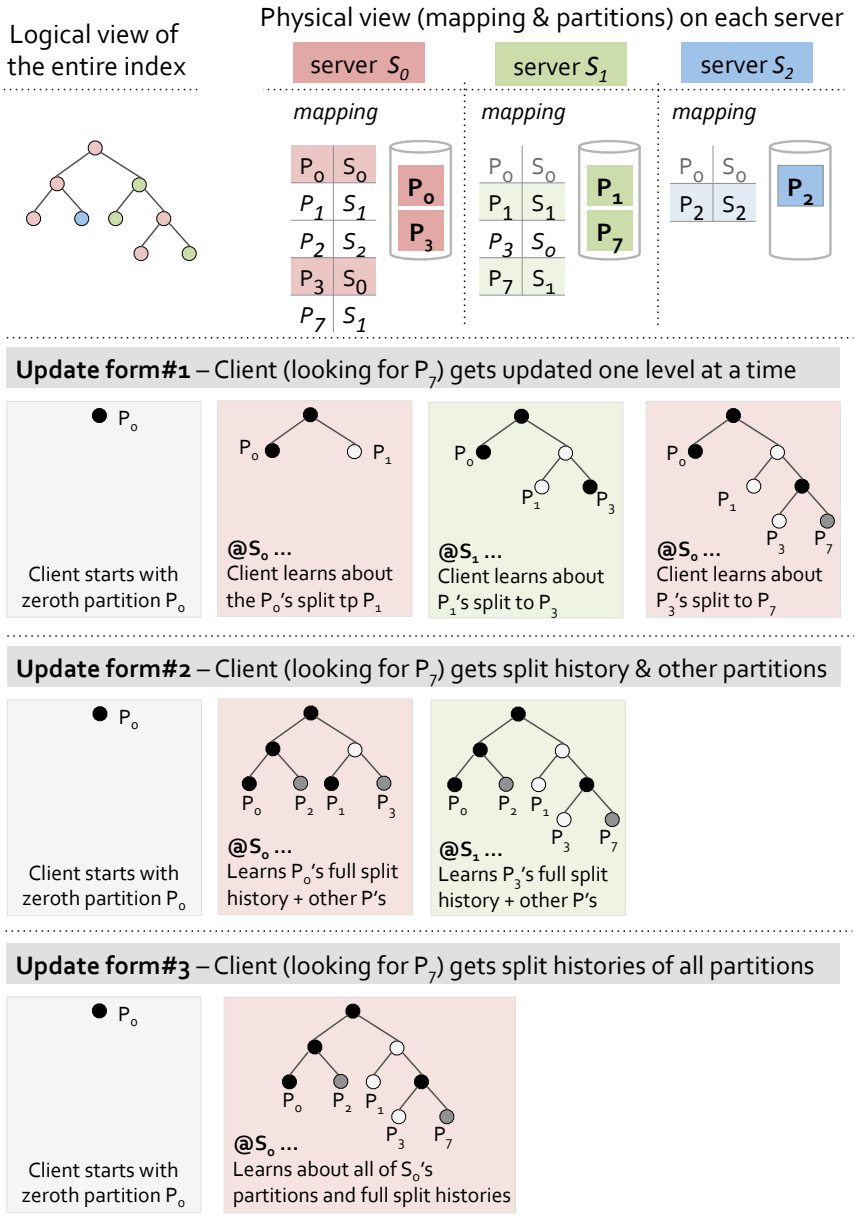

Figure 5.1 —Update message sent in response to addressing errors from clients with inconsistent mapping state.

Using a sample large directory split across three servers (show in the top part of this figure), this illustration shows three different forms of update messages each with more new indexing state. Although GIGA+ relies of the highest encoding message labeled *update form #2* (which needs more space than others), it allows GIGA+ clients to be learn about the state of the index much faster that the other forms of update that are more space-efficient (but incur more addressing errors).

form of updates allows GIGA+ clients to traverse multiple levels down the tree in one update message. However, these update messages only tell the client about a fragmented part of the hash tree as illustrated in Figure 5.1(b) with the label *update form #2*.

To further increase the new indexing state in an update message, GIGA+ includes the split history of not just the incorrectly addressed partition on a server, but the *split history of all partitions stored on that server*. Figure 5.1 shows that this form, labeled as *update form #3*, of update message allows a client to learn about more partitions quickly. In fact, this bounds the number of addressing errors to be no more than the number of GIGA+ servers in the system. In other words, for a given directory, a new client needs to visit each server at most once to have a complete index of partitions.

To represent the split history of all partitions on a server, GIGA+ uses a bitmap-based encoding to represent the partition state on each client and server. In this bitmap, a value '1' at the i^{th} bit indicates that P_i has been created, and value '0' indicates that P_i 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. Figure 5.2 illustrates how bitmaps are used when partitions are split and when partitions are accessed in GIGA+ .

For every directory, each GIGA+ server creates and maintains a separate bitmap. This bitmap tracks a server's partial view of the directory comprising of all partitions stored on a server and split histories of those partitions. And since GIGA+ servers split independently, each of them has a different bitmap. Union of bitmaps from all servers represents all the partitions of a large directory.

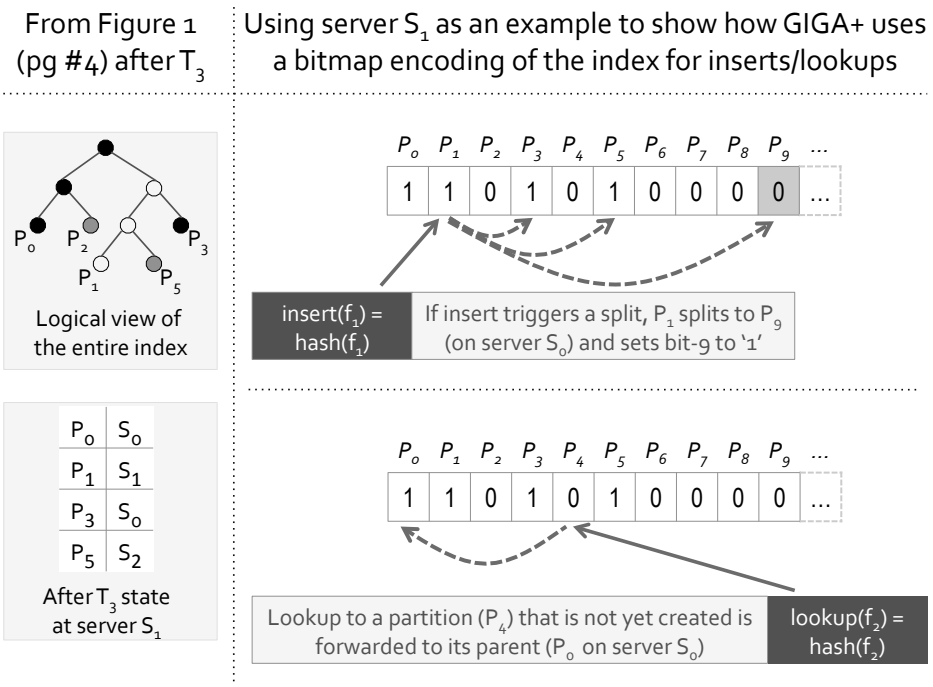

Figure 5.2 — Bitmap-encoding to store partitions and their split histories.
A bit-value of "1" indicates the presence of a partition on a server, and bit-value "0" indicates the absence of the partition on a server. This example show how bitmaps are used to choose a new partition to split into and to lookup the partition that holds the desired filename.

This bitmap encoding reduces both the memory footprint in clients and servers, and the network traffic caused by update messages from servers to clients. In the simplest bitmap scheme, the size of the bitmap is proportional to the number of partitions in that directory. However, for large directories with many partitions, GIGA+ can further compact these bitmaps by using the knowledge of how the GIGA+ hash-space tree splits.

Recall from Section 4.2 that the hash tree grows deeper with more splits; in other words, most partitions are either at level r or level $r + 1$ in the hash tree because of repeated splits of their parent partitions. Thus the bitmap, such as Figure 5.2, is likely

going to be a long sequence of 1s (indicating interior nodes of the tree representing parent partitions that have split) followed by a sequence mixed with 1s and 0s (indicating leaf nodes of the hash tree representing partitions at tree depth r or $r + 1$). Encoding of such bitmaps can be more efficient using compression techniques like run-length encoding [Salomon 2004] or using techniques that enable provide space efficiency at the cost of higher addressing errors [Litwin 1996].

The latter trade-off is particularly interesting. The GIGA+ encoding takes more space (to represent split histories of all partitions on a server) and allows clients to learn about the hash tree faster, which results in fewer addressing errors. In contrast, the distributed variant of linear hashing, called LH*, takes a complimentary approach: a space-efficient index representation for a higher number of addressing errors [Litwin 1996]. LH* uses a two variables to represent the current state of the hash index: the highest depth of the hash tree at which all partitions have split and the index of the most recently created partition. But the cost of this two-variable representation is that LH* clients may suffer from addressing errors even when the index stops splitting on the LH* servers, particularly when they are using optimizations for high concurrency splitting [Litwin 1996].

A similar scheme can be used to further optimize the GIGA+ bitmap encoding. GIGA+ can also use two variables corresponding to LH*: the lowest index in the bitmap that is set to `0' and the highest index in the bitmap that is set to `1'. However, the GIGA+ design chose to incur fewer addressing errors at the cost of a minimal space overhead, and GIGA+ guarantees that these addressing errors will stop once servers stop splitting a directory's partitions.

5.3 Effect of directory mutation rates

The cost of using inconsistent indexing state is governed by another factor: the frequency at which clients receive these update messages. The client update frequency is a function of two variables that are dependent on the workload. The first variable, called insert rate, is the rate at which servers are splitting partitions. The second variable, called lookup rate, is the rate at which clients are accessing (but not mutating) that directory.

If the insert rate is lower than the lookup rate, the partitions are splitting slower than the rate at which clients' mapping state becomes inconsistent. And even if a client's mapping becomes stale due to a recent split, it can catch up quickly because it already knows about most other existing partitions. If the insert rate is higher than the lookup rate, clients' mapping state becomes inconsistent more often. In other words, the server-side view of the hash tree is deeper than the client-side view of the tree. This requires clients to get updated quickly either through frequent addressing errors or through update messages that convey more new partitions.

To understand the cost of using inconsistent mapping, Figure 5.3 measures the incorrect addressing overhead, on Y-axis, as the fraction of all client requests that were re-routed during a create-intensive benchmark for different configurations (on X-axis). In this benchmark, an N-server configuration stores an empty directory that will contain $N/2$ million files created concurrently by a total of N remote client machines, i.e. 1-server configuration has 500,000 files created by 8 application threads running on one client machine, 2-server configuration has 1,000,000 files created by 16 application threads

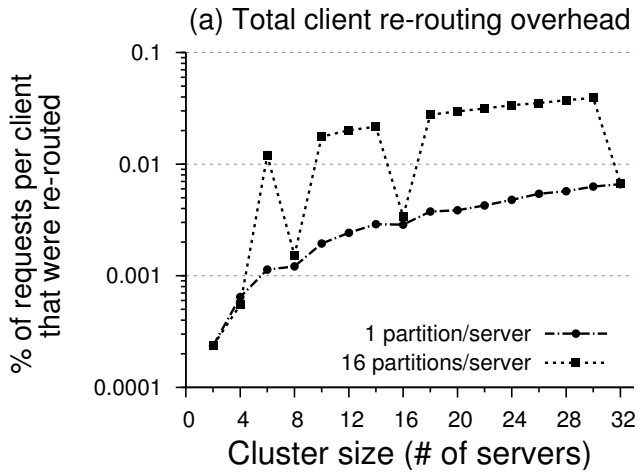

Figure 5.3 — Fraction of requests that incur addressing errors due to inconsistent indexing state at GIGA+ clients.
This graph measures the fraction of requests that are addressed to incorrect servers for configurations with varying number of GIGA+ servers. For every configuration, the experiment was repeated with one partition per server and with 16 partitions per server. The observed results show a very neglible incorrect addressing overhead (less than 0.05% of total requests).

running on two client machines, and so on. For all server setups, this figure shows results of GIGA+ configurations that use one partition per server and 16 partitions per servers.

Figure 5.3 shows that, in absolute terms, fewer than 0.05% of the requests are addressed incorrectly; this is only about 250 requests per client because each client is doing 500,000 file creates. The number of addressing errors increases proportionally with the number of partitions per server because it takes longer to create all partitions. In cases where the number of servers is a power-of-two, after each server has at least one partition, subsequent splits yield two smaller partitions on the same server, which will not lead to any additional addressing errors.

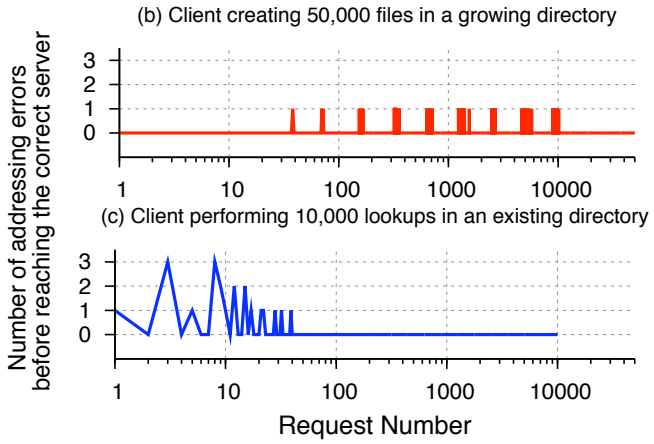

Figure 5.4 — Occurrence of addressing errors based on the workload executed by a GIGA+ clients.
This graph shows how often addressing errors occur and when do they stop for two separate GIGA+ clients, one that performs inserts in a growing directory (top graph) and other that performs lookups in a directory that it has never accessed before (bottom graph). In both cases, incorrect addressing occurs for only for initial few requests until the clients learns about all the servers in the system.

To better understand how addressing errors occur and when they stop happening, Figure 5.4 takes a closer look at the worst case from Figure 5.3: the 30-server configuration with 16 partitions on each server. This figure consists of two graphs that report the number of addressing errors (on Y-axis) incurred by two GIGA+ clients during the execution of their respective workloads.

The top graph, Figure 5.4(b), shows the number of errors encountered by each request generated by one client thread (i.e., one of the eight workload generating threads per client) as it issues 50,000 file create requests (on the X-axis) in a directory. This client thread is one of 240 client threads that are all simultaneously inserting 50,000 files in a single directory that is spread on 30 GIGA+ servers with 16 partitions on each server.

Figure 5.4(b) has three observations. First, the index update that this client thread receives from an incorrectly addressed server is always sufficient to find the correct server on the second probe. Second, addressing errors are bursty, one burst for each level of the index tree needed to create 16 partitions on each of 30 servers, or 480 partitions ($2^8 < 480 < 2^9$). And finally, that the last 80% of the work is done after the last burst of splitting without any addressing errors.

To further emphasize how little incorrect server addressing clients generate, Figure 5.4(c) shows the addressing errors of a new client that executes a lookup workload. This new GIGA+ client issues 10,000 lookup requests (on X-axis) in a large directory that it has not accessed before but was created before it starts issuing lookup requests. This large directory is created in the previous experiment, i.e. 15 million files in a directory distributed on 30-server configuration with 16 partitions per servers. Compared to the insert-only workload in Figure 5.4(b), this lookup-intensive client incurs frequent addressing errors at the beginning because it starts with no state about the GIGA+ index. This client makes no more than 3 addressing errors for a specific request, and no more than 30 addressing errors total and makes no more addressing errors after the 40th request. This lookup-intensive client stops incurring addressing errors much faster than then insert-only client, which stops getting addressing errors after about 10,000 of its 50,000 requests, because the latter keeps getting periodic addressing errors until all partitions are created in the system.

5.4 Summary

GIGA+ allows clients to have inconsistent indexing state which may cause them to send requests to incorrect servers. The server-side split histories enables incorrectly addressed servers to detect this addressing error and provide hints to fix this error. These hints allow GIGA+ clients to update their indexing state and re-address their request to another server.

GIGA+ minimizes these addressing errors incurred by the client by making one key trade-off: it increases the new indexing state sent in an update message to decrease the number of addressing errors. GIGA+ uses a bitmap-encoding that allows a server to send split histories of all its partitions. This encoding enables clients to learn about new partitions quickly

Furthermore, GIGA+ ensures that addressing errors are bounded by the number of servers in the system: if GIGA+ servers have stopped splitting partitions, clients will not incur addressing errors after they have visited each server once.

Chapter 6

Interaction with backend stores

The GIGA+ architecture presented in Chapter 3 (and illustrated in Figure 3.1) briefly discussed how the choice of backend stores affects the behavior and performance of the system. This chapter presents details about how GIGA+ servers use two kinds of backends — a local on-disk store available only on that server, and a shared disk storage system that can be accessed through clients and servers — and how their implementations affect the behavior of GIGA+ .

6.1 Using local file system as backend

Many modern cluster file systems rely on local file systems to store data persistently on disks or SSDs attached to storage servers. Examples of such file systems include the Google file system [Ghemawat 2003], Hadoop distributed file system [Shvachko 2010], Lustre [Lustre 2010b], and PVFS [PVFS2 2010].

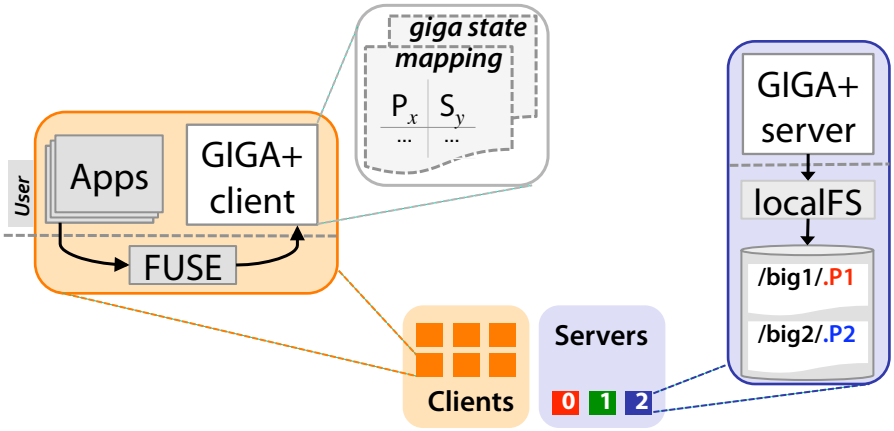

Figure 6.1 — GIGA+ that uses a local file system as the backend store.
GIGA+ servers are responsible for managing and storing hash partitions. These partitions are stored in a local file system as a regular directory; this allows GIGA+ to uses a separate logical namespace (that is seen by an application using a GIGA+ client) and physical namespace (that is stored in backends stores).

When GIGA+ servers use a local file system as a backend store, the server is responsible for both managing and storing directory entries. GIGA+ servers perform the indexing functionality and call in the local file system to access its hash partitions. Figure 6.1 shows a configuration where GIGA+ server stores the hash partitions as directories in a local file system.[1]

GIGA+ partitions are stored in the local file system as regular directories. In this configuration, a logical large directory is represented physically as many small directories corresponding to the partitions associated with GIGA+ servers. Figure 6.2 shows how the logical namespace, seen by the application running a client, is different from the physical namespace, seen on the backend stores in GIGA+ servers. The example in this figure

[1] Although Figure 6.1 shows a single disk on one physical server, there a multiple disks running on real-world clusters.

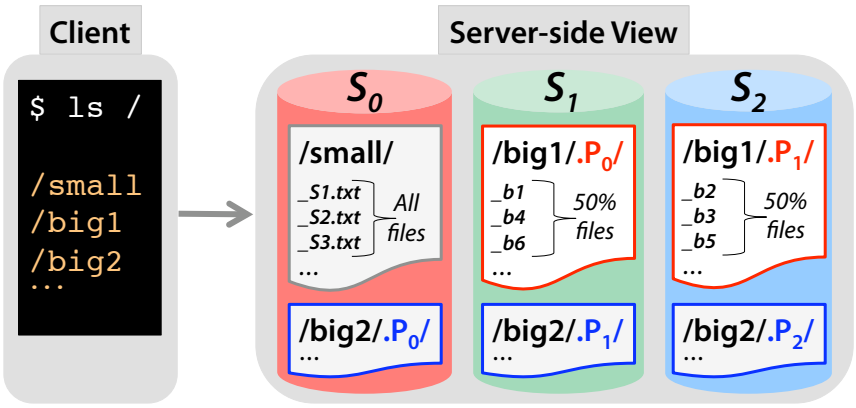

Figure 6.2 — Namespace separation in GIGA+ : application namespace is different from physical namespace.
By storing partitions as directories in a local file system, GIGA+ separates the logical namespace (that is seen by applications running on client) from the physical namespace (that is stored in backends stores on GIGA+ servers). In this example, client visible directories are stored on one or more servers (depending on their size) as local file system directories on GIGA+ servers.

shows that a large directory */big1* is divided in two partitions that are stored as local file system directories */big1/.P0* and */big2/.P0* on servers S_1 and S_2. If the application runs a *"ls /big1"* commands, the GIGA+ client fetches and returns files from both partitions.

When a partition splits, the GIGA+ server performs a *readdir*() scan of the physical directory associated with that partition and migrates the appropriate directory entries (along with the data, if any) to the destination of the newly created partition. The recipient GIGA+ server creates a new directory corresponding to the newly created partition and inserts the directory entries in that directory. The main drawback of such splits is that the GIGA+ server also needs to move the file data associated with these directory entries because local backend stores both the directory entry and its associated file data. Depending on the size of files, which can easily be gigabytes or more, partition splits

become very expensive. The experiments in this section focus on directory entries and, hence, use zero-byte files.

GIGA+ servers have used both Linux file systems, Ext3 and ReiserFS, as backend stores. Implementations of these file systems have significant differences that have surprising effects on the performance of the system. Figure 6.3 shows how the file create rate varies in a 16-server configuration for four different configurations: (1) one partition per GIGA+ server that uses Linux Ext3, (2) one partition per GIGA+ server that uses ReiserFS, (3) 16 partitions per GIGA+ server that uses Linux Ext3, and (3) 16 partitions per GIGA+ server that uses ReiserFS. Each of these configurations starts with a new file system. The experiment creates an empty directory that is populated with 8 million files, causing the large directory to be striped on 16 GIGA+ servers.

Figure 6.3 shows two interesting phenomena. First, the benchmark running time varies by a factor of six, from about 100 seconds to over 600 seconds, and second, the backend file system yielding the faster performance is different when there are 16 partitions on each server than with only one.

Comparing a single partition per server in GIGA+ over ReiserFS and over Ext3 (left column in Figure 6.3), benchmark completion time increases from about 100 seconds using ReiserFS to nearly 170 seconds using Ext3. For comparison, the same benchmark completed in 70 seconds when the backend was the in-memory *tmpfs* file system.

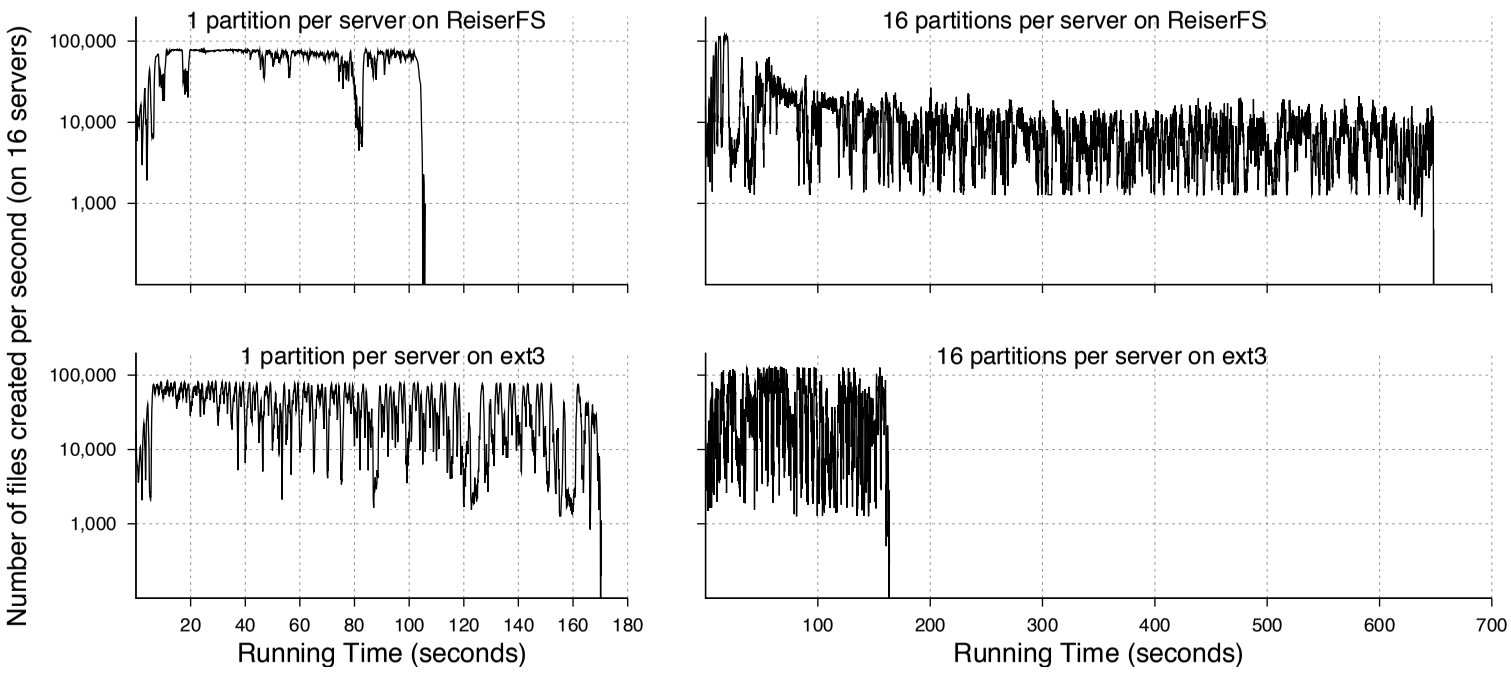

Figure 6.3 — Effect of underlying file systems.
This graph shows the concurrent create benchmark behavior when the GIGA+ directory service is distributed on 16 servers with two local file systems, Linux Ext3 and ReiserFS. For each file system, two different numbers of partitions per server, 1 and 16, are shown in this figure.

Directories are indexed in Linux Ext3 using the h-tree data-structure [Cao 2007] and in ReiserFS using the balanced B-tree structure [Reiser 2004]. Looking more closely at Linux Ext3, as a directory grows, Ext3's journal also grows and periodically triggers Ext3's *kjournald* daemon to flush a part of the journal to disk. Because directories are growing on all servers at roughly the same rate, multiple servers flush their journal to disk at about the same time leading to a significantly lower aggregate file create rate. This behavior was observed for all three journaling modes supported by Ext3 [Cao 2007, Prabhakaran 2005].

To confirm the effect of journaling, the above experiment was repeated with a multi-disk configuration where the journal was mounted on a second disk in each server. This eliminated most of the throughput variability observed in Ext3, completing the benchmark almost as fast as with ReiserFS.

For ReiserFS, however, placing the journal on a different disk had little impact. This is explained by the second phenomenon observed in the right column of Figure 6.3. It shows that for GIGA+ with 16 partitions per server, Ext3 (which is insensitive to the number of partitions per server) completes the create benchmark more than four times faster than ReiserFS. This results, possibly, from the on-disk directory representation. ReiserFS uses a balanced B-tree for *all objects* in the file system, which re-balances as the file system grows and changes over time [Reiser 2004]. When partitions are split more often, as in case of 16 partitions per server, the backend file system structure changes more, which triggers more re-balancing in ReiserFS and slows the create rate.

6.2 Shared storage backend with optimized layout

The previous section described two hard problems from using local file systems as backends. First, on-disk structures and directory representations of modern Linux local file systems are inefficient at handling large directories. Second, GIGA+ splits become expensive when they migrates both directory entries and their file contents across servers. To overcome these two challenges, GIGA+ uses a backend configuration that combines a metadata-optimized on-disk representation on each GIGA+ server and a shared disk storage system that can be accessed from both GIGA+ clients and servers.

GIGA+ uses a persistent key-value storage library, called LevelDB , to store all metadata associated with directory entries and GIGA+ specific indexing state. [LevelDB 2012]. This LevelDB based representation delivers higher performance than local file systems for metadata-intensive workloads [Ren 2013].

Using shared storage allows cross-server GIGA+ splits to migrate only the directory entries and not the file contents. Since the file contents are stored in the shared storage system, GIGA+ clients can access the file contents directly without going through the GIGA+ servers. Figure 6.4 shows the architecture that allows GIGA+ clients to decouple metadata path from the data path by using the GIGA+ indexing modules to handle only metadata operations such as *create()*, *mkdir()* and *open()*. These metadata operations return a pointer (or a file handle) to the actual file contents stored in the backend share storage such as a cluster file system in Figure 6.4.

This section describes more details: Section 6.2.1 shows how LevelDB stores all file system metadata using a single on-disk structure on each server, Section 6.2.2 describes

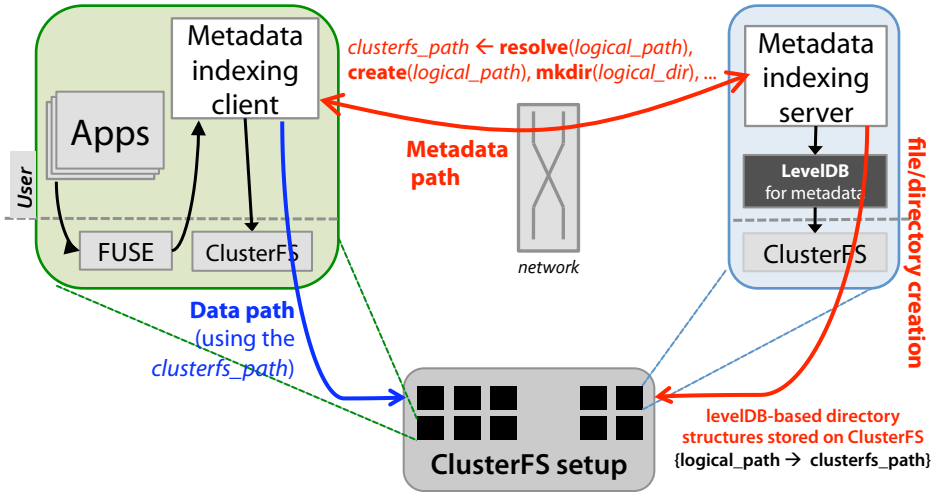

Figure 6.4 — Different data and metadata paths when GIGA+ is layered on a shared storage backend.
GIGA+ is integrated with a tabular metadata-optimized on-disk layout (using LevelDB) on each server and a shared storage space that allows efficient cross-server operations (such as splitting without migrating file contents).

---

the challenges in effectively integrating GIGA+ and LevelDB to work with existing cluster file systems, and Section 6.2.3 analyzes the scale of performance of using GIGA+ with LevelDB .

6.2.1 *Overview of LevelDB*

LevelDB is an open-source key-value storage library for on-disk storage [LevelDB 2012]. It is based on the Google BigTable's server-side tablet architecture [Chang 2006] that implements the log-structured merge (LSM) tree data structure [O'Neil 1996]. LSM trees enable high-speed write performance using an in-memory buffer that delays writing new and changed entries until it has a significant amount of change to record on disk. The process

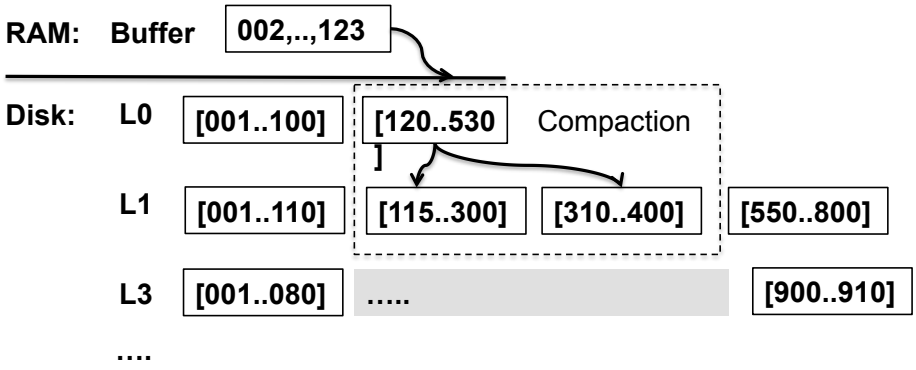

Figure 6.5 — Structure of LevelDB
LevelDB represents data on disk in multiple SSTables that store sorted key-value pairs.

of writing in-memory data to an on-disk representation uses multi-level trees that are ordered on the key used by the application.

Using LevelDB as a local storage representation for metadata can transform metadata updates to large, non-overwrite, sorted and indexed logs on disks, which greatly reduces random disk seeks [Ren 2013]. The detailed design of LevelDB and how to use LevelDB to store metadata is explained below.

LevelDB and LSM-trees — LevelDB buffers all modifications, which includes writing new entries and updating old entries, to its entries in a sequential log kept in an in-memory buffer. These modifications are spilled to disk when the in-memory buffer exceeds 4 MB; this process is called a minor compaction [Chang 2006]. When a spill is triggered, buffered entries are sorted, indexed and written to disk in a format called SSTable [Chang 2006]. These entries may then be discarded from the in memory buffer and can be reloaded by searching each SSTable on disk, possibly stopping when the first match

occurs if the SSTables are searched most recent to oldest. The number of SSTables that need to be searched can be reduced by maintaining a Bloom filter on each, but, with time, the cost of finding a record not in memory still increases [Chang 2006]. To avoid keeping large numbers of SSTable files, LevelDB triggers a background process, called major compaction, that combines multiple SSTables into a smaller number of SSTables by merge sort.

As illustrated in Figure 6.5, LevelDB extends this simple approach to further reduce read costs by dividing SSTables into several levels. In Level-0, each SSTable may contain entries with any key value, based on what was in memory at the time of its spill. The higher levels of SSTables are the results of compacting SSTables from their own or lower levels. In these higher levels, LevelDB maintains the following invariant: the key range spanning each SSTable is disjoint from the key range of all other SSTables at that level. So querying for an entry in the higher levels only needs to read at most one SSTable in each level. LevelDB also sizes each of the higher levels differentially: all SSTables have the same maximum size and the sum of the sizes of all SSTables at level L will not exceed 10^L MB. This ensures that the number of levels grows logarithmically with increasing numbers of entries. LevelDB compactions are inspired by LSM trees [O'Neil 1996], but such background operations are also found in newer data-structures such as streaming B-trees in the Toku file system [Esmet 2012, Bender 2007] and stratified trees in the Acunu storage engine [Twigg 2011].

For data consistency, LevelDB offers the same guarantees as most local file systems. To avoid high performance penalty, LevelDB does not synchronize all data to disk immediately; it flushes the data to disk every 5 seconds [Ren 2013]. However, users can con-

figure this duration depending on their desired consistency-performance tradeoff. This periodic synchronization is similar to local Linux file systems that also use a 5-second (or 30 seconds, in some older versions) interval to flush the journal to persistent storage [Prabhakaran 2005].

Table schema — LevelDB stores records of key-value pairs indexed lexicographically on the key. The file system metadata is stored in this LevelDB abstraction based on the metadata schema proposed by TableFS [Ren 2013]: directory entries and i-node attributes are aggregated in a single LevelDB table with a row for each file and directory. To link together the hierarchical structure of the user's namespace, rows of this table are ordered by a 224-bit key consisting of the 64-bit i-node number of a file's parent directory and a 160-bit SHA1 hash value of its filename string (final component of its pathname). The value of a row contains the file's full name, its i-node attributes (from *struct stat* in Linux), and a symbolic link that contains the actual path of the file object in the shared storage system (such as a cluster file system). Figure 6.6 shows an example of storing a sample file system's metadata into one LevelDB table.

All the entries in the same directory have rows that share the same first 64 bits in their the table's key. For *readdir*() operations, once the i-node number of the target directory has been retrieved, a scan sequentially lists all entries having the directory's i-node number as the first 64 bits of their table's key. To resolve a single pathname, the metadata server starts searching from the root i-node, which has a well-known global i-node number (0). Traversing the user's directory tree involves constructing a search key by

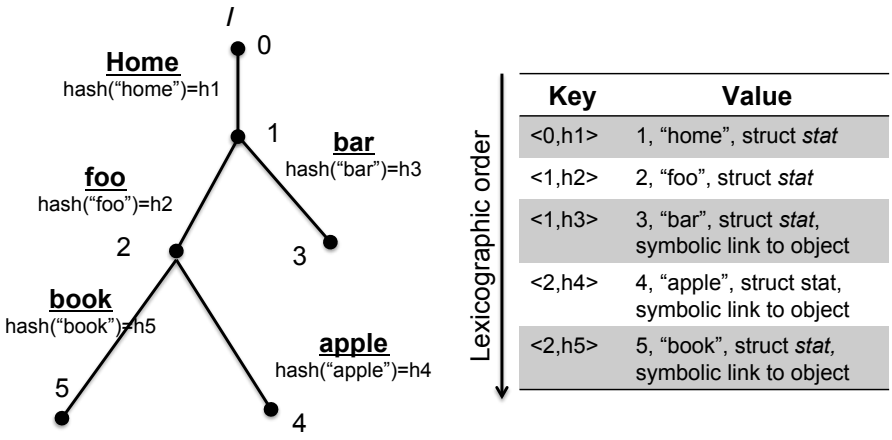

Figure 6.6 — Schema used in LevelDB to store file system metadata
An example illustrating how the file system metadata is stored in LevelDB records.

concatenating the i-node number of current directory with the hash of next component name in the pathname.

6.2.2 Integrating LevelDB with GIGA+

Each GIGA+ server manages its local LevelDB instance that is stored on a local disk attached to the server.[2] In the current GIGA+ prototype, GIGA+ servers use a NFS mounted volume, accessible from all GIGA+ servers, to emulate shared storage.

This integration of GIGA+ indexing, shared storage and LevelDB requires three features to be efficient: a schema to represent directory entries and partitions in LevelDB , a mechanism to split overflow partitions across servers without transferring the data, and a data access path that is decoupled from the metadata access path to scale reading and writing files.

[2]If a shared storage system is available on all nodes of the cluster, LevelDB instance can be stored in that system but each GIGA+ server's instance should be unique.

Metadata representation — LevelDB stores all metadata including GIGA+ hash partitions for directories, entries in each hash partition, and other bootstrapping information such as root entry and GIGA+ configuration state. The general schema used is:

KEY	*parentDirID, gigaPartitionID, hash(dirEntry), dirEntry*
VALUE	*attr(dirEntry), [symlink | data | gigaMetaState]*

The main difference from the schema used in TableFS described in Section 6.2.1 is the addition of two GIGA+ specific fields: *gigaPartitionID* to identify a GIGA+ hash partition and *gigaMetaState* to store the hash partition related mapping information. These GIGA+ related fields are used only for large directories that are distributed on multiple GIGA+ servers.[3]

Partition splitting — Each GIGA+ hash partition and its directory entries are stored in SSTable files in a local LevelDB instance. Recall that each GIGA+ server process splits a hash partition P on overflow and creates another hash partition P' which is managed by a different server. This split involves migrating approximately half the entries from old partition P to the new hash partition P' on another server during which the key range in write is locked. Cross-server splits of LevelDB partitions can be done in several ways.

A simple approach to splitting is to perform a LevelDB range scan on partition P and copy about half the results (corresponding to the keys that are migrated to the new

[3] Since the hash(dirEntry), which is the hash value of the directory entry, is a part of the key, it is possible to use this hash value to identify hash partitions if the same hash function is used for both GIGA+ and LevelDB keys. This optimization can eliminate the need for gigaPartitionID in the schema; the current GIGA+ prototype does not implement this optimization yet.

partition) from P in a buffer. This buffer of entries is copied in an RPC message that is sent to the server that holds the new partition P'. The recipient server inserts each key from the received buffer in its own LevelDB instance. This approach is attractive for its simplicity, but it can be slow because the server that receives the split inserts each key incrementally. To speed up splits, GIGA+ uses a bulk insertion optimization in LevelDB that communicates splits through a shared storage volume.

The immutability of LevelDB SSTables makes it possible to implement a fast bulk insertion that adds an SSTable to Level 0 without pushing its data through the write-ahead log and minor compaction process. To leverage this optimization, LevelDB was modified to support a three-step split operation. First, the split initiator performs a range scan on its LevelDB instance to find all entries that need to be moved to the new partition on another server. Results of this scan are written in an SSTable file (understood by LevelDB) that is stored in the shared storage volume. Second, the split initiator sends an RPC to the split receiver with the location of this intermediate SSTable file; this RPC is much smaller in size than an RPC that sends all the keys over the wire. The split receiver then reads the intermediate SSTable file and bulk inserts the file into the LevelDB tree structure; this bulk insert is also faster than the previous iterative insertion of one at a time. The final step is a clean-up and commit phase: after the receiver completes the bulk insert operation, it notifies the initiator to delete the migrated hash-range from its LevelDB instance and unlock the range.[4]

[4]This three-phase split can be refined even further: LevelDB can use symbolic links to the intermediate files without explicitly copying the files through shared storage volume. Because the current release of LevelDB does not have support for links, this optimization is not a part of the current prototype.

Decoupled data and metadata path — All metadata operations go through the GIGA+ server; however, following the same path for data operations would incur an unnecessary performance penalty of shipping data over the network an extra time. That is, data is copied from the shared storage system to the GIGA+ server, and then copies it again from the GIGA+ server to the GIGA+ client. This penalty can be significant in HPC use-cases where files can easily be tens to hundreds of gigabytes in size.

This migration cost is mitigated by performing all data path operations directly through the shared storage module (or client module of a cluster file system) on the machine running the GIGA+ client. Figure 6.4 shows this data path in blue color. After a GIGA+ client completes a lookup on a desired file name through GIGA+ servers, it gets back a symbolic link to the physical path in the shared storage system. Recall that the schema, shown in 6.6, used in LevelDB includes a symbolic link to the actual on-disk location of the object. All subsequent accesses using this symbolic link will force the client operating system to resolve this link into the underlying shared storage system (either a NFS-mounted volume or a cluster file system). While the file is open, some of its attributes (e.g., file size and last access time) may change relative to copy of attributes stored in LevelDB instance of the GIGA+ server. GIGA+ server will capture these changes on file close on the metadata path. Other attribute changes relative to permissions can be updated in-flight through the GIGA+ servers.

6.2.3 *Analysis*

To emulate shared storage for split operations, the current GIGA+ prototype uses a NFS-mounted volume accessible from all machines; this volume was only used for the cross-server LevelDB split optimization described in Section 6.2.2. This experimental setup is used to evaluate the performance of a single-node LevelDB metadata store and the scalability of GIGA+ distributed directories on a 64-server setup (using the hardware described in Chapter 3).

The baseline performance of a single-node LevelDB metadata store was measured using a workload that creates 100 million zero-length files in a single directory. Figure 6.7 reports the instantaneous throughput, measured as files created per second, on Y-axis during the run time of the workload (on X-axis).

Figure 6.7 compares the instantaneous throughput of LevelDB metadata store with three Linux file systems: Ext4 [Mathur 2007], XFS [Sweeney 1996], and BTRFS [Btrfs 2012]. All systems perform well at the beginning of the test, but the file create throughput drops gradually for all systems. BTRFS suffers the most serious throughput drop, slowing down to 100 operations per second. LevelDB incurs a gradual degradation in throughput; this happens due to the periodic background compactions that occur during the entire experiment as more files get created over time. If there are pre-existing files from previous compactions, LevelDB merges these old files with newly arrived in-memory logs to write out most recent sorted sequential files. These compactions cause both read and write disk I/O: existing sorted files are read into memory and newly merged files are written back to disks. When there are more entries already existing in LevelDB, it requires more

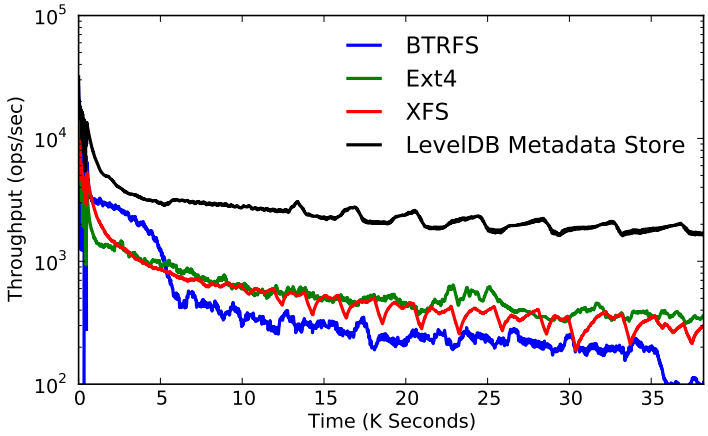

Figure 6.7 — Single-node baseline performance on an on-disk LevelDB backend. LevelDB-based metadata store is 10X faster than modern Linux file systems for a workload that creates 100 million zero-length files. X-axis only shows the time until LevelDB finished all insertions because the other file systems were much slower. Y-axis uses a log scale.

compaction work to maintain LevelDB invariants and to perform a negative lookup before each create has to search more SSTables on disk. The LevelDB metadata store, however, maintains a more steady performance with an average speed of 2,200 operations per second respectively, and is 10X faster than all other tested file systems.

After establishing the baseline performance, the next experiment evaluates the scalability of the distributed directory setup that uses LevelDB store on GIGA+ servers. Figure 6.8 shows the instantaneous throughput, measured as file creates per second, on Y-axis during the entire run time of the workload that creates many files in a single directory striped on different number of GIGA+ servers. This workload creates millions of files in a single directory in a strong scaling setup: an N-server configuration stores an empty directory that will contain N million files created concurrently by a total of $8N$-threads running on N remote client machines, i.e. 1-server configuration has 1 million files cre-

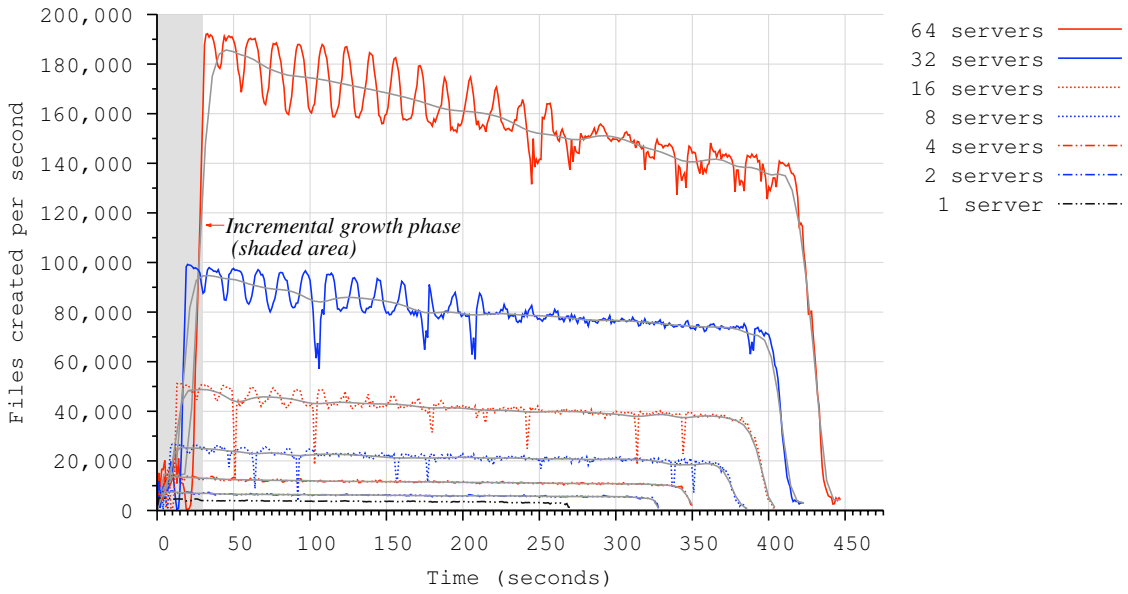

Figure 6.8 — Scale-out performance on an on-disk LevelDB backend.
GIGA+ using a disk-based LevelDB backend shows promising scalability up to 64 servers. Note that at the end of the experiment, the throughput drops to zero because clients stop creating files as they finish 1 million files per client. However the interaction between LevelDB 's compaction policies and the Linux Ext3's implementation policies causes periodic throughput variance that degrades as the the number of directory entries in each LevelDB increases. *Solid lines in each configuration are Bezier curves to smooth the variability.*

ated by 8 application threads running on one client machine, 2-server configuration has 2 million files created by 16 application threads running on two client machines, and so on up to 64 million files on 64 servers. Each GIGA+ server stores its hash partitions in LevelDB that is stored on an on-disk Linux file system managing a single 7200RPM disk; cross-servers splits in LevelDB are communicated through a NFS mounted volume shared by all GIGA+ servers.

The main result in Figure 6.8 is that as the number of servers doubles the throughput of the system also scales up. With 64 servers, GIGA+ can achieve a peak throughput of

about 160,000 file creates per second. The prototype delivers peak performance after the directory workload has been spread among all servers. Reaching steady-state, the throughput quickly grows due to the splitting policies adopted by GIGA+ .

After reaching the steady state, throughput slowly drops as LevelDB builds a larger metadata store. In fact, in large setups with 8 or more servers, the peak throughput drops by as much as 25% (as in case of the 64-server setup). This is because when there are more entries already existing in LevelDB , it requires more compaction work to maintain LevelDB invariants and to perform a negative lookup before each create has to search more SSTables on disk. In theory, the work of inserting a new entry to a LSM-tree is $O(\log_B(n))$ where n is the total number of inserted entries, and B is a constant factor proportional to the average number of entries transferred in each disk request [Bender 2007]. Thus the formula $\frac{a \cdot S + b}{\log T}$ can approximate the throughput timeline in Figure 6.8. In this formula, S is the number of servers, T is the running time, and a as well as b are constant factors relative to the disk speed and splitting overhead. This estimation projects that when inserting 64 billion files with 64 servers, the system may deliver an average of 1,000 operations per second per server, i.e. 64,000 operations per second in aggregate.

6.3 Summary

The GIGA+ file system directory service is designed to be layered on top of existing, unmodified storage systems that need highly concurrent directory accesses. This makes it important that GIGA+ be layered in a manner that uses the lower-level system effectively. Splitting partitions, in particular, is tightly dependent on the choice of the backend store.

When GIGA+ is layered on local file systems managed by GIGA+ servers, split operations are very expensive. Because a local file system stores the directory entry and its associated file contents on the same server, cross-server GIGA+ splits need to migrate both the directory entries and the associated contents (which can be gigabytes or more in size). Even when this file-related data migration was avoided (by using zero-byte files in synthetic workloads) during experimentation, GIGA+ scalability is tightly dependent on how the local file systems, such as Linux Ext3 and ReiserFS, store directories on-disk.

To avoid costly data migration and inefficient on-disk representation, GIGA+ was layered on a metadata-optimized backend store that used shared storage for cross-server splits. Such a setup is an ideal setup for GIGA+ because splits only move the directory entries managed by GIGA+ servers without migrating any file data that is stored in the shared storage system. GIGA+ used LevelDB for high-performance on-disk metadata representation along with a NFS mounted shared volume to emulate shared storage. This LevelDB-based metadata store outperforms modern Linux file systems by an order of magnitude on a single machine. For distributed directories, this GIGA+ configuration scaled linearly up to 64 servers delivering a peak throughput of more than 160,000 file creates per second.

Experiments with different backend file systems demonstrate that the design and implementation of these backends have subtle and significant side-effects on the performance of the GIGA+ directory service. This analysis recommends further research, which is outside the scope of this dissertation, in on-disk representations that will be suitable for emerging metadata-intensive workloads such a large file system directories.

Chapter 7

Conclusion

Growth in application-level parallelism and diversity among data-intensive workloads is posing new scale and concurrency challenges for cluster file systems. For decades, cluster file systems have focused on scalable performance on the data path and ignored scaling the metadata path. As applications start becoming more metadata-intensive, cluster file systems need to evolve their support for highly parallel and distributed access to the metadata such as file system namespace and directories. While some cluster file systems have evolved to support distributed namespace operations, most of these systems do not provide decentralized directory access. Few cluster file systems that provide distributed directories suffer from serialization and consistency bottlenecks when applications are accessing a single directory concurrently at high speeds.

This dissertation presented a new directory subsystem called GIGA+ that delivers promising scale and performance for workloads that need to store millions to billions of files in a single directory at hundreds of thousands of file operations per second. The

central idea of GIGA+ is a concurrent hash-based indexing technique that embodies two principles — asynchrony and inconsistency — to distribute large directories on many servers. GIGA+ splits a growing directory into several partitions in a manner that spreads the directory incrementally across multiple servers without any system-wide serialization or synchronization. Each GIGA+ server makes an independent decision to split its directory partition on another server and keeps a history of splits for each of its partitions. As the index spreads over multiple servers, GIGA+ allows the indexing state (partition-to-server mapping) at clients to become stale and out-of-date. This client state is eventually updated in a lazy manner by a GIGA+ server that is addressed incorrectly. GIGA+ relies on asynchrony and inconsistency to add new servers: an existing directory service expands on to new servers asynchronously through minimal data migration and re-balancing while lazily disseminating information about new servers as needed.

GIGA+ indexing explores several factors that control the tradeoff between consistency and concurrency of the indexing state. Table 7.1 summarizes these trade-offs and lessons learnt from their analysis.

GIGA+ file system directory prototype is built as a user-level client-server architecture that layers its indexing technique on an unmodified backend storage system. This architecture allows modularity by separating how directory entries are indexed on multiple servers from how they are stored on persistent storage; it decouples the logical namespace used by applications from the physical namespace stored on the backend stores. GIGA+ servers perform this translation. Applications use the traditional hierarchical namespace to access a file through GIGA+ clients and send the request to GIGA+ servers that access the file using the physical namespace used by the backend storage system.

Trade-off #1	Incremental scale-out (on one server at a time) instead of aggressive scale-out (over all servers at once) Incremental scale-out takes longer to harness parallelism than aggressive scale-out, but it achieves more efficient directory scan performance particularly for small directories which form the majority of directories in a file system. Because GIGA+ aims to provide a general-purpose directory service, it aims to improve large directory performance without affecting small directory performance.
Trade-off #2	Stop inter-server indexing after it is fully distributed and parallel, and rely on backend store for out-of-core indexing for overflow partitions Splitting partitions on multiple servers allows GIGA+ to distribute work and harness available parallelism. These splits, however, may involve cross-server migration of entries from one partition to another. To avoid this overhead, GIGA+ stops splitting a growing directory after it has spread on all servers and is load balanced. GIGA+ allows overflow partitions to be indexed by the backend storage system's out-of-core indexing technique for efficient access.
Trade-off #3	Tolerate inconsistent, out-of-date indexing state for unsynchronized, non-blocking server-side growth GIGA+ servers split partitions in an uncoordinated, asynchronous manner, which may change partition-to-server mapping rapidly. GIGA+ allows clients to use their possibly out-of-date state; these clients may incorrectly address a wrong server that uses its split history to update the client's state. The number of addressing errors is dependent on the workload (ratio of clients' lookup rate and server-side growth) and is negligible even for insert-intensive workloads. Moreover, these errors happen only when the directory is mutating over servers.
Trade-off #4	Faster update of inconsistent client-side indexing state at the cost of encoding more state in update messages GIGA+ encodes a directory's index using a bitmap that is used by servers to send updates to clients. This encoding uses more space (than theoretical optimal) to allow clients to learn about the index quickly.

Table 7.1 — Summary of GIGA+ trade-offs analyzed in this dissertation.
Tradeoffs between consistency and concurrency of indexing state.

This dissertation explores how the choice of backend storage systems affects GIGA+ behavior, particularly how GIGA+ hash partitions are represented in the backend and how GIGA+ splits these partitions efficiently. The current GIGA+ prototype was studied on local file system backends, where splits result in migration of directory entries and the file data, and shared storage backends, where splits are more efficient and move only the directory entries.

7.1 Future work

Although GIGA+ has shown promising scalability and performance as a research prototype, there are several avenues of further work that may be relevant for real-world deployments and future metadata subsystems.

7.1.1 GIGA+ without client and server processes

The design of GIGA+ implicitly assumes the ability to run client and server processes on a compute cluster. In practice, particularly in HPC deployments, it is often infeasible to run such processes because it conflicts with fault zoning and performance isolation requirements of long-running applications. Furthermore, assuming the ability to modify the operating system kernel on the cluster nodes may be unrealistic; thus FUSE (and its kernel module) may not be available for running a GIGA+ client. A version of GIGA+ without clients and servers can be designed as a middleware library that applications can link and unlink as needed.

The key challenge in building this GIGA+ middleware is to logically distribute GIGA+ server's responsibility across all nodes running the middleware. The main purpose of a GIGA+ server

process is to serialize access to the partition managed by that server and to communicate addressing updates from partition splits to the clients. One approach would be designate certain application threads as "server proxies" that act as a logical controller to access partitions. These proxies should also be communicate with each other for splitting partitions and handling addition of new servers. This communication can be performed through a message passing interface (MPI) that is widely used in HPC deployments.

7.1.2 FUSE extensions for cluster systems

In the current GIGA+ prototype, once the application gets a symbolic link pointing to the physical location of the file, it will rely on FUSE and VFS to dereference the symbolic link to access the physical file. To avoid these extra indirections going in and out of VFS, it would be beneficial to extend the FUSE kernel module to detect a file handles coming from a cluster file system for future use.

Another extension in FUSE would be for file system notifications. In a layered file system, the higher layers will benefit from support for notification calls like *inotify()* and *dnotify()* to monitor changes to the underlying file system. For cluster file systems, such a notification system may enable the higher layers to make informed decisions about fault tolerance, data migration and configuration changes.

Given the popularity of FUSE in large-scale cluster systems, such extensions may benefit a large community of researchers and developers.

7.1.3 *Efficient backends for metadata-intensive workloads*

The analysis presented in this dissertation discovered the tight coupling of the distributed file system with the on-disk representation of directories. In most local file systems, file system metadata, including i-nodes, block management and directories, is stored in an area of the file system separate from the disk blocks that store file contents. Furthermore, to ensure metadata consistency, techniques such as journaling and update ordering require multiple writes for each metadata mutation. Such random disk accesses reduces disk bandwidth utilization and throughput.

Inspired by log-structured designs that convert random accesses to sequential accesses [Rosenblum 1992, O'Neil 1996], storage systems have begun exploring the use of optimized data-structures for metadata-intensive workloads. Both TokuFS's use of fractal trees and TableFS's use of LevelDB has demonstrated a ten-fold increase in the throughput of metadata-intensive workloads [Ren 2013, LevelDB 2012, Esmet 2012, Bender 2007].

Inspired by TableFS's finding, this dissertation showed the benefits of using LevelDB to enhance directory performance in a distributed system. However, this needs further exploration. Compaction policies, for instance, used by LevelDB have a significant impact on the foreground performance of applications. Compactions are a necessary evil: in order to speed up future reads and scans, they steal resources from foreground operations that happen simultaneously with these background operations. Exploring heuristics that can minimize the impact of foreground operations for metadata-specific will enhance performance [Sears 2012, Esmet 2012, Twigg 2011].

Bibliography

[Acunu 2011] Acunu. Acunu Storage Engine. http://http://www.acunu.com/technology.html, 2011.

[Adya 2002] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston MA, November 2002.

[AFS 2013] AFS. Open Andrew File System (AFS). http://www.openafs.org/, 2013.

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

[Agrawal 2008] Rakesh Agrawal, Anastasia Ailamaki, Philip A. Bernstein, Eric A. Brewer, Michael J. Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J. Franklin, Hector Garcia-Molina, Johannes Gehrke, Le Gruenwald, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, Henry F. Korth, Donald Kossmann, Samuel Madden, Roger Magoulas, Beng Chin Ooi, Tim O'Reilly, Raghu Ramakrishnan, Sunita Sarawagi, Michael Stonebraker, Alexander S. Szalay, and Gerhard Weikum. The Claremont report on database research. *ACM SIGMOD Record*, 37(3), September 2008.

[Alam 2011] Sadaf R Alam, Hussein N El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel I/O and the Metadata Wall. In *Proceedings of the Petascale Data Storage Workshop (PDSW '11)*, Seattle WA, November 2011.

[Anderson 2003] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface - SCSI vs. ATA. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco CA, March 2003.

[Anderson 1995] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network Filesystem. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Cooper Mountain Resort CO, December 1995.

[Artiaga 2010] Ernest Artiaga and Toni Cortes. Using Filesystem Virtualization to Avoid Metadata Bottlenecks. In *Proceedings of the Design, Automation and Test in Europe (DATE '10)*, Dresden, Germany, March 2010.

[AWS 2012] AWS. Amazon Web Services. http://aws.amazon.com/, 2012.

[Barr 2012] Jeff Barr. Amazon S3 - The First Trillion Objects. http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillion-objects.html, 2012.

[Basho 2013] Basho. Riak, an open-source distributed database. http://basho.com/riak/, January 2013.

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

[Bender 2007] Michael Bender, Martin Farach-Coulton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '07)*, June 2007.

[Bent 2009] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the ACM/IEEE Transactions on Computing Conference on High Performance Networking and Computing (SC '09)*, Portland OR, November 2009.

[Braam 1999] Peter Braam, Michael Callahan, and Phil Schwan. The Intermezzo File System. In *Proceedings of the 3rd Perl Conference, O'Reilly Open Source Convention*, Monterey CA, August 1999.

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

[Brandt 2006] Scott A. Brandt, Lan Xue, Ethan L. Miller, and Darrell D. E. Long. Efficient metadata management in large distributed file systems. In *Proceedings of 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '06)*, College Park MD, May 2006.

[Btrfs 2012] Btrfs. btrfs: A new Copy-on-Write File System. https://btrfs.wiki.kernel.org/, 2012.

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

[Calderon 2010] Alejandro Calderon. On Evaluating GPFS. arcos.inf.uc3m.es/~dcio/ALEX-gpfs.ppt, November 2010.

[Cao 2007] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the Art: Where we are with the ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, June 2007.

[ceph devel 2013] ceph devel. Crash and strange things on MDS. http://www.spinics.net/lists/ceph-devel/msg12713.html, February 2013.

[ceph users 2013] ceph users. Improving cephfs file listing speed. http://comments.gmane.org/gmane.comp.file-systems.ceph.user/792, 2013.

[Chang 2006] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle WA, November 2006.

[CIFS 2013] CIFS. Common Internet File Systems. http://www.samba.org/cifs/, 2013.

[Comer 1979] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), 1979.

[Congleton 2011] Ben Congleton. You can list a directory containing 8 million files! But not with ls.. http://www.olark.com/spw/2011/08/you-can-list-a-directory-with-8-million-files-but-not-with-ls/, August 2011.

[Cope 2005] James Cope, Mike Oberg, Henry M. Tufo, and Mike Woitaszek. Shared Parallel File Systems in Heterogeneous Linux Multi-Cluster Environments. In *6th LCI International Conference on Linux Clusters: The HPC Revolution*, April 2005.

[Corbett 2012] James C. Corbett, Jeff Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood CA, October 2012.

[Corbett 1996] Peter F. Corbett and Dror G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems (TOCS)*, 14(3), 1996.

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

[Daley 1965] Rick Daley and Paul Neumann. A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference, Part 1*, 1965.

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

[Dean 2009] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. Keynote at LADIS Workshop 2009 - http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf, 2009.

[DeCandia 2007] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson WA, October 2007.

[Dilger 2012] Andreas Dilger. Lustre Metadata Scaling. Tutorial at IEEE MSST 2012 conference. http://storageconference.org/2012/Presentations/T01.Dilger.pdf, May 2012.

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

[Drepper 2007] Ulrich Drepper. If not readdir() then what? LKML Mailing List at http://lkml.org/lkml/2007/4/7/107, April 2007.

[Ellis 1983] Carla Ellis. Extendible Hashing for Concurrent Operations and Distributed Data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '83)*, Atlanta GA, March 1983.

[Ellis 1985] Carla Ellis. Distributed data structures: A case study. *IEEE Transactions on Computing*, 34(12), December 1985.

[Esmet 2012] John Esmet, Michael Bender, Martin Farach-Coulton, and Bradley C. Kuszmaul. The TokuFS Streaming File System. In *Proceedings of the Workshop on Hot Topics in Storage Systems (HotStorage '12)*, Boston MA, June 2012.

[Fagin 1979] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing -- A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3), September 1979.

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

[Frings 2011] Wolfgang Frings and Michael Hennecke. A system level view of Petascale I/O on IBM Blue Gene/P. *Computer Science - R and D*, 26(3-4), 2011.

[FUSE 2010] FUSE. Filesystem in Userspace (FUSE). http://fuse.sf.net/, December 2010.

[Geschwinde 2002] Ewald Geschwinde and Hash-Jurgen Schonig. *PostgreSQL: Developer's Handbook*. SAMS Publishing, 2002.

[Ghemawat 2003] Sanjay Ghemawat, Howard Gobioff, and Shuan-Tek Lueng. Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing NY, October 2003.

[Gibson 1998] Garth A. Gibson, Dave F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '98)*, San Jose CA, October 1998.

[Goldstein 2012] Ted Goldstein. Extracting Medical Knowledge from High Throughput Genomics. XLDB 2012 workshop talk at http://www-conf.slac.stanford.edu/xldb2012/talks/xldb2012_tue_1010_Goldstein.pdf, September 2012.

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

[Gray 2005] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alexander S. Szalay, David DeWitt, and Gerd Heber. Scientific Data Management in the Coming Decade. *ACM SIGMOD Record*, 34(4), December 2005.

[Gray 1992] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[Hadoop 2011] Hadoop. Apache Hadoop Project. http://hadoop.apache.org/, 2011.

[Harizopoulos 2008] Stavros Harizopoulos, Daniel J. Abadi, Samuel R. Madden, and Michael Stonebraker. OLTP Through the Looking Glass, And What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, June 2008.

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

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

[HDFS 2010] HDFS. The Hadoop Distributed File System. http://hadoop.apache.org/hdfs/, 2010.

[Hedges 2010] Richard Hedges, Keith Fitzgerald, Mark Gary, and D. Marc Stearman. Comparison of leading parallel NAS file systems on commodity hardware. Technical Report LLNL-TR-461793, Lawrence Livermore National Laboratory, November 2010.

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

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

[Joyent 2012] Joyent. The Joyent Cloud. http://joyent.com/products/joyent-cloud, 2012.

[Kantor 2010] Jeff Kantor and Arun Jagatheesan. Storage Challenges for LSST: When Science Is Bigger Than Your Hardware. Talk at IEEE MSST 2010 Conference, 2010.

[Karger 1997] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the ACM Symposium on Theory of Computing (STOC '97)*, El Paso TX, May 1997.

[Katsov 2012] Ilya Katsov. Distributed Algorithms in NoSQL Databases. http://highlyscalable.wordpress.com/2012/09/18/distributed-algorithms-in-nosql-databases/, September 2012.

[Kerzner 2009] Mark Kerzner. HDFS - millions of files in one directory? Email thread at http://mail-archives.apache.org/mod_mbox/hadoop-core-user/200901.mbox/%3cc9b0d8bd0901231503k28cee90eg472e319e3ca4ce24@mail.gmail.com%3e, January 2009.

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

[Kumar 1990] Vijay Kumar. Concurrent operations on extendible hashing and its performance. *Communications of the ACM*, 33(6), June 1990.

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

[Lee 1996] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, Cambridge MA, October 1996.

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

[LevelDB 2012] LevelDB. leveldb: A fast and lightweight key/value database library by Google. http://code.google.com/p/leveldb/, 2012.

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

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

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

[Litwin 1996] Witold Litwin, Marie-Anne Neimat, and Donovon A. Schneider. LH* - A Scalable, Distributed Data Structure. *ACM Transactions on Database Systems*, 21(4), December 1996.

[LRZ 2013] LRZ. Optimizing concurrent access to files and directories. http://www.lrz.de/services/compute/supermuc/filesystems/GPFS-Usage/, March 2013.

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

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

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

[MacCormick 2004] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco CA, November 2004.

[Mathur 2007] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, June 2007.

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

[Metarates 2010] Metarates. UCAR Metarates Benchmark. www.cisl.ucar.edu/css/software/metarates/, 2010.

[Muthitacharoen 2002] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston MA, November 2002.

[MySQL 2013] MySQL. LINEAR HASH Partitioning. http://dev.mysql.com/doc/refman/5.1/en/partitioning-linear-hash.html, 2013.

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

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

[NFSv4 2013] NFSv4. Network File System version 4. http://tools.ietf.org/wg/nfsv4/, 2013.

[Nunez 2012] James Nunez. Multi-Dimensional Hashed Indexed Metadata/Middleware (MDHIM) Project. Talk at http://www.pdl.cmu.edu/SDI/2012/slides/Nunez-May2012.pdf, May 2012.

[Olson 1999] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '99)*, Monterey CA, June 1999.

[O'Neil 1996] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structure Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), July 1996.

[OrangeFS 2010] OrangeFS. Distributed Directories in OrangeFS v2.8.3-EXP. http://orangefs.net/trac/orangefs/wiki/Distributeddirectories, November 2010.

[Phillips 2001] Daniel Phillips. A Directory Index for Ext2. In *5th Annual Linux Showcase and Conference*, Oakland CA, November 2001.

[Prabhakaran 2005] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '05)*, Anaheim CA, June 2005.

[PROBE 2012] PROBE. Probe: Parallel reconfigurable observational environment. http://marmot.nmc-probe.org/, 2012.

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

[Raicu 2011] Ioan Raicu, Ian T. Foster, and Pete Beckman. Making a Case for Distributed File Systems at Exascale. In *Proceedings of the ACM Workshop on Large-scale System and Application Performance (LSAP '11)*, San Jose CA, June 2011.

[Rajgarhia 2010] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user-space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, Sierre, Switzerland, March 2010.

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

[Ren 2013] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System, June 2013.

[Rhea 2003] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the Oceanstore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco CA, March 2003.

[Rhea 2004] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in DHT. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '04)*, Boston MA, June 2004.

[Rhea 2005] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '05)*, Philadelphia PA, August 2005.

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

[Robson 2010] Alex Robson. Consistent Hashing. http://sharplearningcurve.com/blog/2010/09/27/consistent-hashing/, September 2010.

[Rosenblum 1992] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1), August 1992.

[Ross 2006] Rob Ross, Evan Felix, Bill Loewe, Lee Ward, James Nunez, John Bent, Ellen Salmon, and Gary Grider. High End Computing Revitalization Task Force (HECRTF), Inter Agency Working Group (HECIWG) File Systems and I/O Research Guidance Workshop 2006. http://institutes.lanl.gov/hec-fsio/docs/HECIWG-FSIO-FY06-Workshop-Document-FINAL6.pdf, 2006.

[Rowstron 2001] Anthony Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of*

the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Banff, Canada, October 2001.

[Salomon 2004] David Salomon. *Data Compression: The Complete Reference, 3rd edition*. Morgan Kaufmann Publishers, 2004.

[Sandberg 1985] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Summer Technical Conference*, Portland OR, June 1985.

[Schindler 2011] Jiri Schindler, Sandip Shete, and Keith A. Smith. Improving Throughput for Small Disk Requests with Proximal I/O. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, San Jose CA, February 2011.

[Schlosser 2005] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On Multidimensional Data and Modern Disks. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco CA, December 2005.

[Schmuck 2010] Frank Schmuck. Private Communication with Frank Schmuck, IBM, February 2010.

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

[Sears 2012] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, June 2012.

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

[Seltzer 2009] Margo Seltzer and Nicholas Murphy. A general-purpose file system for secondary storage. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '09)*, Monte Verita, Switzerland, May 2009.

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

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

[SMB 2013] SMB. Microsoft SMB Protocol and CIFS Protocol Overview. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365233(v=vs.85).aspx, 2013.

[Soltis 1996] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Global File System. In *Proceedings of the IEEE Transactions on Computing Symposium on Mass Storage Systems and Technologies (MSST '96)*, Chicago IL, August 1996.

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

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

[Stonebraker 2007a] Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. One Size Fits All? Part 2: Benchmarking Studies. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR '07)*, Asilomar CA, January 2007.

[Stonebraker 2009] Michael Stonebraker, Jacek Becla, David Dewitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stan Zdonik. Requirements for Science Data Bases and

SciDB. In *Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR '09)*, Asilomar CA, January 2009.

[Stonebraker 2007b] Michael Stonebraker and Ugur Çetintemel. One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, April 2007.

[Stonebraker 2007c] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, Trondheim, Norway, September 2007.

[Studham 2005] R. Scott Studham. Lustre: A Future Standard for Parallel File Systems. Invited presentation at International Supercomputer Conferencei 2005. Heidelberg, Germany, 2005.

[Sweeney 1996] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '96)*, Boston MA, June 1996.

[Thekkath 1997] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint-Malo, France, October 1997.

[Tokutek 2010] Tokutek. TokuDB. http://www.tokutek.com, December 2010.

[Top500 2012] Top500. Top 500 Supercomputer Sites. http://www.top500.org, December 2012.

[Tweed 2008] David 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.

[Twigg 2011] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified B-trees and Versioned Dictionaries. In *Proceedings of the Workshop on Hot Topics in Storage Systems (HotStorage '11)*, Portland OR, June 2011.

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

[Voldemort 2010] Voldemort. Project Voldemort. A distributed database. http://www.project-voldemort.com/voldemort/, December 2010.

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

[Weil 2006b] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzhan. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the ACM/IEEE Transactions on Computing Conference on High Performance Networking and Computing (SC '06)*, Tampa FL, November 2006.

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

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

[Welch 2008] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose CA, February 2008.

[Welch 1992] Brent B. Welch. Naming, State Management, and User-Level Extensions in the Sprite Distributed File System. 1992. Available as Technical Report UCB/CSD 90/567.

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

[White 2002] Brian White, Jay Lepreau, Shashi Guruprasad, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston MA, November 2002.

[White 2009] Tom White. The Small Files Problem. http://www.cloudera.com/blog/2009/02/02/the-small-files-problem/, February 2009.

[Whitehouse 2007] Steven Whitehouse. The GFS2 Filesystem. In *Proceedings of the Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, June 2007.

[Wu 2004] Changxun Wu and Randal Burns. Tunable randomization for load management in shareddisk clusters. *ACM Transactions of Storage*, 1(1), December 2004.

[Yang 2011] Shuangyang Yang, Walter B. Ligon III, and Elaine C. Quarles. Scalable Distributed Directory Implementation on Orange File System. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O (SNAPI '11)*, Denver CO, May 2011.

[Yaschenko 2011] Eugene Yaschenko. Sequence Read Archive: Validation, Archival, and Distribution of Raw Sequencing Data. XLDB 2011 workshop talk at http://www-conf.slac.stanford.edu/xldb2011/talks/xldb2011_wed_0905_Yaschenko.pdf, September 2011.

[Zadok 2006] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. On Incremental File System Development. *ACM Transaction on Storage (TOS)*, 2 (2), May 2006.

[ZFS 2007] ZFS. Concurrent, constant time directory operations in ZFS. http://www.solarisinternals.com/wiki/index.php/ZFS_Performance, 2007.

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

[Zhen 2011] Liang Zhen. Parallel Directory Operations of Lustre. http://www.opensfs.org/wp-content/uploads/2011/11/single_mds_performance.pdf, November 2011.

[ZooKeeper 2011] ZooKeeper. Apache ZooKeeper Project. http://hadoop.apache.org/zookeeper/, 2011.