λFS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions

Benjamin Carver
George Mason University
Fairfax, VA, USA
bcarver2@gmu.edu

Runzhou Han
Iowa State University
Ames, IA, USA
hanrz@iastate.edu

Jingyuan Zhang
George Mason University
Fairfax, VA, USA
jzhang33@gmu.edu

Mai Zheng
Iowa State University
Ames, IA, USA
mai@iastate.edu

Yue Cheng*
University of Virginia
Charlottesville, VA, USA
mrz7dp@virginia.edu

ABSTRACT

The metadata service (MDS) sits on the critical path for distributed file system (DFS) operations, and therefore it is key to the overall performance of a large-scale DFS. Common "serverful" MDS architectures, such as a single server or cluster of servers, have a significant shortcoming: either they are not scalable, or they make it difficult to achieve an optimal balance of performance, resource utilization, and cost. A modern MDS requires a novel architecture that addresses this shortcoming.

To this end, we design and implement λFS, an elastic, high-performance metadata service for large-scale DFSes. λFS scales a DFS metadata cache elastically on a FaaS (Function-as-a-Service) platform and synthesizes a series of techniques to overcome the obstacles that are encountered when building large, stateful, and performance-sensitive applications on FaaS platforms. λFS takes full advantage of the unique benefits offered by FaaS—elastic scaling and massive parallelism—to realize a highly-optimized metadata service capable of sustaining up to 4.13× higher throughput, 90.40% lower latency, 85.99% lower cost, 3.33× better performance-per-cost, and better resource utilization and efficiency than a state-of-the-art DFS for an industrial workload.

ACM Reference Format:
Benjamin Carver, Runzhou Han, Jingyuan Zhang, Mai Zheng, and Yue Cheng. 2023. λFS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3623278.3624765

1 INTRODUCTION

Many different fields in computing have enjoyed successes in part due to the availability of large amounts of data [2, 20, 23, 28, 33, 34, 36, 41, 65]. Data-intensive applications [18, 24] in these fields are characterized by varied, heterogeneous I/O patterns in which I/O bottlenecks are not uncommon [35, 46, 64]. Large-scale, distributed file systems (DFSes), such as Google File System (GFS) [38] and Hadoop Distributed File System (HDFS) [57], are commonly used by these data-intensive applications. DFSes often use an architecture that *decouples* metadata management from file I/O [14, 38, 57]. DFS metadata tracks global file system namespace information, including hierarchical directories and file names. These DFSes use a centralized metadata management component called a metadata service (MDS), which executes file system namespace operations, such as file `open`, `close`, and `mv`. In a DFS, client applications acquire a file's permission and location information from the MDS before accessing the file's contents. Therefore, the performance of the MDS is key to the overall efficiency of a DFS.

Scaling the performance of an MDS is challenging. Using a dedicated server (e.g., GFS [38], HDFS [57]) to host the MDS is not scalable and may suffer from poor performance during highly dynamic workloads.

Researchers have proposed various ways to overcome the scalability challenges of DFS MDSes. IndexFS [54] is a middleware that offloads metadata storage and processing to a scaled-out, table-based, key-value store cluster that is co-located with the data storage cluster. InfiniFS [49] uses the same scaled-out cluster architecture as IndexFS but with deep optimizations along the metadata processing path. HopsFS [51], built on HDFS, further decouples metadata request handling and metadata storage: it offloads the metadata storage to a distributed, sharded, in-memory database (MySQL NDB Cluster [16]) and utilizes a cluster of *stateless* NameNodes (metadata servers in HDFS terminology) to scale DB query handling.

While these systems offer scalable MDS solutions with different tradeoffs, they have a common issue: they lack elasticity support at the MDS level. IndexFS and InfiniFS employ a fixed cluster of metadata servers and use client-side metadata caching extensively for performance improvement. HopsFS provides no metadata caching on the stateless NameNode side and uses the distributed NameNodes only for handling and scaling client requests. Therefore, HopsFS' performance is capped by the capacity of the backend NDB cluster. All three of these systems require explicit server management and a large amount of server resources to be reserved to host the MDS cluster. As reported in [51], HopsFS requires as many as 60 NameNodes and 12 NDB servers in order to significantly outperform vanilla HDFS, the latter of which typically uses a small cluster

*Corresponding author

of NameNodes for high availability but not performance. Worse, under low load conditions, the scaled-out MDS cluster suffers from low resource utilization.

Serverless computing or Function-as-a-Service (FaaS) has emerged as a new cloud computing model [4, 42]. FaaS enables developers to break traditionally monolithic, server-based applications into finer-grained serverless (or cloud) functions, thereby providing a new way of building and scaling applications and services. Developers are tasked with providing the function logic while the FaaS provider is responsible for the notoriously tedious tasks of provisioning, scaling, and managing backend servers that host the serverless functions [39].

We find that serverless functions provide an appealing environment in which to host and scale the metadata management component of a large-scale DFS. Using serverless functions provides several key advantages. First, serverless functions have CPU and memory resources that are elastically scaled out and in with the functions. This enables the construction of an elastic MDS that can achieve optimal performance by dynamically adapting the amount of resources as the workload shifts. Second, the elasticity offered by FaaS can greatly improve cost-efficiency and resource utilization as resources are allocated/deallocated in an on-demand manner and used more efficiently. Third, the auto-scaling property also alleviates the need for tedious server management.

The aforementioned challenges pertaining to MDS efficiency and the emergence of serverless computing together raise a research question: *Can we use serverless functions in a novel way to build a high-performance, cost-efficient, elastic, and resource-efficient MDS?*

To answer this question, we present λFS, the first serverless-function-based, elastic MDS for large-scale DFSes. In a nutshell, λFS features a novel MDS architecture that combines an elastically-scalable, FaaS-based metadata cache with a persistent, strongly-consistent metadata store. To minimize network overhead, λFS uses the collective memory of a dynamic fleet of serverless functions for metadata caching. However, simply implementing a metadata caching layer is insufficient. λFS further enables elastic and massively-parallel metadata caching by taking advantage of the auto-scaling offered by FaaS. Not only does this elasticity improve metadata query performance, but it also enables high resource efficiency and low cost. Moreover, λFS effectively decouples the management of metadata caching (and thus, metadata request processing) and metadata storage so that compute and storage can scale independently. This fully-disaggregated architecture is driven by the observations that real-world MDS workloads are bursty [35, 55, 60] and that it is often difficult to manually determine the right MDS deployment scale offline [27, 58].

Building an elastic serverless MDS for large-scale DFSes requires addressing two sets of unique challenges:

- First, FaaS platforms have a series of constraints and limitations that make it challenging to support data-intensive, stateful applications efficiently: (1) individual serverless functions have limited CPU, memory, and network resources, and thus offer limited data processing, storage, and transfer capacity. (2) Serverless functions occasionally suffer from long cold start times and execution timeouts. (3) The typical method to communicate with serverless functions is via HTTP requests, but this can be slow. As such, naively porting the stateful MDS of a large-scale DFS to a serverless platform leads to poor performance.
- Second, while FaaS platforms offer auto-scaling and elasticity, a careful, holistic MDS redesign is required to fully utilize these benefits: (1) Performance-sensitive systems such as MDSes require careful treatment to balance the performance and auto-scaling tradeoff. (2) Partitioning the file system namespace across a dynamic fleet of serverless functions introduces interesting tradeoffs in a FaaS environment. (3) The lack of addressability of serverless functions means that a metadata entry may be stored on multiple functions, therefore introducing metadata consistency issues.

λFS addresses these challenges by synthesizing several techniques into an end-to-end, serverless MDS system. First, we find that λFS can achieve strong performance using a large number of serverless NameNodes each having relatively small CPU and memory resources compared to their serverful counterparts. λFS also leverages a hybrid HTTP-TCP RPC mechanism to enable agile, lightweight, and performance-preserving auto-scaling.

Second, λFS' FaaS-powered metadata cache consists of *n* unique serverless function deployments. λFS uses path-based hashing of parent directories to partition the file system namespace among the *n* deployments in order to support efficient metadata `read` operations. Each deployment can automatically scale out to an arbitrary number of concurrently-running function instances that elastically support bursts of metadata requests on hot directories. λFS trades function-deployment-based auto-scaling (i.e., there is a *fixed* number *n* of deployments) for easy-to-manage, deterministic metadata partitioning. Third, λFS implements a serverless coherence protocol to provide strong consistency in the presence of an arbitrary number of running "function instances" and clients.

Finally, λFS re-implements many DFS maintenance features, such as block reports and DataNode discovery, in a serverless-compatible way by publishing information to the persistent metadata store on a regular interval.

There are a number of benefits and advantages of using FaaS as the underlying platform for the MDS of a large-scale DFS. Notably, these benefits would be difficult or impossible to realize using a traditional, serverful MDS architecture. First, by using a large number of relatively lightweight serverless functions, overall resource utilization can be improved, which ultimately leads to better performance and cost-efficiency. This cost-efficiency is further enhanced by the pay-per-use pricing model of FaaS, which drastically lowers tenant-side costs without negatively impacting performance. This is quantified using a *performance-per-cost* metric in §5.2.5.

In addition to cost-related benefits, the MDS can take advantage of FaaS-based auto-scaling to automatically adapt to changes in request volume without requiring management by users or admins. When request volume increases, the MDS automatically scale-outs to serve the additional requests. When request volume decreases, the MDS will scale-in, avoiding the problem of low resource utilization and poor cost-efficiency. This completely circumvents the typical under/over-provision problem faced by serverful systems.

In summary, this paper makes the following contributions:

- We identify scalability and performance issues of HopsFS, a state-of-the-art DFS with a scaled-out MDS design.

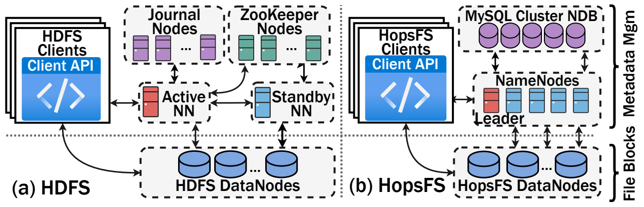


Figure 1: Architecture of HDFS and HopsFS.

- We explore the design space of serverless metadata management for large-scale DFSes. We identify the key opportunities and challenges of using FaaS for this purpose, and we share insights into how to address these challenges.
- We present the design and implementation of λFS, a novel metadata service that uses massively-parallel serverless functions to cache and elastically scale the metadata workload. To the best of our knowledge, λFS is the first serverless-function-based MDS for large distributed file systems.
- We demonstrate λFS' generality by porting λFS to two practical DFSes that have different, scaled-out MDS architectures: HopsFS and BeeGFS [6] enhanced with IndexFS.
- We extensively evaluate λFS using real-world workloads and microbenchmarks. Our results show that λFS achieves up to 4.13× higher throughput, 90.40% lower latency, and 85.99% lower cost than HopsFS and 3.33× greater performance-per-cost than HopsFS augmented with a metadata cache while providing better resource utilization.

2 BACKGROUND AND MOTIVATION

Different DFS architectures have different tradeoffs, but there is one commonality: all existing solutions use an architecture that separates metadata management and file data storage management. In this section, we use HDFS and HopsFS, two representative, production-ready DFSes, as examples to illustrate the two generations of MDS architectures used by today's DFSes. Specifically, HopsFS uses a cluster of scaled-out, stateless metadata servers in front of a scaled-out, strongly-consistent metadata store to support a scalable MDS for the widely used HDFS. Therefore, HopsFS provides an ideal platform to experiment with and demonstrate the efficacy of λFS. This section describes the common limitations of state-of-the-art MDS solutions to further motivate λFS.

Hadoop Distributed File System. The Hadoop Distributed File System (HDFS) is an open-source implementation of GFS [38] and is widely used in practice [57]. HDFS stores metadata in the memory of a Java process referred to as the Active NameNode. This metadata is replicated to a Standby NameNode, which is used for checkpointing and failure recovery. File system operations are executed atomically by the Active NameNode, thereby providing strong consistency for file system metadata. POSIX semantics are relaxed in order to allow for streaming access to the system data. See Figure 1(a).

DataNodes are responsible for storing file data. Each DataNode connects to both the Active and Standby NameNodes. A DataNode periodically generates reports that are sent to the NameNode. These reports are used to ensure the NameNode's block map is consistent with the actual data stored in DataNodes. The JournalNodes are used to synchronize state between the Active and Standby NameNodes. ZooKeeper [40] provides *automatic failover* and *leader election* for the NameNodes.

HopsFS. HopsFS [51] is a distributed file system developed as an extension of HDFS. HopsFS provides a scaled-out metadata management layer by decoupling the storage and manipulation of metadata. Specifically, HopsFS supports multiple stateless NameNodes. The NameNodes persists the metadata to a pluggable storage backend and collectively serve metadata requests made by clients. HopsFS uses MySQL Cluster NDB [15] for this persistent backend data store. The architecture of HopsFS is shown in Figure 1(b).

Each NameNode uses a Data Access Layer (DAL) that provides a generic interface to an arbitrary persistent storage backend. This interface is used to manipulate the metadata stored within NDB. All file system operations require the resolution of each path component in order to check for permissions and path validity. HopsFS introduces techniques to mitigate the performance impact of path resolutions, such as an "INode Hint Cache", which allows clients to cache metadata prefixes locally to reduce the number of round trips required for path resolution from N round trips (for a path of length N) to just one single batch query. The cluster of stateless NameNodes cooperates to handle DataNode failures. The NameNodes elect a leader NameNode to perform administrative tasks.

Limitations of Today's Scaled-Out MDSes. For the remainder of the paper, we do not focus on issues that HopsFS has already addressed—λFS uses the same decoupled compute-and-storage MDS architecture and uses the same DAL to interface with the persistent metadata store used by HopsFS. Instead, we focus on the scalability and elasticity problems with HopsFS' statically-fixed, stateless NameNode cluster design, which we describe next.

There are several aspects of HopsFS' design that hinder HopsFS' MDS efficiency. First, the use of *stateless* NameNodes necessitates the retrieval of metadata from the persistent metadata store for every single metadata operation. This means that HopsFS' performance is capped by the capacity of the backend NDB cluster. The compute (NameNode) and storage (NDB) resources, though physically decoupled, are essentially *logically-bundled* resources that need to be configured together. Otherwise, system performance can rapidly degrade if either of the two layers becomes a bottleneck.

Second, HopsFS and other scaled-out MDS solutions [49, 54, 63] lack elasticity and require an admin to empirically configure a statically-fixed deployment of compute and storage resources for the serverful MDS cluster. This leads to a choice between resource under-utilization and degraded performance: if the admin provisions compute resources for the peak load of the metadata workload, the system wastes both compute and storage resources; if the admin provisions resources for the average demand, then the performance degrades when the load increases beyond the provisioned capacity.

Terminology. Before describing λFS' design, it is necessary to define some terminology. First, λFS' NameNodes are organized into multiple serverless *function deployments*. Function deployments consist of user-written code to be executed when the serverless function runs, configuration info, and metadata, all of which is registered under a unique name with the FaaS platform. The code for a NameNode is written (in Java) as the body of a serverless function. λFS registers a configurable number of uniquely named

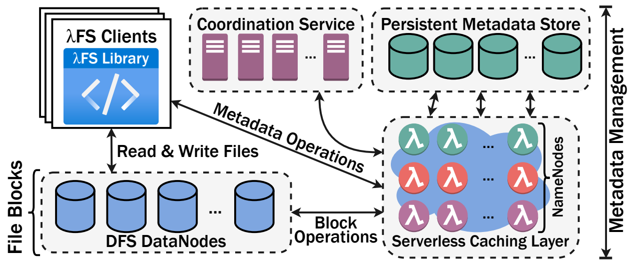


Figure 2: The λFS architecture.

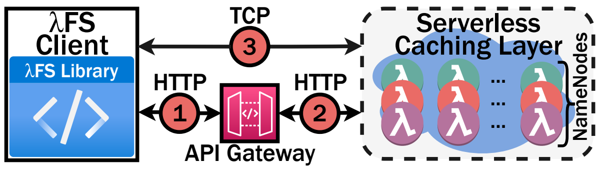


Figure 3: λFS supports two different types of metadata RPC requests: HTTP-based and TCP-based RPCs.

serverless NameNode functions with the FaaS platform. (The bodies of these functions are identical; the names are different.)

When a user invokes a serverless function defined by a particular deployment, the FaaS platform automatically provisions an *instance* of that function based on the configuration info specified when the deployment was registered. Thus, a *function instance* refers to an instantiated, running serverless function. A *NameNode* then refers to the Java application executing within the function instance. When we say that a function instance "belongs" to a deployment, we mean that the instance is an instantiation of the serverless function defined by that deployment. Only one NameNode can execute within a function instance, so the two terms are used interchangeably.

3 λFS DESIGN

λFS enables elastic metadata service for large-scale DFSes such as HDFS. λFS uses a hybrid, FaaS-optimized RPC mechanism that uses both TCP-based RPC and HTTP-based RPC together to enable high throughput and reduced latency (§3.2). λFS uses a serverless-function-based memory caching layer for DFS metadata caching (§3.3) and features an agile auto-scaling policy to enable elastic and parallel metadata processing at the caching layer (§3.4). While caching reduces the number of network hops per request, λFS' auto-scaling significantly improves the cache's throughput. λFS also introduces a simple coherence protocol to ensure strong consistency of metadata operations within the serverless cache (§3.5). Note that λFS is also usable with other DFSes: as we will show in §4, λFS can enhance BeeGFS [6] as a drop-in replacement for IndexFS.

3.1 λFS Overview

Figure 2 shows the architecture of λFS. Clients issue RPC metadata requests to NameNodes, just as clients do in HopsFS. The difference is that each λFS NameNode is a Java serverless function executing within a container managed by the serverless platform. RPC metadata requests are initially performed as HTTP invocations directed towards the platform's API gateway. The serverless platform routes HTTP requests it receives to already-running NameNodes, if available, or it starts a new NameNode if none are running.

Once a NameNode is up and running, it can establish direct TCP connections back to clients (after first interfacing with clients through HTTP requests). TCP-based RPC requests serve as a lower-latency alternative to HTTP-based RPC requests, as only one network hop (client to NameNode) is required for a TCP RPC. Hybrid TCP and HTTP RPC mechanisms are discussed in §3.2.

Another key difference between λFS and HopsFS is that the serverless NameNodes in λFS: are (1) *not* stateless, and (2) are *elastic*. This allows the dynamic cluster of serverless NameNodes to collectively form an *elastic metadata caching layer*. When a NameNode receives a metadata request, it checks whether the requested metadata was retained on the NameNode from a previous function invocation. The retained metadata on the NameNodes thus form a cache. The caching system is discussed further in §3.3.

To support *elastic caching*, λFS' NameNodes are organized into n individual *function deployments*. We partition the namespace among the function deployments by consistently hashing on the parent directory path of each file/directory. For example, we may hash the file "`/dir/note.pdf`" to the deployment named "`NameNode5`". In this case, the client would issue an HTTP RPC for the deployment "`NameNode5`", or issue a TCP RPC to an already-running function instance of the `NameNode5` deployment. If the NameNode that serves this request already has the target metadata cached locally, then a network hop to the persistent metadata store is avoided, resulting in lower latency for the request. Individual function deployments automatically scale out in response to the sudden increase of metadata requests. Our agile auto-scaling policy is described in §3.4.

λFS uses a pluggable "Coordinator" service for tracking NameNode liveness and coordinating NameNodes during write operations. λFS currently supports both ZooKeeper and MySQL Cluster NDB. The Coordinator is used in the coherence protocol described in §3.5.

λFS' design capitalizes on the unique benefits of serverless computing while accounting for the challenges that the platform presents. First, λFS takes advantage of the intra-deployment auto-scaling offered by FaaS to rapidly and transparently scale-out in response to bursts of work. Not only does this enable λFS to responsively and elastically adapt to changing workload characteristics in real-time, but it also improves λFS' resource efficiency and resource utilization. When system throughput returns to normal levels, λFS will transparently scale-in to avoid incurring additional costs. Using traditional, serverful VMs in place of serverless functions would result in significantly reduced elasticity and either wasted resources or poor performance (depending on how resources are provisioned).

3.2 Hybrid Serverless RPC Mechanism

λFS utilizes two different RPC pathways in order to provide high system throughput and high elasticity. Specifically, HTTP RPCs directed to the serverless platform's API gateway are used to scale-out the number of serverless function instances as the load increases. At the same time, NameNodes establish direct TCP connections back to clients. Clients can then use this direct connection as a low-latency alternative to HTTP requests. During our experiments, we found that the average end-to-end latency for read operations was 1-2ms for TCP RPCs and 8-20ms for HTTP RPCs. Clients issue TCP

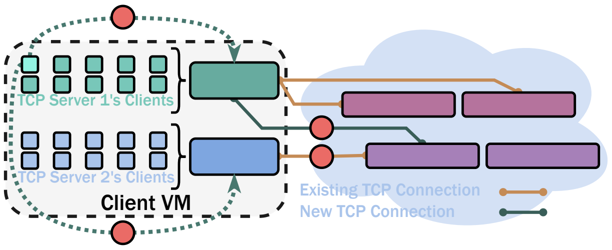


Figure 4: λFS' "connection sharing" mechanism.

RPCs whenever possible due to the significantly lower latency. (TCP RPCs also experience much smaller end-to-end latency variance compared to HTTP RPCs.) The lower latencies lead to substantially higher throughput and overall much lower costs due to the reduced overhead for each file system operation.

Figure 3 depicts the two different types of metadata RPCs that λFS supports. The first, labeled as step (1), is a standard HTTP invocation directed to the API gateway of the FaaS framework (e.g., OpenWhisk). At step (2), the FaaS API gateway will route this request to a serverless function invoker, which will submit the request to an existing NameNode, or the invoker will provision a new instance if none exist or all are busy serving other requests. A NameNode that serves an HTTP request will subsequently establish TCP connections back to the clients that issued the HTTP request if no such connection already exists as shown in step (3).

By default, all clients on the same VM will use the same TCP server (on that VM) to communicate with serverless NameNodes. Users can optionally configure λFS to assign at-most n clients to each TCP server. New TCP servers are automatically created for new clients as needed. Clients transparently include their IP address and the ports for all TCP servers on their VM within HTTP request payloads, which enables the NameNodes to proactively connect to the servers.

Clients also temporarily *share connections* with one another. Consider the process illustrated in Figure 4. Client a wishes to submit a metadata request to deployment 2 ($D2$). In step 1, a finds that there is no existing connection between its TCP server and an instance of $D2$. Thus, in step 2 client a contacts the other TCP servers on its VM and finds that TCP Server 2 has an existing connection to an instance of $D2$. In step 3, a uses TCP Server 2 to issue its metadata request. After fulfilling the request, NameNode 2a establishes a TCP connection back to client a's assigned TCP server.

When HTTP requests time out, clients could resubmit the requests to the FaaS platform immediately, causing a request storm that could overwhelm the FaaS platform and lead to the over-provisioning of NameNodes. We designed the client library so that clients sleep before resubmitting requests, following an exponential backoff delay pattern with randomized jitter added.

Similarly, if a TCP connection between a client and a NameNode is dropped, then any incomplete requests are transparently re-submitted by the client. The client will first determine if there are any other active TCP connections to instances from the target deployment. If so, then these connections will be used to re-submit the requests. If not, then the client queries the other TCP servers on its VM, if any, for active connections, and uses any connection it finds to resubmit its request. If no such TCP connections exist, then the client will simply fall-back to HTTP to re-submit the request. Additionally, NameNodes temporarily cache results returned to clients in the event that network delays or other failures prevent the client from receiving the result. When the NameNode receives a re-submitted request, it will attempt to return cached results before re-performing the requested operation.

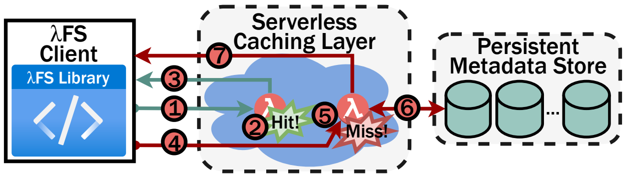


Figure 5: A walkthrough of λFS' caching protocol.

3.3 Serverless Metadata Cache

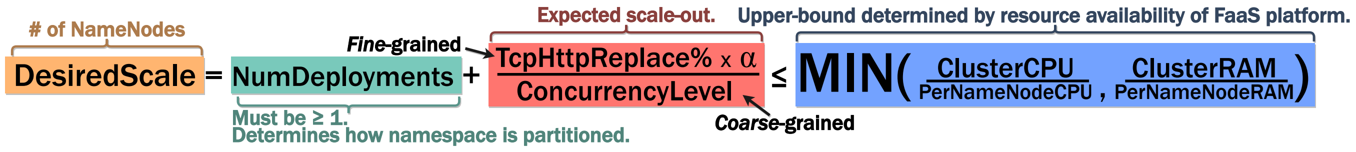
λFS NameNodes provide a serverless caching layer for performance. We partition the file namespace across all the NameNode deployments by consistently hashing on the parent INode ID. Each deployments' NameNodes are then responsible for caching a partition of the namespace, and clients route metadata RPCs based on this partitioning scheme.

NameNodes cache more than just the metadata associated with the terminal INode in a particular path. Specifically, they cache the metadata for *all* INodes contained within a particular path. Cached metadata is stored in a *trie* data structure maintained in-memory on the NameNode. This caching scheme allows for metadata read operations to avoid going to the persistent metadata store, as NameNodes can serve the read entirely from their local cache, called a cache *hit*. A cache *miss* occurs when missing metadata must be retrieved from the metadata store. Once the metadata is retrieved, the NameNode will cache the metadata in its local cache for future read operations.

Figure 5 provides an illustration of λFS' serverless metadata caching. In step (1), the client issues a metadata request for the file "`/nts/notes.txt`". This results in a cache *hit* on the NameNode in step (2). As a result, this NameNode can return the metadata directly to the client in step (3) without having to first retrieve it from the metadata store. Next, the client issues another metadata RPC for the file "`/bks/book.pdf`". This request is routed to a different NameNode in step (4) and results in a cache *miss* in step (5). In step (6), the consulted NameNode retrieves the metadata from the metadata store, which caches the metadata on the NameNode. Finally, the metadata is returned to the client in step (7).

3.4 Agile Serverless NameNode Auto-Scaling

A metadata cache reduces per-operation latency and improves system throughput; however, a cache alone is insufficient for supporting large-scale, bursty workloads while maximizing cost-effectiveness and resource efficiency. An MDS equipped with a cache must still be statically provisioned by the user ahead of time. This creates a dilemma for the user, which is to over- or under-provision the resources, trading off performance and cost-efficiency. In order to avoid this trade-off, we implement an agile and lightweight auto-scaling policy for managing coordinated scaling within a serverless



# of NameNodes
DesiredScale
=
NumDeployments
+
Must be ≥ 1.
Determines how namespace is partitioned.
Fine-grained
Expected scale-out.
TcpHttpReplace% x α
ConcurrencyLevel
Coarse-grained
≤
Upper-bound determined by resource availability of FaaS platform.
MIN(ClusterCPU / PerNameNodeCPU , ClusterRAM / PerNameNodeRAM)

Figure 6: Mathematical model of the agile and lightweight FaaS auto-scaling in λFS.

framework. We begin by motivating the design of our auto-scaling policy before discussing the general model in detail.

When clients issue metadata requests, they issue either a TCP RPC or an HTTP RPC (§3.2). Clients will choose to issue a TCP RPC whenever a TCP connection exists to a NameNode in the target deployment. This is because TCP RPCs incur significantly lower overhead compared to HTTP RPCs; clients only choose HTTP RPCs as a last resort, when no TCP connections exist. However, TCP RPCs are *not* FaaS-aware. Only HTTP RPCs are routed through the FaaS platform and thus enable the platform to detect when additional containers are needed. Therefore, exclusive use of TCP RPCs will ultimately lead to poor scalability and elasticity by preventing auto-scaling, which leads to overload and performance degradation. This scenario highlights an interesting trade-off between performance (low latency and high throughput) and elasticity.

To address the scenario above, we implement an agile and lightweight auto-scaling policy based on a randomized HTTP-TCP replacement mechanism. Each TCP RPC is probabilistically replaced by an HTTP RPC, with a configurable probability. As the request load increases, the absolute number of HTTP RPCs should increase, enabling the FaaS platform to provision additional serverless containers as needed. In essence, the randomized replacement mechanism allows for a majority of RPCs to be TCP-based while still enabling auto-scaling to occur, leading to better elasticity and scalability while achieving low latency and high throughput.

The auto-scaling policy can be modeled using the equation in Figure 6, where α is a parameter encoding the load level (requests per second and load concurrency), and **ConcurrencyLevel** is the *function-level concurrency* of each individual NameNode. To support function-level concurrency, we extended OpenWhisk [3] to enable control over how many unique HTTP RPCs a single function instance can serve *simultaneously*. This parameter provides *coarse-grained* control over the degree of auto-scaling, as small changes in this value will have a large impact on the number of provisioned NameNodes. The closer that the **ConcurrencyLevel** is to its minimum value of 1, the greater the degree of auto-scaling. Meanwhile, the HTTP-TCP replacement probability provides *fine-grained* control over auto-scaling. We find that empirically setting the probability of random HTTP-TCP replacement to a value $\leq 1\%$ tends to provide the best performance for the request loads and resource limits we used.

λFS' auto-scaling policy reuses the FaaS platform's existing auto-scaling facility while remaining agile and performance-preserving. We choose not to use sophisticated feedback-based policies, such as Kubernetes' Horizontal Pod Autoscaling algorithm [10], as these policies typically require a long feedback-loop delay, which cannot be tolerated if sudden load bursts must be dealt with quickly. *We envision that this model is readily applicable to and useful for future performance-sensitive FaaS-based systems. It provides an effective methodology that enables FaaS platforms to embrace high-throughput, low-latency stateful applications.*

Algorithm 1 λFS Coherence Protocol

```
(1) For each $d \in \mathcal{D}$, $N_L$ subscribes to and listens for liveness and
    ACK notifications before issuing an INV, whose payload includes
    the metadata to be invalidated, to that deployment. All of this
    is performed using the Coordinator. ACKs are *not* required from
    NameNodes that terminate mid-protocol.
(2) Upon receiving an INV, NameNodes in each $d \in \mathcal{D}$ first invali-
    date their caches before responding with an ACK.
(3) Once $N_L$ has received all required ACKs, the write operation can
    safely continue. Metadata changes/updates are persisted to the
    persistent datastore.
```

3.5 Coherence Protocol

Supporting stateful and parallel caching atop serverless NameNodes requires special treatment for concurrent metadata operations, as multiple function instances for the same NameNode deployment may cache replicas of the same metadata. Like HDFS, λFS' metadata operations fall into the following two categories: single INode operations that operate on a single file or directory (e.g., `read/create` file), and subtree operations that operate on one or more directories spanning many INodes (e.g., recursive `mv` and `delete`).

Inspired by cache/memory coherence algorithms [43, 47], we designed a modular, serverless memory coherence protocol that guarantees data consistency for DFS metadata. The protocol uses a simple `ACK-INV` mechanism to ensure that NameNodes have invalidated their caches before any new metadata is persisted to the metadata store. That is, when a NameNode performs a write operation on an INode, it issues an invalidation (`INV`) to the instances in the deployment responsible for caching each piece of metadata related to the modified INode. The write operation blocks until all active NameNodes have acknowledged (`ACK'd`) this `INV`, at which time the write operation can safely proceed. Our coherence protocol utilizes the pluggable "Coordinator" service to facilitate communication among the NameNodes. The Coordinator is used to keep track of which NameNode instances are actively running in which deployments and to deliver the `ACK`s and `INV`s. λFS builds a subtree coherence protocol atop the simple, single-INode-based protocol (see Appendix D).

To describe the coherence protocol, we use the following notations. First, recall that there are n deployments across which the NameNodes are partitioned. Let $\mathcal{D}$ denote the set of deployments caching at least one piece of metadata in the target path of a write operation. Next, let N_L denote the "leader" NameNode, which is the NameNode performing the write operation. To orchestrate the

Table 1: Lines of code required by the different components involved in the development and evaluation of λFS.

Component	LoC	Component	LoC
Benchmark drivers	15,000	Docker images	2,100
λFS	36,685	Python scripts	4,493
hammer-bench	3,160	λIndexFS	4,472
SQL+Shell scripts	435	**Total:**	**67,352**

coherence protocol, N_L actively communicates with other NameNodes via the Coordinator.

The coherence protocol is described in Algorithm 1. It is conceptually divided into three steps. There is also a small amount of clean-up that is performed after the protocol terminates; this step is omitted for simplicity. The protocol guarantees the serialization of concurrent writes by utilizing exclusive locks in the persistent datastore. First, consider how once the leader NameNode N_L receives all ACKs from its followers, it is necessarily true that all other NameNodes will have invalidated their caches. Next, N_L will have taken exclusive write-locks on the metadata in the persistent datastore, so it will be impossible for another NameNode to read and cache the metadata before it is updated. This effectively serializes write operations against any other concurrent writes on the same data, thereby guaranteeing strong consistency.

3.6 Fault Tolerance

By default, both single INode and subtree operations do not span multiple NameNodes; however, a multi-node subtree batching mechanism (described in-detail in Appendix D) may be enabled to reduce the latency of subtree operations. λFS reuses HopsFS' transaction model, and thus both individual request- and NameNode-level failures are handled exactly as HopsFS handles them. Clients transparently resubmit subtree operations to other NameNodes in the event of a crash. In the multi-node case, the failure of any node will be treated as though the entire operation failed, and clients will simply resubmit the operation. Since λFS' persistent data store provides ACID transaction semantics, and coupled with λFS' consistency protocol, failures cannot leave the namespace in an inconsistent state. Likewise, λFS' Coordination service ensures that crashes are detected, enabling the easy removal of locks held by crashed NameNodes.

4 IMPLEMENTATION

Implementing λFS. λFS is implemented as a fork of HopsFS 3.2.0.3. Both λFS [11] and the benchmarking application [12] are open-sourced. λFS can be used as a drop-in replacement for HopsFS since λFS' client API is a superset of the HopsFS API. λFS uses a deployment of Apache OpenWhisk [3] as its FaaS platform. λFS also supports other FaaS platforms including Nuclio [19]. Notably, adding support for Nuclio required just 108 additional lines of Java code in λFS. Additionally, λFS uses MySQL Cluster NDB 8.0.26 as its persistent metadata store and ZooKeeper [40] as its "Coordinator" service.

Porting λFS to IndexFS. We have also ported λFS to IndexFS, a scalable middleware MDS [54] for DFSes such as BeeGFS [6]. Thanks to λFS' modular design, the integration of λFS and IndexFS is conceptually similar to that of λFS and HopsFS, as shown in Figure 7. We briefly discuss a few key differences below.

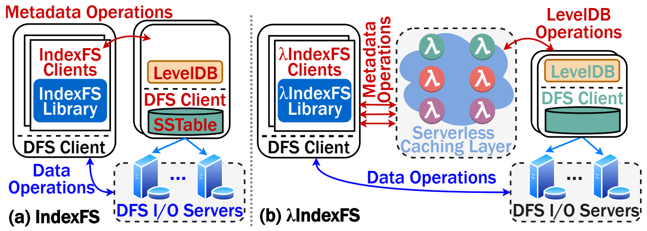


Figure 7: Porting λFS to IndexFS.

First, vanilla IndexFS relies on LevelDB to pack metadata into SSTables [13]. We decouple in-memory metadata handling from backend LevelDB by packaging the logic into serverless functions, and only using LevelDB as the persistent metadata store. Second, IndexFS leverages a sophisticated metadata partitioning algorithm adapted from GIGA+ [53]. After discussions with the IndexFS authors, we developed an alternative partitioning scheme that is easier to integrate with λFS. This scheme uses hashing to partition directories across LevelDB SSTables by directory names. Third, to make C++ based IndexFS compatible with λFS' Java-based serverless functions, we addressed multiple engineering challenges involving cross-language data types and library compatibility (e.g., Java's KryoNet [8] is not available for C++). Overall, we find that porting λFS is managable as λFS is designed to be modular and generalizable. For simplicity, we refer to our λFS-ported IndexFS as λIndexFS.

Porting λFS to Commercial FaaS Platforms. It is straightforward to port λFS to commercial FaaS platforms such as AWS Lambda. λFS' core techniques are not dependent on any particular FaaS platform. This includes λFS' RPC mechanism, as other frameworks have successfully used TCP-RPC-like mechanisms on commercial FaaS platforms in the past [37]. λFS could in theory be deployed on any FaaS platforms that support custom-container-based function deployment [1, 9, 17]. One challenge is how to minimize the performance impact of warm function reclamation [62], which we leave as our future work.

Summary of Implementation Efforts. We have implemented λFS and the software used for its evaluation in roughly 63,624 lines of Java/C++ code (see Table 1), completed over the course of more than two person-years. The benchmarking software constitutes 18,160 LoC, while λFS and λIndexFS together are composed of approximately 41,157 LoC.

5 EVALUATION

5.1 Experimental Setup & Methodology

In order to elucidate the effectiveness of λFS' various techniques and optimizations, we evaluated λFS against a number of other file systems. Specifically, we compared λFS against three state-of-the-art distributed file systems: HopsFS, IndexFS [54], and CephFS [63]. We also performed experiments that evaluated λFS' performance against that of a modified HopsFS, denoted "HopsFS+Cache", whose NameNodes had been augmented with an in-memory metadata cache similar to that of λFS. HopsFS+Cache serves as a serverful, cache-based DFS baseline. Finally, we compared λFS with InfiniCache [61], an in-memory object cache implemented atop FaaS, in order to better understand the efficacy of λFS' FaaS caching layer. InfiniCache uses a static, fixed-size deployment of cloud functions to serve I/O operations via short TCP connections that require invoking functions for every operation. InfiniCache thus serves as an approximation of λFS with no auto-scaling or long-lived TCP-RPC request mechanism. All experiments were performed on Amazon Web Services (AWS), and results were verified to be consistent with results obtained on Google Cloud Platform (GCP).

The experiments used deployments of the OpenWhisk serverless platform and MySQL Cluster NDB. Like λFS, HopsFS uses MySQL Cluster NDB as its persistent metadata store. OpenWhisk was deployed on AWS Elastic Kubernetes Service (EKS). All other VMs were deployed on AWS EC2. All AWS VMs used the `r5.4xlarge` instance type (16 vCPU and 128GB RAM). MySQL Cluster NDB 8.0.26 was deployed on GCE and EC2 with a single master node and four data nodes. We configured each NDB storage node according to the sample configuration provided by HopsFS. Unless otherwise specified, each λFS NameNode was configured with 6.25 vCPU and 30GB RAM. HopsFS' NameNodes were configured with 16 vCPU, 64GB RAM, and 200 RPC handlers. For clarity, we defer the description of the setup for the portability experiment with IndexFS to §5.7 as IndexFS' architecture is different from HopsFS.

To ensure a fair comparison between λFS and HopsFS, we allocated an equal amount of vCPUs and RAM to each framework's NameNode cluster (unless otherwise specified). However, imposing a fixed, total vCPU limitation on λFS' NameNodes implicitly restricted the maximum performance of λFS compared to what λFS could have achieved with the nearly unbounded resources typically provided by FaaS platforms. Because of this self-imposed bound on vCPUs, unrestricted λFS scale-outs would have over-used resources, leading to thrashing and severe performance degradation. To prevent this, λFS' scaling behavior was "toned down", and consequently λFS never actively provisioned more than 92.77% of the available vCPUs during these experiments. (We describe λFS' anti-thrashing technique in Appendix C.) The resource scaling tests presented in §5.3.2 illustrate how λFS' performance improves as more resources are allocated to λFS, thereby providing insight into how λFS would perform with nearly unbounded resources.

Our evaluation aims to answer the following questions:

- How does λFS perform under industrial workloads (§5.2)?
- To what extent does λFS' elasticity improve performance, scalability, resource- efficiency, and cost-efficiency (§5.2.4, §5.4, §5.2.5, §5.3.3)?
- How does λFS scale for individual DFS operations compared to other large-scale DFSes (§5.3)?
- Is λFS resilient to serverless NameNode failures (§5.6)?
- Can λFS benefit other DFSes besides HopsFS (§5.7)?

5.2 Industrial Workload

In this section, we present and discuss the results of executing a real-world, industrial workload on both λFS and HopsFS. The workload is based on the one used in HopsFS' evaluation, which was generated using statistics from traces of Spotify's 1600-node HDFS cluster. The frequencies of the file system operations are shown in Table 2.

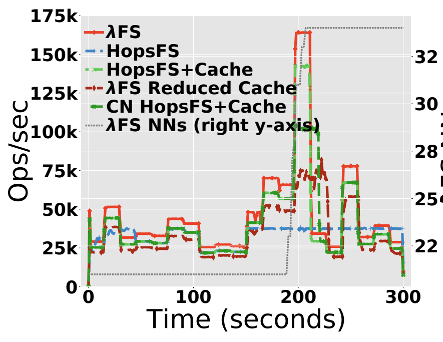
Table 2: Relative frequency of the file system operations used in the Spotify workload experiment.

Operation	Percentage	Operation	Percentage
`create` file	2.7%	`read` file	69.22%
`mkdirs`	0.02%	`stat` file/dir	17%
`delete` file/dir	0.75%	`ls` file/dir	9.01%
`mv` file/dir	1.3%	**Total Read Ops**	**95.23%**

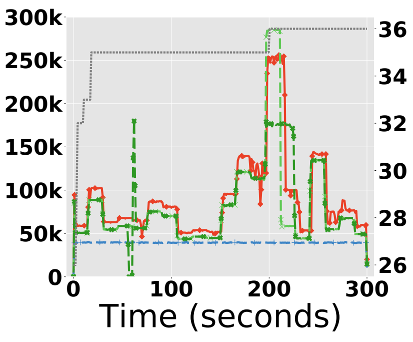
5.2.1 Experimental Setup. We implemented a DFS benchmark that can generate bursty file system loads by modifying the `hammer-bench` utility used to conduct HopsFS' evaluation [26, 51]. Our benchmark randomly varies system throughput over the course of the workload's execution in order to accurately simulate a real-world DFS workload [55]. Specifically, the workload is executed for 5 minutes. Every 15 seconds, the benchmark generates a random throughput value Δ from a Pareto distribution with a shape parameter $\alpha = 2$. (Please refer to [55] for a discussion on why the Pareto distribution is useful in this scenario.) Each client VM will attempt to sustain $\delta = \frac{\Delta}{n}$ ops/sec, where n is the total number of client VMs. If less than δ operations are completed in a given second, then the remaining operations roll over to the next second.

In order to demonstrate λFS' ability to elastically scale in response to bursts of metadata requests, the benchmark randomly generates throughput spikes up to 7× greater than the *base* throughput. We ran 2 different versions of the workload: one in which the Pareto distribution's scale parameter $x_t = 25,000$ and the other in which $x_t = 50,000$. The value of x_t determines the workload's base throughput. Both workloads were executed by 1,024 clients across 8 VMs. We allocated 512 vCPUs to HopsFS in order to maximize its performance during these tests. While allocating less vCPUs would've been cheaper, HopsFS' performance would've suffered, as shown by the resource scaling tests. Each λFS NameNode was allocated 5 vCPUs, and in the 25,000 ops/sec workload, λFS' NameNode cluster was collectively allocated just 50% of the total vCPU allocated to HopsFS' NameNode cluster in order to better illustrate λFS' resource and cost efficiency.

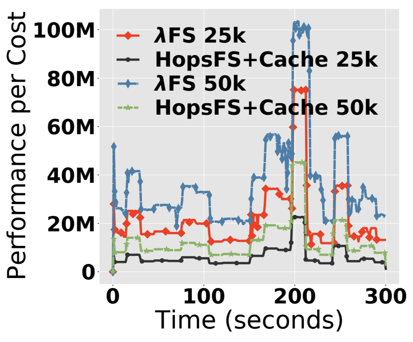
5.2.2 Throughput & Latency. Figure 8(a) shows a throughput comparison between λFS, HopsFS, and HopsFS+Cache during an execution of the Spotify workload with a base throughput of 25,000 ops/sec. Note that each of λFS' NameNodes was configured with 6GB of RAM for the 25,000 ops/sec workload. λFS achieved an average throughput of 45,690.34 ops/sec and an average latency of 1.02 ms during the execution of this workload. HopsFS achieved an average throughput of 38,134.35 ops/sec and an average latency of 10.58 ms. HopsFS+Cache achieved an average throughput of 45,945.1032 ops/sec and an average latency of 3.348 ms. Summarily, λFS achieved 90.40% (10.41×) lower latency and 16.53% (1.19×) higher throughput on average than HopsFS, while using 39.45% less resources. Compared to HopsFS+Cache, λFS achieved equivalent average throughput and 69.53% (3.28×) lower latency on average. λFS was successfully completed the entire workload, including the entire 15-second 163,996 ops/sec burst generated at time $t = 200$. Meanwhile, HopsFS' clients struggled to sustain loads above 38,000 ops/sec. When the



(a) Base: 25,000 ops/sec.



(b) Base: 50,000 ops/sec



(c) Performance-per-cost comparison.

Figure 8: Throughput and performance-per-cost comparison between the various systems during the Spotify workload. The number of active λFS NameNodes ("NNs") is shown on the secondary y-axis in both Figure 8(a) and 8(b).

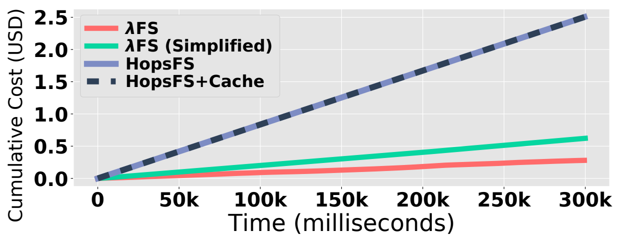


Figure 9: Cumulative cost of the 25k ops/sec Spotify workload. HopsFS' cost was \$2.50. λFS' cost was \$0.35 using AWS Lambda's prices, which are \$0.0000166667 per GB-second, charged at 1ms granularity, and \$0.20 per 1M requests [5]. Under the "simplified" cost model, λFS NameNodes incur cost while they're provisioned, similar to VMs, which overcharges compared to AWS Lambda's pay-per-use pricing model.

burst occurred, HopsFS had already "fallen behind" and was struggling to execute operations generated nearly a minute prior. So, λFS' peak sustained, throughput was 4.3× higher than that of HopsFS.

Figure 8(b) shows a throughput comparison during an execution of the Spotify workload with a base throughput of 50,000 ops/sec. For this test, λFS' FaaS platform was allocated 512 vCPU but used at most 180/512 (35.15%) of the available vCPUs. λFS achieved an average throughput of 90,875.60 ops/sec and an average latency of 4.31 ms during this workload. HopsFS achieved an average throughput of 44,956.28 ops/sec and an average latency of 22.40 ms.

HopsFS was unable to achieve the base throughput of 50,000 ops/sec, so it spent the duration of the workload attempting to "catch up". Meanwhile, λFS sustained approximately 250,000 ops/sec during the burst at around $t = 200$. To this end, λFS' peak, sustained throughput was 456.09% (5.56×) higher than that of HopsFS.

λFS' average throughput was 102.14% (2.02×) greater than HopsFS'; similarly, λFS' average latency was 80.76% (5.19×) lower than HopsFS'. For "read" operations, λFS achieved an average latency anywhere from 6.93×—20.13× lower than that of HopsFS (see Figure 10). However, λFS was unable to complete "write" operations as quickly as HopsFS because of the added overhead required by λFS' coherence protocol. Summarily, HopsFS achieved 1.5×—5.55× shorter "write" latencies compared to λFS.

InfiniCache failed to complete either of the two Spotify workloads. The FaaS platform became overwhelmed by the volume of HTTP requests: the high-latency HTTP requests and static, fixed-size deployment were insufficient for both the base throughput and bursts of work during the workloads.

Because FaaS assumes a near-unbounded amount of resources, fixing the amount of vCPU allocated to the platform results in poor performance and scalability. To perform a fair comparison and highlight the cost-saving benefits of FaaS, we also compared λFS against a "cost-normalized" configuration of HopsFS+Cache, referred to as "CN HopsFS+Cache". Specifically, we configured CN HopsFS+Cache with 72 and 144 vCPU for the 25,000 and 50,000 ops/sec Spotify workloads, respectively. In doing so, CN HopsFS+Cache incurred the same monetary cost as λFS. Considering first the 25,000 ops/sec workload as shown in Figure 8(a), CN HopsFS+Cache achieved lower throughput than λFS, failing to sustain the burst of requests around the 200th second of the workload. This phenomenon occurs again during the 50,000 ops/sec workload as shown in Figure 8(b).

5.2.3 In-Memory Metadata Cache. To measure the performance impact of λFS' metadata caching layer, we executed another instance of the $x_t = 25,000$ workload in which we decreased the capacity of the serverless NameNode cache to less than half the working set size (WSS) of the workload. As shown in Figure 8(a), "reduced-cache λFS" achieved better performance than HopsFS, sustaining between 70,000—80,000 ops/sec during the largest burst. Despite failing to sustain 163,996 ops/sec, "reduced-cache λFS" quickly caught up and completed the remainder of the workload.

5.2.4 Elastic Auto-Scaling. The results of the Spotify workload demonstrate λFS' ability to handle large bursts of work. Figures 8(a) and 8(b) show that λFS provisioned additional NameNodes to satisfy the influx of requests as soon as the workload started. λFS quickly scaled-out again near the 200-second mark, which is when the 7× request burst occurred, demonstrating the effectiveness of λFS' auto-scaling policy. With unbounded resources, λFS could rapidly scale-out to much higher load spikes. This is supported by the trend shown in the resource scaling experiments (Figure 12).

5.2.5 Monetary Cost. Figure 9 shows the cumulative cost for the 25,000 ops/sec Spotify workload for λFS, HopsFS, and HopsFS+Cache. For λFS, the cost was computed as follows: for every 1 ms interval of the workload, we billed each NameNode actively serving an HTTP or TCP request using AWS Lambda's prices (as described in

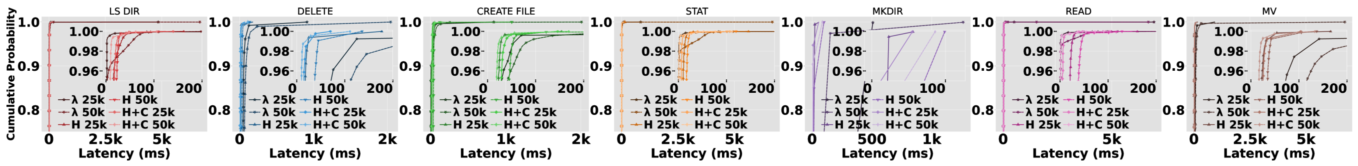


Figure 10: Latency CDFs of λFS ("λ"), HopsFS ("H"), and HopsFS+Cache ("H+C") for both versions of the Spotify workload.

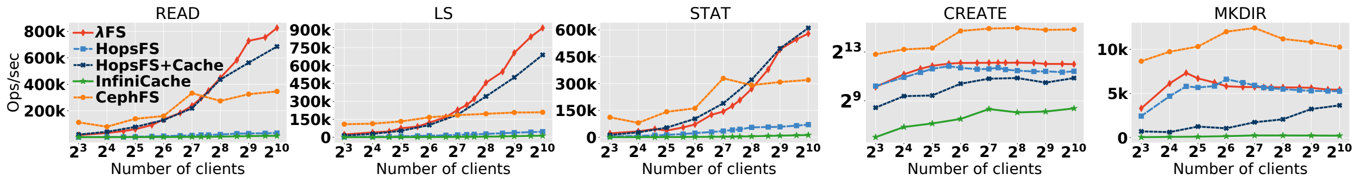


Figure 11: *Client-driven* scaling comparison between the various systems. The total amount of vCPUs allocated to λFS and HopsFS was held constant at 512 vCPUs, and each client performed 3,072 operations. The number of clients ranged from 8 to 1,024. Note that the y-axis for the `create` operation is log-scale.

Figure 9). If no requests were actively being served by a particular NameNode, then that NameNode incurred no cost. For HopsFS and HopsFS+Cache, we billed the cost of the entire 512-vCPU cluster for each 1 ms interval of the workload. By the end of the workload, the cumulative cost of HopsFS and HopsFS+Cache was \$2.50 while λFS' cumulative cost was just \$0.35. By taking advantage of FaaS' pay-per-use pricing model and our agile auto-scaling policy, λFS reduced the cost of executing the workload by 85.99% (7.14×) compared to HopsFS and HopsFS+Cache while achieving better performance with fewer resources.

We also computed the cost of λFS using a *simplified* pricing model, which is shown in Figure 9 as "λFS (Simplified)." Under this model, active NameNode instances incurred cost as long as they are provisioned [22], which doubled the cost of λFS compared to the pay-per-use FaaS pricing model. This illustrates how λFS takes advantage of FaaS' pay-per-use pricing model to greatly reduce tenant-side costs.

While the use of FaaS can yield improved elasticity, scalability, and performance, other primary benefits of FaaS are reduced tenant-side cost and increased cost-effectiveness. In particular, λFS' cost-effectiveness arises from its ability to achieve superior or equivalent performance while using a smaller amount of resources. By saturating a large number of relatively small, individual serverless NameNodes, λFS exhibits high resource utilization and resource efficiency with respect to the resources provisioned to it by the FaaS provider. Likewise, by leveraging the pay-per-use property of FaaS, λFS is ultimately able to reduce workload costs while delivering equivalent or better performance.

To quantify this notion of cost-efficiency, we define a new metric *performance-per-cost*, given as one of $\frac{\text{instantaneous throughput}}{\text{instantaneous cost}}$ or $\frac{\text{average throughput}}{\text{total cost}}$. The units are $\frac{operations/second}{\$}$ $(=\frac{operations}{second \times \$})$, or operations-per-second-per-dollar. This metric provides a measurement of cost-efficiency — a higher value indicates that the associated framework is able to achieve a higher performance-cost ratio, which is desirable. Increasing the value of this metric can be done using some combination of increasing throughput and decreasing cost.

Figure 8(c) shows the performance-per-cost for each second of the real-world Spotify workload for λFS and HopsFS+Cache. The cost for HopsFS+Cache is computed as the cost of running the 32 NameNode VMs for one second, whereas the cost for λFS is calculated using the pay-per-use pricing model of FaaS. Specifically, the resources allocated to an active NameNode are only billed if that NameNode served a request within that second. λFS achieved significantly higher performance-per-cost compared to HopsFS+Cache. This is because λFS experienced equal or greater throughput compared to HopsFS+Cache for the entirety of the workload while using significantly fewer resources (at-most 165 or 180 vCPU for λFS, depending on the workload, compared to the 512 vCPU used by HopsFS+Cache during both workloads).

5.3 Scalability

Next, we evaluate the scalability of λFS, HopsFS, HopsFS+Cache, InfiniCache, and CephFS using two micro-benchmarks covering key DFS operations including `read`, `stat file`, `ls`, `mkdir`, and `create file`. All operations target random files and directories across an existing directory tree. The first microbenchmark tests λFS' *client-driven scaling*: the ability of λFS to automatically scale-out as the number of clients increases, given a fixed resource cap. The second microbenchmark, which we call *resource scaling*, tests horizontal scalability (i.e., performance scaling with more deployments) and intra-deployment, vertical auto-scaling. The results of these tests illustrate λFS' ability to transparently adapt to increases in both request load and available resources in order to maximize performance. We allocated a maximum of 512 vCPUs to all systems during these tests, and for λFS we provisioned at-most 76 NameNodes, each with 6.25 vCPUs, meaning λFS used at-most 76 × 6.25 = 475/512 (92.77%) of its allocated vCPUs during these tests.

5.3.1 Client-Driven Scaling. In this test, the amount of vCPUs allocated to both systems was fixed at 512 vCPUs to maximize performance, and each client executed 3,072 operations. The total number of clients configured for each framework was varied between 8 and 1,024 in order to provide a wide range of scales to evaluate the frameworks. The results of this experiment are shown in Figure 11.

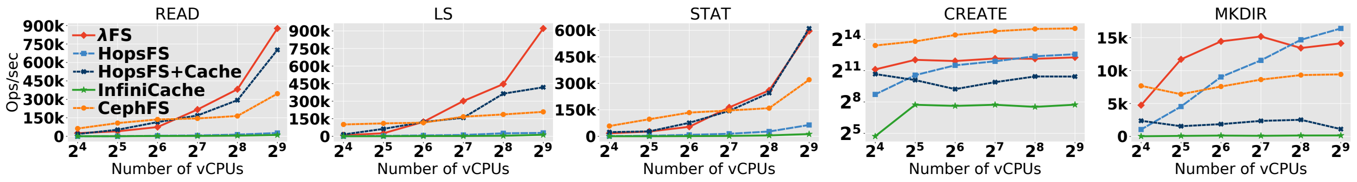


Figure 12: *Resource scaling* comparison between the various systems. The amount of vCPUs allocated to the systems ranged from 16 to 512. For each problem size, all systems used the same number of clients, each of which performed 3,072 operations.

λFS ultimately achieved higher throughput for all read operations (i.e., `read`, `stat`, and `ls`) for all problem sizes: λFS averaged 28.91×, 8.22×, and 20.53× higher throughput than HopsFS for `read`, `stat`, and `ls`, respectively. CephFS outperforms the other file systems for `read`, `ls`, and `stat` for the first 4-5 problem sizes but fails to scale well beyond this point, only outperforming HopsFS and InfiniCache. λFS outperforms HopsFS+Cache for `read` and `ls` and achieves comparable performance for `stat`.

There are several reasons for the throughput differences. First, λFS' elastic caching layer efficiently serves metadata to clients from the memory of serverless functions rather than the persistent metadata store that is 2 network hops away. This also decreases the likelihood of the persistent metadata store becoming a bottleneck, as it is with HopsFS. Second, λFS elastically scales-out its NameNodes in accordance with its agile auto-scaling policy in order to satisfy the increasing request load, whereas HopsFS is limited by its fixed-scale deployment. Under λFS, there were 20 active NameNode instances for the 8 client case during the `read file` test. λFS then scaled-out to 74 NameNodes for the 1,024 client case, illustrating the efficacy of λFS' agile auto-scaling policy. It is also worth noting that λFS used at-most 462.5, 425, and 475 of the 512 available vCPU during the `read`, `ls`, and `stat` client-driven scaling tests, respectively. This illustrates λFS' resource efficiency, as λFS achieves strong performance with a fraction of the available resources.

For `create file` and `mkdir`, the performance disparity between λFS and HopsFS was not as significant as it was with read-based operations. The magnitude of the throughput achieved by both systems is also considerably lower than that of read operations. Specifically, λFS achieved 49.09% (1.49×) higher throughput than HopsFS for `create file`. For `mkdir`, the two systems achieved roughly the same throughput. The reason both systems achieved significantly lower throughput for write operations is because the persistent metadata store quickly becomes a bottleneck. InfiniCache experienced poor performance for similar reasons as with the `read` operations. HopsFS+Cache also experienced low throughput, as the consistent hashing scheme used by clients can be bottle-necked by hot directories. CephFS achieved higher throughput than the other frameworks. One possible explanation for this is because CephFS' "capabilities" system [7] enables more efficient `write` operations compared to the permission system used by HopsFS and λFS.

5.3.2 *Resource Scaling.* For the resource scaling experiments, the total amount of vCPUs allocated to each framework was varied between 16 and 512. As such, this experiment helps to elucidate how λFS would scale both horizontally and vertically with nearly unbounded (or at least additional) resources. Note that for each vCPU value, all systems used the same number of clients, each of which performed 3,072 operations. The largest throughput obtained is reported.

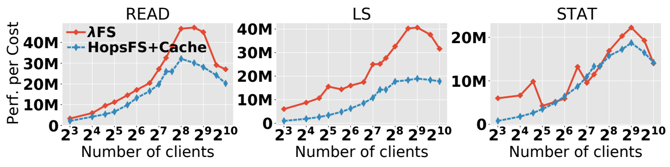


Figure 13: Performance-per-cost comparison between λFS and HopsFS+Cache for read-based file system operations.

Figure 12 shows the results. For read operations (`read`, `stat`, and `ls`), λFS exhibited significantly better scaling than HopsFS, InfiniCache, and CephFS, with higher throughput as the resources scaled. λFS achieved equivalent or superior throughput compared to HopsFS+Cache for all operations. For the largest problem size, λFS achieved 30.67×, 9.30×, and 20.69× higher throughput than HopsFS for `read`, `stat`, and `ls`, respectively. Likewise, λFS' throughput increased by 34.60×, 34.80×, and 72.08× This occurred because allocating more resources to λFS enables a higher degree of auto-scaling. For smaller vCPU allocations, λFS' auto-scaling is limited and it cannot dynamically adapt to the workload, resulting in worse performance. The performance trend is less dramatic for write operations since the persistent metadata store is the bottleneck.

λFS' superior scaling behavior can once again be attributed to its metadata cache and its agile auto-scaling policy. As the total amount of vCPUs increases, λFS provisions an increasingly large pool of concurrently-running serverless NameNodes. By using a large pool of relatively "small" serverless NameNodes, λFS achieves high resource utilization, as individual NameNodes utilize a majority of their allocated resources. This in turn enables λFS to achieve high performance with relatively modest resource allocations.

Though each HopsFS NameNode was configured with 200 RPC handler threads, HopsFS was not able to fully utilize the allocated resources, because its stateless NameNodes essentially serve as proxies, forwarding requests/responses between clients and the metadata store. This also explains why HopsFS' NameNodes had a consistently low CPU utilization at around 70%. Adding more NameNode servers may help, but again, it is difficult to pre-determine how many NameNodes to deploy for optimal performance and resource utilization.

5.3.3 *Cost-Efficiency.* Figure 13 shows the average performance-per-cost for λFS and HopsFS+Cache for the *client-driven* scaling tests. HopsFS+Cache's cost was computed as before: the cost of the 32 NameNode VMs running for the duration of the test. The cost of λFS was calculated using the simplified pricing model, which

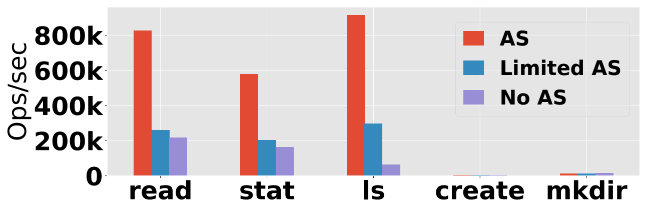


Figure 14: Performance impact of auto-scaling for λFS.

may have inflated the reported cost. However, since all active NameNodes were likely busy serving the high request volume for the entire experiment, the reported cost is likely close to the true cost.

λFS achieved higher performance-per-cost values for both `read file` and `ls` for all problem sizes. For `stat file`, λFS achieved higher performance-per-cost for 8 problem sizes and roughly equivalent performance-per-cost for the others. λFS achieved higher performance-per-cost for `read file` because λFS achieved equivalent or higher throughput than HopsFS+Cache using a fraction of the resources; while HopsFS+Cache used 512 vCPU, λFS used at-most 475 vCPU by the largest problem size. This phenomenon occurred to an even greater degree for `ls` for which λFS achieved 32.74% higher throughput with fewer resources. For `stat file`, λFS achieved equal or better cost-effectiveness compared to HopsFS+Cache, as the two frameworks achieved similar performance, but λFS used fewer resources. Note that λFS' cost-efficiency decreased for the final few problem sizes. This occurred because λFS saturated an increasingly large percentage of its available 512 vCPU resources. This trend can be avoided by increasing the resources allocated to the FaaS platform, enabling λFS to scale-out further.

5.4 Auto-Scaling

Figure 14 shows the impact on system throughput of enabling or disabling horizontal, intra-deployment auto-scaling for λFS across various file system operations. With auto-scaling "enabled", individual deployments were free to scale-out as they did in the other experiments. With limited auto-scaling, deployments could scale-out to at most 2-3 active instances. With auto-scaling disabled, each deployment was limited to a single active NameNode instance.

λFS achieved $2.85-3.17\times$ and $3.53-3.80\times$ higher throughput for `read` and `stat file` operations with auto-scaling enabled compared to limited and disabled auto-scaling, respectively. This trend is even more pronounced for `ls`, with λFS achieving $3.07\times$ and $14.37\times$ higher throughput with auto-scaling enabled compared to limited and disabled auto-scaling, respectively. The difference is less severe for write operations, as the bottleneck for writes is the persistent metadata store. These results further illustrate the importance of the FaaS-enabled agile auto-scaling policy within λFS' design as well as its significant impact on λFS' performance.

5.5 Subtree Operations

Table 3 shows the end-to-end latency of the `mv` operation performed on directories whose sizes varied between 2^{18} and 2^{20} files. On average, λFS completed the `mv` operation in 16.35% and 13.39% less time than HopsFS for 2^{18}-file and 2^{19}-file directories, respectively. End-to-end latency was roughly equal between HopsFS and λFS for 2^{20}-file directories. For large subtree operations, the persistent metadata store becomes the bottleneck, as every write operation must update persistent state in the database.

Table 3: Average end-to-end latency (ms) of subtree `mv` operations for varying `dir` sizes.

Directory Size	HopsFS	λFS
262k (2^{18})	7,511.60	6,455.80
524k (2^{19})	14,184.80	12,509.20
1.04M (2^{20})	25,137.00	25,220.80

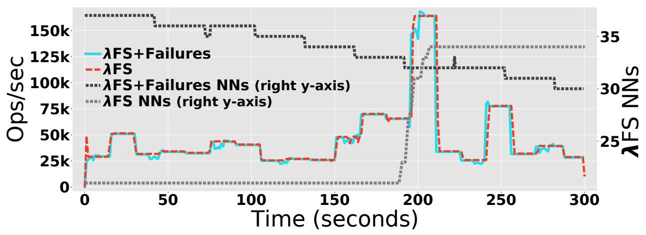
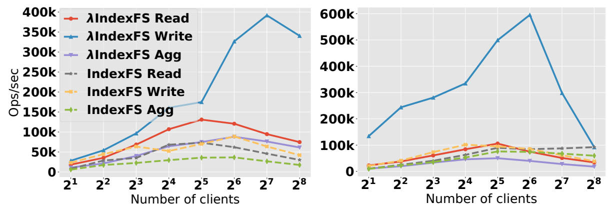


Figure 15: Fault tolerance test under the Spotify workload.



(a) Fixed-sized workload. (b) Variable-sized workload.

Figure 16: Comparison between λIndexFS and IndexFS on BeeGFS. `Agg` denotes the writes-followed-by-reads workload.

5.6 Fault Tolerance

To evaluate λFS' fault tolerance mechanisms, we executed the 25,000 ops/sec Spotify workload and manually terminated an active NameNode once every 30 seconds, targeting each deployment in a round-robin fashion. λFS began the workload with 36 active NameNodes (225/512 vCPU).

The results of this test are shown in Figure 15. Despite the failures, λFS completed the workload as generated, even during the 163,996 ops/sec burst. The darker dashed line shows the number of active λFS NameNodes. λFS' throughput decreased slightly following a termination event, as some clients were blocked, waiting for responses to requests that had been sent to the terminated NameNode. Once these requests timed-out, they were automatically resubmitted by clients. System throughput then rose briefly as clients temporarily increased their request rate to "make up" for the drop in throughput that followed the termination event.

5.7 λIndexFS vs. IndexFS

To further demonstrate λFS' portability and performance, we compare λIndexFS with IndexFS. For this test, we used a 7-VM BeeGFS cluster with 1 management sever, 1 metadata server, 1 storage server, and 4 BeeGFS client VMs. The cluster had 112 vCPUs and 448GB RAM. IndexFS was deployed on the 4 BeeGFS client VMs, which adheres to IndexFS' co-location principle [54]. λIndexFS ran 1 LevelDB instance on each BeeGFS client VM and used an OpenWhisk cluster with 64 vCPUs and 256GB RAM to host the serverless functions.

We evaluated λIndexFS using IndexFS' built-in benchmarking tool `tree-test`. We performed the following two client-driven scaling experiments.For the variable-sized workload, each client executed 10,000 `mknod` write operations followed by 10,000 random `getattr` read operations. For the fixed-sized workload, the total number of operations was fixed at 1 million writes followed by 1 million random reads. The number of clients varied from 2 to 256.

For read operations, λIndexFS' throughput is consistently higher than that of IndexFS, since most of the metadata is cached in serverless functions (Figure 16). Notably, for both workloads, λIndexFS significantly outperforms IndexFS in terms of write throughput, largely benefiting from λIndexFS' auto-scaling. λIndexFS' write throughput decreases when serving more than 2^6 clients due to OpenWhisk's limited resources (64 vCPUs). Despite the limited resources, λIndexFS still out-performs IndexFS, demonstrating the efficacy and portability of λFS.

6 RELATED WORK

InfiniFS [49] implements a technique called *speculative path resolution* and a client-side directory cache to optimize path resolution. Instead, λFS opts to use HopsFS' existing "INode Hint Cache" to optimize path resolution. λFS uses cloud-function-side caching rather than client-side caching.

IndexFS [54] is a layered, scaled-out MDS middleware built atop an existing DFS (e.g., PVFS [30] and BeeGFS [6]). IndexFS supports client-side, stateless caching, similar to the "INode Hint Cache" used in HopsFS and λFS. λFS goes beyond IndexFS by offering MDS elasticity with a consistent distributed metadata cache built atop serverless functions.

LocoFS [48] co-locates the metadata of a single directory on the same server, similar to how λFS' partitioning mechanism will co-locate metadata from single directories on the same deployments. This scheme can lead to single-node bottlenecks, as one metadata server can end up serving all requests for a hot directory. λFS avoids this bottleneck by leveraging FaaS' auto-scaling to scale-out overloaded deployments.

Lustre [14] hashes on file names. CalvinFS [59] hashes on full pathnames, while Giraffa [25] uses full file paths as primary keys to the associated metadata. BetreFS uses the pathname as a file index into the local file system. Lazy Hybrid [29] combines both directory subtree management and that hashing-based approach with lazy metadata relocation and lazily updated dual-entry access control lists. λFS also uses hashing but hashes on a file's parent directory.

InfiniCache [61] exploits the memory of AWS Lambda functions for caching large, read-only objects for low-throughput web apps. InfiniStore [66] is built atop InfiniCache, incorporating a tiered storage design that adds serverless memory elasticity and persistence. Faa$T [56] co-locates a key-value memory cache with a FaaS application to optimize the FaaS application's I/Os. Pocket [45] and Jiffy [44] provide elastic, serverful, ephemeral storage for serverless analytics. The DFS workloads that λFS targets are dramatically different than the applications mentioned above, therefore requiring new treatments when designing a serverless MDS system.

7 DISCUSSION AND LESSONS

While designing and implementing λFS, we learned several interesting lessons that have applicability beyond the scope of serverless DFS metadata management. First, creating latency-sensitive applications atop FaaS requires techniques to mitigate the high invocation overhead of serverless functions. Relying exclusively upon HTTPinvocations does not enable systems to achieve high throughput and low latency; instead, mechanisms such as λFS' hybrid invocation scheme are necessary for achieving good performance.

Notably, introducing techniques to circumvent the large overhead of HTTP invocations can reduce the system's ability to harness the auto-scaling property of FaaS. Such techniques must be designed with care so as to effectively optimize the trade-off between maximizing performance and maximizing elasticity and scalability. This can be considered an instance of the performance-parallelism trade-off—a trade-off that has been observed in FaaS systems from other domains [31, 32].

The use of FaaS also introduces a number of relatively complicated error states. Serverless functions can be reclaimed by the cloud provider at any point. If TCP-RPC connections are dropped unexpectedly, re-establishing connections is non-trivial due to the lack of addressibility of serverless functions. Additionally, naively resubmitting erred tasks via HTTP can result in request storms that overwhelm the serverless platform, leading to extreme over-provisioning of resources. This can ultimately cause a significant drop in performance and can cause errors elsewhere in the system—for example, the persistent metadata store may experience a temporary performance drop due to a wave of new connections from newly-provisioned NameNodes. To address this, FaaS-based systems must develop clever techniques to provide fault tolerance that avoids the aforementioned problems.

Similarly, FaaS-based systems are intended to support hundreds or thousands of clients. If thousands of clients concurrently issue HTTP invocations, then the FaaS platform may scale-out more rapidly than is desired, quickly increasing parallelism beyond what is necessary to sustain good performance. This can lead to increased costs and thrashing-like behavior, as the system rapidly over-corrects to changes in traffic patterns.

8 CONCLUSION

λFS is, to the best of our knowledge, the first cloud-native DFS metadata service, which uses the memory of serverless functions to cache and elastically scale a DFS' metadata workload. λFS achieves high-throughput, low-latency, low cost, and high resource efficiency by synthesizing a series of techniques built around a FaaS-based metadata cache. We have ported λFS to both HopsFS and IndexFS. We hope that this work will provide insight for building new, cloud-native, performance-sensitive backend services on FaaS. λFS is open-sourced and is available at:

https://github.com/ds2-lab/LambdaFS.

ACKNOWLEDGMENTS

We are grateful to the anonymous ASPLOS reviewers for their valuable feedback and comments that significantly improved the paper. This work was sponsored in part by NSF grants: CNS 1943204 / 2045680 / 2322860 (NSF CAREER Awards), an NSF CloudBank grant, CCF 1910747 / 1919075 / 1919113 / 2318628, OAC-2106446, and supported by an Adobe Research gift. Benjamin Carver was supported by a Presidential Scholarship from George Mason University.

REFERENCES

[1] Alibaba Cloud Function Compute Custom Container Runtime. https://www.alibabacloud.com/help/doc-detail/179368.htm.
[2] Apache Hadoop. http://hadoop.apache.org/.
[3] Apache OpenWhisk. https://github.com/apache/incubator-openwhisk.
[4] AWS Lambda. https://aws.amazon.com/lambda/.
[5] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.
[6] BeeGFS. https://www.beegfs.io/c/.
[7] Capabilities in CephFS. https://docs.ceph.com/en/quincy/cephfs/capabilities/.
[8] GitHub EsotericSoftware/kryonet. https://github.com/EsotericSoftware/kryonet/blob/03a135e2039bd7eb20e436ad70539238563d15a4/README.md.
[9] Google Cloud Run. https://cloud.google.com/run.
[10] kubernetes: Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[11] λFS Source Code. https://github.com/ds2-lab/LambdaFS.
[12] λFS Workload Driver. https://github.com/ds2-lab/LambdaFS-Benchmark-Utility.
[13] LevelDB. https://github.com/google/leveldb.
[14] Lustre file system. http://lustre.org/.
[15] MySQL :: MySQL 8.0 Reference Manual :: 23 MySQL NDB Cluster 8.0.
[16] MySQL Cluster NDB. https://www.mysql.com/products/cluster/.
[17] New for AWS Lambda – Container Image Support. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/.
[18] NSF Computational and Data-Enabled Science and Engineering (CDS&E). https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=504813.
[19] Nuclio. https://nuclio.io/.
[20] NumPy: the fundamental package for scientific computing with Python. http://www.numpy.org/.
[21] Preventing Long Tail Latency. https://www.section.io/blog/preventing-long-tail-latency/.
[22] Provisioned Concurrency for Lambda Functions. https://aws.amazon.com/blogs/aws/new-provisioned-concurrency-for-lambda-functions/.
[23] PyTorch: A Deep Learning Framework for Fast, Flexible Experimentation. https://pytorch.org/.
[24] REALIZING THE POTENTIAL OF DATA SCIENCE: Final Report from the National Science Foundation Computer and Information Science and Engineering Advisory Committee Data Science Working Group. https://www.nsf.gov/cise/ac-data-science-report/CISEACDataScienceReport1.19.17.pdf.
[25] Scaling Namespace Operations with Giraffa File System | USENIX. https://www.usenix.org/publications/login/summer2017/shvachko.
[26] Smkniazi/Hammer-Bench: HDFS-Distributed-BenchMark. https://github.com/smkniazi/hammer-bench.
[27] The exabyte club: LinkedIn's journey of scaling the Hadoop Distributed File System. https://shorturl.at/agoyH.
[28] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.
[29] S. A. Brandt, E. L. Miller, D. D. E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, pages 290–298, 2003.
[30] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *4th Annual Linux Showcase & Conference (ALS 2000)*, Atlanta, GA, October 2000. USENIX Association.
[31] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *ACM Symposium on Cloud Computing 2020 (SoCC'20)*, 2020.
[32] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In search of a fast and efficient serverless dag engine. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 1–10, 2019.
[33] Ryan Chard, Tyler J. Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. Serverless supercomputing: High performance function as a service for science, 2019.
[34] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.
[35] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, August 2012.
[36] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
[37] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.
[38] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.
[39] Jim Gray. Why do computers stop and what can be done about it?, 1985.
[40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.
[41] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *ACM SoCC '17*, 2017.
[42] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.
[43] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 244–254, 1986.
[44] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery.
[45] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, 2018. USENIX Association.
[46] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 213–226, USA, 2008. USENIX Association.
[47] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, nov 1989.
[48] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. Locofs: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.
[49] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An efficient metadata service for Large-Scale distributed filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, Santa Clara, CA, February 2022. USENIX Association.
[50] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
[51] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, February 2017. USENIX Association.
[52] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.
[53] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, February 2011. USENIX Association.
[54] K. Ren, Q. Zheng, S. Patil, and G. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 237–248, 2014.
[55] Zujie Ren, Biao Xu, Weisong Shi, Yongjian Ren, Feng Cao, Jiangbin Lin, and Zheng Ye. igen: A realistic request generator for cloud file systems benchmarking. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 343–350, 2016.
[56] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. *Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications*, page 122–137. Association for Computing Machinery, New York, NY, USA, 2021.

[57] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST ’10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[58] Konstantin V Shvachko. Hdfs scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE*, 35(2):6–16, 2010.

[59] Alexander Thomson and Daniel J. Abadi. {CalvinFS}: Consistent {WAN} Replication and Scalable Metadata Management for Distributed File Systems. pages 1–14, 2015.

[60] Huangshi Tian, Yunchuan Zheng, and Wei Wang. Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’19, page 139–151, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[62] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[63] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *SC ’04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pages 4–4, 2004.

[64] Lianghong Xu, James Cipar, Elie Krevat, Alexey Tumanov, Nitin Gupta, Michael A. Kozuch, and Gregory R. Ganger. Springfs: Bridging agility and performance in elastic distributed storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 243–255, Santa Clara, CA, February 2014. USENIX Association.

[65] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI 12*, 2012.

[66] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Vasily Tarasov, Dimitrios Skourtis, Feng Yan, and Yue Cheng. Infinistore: Elastic serverless cloud storage. *Proc. VLDB Endow.*, 16(7):1629–1642, may 2023.

A ARTIFACT APPENDIX

A.1 Abstract

This section provides supplementary material and instructions aimed at facilitating the reproducibility and further exploration of λFS and the experiments used to evaluate λFS and HopsFS. This collection encompasses a diverse set of resources, including source code, datasets, configurations, and tools utilized in the experimental setup. Additionally, detailed instructions for setting up the environment and executing experiments are provided primarily within the referenced GitHub repositories.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** AWS, Linux.
- **Hardware:** EC2 virtual machines.
- **Metrics:** Throughput, latency, monetary cost.
- **Output:** Numerical statistics.
- **Experiments:** Microbenchmarks, real-world workload trace executions.
- **How much disk space required (approximately)?:** 10s of GB across multiple virtual machines.
- **How much time is needed to prepare workflow (approximately)?:** Under 10 minutes once components are deployed and running. The installation process should take 45 - 75 minutes at most – *significantly* less if there are no errors.
- **How much time is needed to complete experiments (approximately)?:** 1-3 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0 license
- **Data licenses (if publicly available)?:** None.

A.3 Description

A.3.1 How to access. λFS and the software used for its evaluation are prepacked in Amazon Machine Images (AMI) on Amazon Web Services. You can find an up-to-date list of publicly-available Amazon Machine Images (AMIs) corresponding to the various components utilized by λFS and HopsFS in the λFS GitHub repository (specifically within the `aws-setup/public_AMIs.md file`). Likewise, the setup scripts and associated documentation are available in the `aws-setup/` directory of the λFS GitHub repository as well.

A.3.2 Hardware and Software dependencies. Our evaluation workloads run on AWS EC2 instances in the `us-east-1` region. They also use an Amazon Elastic Kubernetes Service (EKS) cluster on which OpenWhisk is deployed. OpenWhisk is the Functions-as-a-Service (FaaS) platform used by λFS.

λFS, HopsFS, and the primary benchmarking application are all written in Java (OpenJDK 64-bit 1.8.0_382) and compiled with Maven 3.6.3. They were developed in an Ubuntu-based environment (Ubuntu 22.04.1 LTS). Some components of the benchmarking software also use Python 3.10.12. These software dependencies are all pre-installed within the publicly-available AMIs.

A.4 Installation

The installation process primarily consists of provisioning the necessary virtual machines on Amazon Web Services, and then adjusting the configuration of software contained within those virtual machines as-needed. In order to simplify the process of creating the necessary VMs, we've provided a number of Amazon Machine Images (AMIs) (whose IDs are listed in the `public_AMIs.md` file within the `aws-setup/documentation/` directory of the λFS GitHub repository). These are all that is required to setup/deploy λFS and Vanilla HopsFS as well as run the experiments from our evaluation.

The `aws-setup/` directory of the λFS GitHub repository contains the latest and most up-to-date scripts and documentation concerning the installation, deployment, and execution of the framework. At a high-level, a majority of the required AWS infrastructure is created automatically using the `create_aws_infrastructure.py` Python script available in the λFS GitHub repository. There are also several additional scripts used to automatically apply patches to various components in order to resolve commonly-encountered deployment problems.

The source code, as well as additional documentation, is available in the following GitHub repositories:

(1) λFS source code (primary artifact)
(2) Benchmarking software (for λFS)
(3) Benchmarking software (for HopsFS)
(4) λFS's OpenWhisk Java runtime (for AWS)
(5) OpenWhisk K8s Helm chart with λFS-specific configuration

Note that we also created a persistent identifier for the main artifact, which is available here.

A.5 Experiment workflow

The Vanilla HopsFS and λFS experiments are primarily orchestrated using our bench-marking software/application. There is a GitHub repository containing the latest version of this software for Vanilla HopsFS in the "`/home/ubuntu/repos/HopsFS-Benchmarking-Utility`" directory of the **Vanilla HopsFS Client** AMI. There is a corresponding repository for the λFS version in the same directory of the **λFS Client** AMI. This benchmarking application provides an real-time, terminal-based interface for interaction with both λFS and HopsFS. The benchmarking application enables users to perform individual file-system operations as well as execute full micro-benchmarks and real-world workloads.

A.6 Evaluation and expected results

The experiments in the benchmarking utility that correspond to those described in the evaluation of λFS are as follows:

(1) **Client-driven and resource scaling**: 17 "Write *n* Files with *n* Threads", 20 "Weak Scaling Reads v2", 21 "File Stat Benchmark", "23 List Directories from File", 24 "Stat File", and 25 "Weak Scaling (MKDIR)".
(2) **Real-world workload**: 26 "Randomly-generated workload"

For additional details on how to replicate the experiments described in this paper, please refer to the `asplos23_experiments.md` file in the `./documentation/` directory of the benchmark utility GitHub repository.

After selecting a benchmark to execute, the application will request specific values concerning the number of clients, the number of operations per client, and in some cases, the number of repeated trials. The values described in Section 5 can be provided at this point in order to reproduce the experimental results.

The application will begin the experiment and report results back to the user through log messages displayed within the terminal. These messages will include, among other things, metrics from each trial of the experiment, including average latency and throughput.

The benchmarking application can be deployed in a *distributed* mode, thereby enabling multiple VMs to create λFS (or HopsFS) clients. In our evaluation, we used up to 8 VMs in total to execute benchmarks. The creation and management of these VMs is delegated to an EC2 auto-scaling group. This auto-scaling group is provisioned by the provided installation scripts.

A.7 Experiment customization

You can modify the real-world Spotify workload by modifying the associated config file. The default location (in the provided AMI) is "`~/repos/HopsFS-Benchmarking-Utility/workload.yaml`". For a description of all configuration parameters for real-world workloads, please refer to the documentation in the λFS benchmark software's GitHub repository.

A.8 Methodology

We performed our evaluation on Amazon Web Services (AWS) and validated it (i.e., replicated the same results) on Google Cloud Platform (GCP).

B STRAGGLER MITIGATION

Tail latencies can have a detrimental impact on application performance and user experience [21, 50, 52]. In order to mitigate the negative impact of tail latencies, we employ a technique referred to as *straggler mitigation*. λFS clients maintain a moving-window average latency. When a request's latency is sufficiently larger than the average (based on a configurable threshold), the request is cancelled and resubmitted to another NameNode. This can reduce the worst-case tail latencies and lead to higher system throughput. We found that the average TCP RPC latency is between 1-5ms, so we default this threshold to 10, meaning TCP requests with a latency $\geq$ 50ms will be resubmitted.

C ANTI-THRASHING MODE

Typically, the FaaS platform is assumed to provide clients with virtually infinite compute and memory resources [4], and clients pay only for the resources they use. However, private clouds have limited cluster resources for hosting a DFS deployment [35, 54]. Additionally, to perform a fair comparison between λFS and HopsFS, some form of normalization is required, such as assigning equal vCPU to both frameworks, or provisioning the frameworks such that they incur the same monetary cost. However, placing a bound on the amount of resources can result in *thrashing* behavior in the FaaS platform. Recall that the serverless functions are organized into n different deployments, and further that the namespace is partitioned across these deployments by hashing on the parent directory's path. Consider a scenario in which the serverless cluster's CPU utilization is approaching 100%. If the FaaS platform attempts to create a new container, it may have to delete an existing container to make room. When this pattern of destroying and creating containers begins to occur frequently, system throughput plummets. This is because cold starts take a non-negligible amount of time, and constantly deleting and creating containers results in a large number of cold starts.

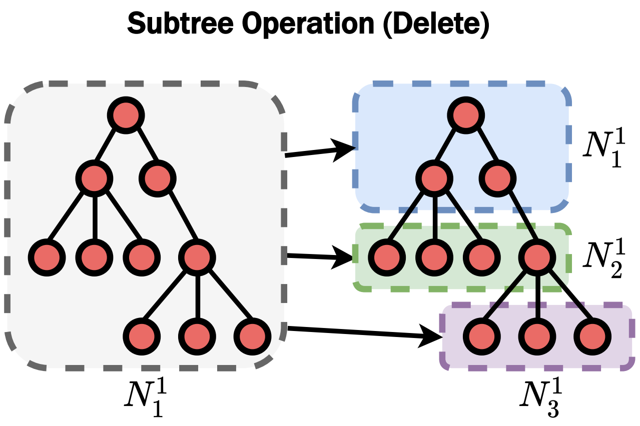


Figure 17: NameNode N_1^1 partitioning the sub-operations of a subtree delete operation to two other NameNodes N_2^1 and N_3^1.

To address this, client processes compute a moving average (with a configurable window size) of the latency of individual file system operations. When a metadata request observes a latency that is $T\times$ greater than the moving average latency, where T is a configurable threshold parameter, the client enters *anti-thrashing mode*. While in anti-thrashing mode, the client will opt to issue TCP RPCs for every metadata operation, even when no TCP connection exists to the NameNode in the deployment that is responsible for caching the requested metadata. By reusing TCP connections instead of issuing HTTP invocations, the FaaS platform will not create additional containers, as clients will issue requests to existing containers whenever possible. This will ultimately limit scaling and potentially result in reduced or leveled-off performance, but it avoids the severe performance degradation that occurs during thrashing. We find empirically that setting the threshold T between 2-3 provides the best performance.

D SUBTREE COHERENCE PROTOCOL

HopsFS implements subtree operations using an application-level distributed locking protocol. Part of this protocol involves partitioning the overall subtree operation into a number of sub-operations that are executed in-parallel.

There are three main phases to this protocol. In Phase 1, an exclusive lock is acquired on the subtree root, and the *subtree lock* flag is persisted to the database (NDB). Active subtree operations are also stored in a table, which is queried before beginning new subtree operations in order to ensure no two operations overlap (i.e., subtree isolation). In Phase 2, the subtree is quiesced by taking and releasing database write locks on all INodes within the tree, using a predefined total ordering to avoid deadlocks. This also builds a tree data structure in-memory for use during the subtree operation. Finally, in Phase 3, the whole subtree operation is partitioned into sub-operations that can execute in-parallel. Batches of INodes are modified in each transaction in order to improve performance.

λFS augments the standard HopsFS subtree protocol above by integrating our serverless memory coherence protocol. A naive integration of the protocol with the standard subtree protocol would involve executing the coherence protocol once for each individual sub-operation. This would result in extremely poor performance for large subtree operations. To address this, λFS performs the coherence protocol just once for the *entire* subtree. This is done using a special type of invalidation, referred to as a *subtree* or *prefix* invalidation. Rather than specifying the individual metadata to be invalidated, λFS specifies the file path *prefix* such that any cached INodes prefixed by this value will be invalidated. We use the subtree root as this prefix. NameNodes then utilize the trie structure of the metadata cache (§3.3) to efficiently invalidate all INodes contained within the subtree.

Subtree invalidations are issued to all deployments responsible for caching at least one piece of metadata in the subtree. These deployments are calculated by a NameNode during a step in the Vanilla HopsFS subtree protocol. Specifically, the NameNode walks through the subtree in a predefined total order, taking out write locks. This is done to quiesce the subtree. It is during this step that we also calculate the set of deployments responsible for caching metadata in the subtree.

As an example, consider a scenario in which the user deletes a subtree rooted at directory "`/foo/`". This directory may contain thousands of files and sub-directories. If the baseline coherence protocol (Algorithm 1) were used here, then thousands of individual invalidations would be required—one for each INode within the subtree. Instead, the leader NameNode simply issues a single *subtree* invalidation with the prefix "`/foo/`" to all deployments caching any metadata within the subtree. Once the leader NameNode has received all the `ACKs`, λFS is free to execute the subtree operation without running any further instances of the coherence protocol.

Elastically Offloading Batched Operations. The sub-operations created during subtree operations are typically executed in-parallel on the NameNode orchestrating the subtree operation. This works well when each NameNode has a large amount of CPU resources allocated to it. Serverless NameNodes, however, typically have a small amount of CPU cores allocated. As a result, executing hundreds or thousands of operations can be slow. To address this, we designed a technique referred to as serverless offloading. That is, λFS offloads batches of sub-operations to other NameNodes by taking advantage of FaaS elasticity in order to increase parallelism and scalability. The overhead of the coherence protocol is therefore minimized, thanks to batching and serverless offloading.

The batch size is configurable. We found that larger batch sizes tend to perform better, as there is a trade-off between increasing parallelism and the network overhead of offloading the operations. The batch size parameter defaults to 512. Figure 17 illustrates an example of this procedure. In Figure 17, we refer to the i^{th} NameNode in the j^{th} deployment as N_i^j. We say that a NameNode N belongs to deployment D_i (the i^{th} deployment) using $N \in D_i$.

In this example, the client sends a "`rm -rf /foo/bar`" operation to NameNode N_1^1, which caches all of the files and sub-directories rooted under `/foo/bar`; N_1^1 offloads level 2 and level 3 of the subtree to a different set of helper NameNodes, N_2^1 and N_3^1, from deployment 2 and 3. This does not create a consistency problem as the helper NameNodes simply help N_1^1 process part of N_1^1's load to speedup the subtree processing.