Appears in the Proceedings of the 16th ACM International Systems and Storage Conference (SYSTOR '23)
Won Best Paper Award.

F3: Serving Files Efficiently in Serverless Computing

Alex Merenstein
Stony Brook University
Stony Brook, New York
mmerenstein@cs.stonybrook.edu

Vasily Tarasov
IBM Research - Almaden
San Jose, California
vtarasov@us.ibm.com

Ali Anwar
University of Minnesota
Minneapolis, Minnesota
aanwar@umn.edu

Scott Guthridge
IBM Research - Almaden
San Jose, California
guthridg@us.ibm.com

Erez Zadok
Stony Brook University
Stony Brook, New York
ezk@cs.stonybrook.edu

ABSTRACT

Serverless platforms offer on-demand computation and represent a significant shift from previous platforms that typically required resources to be pre-allocated (*e.g.*, virtual machines). As serverless platforms have evolved, they have become suitable for a much wider range of applications than their original use cases. However, storage access remains a pain point that holds serverless back from becoming a completely generic computation platform.

Existing storage for serverless typically uses an object interface. Although object APIs are simple to use, they lack the richness, versatility, and performance of file based APIs. Additionally, there is a large body of existing applications that relies on file-based interfaces. The lack of file based storage options prevents these applications from being ported to serverless environments.

In this paper, we present F3, a file system that offers features to improve file access in serverless platforms: (1) efficient handling of ephemeral data, by placing ephemeral and non-ephemeral data on storage that exists at a different points along the durability-performance tradeoff continuum, (2) locality-aware data scheduling, and (3) efficient reading while writing. We modified OpenWhisk to support attaching file-based storage and to leverage F3's features using hints. Our prototype evaluation of F3 shows improved performance of up to 1.5–6.5× compared to existing storage systems.

CCS CONCEPTS

• **Information systems** → **Information storage systems**; *Computing platforms*; Storage replication; **Cloud based storage**; *Distributed storage*; *Computing platforms.*

KEYWORDS

serverless storage, file systems, ephemeral data, performance optimization

SYSTOR '23, June 5–7, 2023, Haifa, Israel

ACM ISBN 978-1-4503-9962-3/23/06…$15.00
https://doi.org/10.1145/3579370.3594771

1 INTRODUCTION

Serverless platforms have already proven their utility in running small web-oriented tasks. They are approaching a turning point, however, where their on-demand computation is expanding to a wider range of applications [16, 31, 52]—possibly any application. To this end, serverless platforms have been relaxing constraints and adding features, for instance, allowing users to run arbitrary containers and increasing execution time limits to support longer-running actions. Here, an "action" is a snippet of code or a standalone executable, and a serverless application is made up of one or more actions [4, 7, 9].

Storage access, however, remains a pain point for generic applications in serverless environments. Serverless platforms typically support only object-based storage. Object is a natural choice for the short, stateless, web-oriented tasks for which serverless platforms were originally designed and used; but more generic applications frequently need functionality not supported by traditional object storage—for example file-based access to data, the ability to perform in-place modifications, support for concurrent writers, and the ability to read data as it is being written. The lack of support for these features has held serverless computing back from becoming a generic computational platform.

Although most serverless platforms still do not offer a way to connect file based storage to serverless applications (*e.g.*, IBM Cloud Functions [28], Google Cloud Functions [25], OpenWhisk [53], or Knative [36]); some (*e.g.*, AWS Lambda) have recently added support for file-based storage [5]. This is encouraging, as it indicates that industry has recognized the need for file-based storage in serverless applications. Existing file systems, however, were not designed for serverless platforms and lack important features that would benefit serverless applications. In particular, existing file systems lack functionalities that could accelerate *intermediate data transfer* between the individual actions that make up a serverless application.

Applications in serverless environments are often split into multiple components forming pipelines, where one component writes its output data sequentially to storage, the next component reads the data as input, then the system discards the intermediate data. By specifically facilitating this usage pattern, a storage system can improve data access and transfer performance. We identified three ways a storage system can aid this pattern: (1) storing the intermediate data on less durable, lower-latency local storage, (2) providing hints about the location of data to serverless schedulers so that subsequent stages of a pipeline can be scheduled close to

the data, and (3) making it possible for the next stage of a pipeline to begin reading before the previous stage has finished writing.

Durability vs. performance tradeoff. Durability provided by storage systems often comes at the cost of performance. For instance, in our experiments, disabling durability features (*e.g.*, erasure coding) increased read/write bandwidth by 42–45%, and using a local disk rather than networked file system further increased read/write bandwidth by 39–86%.

The data transferred in serverless applications is usually *ephemeral* (*i.e.*, short lived) and is needed only until it has been consumed by the reader. This enables a different durability-performance tradeoff to be made. For example, ephemeral data does not necessarily need strong durability features such as replication or erasure coding that are provided by many storage systems. Although durability features can sometimes be disabled in a given storage system, they are typically configurable only at volume or file system granularity. As a result, it is difficult to optimize for workloads that store both ephemeral and non-ephemeral data: both must exist at the same point along the durability-performance continuum.

Locality. For data to remain local to a server, the serverless platform's scheduler needs to know where the data a serverless application will consume are located within the cluster. Current storage systems either do not convey this information to serverless platforms, or are designed such that the information is not even applicable (*e.g.*, if, for data protection, the data is distributed across multiple nodes in the cluster). Either way, the result is that data transfers between components within a serverless application consume network bandwidth and incur the performance penalty associated with transmitting data across the cluster's network.

Reading while writing. Finally, it is often desirable to process data in a streaming fashion, *i.e.*, to read and process data while it is written to a file. Doing so speeds up end-to-end processing because a subsequent stage can begin without having to wait for the previous stage to finish. In object storage, it is not possible for an object to be open by a writer and reader at the same time. In distributed file systems, however, it is possible but file systems often make the conservative assumption that when a file is open for reading by one client and for writing by another client, that both clients must use unbuffered file accesses to ensure that readers and writers maintain consistency [13].

Unbuffered access significantly slows both the reader and the writer, negating any performance benefit of the read-while-writeaccess pattern. For data transfer in serverless applications, this is an overly conservative assumption since both reader and writer access the data only sequentially (*i.e.*, data is never modified once written).

In this paper we address the storage access and data transfer problems for serverless environments. First, we added file system support to a popular open-source serverless platform, OpenWhisk [53], to demonstrate how existing file storage solutions can work with a serverless platform. Next, we implemented a stackable file system, F3, that is designed to optimize the transfer of data between serverless applications and the individual components of a serverless application. F3 distinguishes ephemeral data from that requiring high durability, and transparently directs ephemeral data to node-local disks. This

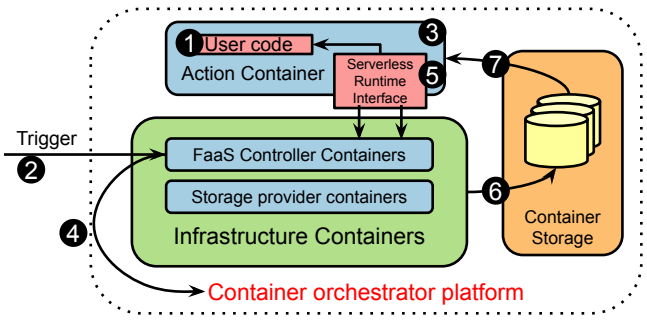

Figure 1: Blueprint architecture of edge serverless platform

enables F3 to perform up to 6.5× faster when writing data and 2.7× faster when reading data compare to the traditional durable storage.

F3 *further* optimizes data transfer by tracking the location of ephemeral files and exposing that information to serverless schedulers. We modified OpenWhisk's scheduler to use data location information when selecting the server to run the function, which in one experiment reduced network traffic used for data transfer from 2GB down to *zero*.

We designed F3 to stack over existing durable storage systems (*e.g.*, Ceph [13], Lustre [42], and GPFS [26]), making F3 a flexible and transparent extension to existing storage solutions. The resulting file system namespace makes both durable and ephemeral files visible to serverless applications.

Though F3 is generic and can be applied in different environments, we focused our empirical evaluation on a specific, rapidly growing use case—Edge Computing. Industrial edge computing is a new market that is predicted grow from $18B to $31B by 2025 [23]. Edge data centers are smaller facilities that range in size from street-side cabinets to cargo container-like structures that house a limited amount of server infrastructure. By having a smaller form factor than typical data centers (typically only 3–10 servers), edge data centers are relatively easy to move and deploy, making them a good fit for housing IT infrastructure at the edge. Serverless computing enables higher resource usage efficiency in these resource constrained environments through its fine-grained sharing [22]. Our experimental platform, workloads, and evaluation methodology are tailored to serverless computing at the edge.

This paper makes the following contributions:

(1) We describe the case for using file systems in serverless computing and extended OpenWhisk to enable attaching actions to file-based storage;
(2) We designed and implemented F3, a file system that extends existing storage systems to enhance data transfers between serverless actions;
(3) We evaluated F3 and several alternatives for edge computing; and
(4) We have made F3 and our modifications to OpenWhisk available as open-source software: *https://github.com/filesystems-for-serverless*.

2 BACKGROUND

In this section we give an overview of how serverless platforms operate (*e.g.*, AWS Lambda [3], Apache OpenWhisk [53]). Figure 1 depicts a serverless platform running on top of a container orchestrator.

Operation. Serverless platforms run processing ❶ on demand in a containerized environment [8]. Traditionally this processing consisted of snippets of code referred to as "functions." As serverless platforms have become more generalized, more and more of the processing is done by standalone executables (*e.g.*, an entire webserver or video processing utility). The term "function" seems insufficient for these more generalized and complex workloads, so we use the more generic term "action" to refer to both traditional function-style processing and newer more generic processing.

Actions run when triggered ❷ by a request to an HTTP endpoint. The trigger can be initiated in response to an event such as an upload to an object store. Information related to the trigger is passed to the action as parameters (*e.g.*, uploaded object name).

Running actions. The containers that run actions ❸ are often managed by a container-orchestration platform such as Kubernetes [59]. When an action is triggered, if there is an appropriate container already running, then that container runs the action. This is referred to as a *warm start*. If no suitable container exists, the serverless platform creates a new container for the action by making a request to the container orchestrator ❹; this is a *cold start*. In either case, a scheduling decision has to be made. If there are multiple warm containers suitable for an action, the serverless platform's scheduler must choose that container to run the action. If a cold start is required, then the containers orchestrator's scheduler must decide the cluster node on which to start the container, possibly using hints from the serverless platform's scheduler.

To avoid the overhead of cold starts, serverless platforms keep action containers running for some time after an action has executed. If the container's resources are needed for something else, however, then the container can be stopped as soon as the action ends. In either case, cluster resources are reserved and paid for only while the action is actually running.

Building and deploying actions. In early serverless offerings, actions were built by writing a snippet of code in a language such as JavaScript or Python. When triggered, the code was run using a container image built by the serverless platform. This approach allowed developers to focus solely on their code, but was somewhat restrictive in that it limited the languages supported. Also, because the serverless platform provided the execution environment, developers had little flexibility in the choice of libraries, runtimes, and other external resources.

The simplicity inherent in this approach is still sometimes desirable, and "Function as a Service" (FaaS) platforms continue to offer this method of building and deploying actions. For many use cases, however, more sophisticated actions are needed. These actions might use external libraries, have multiple executables, or require a specific execution environment (*e.g.*, a specific Linux distribution). To support these actions, most modern serverless platforms now allow developers to run an arbitrary container image in response to a trigger. On startup, these containers run a "Serverless Runtime Interface" executable ❺ that interfaces with the serverless platform. When triggered, the container image runs the Serverless Runtime Interface, which retrieves the action's input parameters, executes the action, and returns the results to the serverless platform. Thus, any application that can be containerized can be run as an action on a serverless platform. This approach opens serverless platforms to many more use cases than were originally designed.

Storage. Early code snippet-based actions were completely stateless, thus did not require access to persistent storage. When stateful serverless actions were later introduced, object stores were the recommended [15, 43] means to hold the state.

This made sense because (1) early serverless applications appeared mainly in web environments where object storage has been the norm, and (2) object stores are easy to access, requiring only the ability to form an outbound HTTP connection.

Although there are a wide variety of file and block storage options [33, 47] that container orchestrators can provision ❻ and attach ❼ to containers, current serverless platforms have not taken advantage of them.

3 STORAGE FOR SERVERLESS COMPUTING

In this section we first discuss the differences between file and object storage, then describe features existing file systems lack that would improve efficiency for serverless applications.

3.1 Object Stores vs. File Systems

In most serverless platforms, the only storage option available to actions is object storage. Object-based storage uses a key to identify an item of data, is typically accessed using through HTML requests, and supports operations PUT, GET, and DELETE. For many serverless applications, this interface is completely adequate and appropriate. We are not suggesting that the option of object storage in serverless platforms should be taken away.

But many applications that run in generic container images expect a file based interface, where files are identified by their names in a hierarchical namespace, and are accessed using operations such as `open`, `read`, and `write`. While file-to-object translation layers that can be embedded with the application exist, they generally do not support the richer functionality of files—including in-place modification, read-after-write consistency and directory-level operations—thus are not adequate for all applications.

Further, file systems typically provide higher performance than object stores [29, 55, 56]. Although high performance object stores could be implemented, applications that require high performance today are mainly file based [51].

One of the commonly cited benefits of serverless platforms is their near-limitless scalability. It might therefore seem counter-intuitive to suggest bringing file systems, often regarded as having limited scalability, to serverless platforms. Nevertheless, several major cloud providers have added file system support to their serverless platforms. This reinforces our belief that file system support is necessary, and that if serverless platforms are to take the next step toward becoming a generic computing platform, they must support file in addition to object interfaces.

3.2 Shortcomings of Existing File Systems

Existing shared file systems such as NFS and CephFS can provide storage for serverless applications. However, these file systems were not designed with serverless platforms in mind and lack features that would benefit serverless environments. Three such features are: (1) support for ephemeral (short-lived) data, (2) the ability to schedule actions based on where their data is located, and (3) support for reading files as they are being written.

Ephemeral data. Serverless applications make heavy use of ephemeral data—one that is short lived and that can be easily recreated. Ephemeral data comes from a variety of sources. For example, pipelines that span multiple actions may produce intermediate results generated by one action, consumed by another, and then discarded. Sensor and other user data generated at the edge is often filtered and pre-processed, with much of the original raw data not retained. Moreover, resources such as machine-learning models are frequently replaced with updated versions.

Many storage systems provide durability and reliability features such as replication or erasure coding. These features come with a performance cost. Since ephemeral data does not need these features, there is an opportunity to trade off decreased data reliability for increased performance.

In the case of node or disk failure, ephemeral data can be recreated by rerunning the original actions that created it. Detecting an action failure and rerunning the original actions requires a serverless execution framework that is beyond the scope of this paper; but we note that a file system could reasonably identify when a disk fails (*e.g.*, EIO errors) and inform the serverless execution framework. This would allow the execution framework to differentiate between regular action failures (*e.g.*, due to an application error) and action failures due to missing or corrupted data caused by disk failure. How a serverless execution framework handles such errors is part of the larger problem of serverless application orchestration (see Section 8).

Data locality-aware scheduling. When running an action, the serverless platform must decide where to run that action. Assuming the platform uses containers to run actions, this entails either (1) finding an available already running container and assigning the action to that container, or (2) starting a new container to run the action.

There has been a significant amount of work done in trying to avoid cold starts, since starting up a new container to run the action can significantly increase action latency and overall runtime. However, another factor that must be taken into account is the location of the data needed by the application. Running the action close to the data avoids the delay and overhead of moving the data to where the action runs.

Most existing storage systems do not provide the necessary data-locality scheduling hints. Those that do, provide them only at a volume granularity, too coarse for per-file-based scheduling. For example, with volume-level scheduling hints, an application's actions cannot simply write their output to a common output directory. Other systems that have incorporated data locality into serverless action scheduling (i) require applications to be structured in a specific way (*e.g.*, as a DAG) [12] and (ii) require information about the structure of the application before the application runs [12, 44].

Reading-while-writing. Pipelines where one process generates data as another process consumes it are common in Unix environments, especially in the form of Unix pipes (*e.g.*, `procA | procB`). Such a pipeline can reduce end-to-end application run times since the second process does not need to wait for the first process to finish before starting its processing.

This technique requires the two processes to share a kernel to facilitate piping the output from one process to the input of the next, and so porting such a pipeline to a serverless platform is not trivial. Note that in Unix pipes, the pipe's data is itself ephemeral and lives temporarily in kernel buffers.

One workaround is to use a temporary file as an intermediary, *e.g.*, `procA >/tmp/f & procB </tmp/f`. This solution can fail, however, since `procB` may read all of `/tmp/f` and exit before `procA` has finished writing, leaving some data unprocessed by `procB`.

A better workaround is to use an intermediary file, but to also have `procB` wait to exit until after `procA` closes `/tmp/f`. This is easy to do with the standard Unix utility `tail`: `procA >/tmp/f & PID=$!; tail --pid=$PID -f /tmp/f | procB`. Here, `tail` waits for additional data until `procA` exits.

This works on a single system where `tail` is able to test if `procA` has exited. However, if `procA` and `procB` are running in different serverless contexts, this workaround does not work.

Because pipelines are such a common idiom in serverless workflows, a file system that optimizes this pattern and increases parallelizablity between stages is highly desirable. When an intermediate file is used to communicate data between two actions, the file system is in a unique position to block the reader as necessary to wait for a concurrently running writer to add additional data to the file, returning end-of-file indication to the reader only after the writer has finished and closed the file.

4 DESIGN

We have designed a proof of concept file system, F3, that has all of the desired properties identified in Section 3. Figure 2 depicts F3's architecture. F3 is designed to layer on top of an existing durable file system, extending it with features benefiting serverless applications. F3 provides faster access to ephemeral data by storing it separately from non-ephemeral data on local, less durable storage without features like replication or erasure coding. Since ephemeral data is stored on node-local devices, F3 interfaces with the serverless platform to aid in scheduling actions on the nodes where their data resides. In the event that this is not possible (*e.g.*, because the load on that node is too high), F3 transparently and efficiently handles transferring the data between nodes.

We describe the design of the three serverless data transfer features in more detail below.

Ephemeral data support. F3 provides a common file system interface for both ephemeral and non-ephemeral data. To do this, F3 merges (1) a distributed, reliable, networked file system with (2) a file system on a fast local disk, and exposes a single mount point. Applications use the mount point exposed by F3, and F3 transparently writes file contents to either the networked file system or to the faster local file system.

The networked file system should be a file system that is accessible by every node in the cluster, such as CephFS or NFS. Each

node should have its own local data store for ephemeral data. This, for instance, is the case in a hyperconverged architecture, where storage is provided by aggregating disks attached to each compute node rather than using dedicated storage servers.

In our current implementation, users can mark a file or directory as ephemeral by setting the appropriate extended attribute on the file or directory or just use a special predefined file name extension. All data under an ephemeral directory is automatically marked ephemeral. In many workflows an application developer or user can easily identify which files are intermediate hence contain ephemeral data. In other cases some files (*e.g.*, stored in `/tmp`) or opened with O_TMPFILE, could be automatically designated as ephemeral. In the future, we can explore using more advanced automation for identifying ephemeral data.

For each volume, F3 creates a different top level directory on the local and networked file systems. This keeps the volume namespaces separate, so files in separate volumes can share the same name and path. Under this top level directory, F3 maintains the same directory hierarchy on both the local file system and the networked file system: the only difference is where the a file's contents are stored. It creates an empty file as a placeholder in the underlying file system where the file is not stored (*e.g.*, the networked file system if the file is an ephemeral file). However, if a F3 volume is created by extending an existing networked file system volume, F3 does not require any initial synchronization. Instead, F3 lazily creates the network file system's directory hierarchy on the local disk as needed.

F3 uses extended attributes on the copies of the files on the networked file system to track F3-specific metadata about a file. For example, we use extended attributes to mark whether the file is ephemeral, and if so which nodes in the cluster have a copy of that file's data. Storing metadata in the network file systems provides high durability for metadata. When a file is opened by an application, F3 checks the file's extended attributes to determine if the file is ephemeral: if so, it opens the copy of the file on the ephemeral data store and returns the file descriptor to the application. Otherwise, F3 opens the copy of the file on the networked file system and returns that file descriptor. If the extended attributes are missing, F3 assumes that the file is non-ephemeral. This can happen if F3 is extending a networked file system that has already been populated with data, for instance.

When F3 opens an ephemeral file, it first checks if the file contents are available locally. If not, F3 uses the extended attributes to find which nodes in the cluster have the file's contents. F3 then uses a per-node client/server communication to do a point-to-point, direct, efficient transfer of the file contents. As soon as the network transfer is initiated, F3 begins transferring the entire file and returns a file descriptor for the file to the application, which can then read the file as it is being downloaded.

The original copy of data is not deleted, and the data on either node can be used by subsequent actions. For our current implementation, we assume that ephemeral data is written once [34] so this copy of data does not need to be updated. As most ephemeral serverless data is written only once, this is a reasonable assumption. At this time we consider it the responsibility of the application developer to ensure that this assumption holds.

If a node or local disk fails and ephemeral data is lost, the action that created the data has to be re-run. This is consistent with the typical requirement that actions are idempotent [6, 30], and the fact that actions may be automatically re-run by the serverless platform in the event of certain kinds of errors [40].

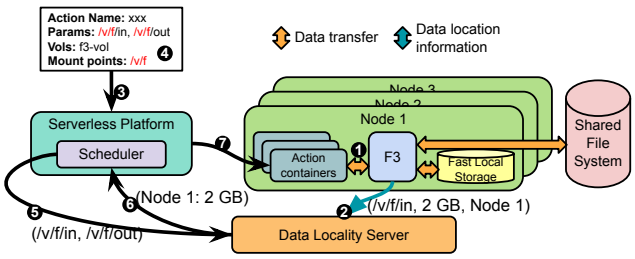

Figure 2: F3 architecture and locality-aware data operations

Our current implementation of F3 does not include any garbage collection to delete old data on the local disk. A simple approach would be to delete data as needed when the disk fills up, using an LRU algorithm to choose what data to delete. If a single action writes enough ephemeral data to fill up the local disk by itself, the current implementation of F3 would return `ENOSPC` to the application. Other approaches might be to have F3 copy the ephemeral data to the shared file system, store the data partially on the local disk and partially on the shared file system, or to have the serverless platform rerun the action and have F3 treat the data as non-ephemeral during the second run. We leave exploration of these options, as well as an implementation of a garbage collection mechanism, to future work.

Data locality hints for action scheduling. Collectively, the F3 file system drivers which run on each node in the cluster know what files are present in their local ephemeral data store. If the serverless platform's scheduler knows what files an action will access, the scheduler can ask the F3 file system for the location of the data and use that information in deciding where to schedule the action.

Rather than making the scheduler query each local instance of F3, F3 includes a simple server that centralizes this data locality information. Each local instance of F3 sends information about what files are on its node to this data locality server. The locality information is sent to the server asynchronously, so the server should not become a bottleneck in data operations.

Figure 2 details how the data locality feature in F3 works. When an ephemeral file is written ❶ to an F3 file system, the local instance of F3 on that node sends ❷ the file name, file size, and node name to the data locality server. F3 sends locality information twice: once when the file is created, and again when the file is closed. The locality information sent when the file is created allows the serverless scheduler to schedule pipelined actions on the same node, since it tells the scheduler where the data will be. The locality information sent when the file is closed allows the scheduler to make scheduling decisions based on the actual amount of data that each host has.

When the serverless platform receives ❸ a new action to run, its scheduler has to choose where to run the action. If there are suitable warm containers available, it chooses one of them; otherwise, it creates a new container. When taking data locality into account, the scheduler tries to identify all files that the action is likely to access. Currently this is done by identifying strings in the action parameters

that contain the mount point of the F3 file system ❹. This was sufficient for the applications that we used for our evaluation. In the future, more sophisticated methods such as predictions based on prior action invocations can be used to identify files likely to be accessed. Additionally, a serverless orchestrator or framework such as Kubeflow [37] that knows the relationship between actions could explicitly provide information about what data an action will produce or consume.

The scheduler then sends the list of files to the data locality server ❺. The data locality server then uses the information supplied by the F3 file system drivers to identify for each file in the list what nodes have the file locally and the size of each file. It sums the amount of data available on each node, and returns this information to the scheduler ❻. The scheduler uses the information to choose a container on a node with the largest amount of data available locally ❼. If there are no suitable containers the scheduler then uses this information to tell the container platform which node the new container should be created on.

Data locality is only one of several factors that the scheduler uses to place actions. For instance, if the node with the best data locality is overloaded, then the scheduler may instead decide to run an action on a less heavily utilized node. Ideally, the serverless scheduler would provide a mechanism for letting users decide how these different pieces of information are used when making scheduling decisions, similar to the flexibility offered by the Kubernetes scheduler [39].

Reading-while-writing. Usually a process consumes a file by issuing `read` system calls in a loop, stopping when `read` returns zero (*i.e.*, when the end of the file is reached). If the file is being written at the same time as it is being read, the reader would need to periodically poll for new data when `read` returns zero.

The challenge here is that the process needs to know when to stop polling because the writer has finished and closed the file. Unix pipes handle this transparently for a process: rather than returning zero, `read` blocks until more data is available as long as the write end of the pipe remains open.

F3 replicates this behavior by blocking `read` calls from returning if `read` would return zero but the file is open for writing by another process. When more data has been added to the file or the writer closes the file, F3 allows `read` to return to the caller. Since F3 spans the entire cluster, this works even if the writing process is running in a different container or a different node.

This feature makes it possible for a serverless scheduler to schedule the next stage of a pipeline before the previous stage has finished, thus improving concurrency. The same locality hints the scheduler uses to place the reader action can also be used to wait for the previous pipeline stage to create the file. Thus pipeline stages can be scheduled in parallel without code changes to either reader or writer.

If one of the pipeline steps fails, there may be subsequent stages that have already read part of the output from the failed step. If the pipeline previously ran on a single node, then it likely already has logic for dealing with this case and such logic can be reused in the serverless environment as well. For example, the application might cleanup the output from failed steps and then rerun. Since objects are written or read in their entirety, rather than incrementally as files are, additional logic may be needed for applications that currently use an object interface for storage. Detecting when a failure occurs and what recovery steps are needed (*e.g.*, failing downstream actions that are currently reading data from the failed action) is the responsibility of the serverless execution system and is out of scope for F3.

5 IMPLEMENTATION

We implemented F3, following the design described in Section 4. We targeted OpenWhisk [53] as the serverless platform, which we deployed on top of Kubernetes as the container orchestration platform. F3's implementation consists of four components and a series of modifications to OpenWhisk, described below. We plan to release these components publicly, as open source, available at *url-redacted*.

1. F3 file system driver. The F3 file system driver is implemented using FUSE [69, 70]. We used FUSE for this prototype rather than implementing a kernel-based driver due to FUSE's relative simplicity and ease of development. We expect that any performance penalty that FUSE imposes is insignificant compared to the benefits provided by F3 (*e.g.*, fewer network transfers). In the future, a kernel version of F3 could be implemented for production uses.

The F3 FUSE driver is implemented in 2,406 lines of C and C++. An instance of the FUSE driver runs on each node, for each F3 volume mounted on that node.

2. File transfer server & client. Ephemeral data written on one node and read on another node must be copied to the reader node via a network transfer. This functionality is implemented in a Go-based client and server, each of which runs on each node of the cluster. Go was chosen due to its strengths as a language for networked applications like file transfer clients and servers [66]. Additionally, Go's ability to compile into a portable executable eases the containerization and deployment of the file transfer and server [41].

The F3 FUSE driver communicates to the client via Unix domain sockets to request that a file's contents be downloaded from another node. The file transfer server and client are written in 574 lines of Go.

3. CSI driver. To integrate F3 with Kubernetes, we implemented a CSI (Container Storage Interface) driver [33] to enable provisioning and attaching F3 volumes to Kubernetes pods. The CSI driver is implemented in 811 lines of Go.For example, the CSI specification website lists 83 CSI drivers with source code: of those, 74 are implemented in Go [33]. When users create an F3 volume, they must also create a volume for the networked file system that F3 will use. The F3 volume definition indicates what networked file system volume to use with the Kubernetes label [38] `f3.target-pvc: foo`, where *foo* is the name of the network file system's volume.

When the CSI driver is instructed to attach an F3 volume (*i.e.*, receives a `NodePublishVolume` CSI command), the driver checks to see if the target networked file system volume is already mounted on the node where the F3 volume is being attached. If not, the F3 CSI driver creates a pod on the target node that is attached to the target networked file system. This forces the networked file system to be mounted on the target node. F3's FUSE file system can then access the mount point. We assume that each node's local data store is mounted in advance.

4. File locality server. The file locality server aggregates data from each F3 file system driver in the cluster. It is implemented in 214 lines of Go. The locality information about ephemeral data is stored on disk in a JSON formatted file. The durability of the locality

information is not critical, since the data itself is ephemeral and the serverless platform can always fall back to data-unaware scheduling.

5. OpenWhisk Modifications. In addition to the new components implemented above, we had to modify the OpenWhisk serverless platform. These changes included (1) adding support for attaching action containers to storage volumes, (2) identifying what files will potentially be accessed by an action, and (3) modifying the OpenWhisk scheduler to query the data locality server and using the response when choosing a container for an action.

In total, we changed about 700 lines of OpenWhisk code, most of it in the Scala language.

5.1 Unmodified Applications in Serverless

One of the advantages of file based storage for serverless is that it enables running unmodified applications. To highlight this capability, we wanted to use unmodified, "off-the-shelf" applications in our evaluation of F3.

During our evaluation we tested many combinations of container images, applications, and application command line options. To simplify this process, we implemented a mechanism that enables easily running a command as an OpenWhisk action. The user runs a command with the `ow-run` utility that we created. The user experience with `ow-run` is similar to that of running a command using the command line on their local machine. For example, consider we want to run this command, normally invoked locally, as follows:

```
trimmomatic /data/0.fastq /data/0.fastq.gz
```

To run that command on OpenWhisk using our `ow-run` utility, the command line invocation would be:

```
ow-run –container-image sunbeam:v0.0.7 –ow-action trim –vol-list f3-pvc –mount-path-list /data trimmomatic /data/0.fastq /data/0.fastq.gz
```

In this example, the user needs to have already configured the resource limits and requests for the `trim` action and created the F3 volume `f3-pvc`. However, the user needs to make no modifications to `trimmomatic` itself. This allowed us to easily and efficiently test a wide range of applications and application settings.

6 EVALUATION

Due to the growing amount of data produced by IoT devices, the rising demand for low-latency on-the-spot computing, as well as privacy and security concerns, applications and infrastructure are increasingly deployed at the Edge rather than in the hyper-scale Clouds [68]. The umbrella project for F3 focuses on the growing Edge business opportunities: thus, we designed our experimental platform and workloads to be representative of edge environments and workloads. Furthermore, our analysis shows that thanks to its higher resource efficiency, the serverless approach could be even more attractive at the resource-constrained Edge than in the Clouds with seemingly unlimited resources.

A typical edge data center is a cluster of only 1–10 servers located either at a customer site (*e.g.*, a factory or a retail sore) or at an Internet access point (*e.g.*, 5G cell tower). The servers in a typical edge data center run standard operating system (*e.g.*, Linux), virtualization software (*e.g.*, KVM), and container orchestrators (*e.g.*, Kubernetes). Due to constraints on clusters' physical footprint, a popular architecture for Edge data centers is hyperconverged setup, where each building block (*e.g.*, a server) provides both compute and storage resources. The testbed described in the following section reflects these characteristics of edge data centers.

6.1 Cluster and Storage Setup

We ran our evaluation on CloudLab [18] using a cluster of nine machines connected via a 1Gbps network, with each node running CentOS Linux 7.9.2009. Each machine had two 2.60GHz, ten-core Intel CPUs with hyperthreading, 160GB of RAM, and one 480GB SATA SSD. The cluster was connected via a private 1Gbps network. Our serverless platform was OpenWhisk 1.0.0, using Kubernetes 1.19.0 as the container orchestrator.

One node was dedicated to running the `etcd` server used by Kubernetes to store cluster state; another node was the Kubernetes control node; and a third node was dedicated to running an NFS server used in evaluation. The remaining six nodes were hyper-converged Kubernetes workers that ran both evaluated workloads and storage systems—CephFS, MinIO, and F3.

In our CloudLab setup every node had only one attached disk. Since F3 requires both a local disk and a shared file system, we used LVM to split the single SSD attached to each node into two volumes. We formatted one volume with ext4 and used that as F3's local disk; we used the other volume for CephFS and MinIO.

In our evaluation we assume the case when an edge cluster already has access to durable storage: CephFS (distributed file system), MinIO (object storage), or central NFS server (NAS). F3 can be layered over these solutions (except MinIO) to provide additional performance benefits in serving ephemeral data to serverless functions. We evaluate MinIO to provide a reference point of how applications perform with a popular object storage solution rather than a file system.

CephFS. Ceph [13] is a popular storage system built on the RADOS object store [57]. It aggregates storage from each node it is deployed on and exposes a single pool of storage. There are several interfaces for Ceph including CephFS, which exposes a file-system interface to applications. Ceph offers several data durability schemes, such as replication and erasure coding. We evaluated three different Ceph configurations: no replication, 3× replication, and 2-1 erasure coding (two data blocks and one erasure block).

CephFS has both kernel- and FUSE-based user-space drivers. We used the FUSE-based user-space drivers, which are typically more up to date and safer to use than their kernel counterparts. We used Ceph version 15.2.7, deployed on Kubernetes with the Rook [61] operator version 1.5.9.

MinIO. MinIO [48] is a popular object store. Like Ceph, it can aggregate storage across multiple nodes and expose a single storage pool. Also like Ceph, MinIO offers several data durability modes. We chose EC-3, which was the default for our sized cluster (six nodes, one disk per node). This mode splits data into three data chunks with three coding chunks. We used MinIO release 2022-09-07T22-25-02Z, deployed on Kubernetes with version 4.5.0 of the MinIO operator.

We used `s3fs` [62] to access MinIO's object API and provide a file-based interface over MinIO. This is representative of the current state of storage for serverless: if a user wishes to run an application on a serverless platform but the application requires a file based storage interface, they would need to use a tool like `s3fs` to access an object store.

NFS. NFS [64] is a well-established file system protocol. Although hyper-converged configurations such as those used by Ceph and MinIO are common, architectures that use standalone NAS storage appliances are still used. NFS is mature, and easier to deploy and configure compared to more sophisticated, distributed or networked file systems like CephFS. We used NFS on a standalone, dedicated server in our cluster—to represent this alternate architecture. We used the standard NFS server included with the Linux kernel to export a local disk formatted with ext4. The NFS version was 4.1, which was the default version available on our operating system (CentOS Linux 7.9.2009).

F3. In most experiments we evaluated F3 using CephFS with no replication as our networked file system. The local disks used as a per-node local data store were formatted with ext4, which is a commonly recommended file system and the default for many operating systems [60]. Although we mainly used CephFS as the networked file system for our evaluations, F3 is capable of stacking on top of any underlying networked file system that supports extended attributes. To test this, we verified that F3 also works on a recent NFSv4.2 server with extended attributes support.

We measured the impact of using different networked file systems (CephFS with no replication, 3× replication, 2-1 erasure coding, and NFS) with F3. We found that the choice of underlying file system had little to no impact on the performance of ephemeral data operations: performance in each case was within 3.1% of each other for reads where the data was not available locally, and less than 0.03% and hence statistically indistinguishable. of each other in all other cases. This is because F3 is designed to avoid the networked file system for ephemeral operations. We used unreplicated CephFS as our networked file system throughout our evaluation: any reference of "F3" in the evaluation means "F3, layered on top of unreplicated CephFS."

Since the focus of this evaluation was on F3's features for ephemeral data, all data in our evaluation was marked as being ephemeral. We leave to future work evaluating the performance of mixed ephemeral and non-ephemeral data operations, as well as how to automatically identify whether data is ephemeral or non-ephemeral.

Disk vs. network speed ratios. When selecting the server for a file system that accesses disks over the network, disk speeds and network speeds should be on par with each other so that neither dominates as the primary bottleneck. We chose network and disk speeds that were representative of real-world ratios. Each of our servers had only a single disk available for the storage systems under evaluation. We measured the disk speed to be 200MB/s, giving a disk to network throughput ratio of approximately 1.6 with the 1Gbps network. Although 1Gbps is slow compared to the networks found in many modern data centers, our disk to network throughput ratio falls within the range typical of real world edge deployments [74]. If we instead had ten disks with a combined throughput of 2000MB/s and a 10Gbps network, the ratio would remain the same.

6.2 Data Transfer Micro-Benchmarks

We evaluated the performance of data exchange and the impact of F3's data exchange optimizations. We first show the performance impact of different replication and erasure coding levels, compared to a baseline of accessing a local disk. This is the motivation behind F3's use of a local disk for ephemeral data.

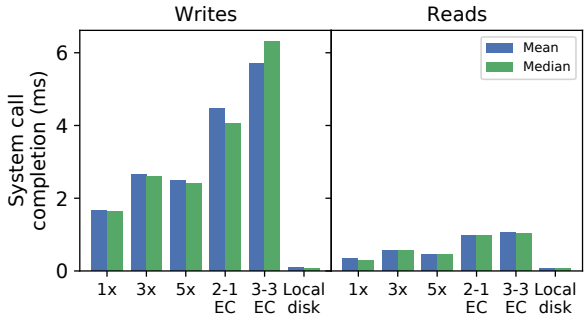

Figure 3: Mean and median system call latencies for different configurations of storage system. 1×, 3×, and 5× refer to the degree of replication; 2-1 and 3-3 EC refer to the erasure coding configuration (2 data and 1 coding chunk, and 3 data and 3 coding chunks, respectively).

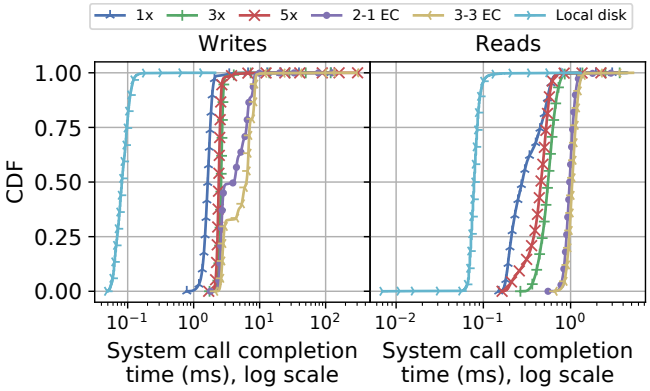

Figure 4: CDFs of `read` and `write` system call latencies, for different storage configurations. We used log scale for the x-axis because the system calls exhibited long tails at higher replication degrees.

We then show the impact of data locality based scheduling, and avoiding the overhead of transferring data across the network. Next, we show the combined impact of F3's data locality based scheduling and F3's use of local disk for ephemeral data. Finally, we show the impact of F3's optimizations for reading-while-writing.

Impact of replication & local disk storage. We evaluated the impact of replication and erasure coding on the latency of `read` and `write` system calls. We ran several experiments to time 100,000 reads and 100,000 writes on CephFS volumes with varying replication and erasure coding options, and compared with the same workload on an ext4 file system on a local disk. Since CephFS uses a FUSE driver, we used a passthrough FUSE file system to access the ext4 file system. This ensured that all `read` and `write` system calls went through a FUSE layer for a more fair comparison. System call times were measured with `strace`, and were generated with `dd` with `bs=4K` and `[o|i]flag=direct`.

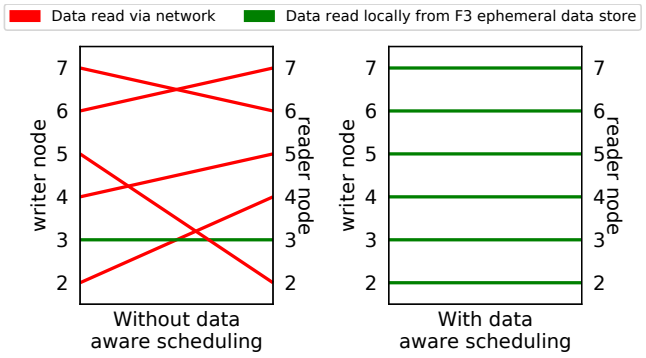

Figure 5: Impact of data aware scheduling. Each line connects a writer with its corresponding reader, with the numbers along each side showing what node in the cluster the writer or reader ran on. Red lines indicate that the reader needed to transfer its file from the writer node via a network transfer. Green lines indicate that the reader and writer ran on the same node and the file was read from F3's local ephemeral data store, with no network transfer was needed.

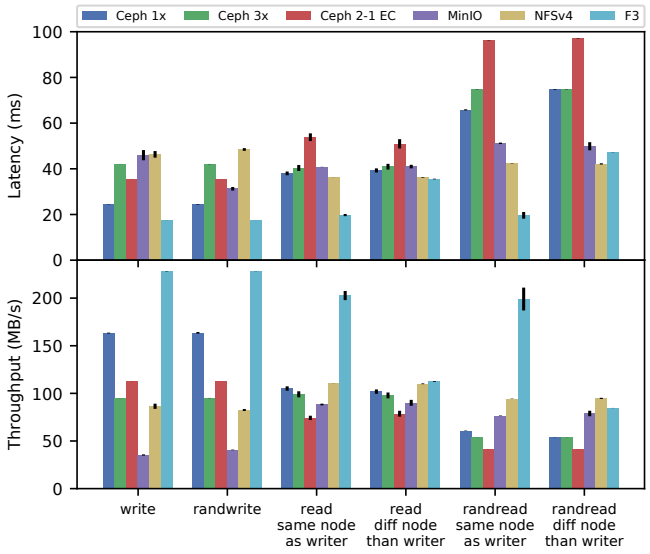

Figure 6: Mean latencies and bandwidths of Ceph, NFS, MinIO, and F3. "Ceph 1×" and "Ceph 3×" are configured with 1× and 3× replication, respectively. "Ceph 2-1 EC" uses erasure coding (data split into two data chunks and one coding chunk). F3 was layered on top of an unreplicated CephFS volume.

Figure 3 shows the mean and median system call latency across multiple storage configurations. The distribution of latencies exhibited a long tail, as can be seen in Figure 4 (note the log scale). This is expected, as there are multiple sources of variability in the storage and networking stacks, and have been observed before [11, 27, 32, 49]. As the degree of replication increases, we see the tail grow longer, which also makes sense as the number of sources of variability increases.

As expected, the local disk performed significantly better than CephFS, especially when writing: 0.1ms vs. 2.2ms for 1× replication). We also see that as the replication degree increased, generally so did system latency. The exception was that for reads, 3× and 5× replication perform about the same or slightly better than 1× replication.

Impact of data locality considerations during action scheduling. To demonstrate the impact of locality aware data scheduling, we wrote and then read six ephemeral files. Writers were run manually on each node, one per node, with each writing a unique 400MB file. For each writer, a corresponding reader was run in an OpenWhisk action that read the entire 400MB file. When the reader and writer both run on the same node, the reader reads its file from F3's local disk. However, when the reader and writer each run on separate nodes, the data must be transferred from the writer node to the reader node over the network.

The left-hand side of Figure 5 depicts the case where OpenWhisk's default scheduling is used. Here, the readers are assigned to nodes without regard to where the input file they need to read is located; we see that only a single reader (green line) ended up running on the same node as its corresponding writer. The red lines depict instances where the reader ran on a different node from its writer, necessitating a 400MB network transfer to copy the data from the writer node to the reader node. In total, using the default OpenWhisk scheduler resulted in $5 * 400 = 2000$MB of data being transferred across the cluster network.

The right side of Figure 5 shows the impact of our modified OpenWhisk scheduler that utilizes F3's data locality hints. All six readers were scheduled on the same node as the corresponding writer, and hence no data was transferred over the network.

Impact of replication, local disk storage, and data locality. We used `fio` [20] to measure sequential and random read and write performance of the storage systems. `fio` ran in a pod (container) via a serverless action. Write and read workloads were generated by separate instances of `fio` running in separate pods. The data written by `fio` was marked as ephemeral, and the reader instance ran after the writer instance finished. We measured read performance where the reader pod ran on the same node as the writer pod, as well as when the reader pod ran on a different node. This demonstrates the difference in performance that data locality can have on an I/O workload. We disabled F3's data locality based scheduling to be able to control whether the reader ran on the same or different node as the writer. We used a large (200GB) dataset to mitigate the impact of caching.

Figure 6 shows the bandwidth reported by `fio`, in MB/s, and the mean latencies, in milliseconds. We ran `fio` in each configuration three times. Error bars show that variance was small, less than 5% of the mean, with one exception: F3 random reads on the same node, where the variation was 7%.

As expected, F3 had the highest read and write performance when the reader was on the same node as the writer. F3's write bandwidth ranged from 1.40× to 6.46× faster than other storage systems; read bandwidth ranged from 1.84× to 2.30× faster. Latency ranged from 1.40× to 2.64× lower when writing and from 1.84× to 2.73× lower when reading. These performance improvements were due to F3's use of local storage. By using local storage, F3 is not limited by the cluster's network capacity as other storage systems are.

F3's read performance when readers and writers ran on different nodes was similar to NFS. In both cases, the data had to be transferred over the network.

Each of the networked file systems was limited by the cluster's 1Gbps (125MB/s) network. The one exception was writing in the unreplicated configuration of Ceph. This was expected because Ceph breaks files into blocks that are then distributed across each of the storage nodes in the Ceph cluster. Because we were using a hyperconverged architecture, the Ceph storage nodes were the same nodes that run user workloads, including our instance of `fio`. Since we had six nodes in our cluster, we expect then that $\frac{1}{6}$ of the data written by `fio` resided on the node running the `fio` program, and as a result was not limited by the cluster's network.

Impact of reading-while-writing. Passing data from one stage of a data processing pipeline to the next is a common pattern. A straightforward implementation is to run the pipeline stages sequentially, where each stage produces an output file that the next stage reads as input. A disadvantage of this approach, however, is that it provides no parallelism between pipeline stages.

Another possible implementation is to run pipeline stages concurrently, streaming the data between stages (*e.g.*, using UNIX pipes to connect them). The added parallelism of streaming can result in lower end-to-end processing times. A limitation of using UNIX pipes, however, is that the stages must all be run on the same node, which is not always convenient.

In this section, we use a third approach where a stage in the pipeline reads input from a file in a shared file system while the previous stage writes the file. We show below how we solved the problem of the reader reaching the end of file before the writer has finished writing all data.

We ran experiments in a serverless environment using CephFS, NFS, and F3 as the shared file system—first with the reader on the same node as the writer, then with the reader on a different node. We used a smaller data set (400MB) than the server's RAM (160GB), so the results reflect the ability of the storage systems to leverage the kernel's page cache rather than being disk bound.

To solve the early EOF problem, we made a few changes to our pipeline stages. First, we modified the writer stage to create an empty file, `/var/data/f.done`, on completion. Next, we split the reader into two parts. The first part was a script that read from `/var/data/f` and wrote to a FIFO pipe, `/tmp/f.pipe`. Whenever the script reached EOF on input, it checked for the existence of the `/var/data/f.done` file, and if not found, slept one second (same duration as `tail -f`), then returned to the top of the loop and continued reading. The second part was the actual reader program (*e.g.*, `grep` or `cat`), except that instead of reading from `/var/data/f`, it read from `/tmp/f.pipe`. Both parts of the reader ran in the same action. This implementation enabled us to run a reading-while-writing workflow in a serverless context.

Because F3 has special support for handling EOF in the reading-while-writing access pattern, it did not require any of the additional implementation: the writer action simply wrote to `/var/data/f` and the reader action simply read from `/var/data/f`.

Note that because CephFS was not designed for this usage pattern, it does not handle the reading-while-writing case efficiently when reader and writer run on different nodes. In this pattern, it falls back to unbuffered reading and writing [14].

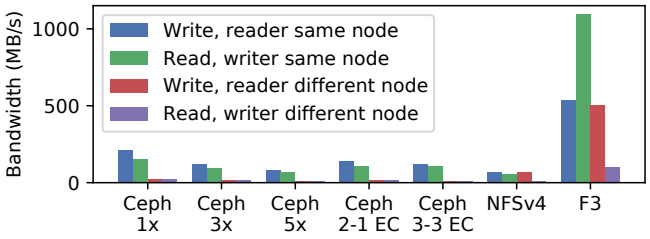

Figure 7: Comparison of read-while-write performance, when readers and writers are on the same or different nodes. For CephFS, having the reader and writer on different nodes significantly degrades both read and write performance. MinIO is absent from this experiment due to its inability to read and write data concurrently. F3 was layered on top of an unreplicated CephFS volume.

MinIO is not capable of reading from an object as it is being written to, so it is omitted from these experiments. This example further highlights the limitation of object-based interfaces.

Figure 7 shows the difference in same-node-reader vs. different-node-reader performance. As expected, for all storage systems, read performance is worse when the reader is on a separate node from the writer. However, Ceph's write performance is also lower when the reader is on a separate node. This is because when both reader and writer are on the same node, Ceph can do buffered reading and writing, as only a single client is accessing the file. When the reader and writer are on separate Ceph nodes, however, there are now multiple clients accessing the same file and Ceph falls back to its slower, unbuffered file accesses (plus the additional overhead of network transfers).

6.3 Case Study: Bioinformatics Pipeline

We developed a bioinformatics case study in collaboration with an industry partner specializing in large scale processing genetic sequence data. The advent of new genetic-sequencing technologies (*e.g.*, nanopore) has made sequencing more portable, affordable, and accessible. Sequencing can now be done anywhere from hospitals to sea-bound ships and is being used for an increasing number of applications [17].

Sequencing typically produces a large amount of data that is then processed using a series of steps run in a pipeline. The pipeline typically begins by cleaning and filtering the data, for example removing artifacts created as a byproduct of the sequencing technology. After cleaning, the sequence data is then analyzed, for example to identify the species present in a sample.

Running all or part of the analysis pipeline at the edge where the sequence data is generated can save significant time and cost associated with moving a large amount of data to the cloud. It is not always possible or desirable to run the entire processing pipeline at the edge, for instance, when the analysis requires more computing power than is available in the edge data center, or when the analysis output is required in the cloud for other reasons (*e.g.*, archival). But running at least the cleaning portion of the pipeline at the edge can still significantly reduce the amount of data uploaded to the cloud.

Because various stages in the pipeline have different resource requirements, running the pipeline in a serverless environment where each stage is run as a separate action provides better resource utilization.

Analysis pipelines are usually built using existing tools developed by other bioinformatics researchers. These tools usually assume a file interface for their inputs and outputs.

We implemented the cleaning stage of a genetic-sequencing pipeline using two commonly used tools: Cutadapt [45] and Trimmomatic [10]. Cutadapt identifies and removes portions of sequences that were added to support the sequencing process and are unrelated to the data being analyzed. Trimmomatic removes sequences that fail to meet a given quality metric. Usually, Cutadapt is run first. Its output becomes the input for Trimmomatic.

Figure 8 describes our implementation. We uploaded a 926MB file of genetic-sequence data to a load-balanced web server ❶, which wrote the file to a data store as ephemeral data. Because the web server uses a load-balancer to distribute requests among nodes, the server that receives and stores the sequence data can be any of the worker nodes in the cluster.

Once the receiving node had saved the file, it ran Cutadapt ❷ and Trimmomatic ❸ as OpenWhisk actions. We ran the pipeline in two modes: sequential and pipelined. In the sequential mode, Trimmomatic was started after the completion of Cutadapt. In pipelined mode, Trimmomatic was run at the same time as Cutadapt, operating on Cutadapt's output as it was being written. Cutadapt's output was 926MB and Trimmomatic's output was 126MB. Together, the two applications reduced the input data size by 7.3×.

Additionally, running the tools in separate actions provided better resource efficiency. The memory requirement of Trimmomatic is 1024MB, while that of Cutadapt is only 32MB. If both steps ran in the same context, then the system would have had to reserve the larger memory requirement for the duration of both pipeline stages. By scheduling them as separate actions, however, the larger memory reservation was needed only for the duration of the Trimmomatic stage. Running in separate actions is enabled by providing access to shared, file based storage.

Figure 9 shows the end-to-end runtimes of the pipeline. The pipeline ran fastest on F3, ranging from 8% to 34% faster than on other storage systems for the sequential mode, and 9% to 47% faster for the pipelined mode. Note that for the pipelined mode, MinIO results are not shown because it is incapable of being run in this mode (simultaneous reading and writing). The pipeline ran slowest on MinIO, not surprising since the pipeline writes a large amount of data during the Cutadapt stage and MinIO has the worst write performance of all evaluated storage systems.

NFS performed similarly to F3, running only 8% slower. There are two factors that contribute to this: the first is that the size of the data used in the experiment is small. This means that the time spent on I/O compared to the overall runtime is relatively small, and so improvements to that I/O time have a small impact on the larger runtime.

Second, the experiment was conducted in what are close to "ideal" conditions for NFS: only a single client and no other network traffic. This allowed the data transfers that take place during the experiment to utilize the entire network capacity. As a quick test, we used `iperf` to generate network traffic and re-ran the experiment for NFS: at 50% network utilization F3 performs 16% faster than

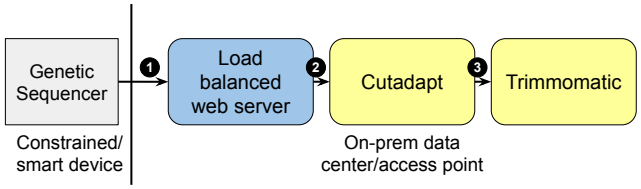

Figure 8: Bioinformatics use case architecture

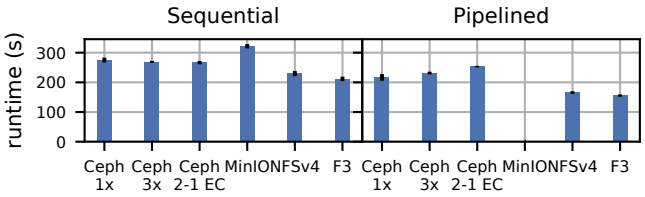

Figure 9: Runtime of Cutadapt + Trimmomatic pipeline. F3 was layered on top of an unreplicated CephFS volume.

NFS, 25% better at 75% utilization, and 59% better at 90% utilization. All networked file systems will be subject to performance variation based on the overall network utilization. F3, by using local disks and data locality scheduling, avoids this problem—performing relatively better and better as the network gets more congested.

7 RELATED WORK

Jonas *et al.* [31] implemented PyWren, which enables the massive parallelization of Python applications using AWS Lambda. This is one of the first cases of researchers using serverless platforms for use cases beyond web applications, and they found that existing storage solutions were lacking. In particular, they reported that the existing storage solutions are incapable of supporting large scale data operations. Following PyWren, Klimovic *et al.* [34] examined the storage use of several FaaS applications and proposed the design of a storage system suitable for these new use cases. Unlike F3, these works do not consider file-based storage for serverless.

Several papers introduce new storage systems for serverless platforms: Locus [54], Pocket [35], and Cloudburst [65]. Other frameworks for writing or running applications on serverless platforms handle storage by abstracting access to one of many possible storage backends. Examples include gg [21] and Ray [50]. In all of these cases, access to storage was exposed via a custom API interface which would require porting existing applications in order to run. Conversely, F3 allows existing applications to run unmodified. Also, F3 could be integrated into frameworks like gg or Ray as an alternative storage backend, or could be layered on top of one of the existing storage backends supported by those frameworks.

Schleier-Smith *et al.* [63] make a similar argument as we do in favor of a file interface for serverless applications. However, they assert that existing shared file systems are too slow and are incompatible with cloud environments where failures and high latencies are common; and they propose a transactional interface. We believe that small edge data centers will have fewer random failures and lower latency than cloud data centers, and that shared file systems can achieve high performance in this setting (see Section 6).

Wukong [12] and SONIC [44] aim to accelerate data transfer in serverless environments by scheduling connected actions together on the same node. However, they require prior knowledge of the workload, such as the graph of which actions call other actions in order to schedule actions that share data together on the same node. F3 does not require prior knowledge about workloads in order to schedule actions close to their data.

Other work has explored transferring data using direct network connections between two serverless actions, made possible via NAT hole-punching [73]. This addresses the issue of data transfer between actions but does not address the need for file-based storage.

Our location-aware data scheduling is similar to the ideas implemented by Hadoop [1] and HDFS [2]. Hadoop and HDFS are designed for map-reduce environments and fit well for data analytics tasks. It is not possible to access HDFS data through the usual `read` and `write` system calls. F3 is created specifically for serverless computing and is suitable for running generic, unmodified applications.

Apache Crail [67] makes a similar argument to us, that some intermediate data generated by applications do not need the durability provided by most storage systems. They introduce an architecture and implementation of a system that provides fast data transfer for ephemeral data. However, unlike F3, Crail exposes a custom API that requires applications to be modified to use.

In HPC environments, burst buffers such as BurstFS [72], and GekkoFS [71] accelerate access to temporary data by adding a faster, less durable storage layer between the application and the cluster's persistent data store. Unlike F3, burst buffers treat all data as ephemeral and do not provide a shared namespace with both ephemeral and non-ephemeral data.

Using non-persistent storage such as RAM for ephemeral data is common (*e.g.*, using Redis [58] or Memcached [46]). These solutions also have no shared namespace with both ephemeral and non-ephemeral data. Additionally, popular memory-based storage systems that are accessible from multiple servers all use object interfaces, rather than file interfaces.

Like F3, the Google File System (GFS) [24] has special support for the read-while-write use case. However, GFS implements a limited number of file operations, making it potentially unsuitable for running unmodified applications. Also, the special support for reading-while-writing is exposed via a new, non-standard operation called *record append*. Unmodified applications therefore cannot benefit from this new feature. In F3, even unmodified applications can benefit from our read-while-write optimizations.

8 CONCLUSION

Serverless platforms have been steadily growing in popularity. Although so far they have been limited to relatively simple web-based tasks, users and researchers are beginning to appreciate the potential of serverless platforms' on-demand computing capabilities. As serverless platforms make the shift to being a platform for any generic task, two significant problems remain: access to storage and data transfer.

Some advanced and existing applications require access to file-based storage. To support these applications, serverless platforms need to allow attaching to file-based storage systems. However, existing storage systems were not designed with serverless applications in mind and lack key features that would accelerate the kind of data transfers commonly found in serverless environments: (1) support for ephemeral data, (2) data locality-aware action scheduling, and (3) support for efficient simultaneous data access (*i.e.*, reading files as they are written).

In this paper, we presented F3, a file system that layers on top of existing storage systems to provide these three key data-transfer features. We additionally described modifications to an open source serverless platform, OpenWhisk, to enable attachment of file-based storage and take advantage of data locality hints provided by F3 when scheduling actions. We evaluated F3 and showed that it is capable of 2.0–6.5× faster write bandwidths and 1.8–2.3× better read bandwidths compared to existing storage systems. Combined with our modifications to OpenWhisk, we demonstrated that F3's data locality hints totally eliminating network traffic caused by data transfers, by enabling OpenWhisk to schedule actions on the same node as the action's data.

Future work. F3's handling of ephemeral data represents an alternate design point in the reliability-performance trade-off continuum: F3 provides higher performance at the cost of lower reliability for some data. We plan to explore additional design points. For example, some temporary files could reside entirely in RAM (much faster but lower reliability); alternatively, we could compute and store an integrity checksum or ECC with ephemeral files, which improves reliability at cost of additional computation. Eventually, users would have multiple "tiers" of reliability-performance service to choose from. We plan to couple having several such tiers with methods for automatically inferring the right tier for different date types.

In cloud settings, failures could be more common. We plan to detect action failures and offer several handling policies: return immediately to the calling application, retry the action up to N times, etc. If failed actions leave behind partial data, periodic garbage-collection would be needed.

9 ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Ram Alagappan, for their constructive feedback. We also thank fellow students Aakarsh Duvvuru, Rishabh Srivastava, and Nasratullah Sultany for their contributions. This work was made possible in part thanks to Dell-EMC, NetApp, Facebook, and IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-1729939, CNS-1900706, CCF-1918225, CNS-1951880, CNS-2106263, CNS-2106434, and CNS-2214980.

REFERENCES

[1] Apache Foundation, The. 2010. Hadoop. *http://hadoop.apache.org*.
[2] Apache Foundation, The. 2010. HDFS Architecture Guide. *https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html*.
[3] AWS Lambda 2022. AWS Lambda. *https://aws.amazon.com/lambda/*.
[4] AWS Lambda Custom Runtimes 2018. AWS Lambda Now Supports Custom Runtimes and Enables Sharing Common Code Between Functions. *https://aws.amazon.com/about-aws/whats-new/2018/11/aws-lambda-now-supports-custom-runtimes-and-layers/*.
[5] AWS Lambda EFS [n.d.]. Using Amazon EFS with Lambda. *https://docs.aws.amazon.com/lambda/latest/dg/services-efs.html*.
[6] AWS Lambda Idempotency 2021. How do I make my Lambda function idempotent? *https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/*.
[7] AWS Lambda Timeouts 2018. AWS Lambda enables functions that can run up to 15 minutes. *https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/*.

[8] AWS Serverless [n.d.]. Serverless Computing. *https://aws.amazon.com/serverless/*.

[9] Michael Behrendt. 2018. IBM Cloud Functions: we're doubling the time limit on executing actions. *https://www.ibm.com/cloud/blog/ibm-cloud-functions-doubling-time-limit-executing-actions*.

[10] Anthony Bolger, Marc Lohse, and Bjoern Usadel. 2014. Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics* 30 (Aug. 2014), 2114–2120. *https://doi.org/10.1093/bioinformatics/btu170*

[11] Zhen Cao, Vasily Tarasov, Hari Raman, Dean Hildebrand, and Erez Zadok. 2017. On the Performance Variation in Modern Storage Stacks, See [19], 329–343.

[12] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. *https://doi.org/10.1145/3419111.3421286*

[13] CephFS [n.d.]. Ceph File System. *https://docs.ceph.com/en/pacific/cephfs/index.html*.

[14] CephFS Capabilities [n.d.]. Capabilities in Ceph. *https://docs.ceph.com/en/latest/cephfs/capabilities/*.

[15] Karen Coombs. 2019. Storing data in a serverless application. *https://www.oclc.org/developer/news/2019/storing-data-in-a-serverless-application.en.htm,*.

[16] Rodrigo Crespo-Cepeda, Giuseppe Agapito, Jose Vazquez-Poletti Luis, and Mario Cannataro. 2019. Challenges and Opportunities of Amazon Serverless Lambda Services in Bioinformatics. In *10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*.

[17] Carlos de Rojas. 2022. Portable Sequencing Is Reshaping Genetics Research. *https://www.labiotech.eu/in-depth/portable-sequencing-genetics-research/*.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. *https://www.flux.utah.edu/paper/duplyakin-atc19*

[19] fast2017 2017. *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. USENIX Association, Santa Clara, CA.

[20] fio [n.d.]. Flexible I/O Tester. *https://github.com/axboe/fio*.

[21] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. *http://www.usenix.org/conference/atc19/presentation/fouladi*

[22] The Linux Foundation. 2019. State of the Edge 2021. *https://www.lfedge.org/wp-content/uploads/2021/08/StateoftheEdgeReport_2021_r3.11.pdf*.

[23] Gartner Predicts [n.d.]. Predicts 2022: The Distributed Enterprise Drives Computing to the Edge. *https://www.gartner.com/document/4007176*.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. *SIGOPS Oper. Syst. Rev.* 37, 5 (oct 2003), 29–43. *https://doi.org/10.1145/1165389.945450*

[25] Google Cloud Functions [n.d.]. Google Cloud Functions. *https://cloud.google.com/functions*.

[26] GPFS 2021. Introducing General Parallel File System. *https://www.ibm.com/docs/en/gpfs/4.1.0.4?topic=guide-introducing-general-parallel-file-system*.

[27] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 263–276. *https://www.usenix.org/conference/fast16/technical-sessions/presentation/hao*

[28] IBM Cloud Functions [n.d.]. IBM Cloud Functions. *https://cloud.ibm.com/functions*.

[29] IBM Obj vs File 2021. Object vs. File vs. Block Storage: What's the Difference? *https://www.ibm.com/cloud/blog/object-vs-file-vs-block-storage*.

[30] idempotent-azure 2022. Designing Azure Functions for identical input. *https://learn.microsoft.com/en-us/azure/azure-functions/functions-idempotent*.

[31] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, 445—451.

[32] Nikolai Joukov, Ashivay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. 2006. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*. ACM SIGOPS, Seattle, WA, 89–102.

[33] K8s CSI Drivers 2022. Drivers - Kubernetes CSI Developer Documentation. *https://kubernetes-csi.github.io/docs/drivers.html*.

[34] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. *https://www.usenix.org/conference/atc18/presentation/klimovic-serverless*

[35] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. *https://www.usenix.org/conference/osdi18/presentation/klimovic*

[36] Knative [n.d.]. Knative is an Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. *https://knative.dev/docs/*.

[37] Kubeflow 2023. Kubeflow. *https://www.kubeflow.org/*.

[38] Kubernetes Labels and Selectors 2022. Labels and Selectors. *https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/*.

[39] Kubernetes scheduler 2022. Kubernetes Scheduler. *https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/*.

[40] lambda-rerun [n.d.]. Error handling and automatic retries in AWS Lambda. *https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html*.

[41] Johanan Liebermann. 2017. Golang. *https://codeburst.io/why-golang-is-great-for-portable-apps-94cf1236f481*.

[42] Lustre [n.d.]. Welcome to the official home of the Lustre(R) filesystem. *https://www.lustre.org/*.

[43] Gilad David Maayan. 2020. Storage Options for Serverless on AWS. *https://hackernoon.com/storage-options-for-serverless-on-aws-fo3x3wsv*.

[44] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. *https://www.usenix.org/conference/atc21/presentation/mahgoub*

[45] Marcel Martin. 2011. Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet.journal* 17, 1 (May 2011), 10–12. *https://doi.org/10.14806/ej.17.1.200*

[46] Memcached 2018. Memcached. *https://memcached.org/*.

[47] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Julie Lee, Lukas Rupprecht, Dimitris Skourtis, Yang Yang, and Erez Zadok. 2021. CNSBench: A Cloud Native Storage Benchmark Native Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*. USENIX Association, Virtual.

[48] MinIO [n.d.]. MinIO. *https://min.io/*.

[49] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. 2019. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 17, 15 pages. *https://doi.org/10.1145/3302424.3303973*

[50] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. *https://www.usenix.org/conference/osdi18/presentation/moritz*

[51] Netappp AWS EFS S3 [n.d.]. EBS Pricing and Performance: A Comparison with Amazon EFS and Amazon S3. *https://cloud.netapp.com/blog/ebs-efs-amazons3-best-cloud-storage-system*.

[52] Xingzhi Niu, Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. 2019. Leveraging Serverless Computing to Improve Performance for Sequence Comparison. In *10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*.

[53] OpenWhisk [n.d.]. Open Source Serverless Cloud Platform. *https://openwhisk.apache.org/*.

[54] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. *https://www.usenix.org/conference/nsdi19/presentation/pu*

[55] PureStorage Obj vs File 2022. Object vs File Storage: When and Why to Use Them. *https://blog.purestorage.com/purely-informational/object-vs-file-storage-when-and-why-to-use-them/*.

[56] Quobyte Obj vs File [n.d.]. File Storage vs. Object Storage: What's the Difference and Why it Matters. *https://www.quobyte.com/storage-explained/file-vs-object-storage*.

[57] RADOS [n.d.]. RADOS – RADOS object storage utility. *https://docs.ceph.com/en/latest/man/8/rados/*.

[58] Redis 2023. Redis. *https://redis.io/*.

[59] RHEL FaaS 2020. What is FaaS? *https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas*.

[60] RHEL FS Choice 2020. How to Choose Your Red Hat Enterprise Linux File System. *https://access.redhat.com/articles/3129891*.

[61] Rook [n.d.]. Open-Source, Cloud-Native Storage for Kubernetes. *https://rook.io/*.

[62] s3fs [n.d.]. s3fs-fuse: FUSE-based file system backed by Amazon S3. *https://github.com/s3fs-fuse/s3fs-fuse*.

[63] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton, and Joseph M. Hellerstein. 2020. A FaaS File System for Serverless Computing. arXiv:arXiv:2009.09845

[64] S. Shepler, M. Eisler, and D. Noveck. 2010. *NFS Version 4 Minor Version 1 Protocol*. RFC 5661. Network Working Group.

[65] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: stateful functions-as-a-service. In *Proceedings of the VLDB Endowment, Volume 13, Issue 12*. 2438–2452.

[66] stackshare golang [n.d.]. Golang. *https://stackshare.io/golang*.

[67] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of Temporary Storage in the NodeKernel Architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 767–782. *https://www.usenix.org/conference/atc19/presentation/stuedi*

[68] TechRepublic Edge Market [n.d.]. Global edge computing market to reach $156 billion by 2030. *https://www.techrepublic.com/article/global-edge-computing-market/*.

[69] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. 2019. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Transactions on Storage (TOS)* 15, 2 (May 2019). *https://doi.org/10.1145/3310148*

[70] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems, See [19], 59–72.

[71] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2018. GekkoFS - A Temporary Distributed File System for HPC Applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 319–324. *https://doi.org/10.1109/CLUSTER.2018.00049*

[72] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. 2016. An Ephemeral Burst-Buffer File System for Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 69, 12 pages.

[73] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *Conference on Innovative Data Systems Research*.

[74] Joe Wigglesworth. 2022. Inside the Storage/Compute Servers of IBM Spectrum Fusion HCI. *https://hardware-fusion.blogspot.com/2022/08/inside-storagecompute-servers-of-ibm.html*.