CNSBench: A Cloud Native Storage Benchmark

Alex Merenstein
Department of Computer Science
Stony Brook University

Research Proficiency Exam

Technical report FSL-20-01

December 14, 2020

Contents

List of Figures

List of Tables

Listings

Abstract

Modern hybrid cloud infrastructures require software to be easily portable between heterogeneous clusters. Application containerization is a proven technology to provide this portability for the *functionalities* of an application. However, to ensure *performance* portability, dependable verification of a cluster's performance under realistic workloads is required. Such verification is usually achieved through benchmarking the target environment and its storage in particular, as I/O is often the slowest component in an application. Alas, existing storage benchmarks are not suitable to generate cloud native workloads as they do not generate any storage control operations (*e.g.*, volume or snapshot creation), cannot easily orchestrate a high number of simultaneously running distinct workloads, and are limited in their ability to dynamically change workload characteristics during a run.

In this report, we present the design and prototype for the first-ever Cloud Native Storage Benchmark—CNSBench. CNSBench treats control operations as first-class citizens and allows to easily combine traditional storage benchmark workloads with user-defined control operation workloads. As CNSBench is a cloud native application itself, it natively supports orchestration of different control and I/O workload combinations at scale. We built a prototype of CNSBench for Kubernetes, leveraging several existing containerized storage benchmarks for data and metadata I/O generation. We demonstrate CNSBench's usefulness with case studies of Ceph and OpenEBS, two popular storage providers for Kubernetes, uncovering and analyzing previously unknown performance characteristics.

Chapter 1

Introduction

The past two decades have witnessed an unprecedented growth of cloud computing [60]. By 2020, many businesses have opted to run a significant portion of their workloads in public clouds [48] while the number of cloud providers has multiplied, creating a broad and diverse marketplace [2, 22, 23, 30]. At the same time, it became evident that, in the foreseeable future, large enterprises will continue (i) running certain workloads on-premises (*e.g.*, due to security concerns), and (ii) employing multiple cloud vendors (*e.g.*, to increase cost-effectiveness or to avoid vendor lock-in). These *hybrid multicloud* deployments [46] offer the much needed flexibility to large organizations.

One of the main challenges when operating in a hybrid multicloud is workload portability—allowing applications to easily move between public and private clouds, and on-premises data centers [58]. Software containerization [14] and the surrounding cloud native [9] ecosystem is considered to be the enabler for providing seamless application portability [49]. For example, a container image [45] includes all user-space dependencies of an application, allowing it to be deployed on any container-enabled host while container orchestration frameworks such as Kubernetes [27] provide the necessary capabilities to manage applications across different cloud environments. Kubernetes's declarative nature [28] lets users abstract application and service requirements from the underlying site-specific resources. This allows users to move applications across different Kubernetes deployments—and therefore across clouds—without having to consider the underlying infrastructure.

An essential step for reliably moving an application from one location to another is validating its performance on the destination infrastructure. One way to perform such validation is to replicate the application on the target site and run an application-level benchmark. Though reliable, such an approach requires a custom benchmark for every application. To avoid this hassle, organizations typically resort to using component-specific benchmarks. For instance, for storage, an administrator might run a precursory I/O benchmark on the projected storage volumes.

A fundamental requirement for such a benchmark is the ability to generate realistic workloads, so that the experimental results reflect an application's actual post-move performance. However, existing storage benchmarks are inadequate to generate workloads characteristic of modern *cloud native* environments due to three main shortcomings.

First, cloud native storage workloads include a *high number of control operations*, such as volume creation, snapshotting, etc. These operations have become much more frequent in cloud native environments as users, not admins [18, 29], directly control storage for their applications. As large clusters have many users and frequent deployment cycles, the number of control operations is high [5, 57, 61].

Second, a typical containerized cluster hosts a *high number of diverse, simultaneously running workloads*. Although this workload property, to some extent, was present before in VM-based environments, containerization drives it to new levels. This is partly due to higher container density per node, fueled by the cost effectiveness of co-locating multiple tenants in a shared infrastructure and the growing popularity of

microservice architectures [63,69]. To mimic such workloads, one needs to concurrently run a large number of distinct storage benchmarks across containers and coordinate their progress, which currently involves a manual and laborious process that becomes impractical in large-scale cloud native environments.

Third, applications in cloud native environments are *highly dynamic*. They frequently start, stop, scale, failover, update, rollback, and more. This leads to various changes in workload behavior over short time periods. Although existing benchmarks allow one to configure separate runs of a benchmark to generate different phases of workloads [52,53], such benchmarks do not provide a versatile way to express dynamicity within a *single* run.

In this report we present *CNSBench*—the first open-source Cloud Native Storage Benchmark capable of (i) generating realistic control operations; (ii) orchestrating a large variety of storage workloads; and (iii) dynamically morphing the workloads as part of a benchmark run.

CNSBench incorporates a library of existing data and metadata benchmarks (*e.g.*, fio [20], Filebench [66], YCSB [51]) and allows users to extend the library with new containerized I/O generators. To create realistic control operation patterns, a user can configure CNSBench to generate different control operations following variable (over time) operation rates. CNSBench orchestrates benchmark containers and, at the end of the run, aggregates disparate results from different benchmarks and system metrics in a central location for simpler analysis and visualization.

Though CNSBench's design is generic, we implemented it for Kubernetes, the most popular container orchestrator at this time. To run a benchmark, a user only needs to define a custom `Benchmark` Kubernetes object describing the storage workloads and the resources (*e.g.*, storage volumes) to be benchmarked.

To demonstrate CNSBench's versatility, we conducted a study comparing cloud native storage providers. We pose three questions in our evaluation: (A) How fast are different cloud storage solutions under common control operations? (B) How do control operations impact the performance of user applications? (C) How do different workloads perform when run alongside other workloads? We use Ceph [6] and OpenEBS [34] in our case study as sample storage providers. Our results show that control operations can vary significantly between storage providers (resulting in *e.g.*, up to 8.5× higher Pod creation rates) and that they can slow down I/O workloads by up to 38%.

In summary, this report makes the following contributions:

1. We identify the need and unique requirements for cloud native storage benchmarking.
2. We present the design and implementation of CNSBench, a benchmark that fulfills the above requirements and allows users to conveniently benchmark cloud native storage solutions with realistic workloads at scale.
3. We use CNSBench to study the performance of two storage solutions for Kubernetes (Ceph and OpenEBS) under previously not studied workloads.

CNSBench is open-source and available for download from *https://github.com/CNSBench/CNSBench*.

Chapter 2

Kubernetes Background

We implemented our benchmark for Kubernetes. In the next sections we use many Kubernetes concepts to contextualize CNSBench's design and use cases. Therefore, we begin with a brief background on how Kubernetes operates.

Overview A basic Kubernetes cluster is shown in Figure 2.1. It consists of control plane nodes, worker nodes, and a storage provider (among other components). Worker and control plane nodes run *Pods*, the smallest unit of workload in Kubernetes that consist of one or more *containers*. User workloads run on the worker nodes, whereas core Kubernetes components run on the control plane nodes. Core components include (1) the API server, which manages the state of the Kubernetes cluster and exposes HTTP endpoints for accessing the state, and (2) the scheduler, which assigns Pods to nodes. Typically, a Kubernetes cluster has multiple worker nodes and may also have multiple control plane nodes for high availability.

The storage provider is responsible for provisioning persistent storage in the form of *volumes* as required by individual Pods. There are many architectures, but the "hyperconverged" model is common in cloud environments. In this model, the storage provider aggregates the storage attached to each worker node into a single storage pool.

The state of a Kubernetes cluster, such as what workloads are running on what hosts, is tracked using different kinds of *resources*. A resource consists of a *desired state* (also referred to as its specification) and a *current state*. It is the job of that resource's *controllers* to reconcile a resource's current and desired states, for example, starting a Pod on node X if its desired state is "running on Node X". Pods and Nodes are examples of resources.

Persistent Storage Persistent block and file system storage in Kubernetes is represented by resources called *Persistent Volumes* (PVs). Access to a PV is requested by attaching the Pod to a resource called a *Persistent Volume Claim* (PVC). Figure 2.1 depicts this process: ❶ A Pod that requires storage creates a PVC, specifying how much storage space it requires and which storage provider the PV should be provisioned from. ❷ If there are no PVs that are at least as large as the amount of storage the Pod has requested, then ❸ a new PV is provisioned from the storage provider specified in the PVC. A PVC specifies what storage provider to use by referring to a particular *Storage Class*. This class is a Kubernetes resource that combines a storage provider with a set of configuration options. Examples of common configuration options are what file system to format the PV with and whether the PV should be replicated across different nodes.

Once the PV has been provisioned, ❹ Kubernetes binds the PV to the PVC, and ❺ the volume is mounted into the Pod's file system. If a PV satisfying the Pod's request already exists when the PVC is created, then steps ❷ and ❸ are skipped.

Kubernetes typically communicates with the storage provider using the *Container Storage Interface*

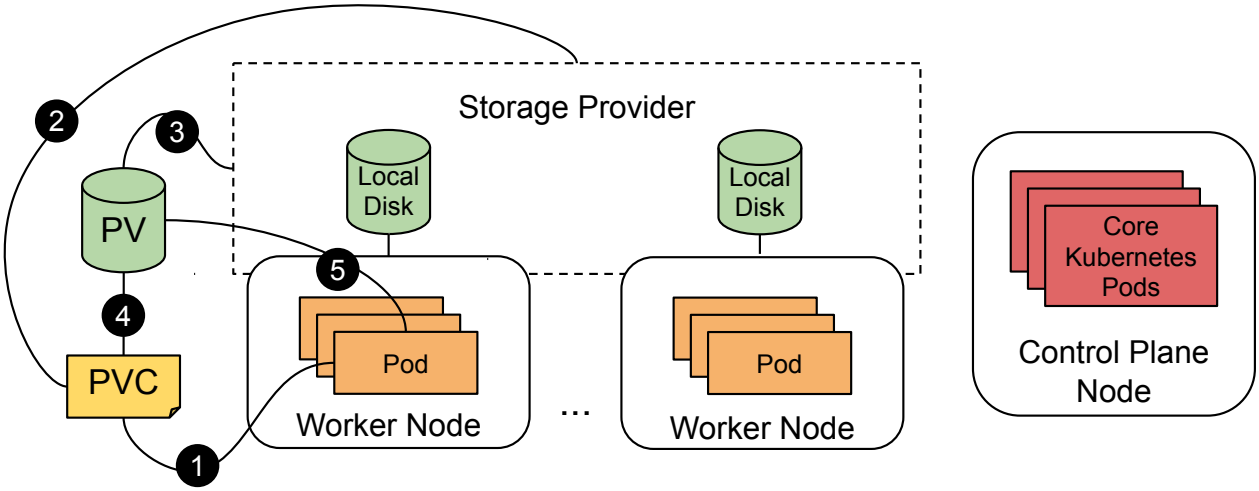

Figure 2.1: Basic topology of a Kubernetes cluster, with a single control plane node, multiple worker nodes, and a storage provider which aggregates local storage attached to each worker node. Also shows the operations and resources involved in providing a Pod with storage.

(CSI) specification [13], which defines a standard set of functions for actions such as provisioning a volume and attaching a volume to a Pod. Before CSI, Kubernetes had to be modified to add support for individual storage providers. By standardizing this interface, a new storage provider needs only to write a CSI driver according to a well-defined API, to be used in any container orchetrator supporting CSI (*e.g.*, Kubernetes, Mesos, Cloud Foundry).

Object Storage Object stores such as Amazon's S3 [1] are another form of persistent storage that are often used because they are more flexible and scalable than block or file system storage. Individual assets such as images are stored as an objects, which are organized into "buckets" and accessed via an API. Currently, there are no Kubernetes objects that represent object stores and no standardized interface for accessing them (*i.e.*, there are no object store equivalents to Persistent Volumes and CSI). Therefore, CNSBench does not support benchmarking object stores. There is some ongoing work to create a CSI equivalent for object stores [12], so we may revisit this decision in the future.

Chapter 3

Need for Cloud Native Storage Benchmarking

In this section we begin with describing the properties of cloud native workloads (§ 3.1), which current storage benchmarks cannot recreate. We then present the design requirements for a cloud native storage benchmark (§ 3.2).

3.1 New Workload Properties

The rise of containerized cloud native applications has created a shift in workload patterns which makes today's environments different from previous generations. This is particularly true for storage workloads due to three main reasons: (i) the increased frequency of control operations; (ii) the high diversity of individual workloads; and (iii) the dynamicity of these workloads.

Control Operations Previously infrequent, control operations became significantly more common in self-service cloud native environments. As an example, consider the frequent creations and deletions of containers in a cloud native environment. In many cases, these containers require persistent storage in the form of a storage volume and hence, several control operations need to be executed: the volume needs to be created, prepared for use (*e.g.*, formatted with a file system), attached to the host where the container will run (*e.g.*, via iSCSI), and finally mounted in the container's file system namespace. Even if a container only needs to access a volume that already exists, there are still at least two operations that must be executed to attach the volume to the node where the container will run and mount the volume into the container.

To get a better idea of how many control operations can be executed in a cloud native environment, consider these statistics from one container cluster vendor: in 2019 they observed that over half of the containers running on their platform had a lifetime of no more than five minutes [43]. In addition, they found that each of their hosts were running a median of 30 containers. Given these numbers, a modestly sized cluster of 20 nodes would have a new container being created every second on average. Although not all of these containers will require access to a storage volume, many will, which results in frequent invocation of control operations.

In addition to being abundant, control operations, depending on the underlying storage technology, can also be data intensive. This makes them slow and increases their impact on the I/O path of running applications. For example, volume creation often requires (i) time-consuming file system formatting; (ii) snapshot creation or deletion, which, depending on storage design, may consume a significant amount of I/O traffic; (iii) volume resizing, which may require data migration and updates to many metadata structures; and (iv) volume reattachment, which causes cache flushes and warmups.

Now that data-intensive control operations are more common, there is a new importance to understanding their performance characteristics. In particular, there are two categories of performance characteristic that are important to understand: (1) How long does it take a storage provider to execute a particular control operation? This is important because in many cases, control operations sit on the critical path of the container startup. (2) What impact does the execution have on I/O workloads? This impact can be significant either due to the increased load on the storage provider or particular design of the storage provider. For example, some storage providers freeze I/O operations during a volume snapshot, which can lead to a spike in latency for I/O operations [39].

Existing storage benchmarks and traces, focus solely on data and metadata operations, turning a blind eye to control operations.

Diversity and Specialization The lightweight nature of containers allows many different workloads to share a single server or a cluster [43]. Workload diversity is fueled by a variety of factors. First, projects such as Docker [15] and Kubernetes [27] have made containerization and cloud native computing more accessible to a wide range of users and organizations, which is apparent in the diversity of applications present in public repositories. For example, on Docker Hub [16] there are container images for fields such as bioinformatics, data science, high-performance computing, and machine learning—in addition to the more traditional cloud applications such as web servers and databases. Additionally, the popularity of microservice architectures has caused traditionally monolithic applications to be split up into many small, specialized components [69]. Finally, the increasingly popular serverless architecture [4], where functions run in dynamically created containers, takes workload specialization even further through an even finer-grained split of application components, each with their own workload characteristics.

The result of these factors is that the workloads running in a typical shared cluster (and on each of its individual hosts) have a highly diverse set of characteristics in terms of runtime, I/O patterns, and resource usage. Understanding system performance in such an environment requires benchmarks that recreate the properties of cloud native workloads. Currently, such benchmarks do not exist. Hence, realistic workload generation is possible only by manual selection, creation, and deployment of several appropriate containers (*e.g.*, running multiple individual storage benchmarks that each mimic the characteristics of a single workload). As more applications of all kinds adopt containerization and are broken into sets of specialized microservices, the number of containers that must be selected to make up a realistic workload continues to increase. Making this selection manually has become infeasible in today's cloud native environments.

Elasticity and Dynamicity Cloud native applications are usually designed to be elastic and agile. They automatically scale to meet user demands, gracefully handle failed components, and developers frequently deploy new versions of applications and their components. Although some degree of elasticity and dynamicity has always been a trait of cloud applications, the cloud native approach takes it to another level.

In one example, when a company adopted cloud native practices for building and operating their applications, their deployment rate increased from rolling out a new version 2–3 times per week to over 150 times in a single day [21]. Other examples include companies utilizing cloud native architectures to achieve rapid scalability in order to meet spikes in demand, for example in response to breaking news [32] or the opening of markets [3].

Currently, benchmarks lack the capability to easily evaluate application performance under these highly dynamic conditions. In some cases benchmark users resort to creating these conditions manually to evaluate how applications will respond—for example manually scaling the number of database instances [51]. However, the high degree of dynamicity and diversity found in cloud native environments makes recreating these conditions manually nearly impossible.

3.2 Design Requirements

The fundamental functionality gap in current storage benchmarks is their inability to generate control-operation workloads representative of cloud native environments. At the same time, the I/O workload (data and metadata, not control operations) remains an important component of cloud native workloads, and is more diverse and dynamic than before. Therefore, the primary goal for a cloud native storage benchmark is to enable combining control-operation workloads and I/O workloads—to better evaluate application and cluster performance. This goal led us to define the following five core requirements:

1. I/O workloads should be specified and created independently from control workloads, to allow benchmarking (i) an I/O workload's performance under different control workloads and (ii) a control workload's performance with different I/O workloads.
2. It should be possible to orchestrate I/O and control workloads to emulate a dynamic environment that is representative of clouds today. In addition, it should be possible to generate control workloads that serve as microbenchmarks for evaluating the performance of individual control operations.
3. I/O workloads should be generated by running existing tools or applications, either synthetic workload generators like Filebench or real applications such as a web server with a traffic generator.
4. It should be possible for users to quickly configure and run benchmarks, without sacrificing the customizability offered to more advanced users.
5. The benchmark should be able to aggregate unstructured output from diverse benchmarks in a single, convenient location for further analyses.

A benchmark which meets these requirements will allow a user to understand the performance characteristics of their application and their cluster under realistic cloud native conditions.

Chapter 4

CNSBench Design and Implementation

To address the current gap in benchmarking capabilities in cloud native storage, we have implemented the *Cloud Native Storage Benchmark*—CNSBench. Next, we describe CNSBench's design and implementation. We first overview its architecture and then describe the new Kubernetes *Benchmark* custom resource and its corresponding controller in more detail.

Overview In Kubernetes, a user creates Pods (one of Kubernetes' core resources) by specifying the Pod's configuration in a YAML file and passing that file to the `kubectl` command line utility. Similarly, we want CNSBench users to launch new instances by specifying CNSBench's configuration in a YAML file and passing that file to `kubectl`. To achieve that, our CNSBench implementation follows the *operator design pattern*, which is a standard mechanism for introducing new functionality into a Kubernetes cluster [37]. In this pattern, a developer defines an *Operator* that comprises a custom object and a controller for that object. For our implementation of CNSBench, we defined a custom *Benchmark* object and implemented a corresponding *Benchmark Controller*. Together, these two components form the *CNSBench Operator*. The Benchmark object specifies the I/O and control workloads, which the controller is then responsible for running.

Figure 4.1 shows the Kubernetes cluster depicted in Figure 2.1 with added CNSBench components shown in blue. The overall control flow is as follow: Ⓐ The Benchmark controller watches the API server for the creation of new Benchmark objects. Ⓑ When a new Benchmark object is created, the controller creates the objects described in the Benchmark's I/O workload: the *I/O Workload Pods* for running the workloads and the Persistent Volume Claims (PVCs) for the Persistent Volumes (PVs) against which the workloads are run. Ⓒ For running the control operation workload, the Benchmark includes a *Rate Generator*, which triggers an *Action Executor* in user-specified intervals to invoke the desired control operations (*actions*).

4.1 Benchmark Custom Resource

The custom Benchmark custom resource lets users specify three main benchmark properties: (1) the control operation workload; (2) the I/O workloads to run; and (3) where the output should be sent for collection.

Control operation workload One of CNSBench's primary requirements is the ability to create realistic control workloads. However, microbenchmarks that purposefully stress only one component or operation of a system are also valuable (*e.g.*, for an in-depth analysis and point optimization of system performance). Useful insights can be derived, for instance, from a benchmark that executes some control operation at a regular interval. Our control workload specification satisfies both use cases, by making it easy to create simple control workloads without sacrificing the ability to define realistic ones.

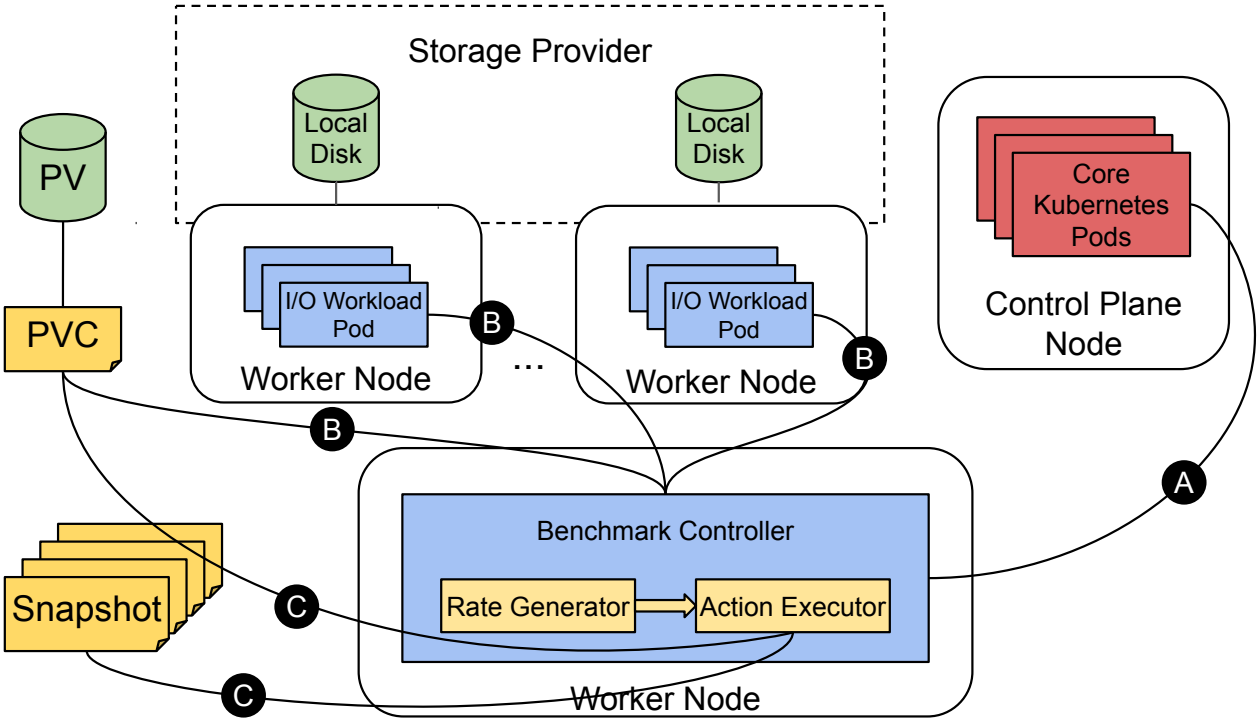

Figure 4.1: CNSBench overview with its components in blue

In CNSBench, control workloads are specified using a combination of *actions* and *rates*. Actions execute operations, for instance *create resource* (*e.g.*, *create Pod or Volume*), *delete resource* (*e.g.*, *delete snapshot*), *snapshot volume*, and *scale resource* (*e.g.*, *scale database deployment*). Rates trigger associated actions at some interval. For our evaluations we used a simple rate which runs actions every T seconds, but more sophisticated rates could be implemented to enable the creation of more realistic control workloads. For example, given a set of cluster traces that logged when different operations were executed, a rate could be implemented that reads those traces and generates a control workload mimicking their specific operating conditions. Actions and rates are deliberately decoupled, so that these more sophisticated rates can be developed independently from CNSBench and then plugged in later.

I/O workload Often, a benchmark's goal is to understand how a particular workload or set of workloads will perform under various conditions. The role of CNSBench's I/O workload component is to either instantiate those workloads or to instantiate a synthetic workload with the same I/O characteristics of a real workload. Specifying these I/O workloads requires defining all of the different resources (*e.g.*, Pods and PVCs) that must be created in order to run the I/O workload. This can be difficult and make benchmark specifications long and complex.

To ease the burden on users and to help them focus on the overall benchmark specification, rather than the specific details of the I/O workload, CNSBench separates the I/O workload specification from the rest of the benchmark specification. The I/O workload specification is defined using a *ConfigMap*—a core Kubernetes resource for storing configuration files and other free-form text. These files contain the specifications for the Pods that will run the I/O workloads, as well as specifications for supporting resources such as PVCs. In addition, they use metadata annotations to specify information such as what output files should be collected and what parsers should be used to process them. Since the specification uses a core Kubernetes resource, it can be accessed using standard Kubernetes tools from anywhere in the cluster.

Users specify which I/O workloads to run in a Benchmark custom resource using a *create resource* action that references (by name) the I/O workload to create. To enable reuse across various use cases and benchmarks, fields in an I/O workload specification can be parameterized and given a value when the workload is instantiated by a specific benchmark.

We are building an open source workload library, available at *https://github.com/CNSBench/workload-library/*, to which community members can contribute their I/O workloads. This library will be a collection of containers and workload specifications that can be used for benchmarking. Ideally, most users will be able to find a suitable I/O workload in the library and not need to define their own. Our initial contributions to the workload library will focus on commonly used benchmarks based on curated benchmark lists [19], open source benchmarking websites [33], and recent storage related conferences [54].

Benchmark output Many of the results of a CNSBench benchmark will be generated by the I/O workload Pods. Collecting this output presents three challenges. First, Kubernetes currently lacks the ability to extract files from Pods in a clean and generic manner [24]. Second, the output produced by some tools can be large, especially for long-running processes that produce output throughout the run. Third, in our experience, many I/O workloads produce output as unstructured text. This can make it difficult to analyze the results using tools such as Kibana [25], especially if the benchmark consists of multiple I/O workloads that all report results in a different unstructured output formats.

To address these issues, we allow I/O workload authors to specify which files should be collected from the workload Pods and to provide a parser script to process the output. Parsing the output allows large files to be reduced to a more succinct size and to output results in a standard fashion. The output files are collected and parsed using a helper container, described in more detail in Section 4.2. Parsers for common I/O benchmarking tools can be included in the Workload Library, either packaged with the tool's workload specification or as a standalone entry. For instance, we include parsers for fio and YCSB in the Workload Library.

The user specifies where the final, parsed results should be sent to in the *output* section of the Benchmark custom resource. Currently CNSBench supports sending the results to a collection server only via an HTTP POST request to a user-specified URL. Support for additional kinds of output, such as simply writing the output to a file, can be easily added.

Example An example Benchmark custom resource is shown in Listing 4.1 and an example of an I/O workload specification is shown in Listing 4.2. Due to space constraints, many of the details of the I/O workload specification are omitted. Figure 4.2 shows the Kubernetes resources that are created as a result of this Benchmark specification.

Lines 6–13 of Listing 4.1 specify the benchmark's I/O workload. Line 8 references the name of the I/O workload that should be run, labeled Ⓐ in both listings. Lines 6–11 of Listing 4.2 specify the resources that make up the I/O workload. These correspond to the Pods and PVCs in Figure 4.2 labeled Ⓑ.

I/O workload specifications can be parameterized to enable their reuse across different use cases and benchmarks. An example of this is on line 10 of Listing 4.2, where the PVC's Storage Class field is parameterized. Label Ⓒ in the two listings and in Figure 4.2 shows how this parameter is set in the Benchmark custom resource specification (line 11 in Listing 4.1), and then how that value is used in the workload's PVCs.

Lines 14-18 of Listing 4.1 specify a *snapshot volume* action. In Kubernetes, volume snapshots are created using a *Snapshot* resource which references a PVC to use as the source of the snapshot. The user indicates which action's PVCs should be snapshotted by referencing the target action by name (line 17 of Listing 4.1). Since all resources created by an action are labeled with that action's name, the controller can map an action name to a set of PVCs (label Ⓓ). These PVCs are then used as the source in the Snapshot resource (label

```
kind: Benchmark
metadata:
  name: fio-benchmark
spec:
  actions:
    - name: fio (D)
      createObjSpec:
        workload: fio (A)
        count: 3
        vars:
          storageClass: obs-r1 (C)
      outputs:
        outputName: es
    - name: snapshots
      rateName: minuteRate
      snapshotSpec:
        actionName: fio (D)
        snapshotClass: obs-csi
  rates:
    - name: minuteRate
      constantRateSpec:
        interval: 60s
  outputs:
    - name: es
      httpPostSpec:
        url: http://es:9200/fio/_doc/
```

Listing 4.1: Sample Benchmark Custom Resource Specification

```
kind: ConfigMap
metadata:
  name: fio (A)
spec:
  data:
    pod.yaml: | (B)
      ...
    pvc.yaml: | (B)
      ...
      storageClass: {{storageClass}} (C)
      ...
```

Listing 4.2: Sample I/O workload specification

(E)).

4.2 Benchmark Controller

The Benchmark Controller watches for newly created Benchmark objects and runs their specified actions. The controller has three main responsibilities: (1) triggering control operations; (2) synchronizing the individual benchmark workloads; and (3) collecting the output of the individual workloads.

Triggering control operations When a new Benchmark resource is created, the Controller starts two *goroutines* (Go's equivalent of a thread) for each of the specified rates: one is responsible for generating

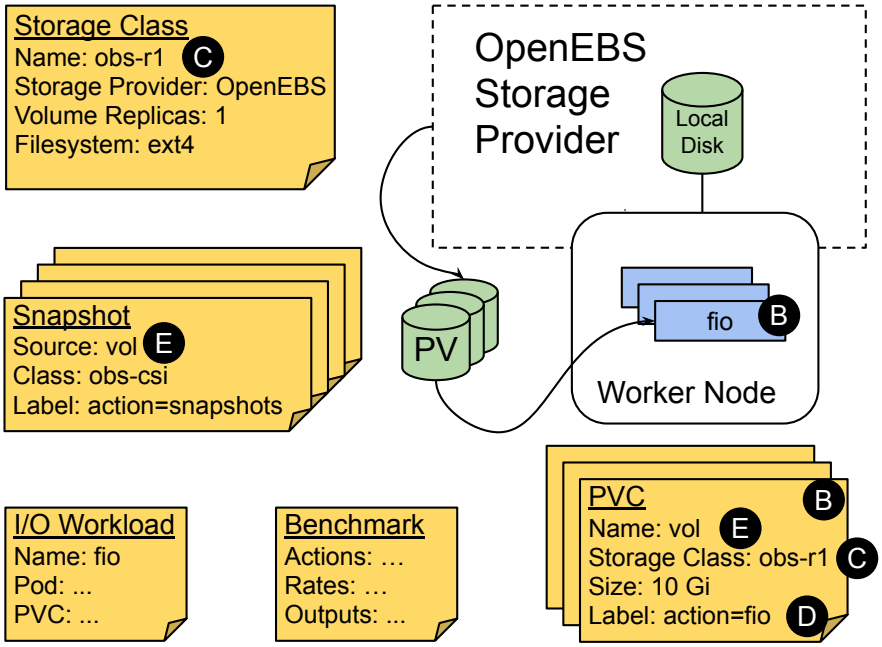

Figure 4.2: Subset of a Kubernetes cluster with a single worker node and a PV. Shows the CNSBench resources that are involved (the I/O Workload and Benchmark), as well as the core Kubernetes resources created by the CNSBench controller according to the Benchmark specification (the Snapshots, PVCs, PV, and workload Pods).

the rate, and the other is responsible for running all of the actions using that rate. The rate goroutine uses a shared channel to tell the executor goroutine when it is time to run an action. As described in Section 4.1, decoupling the rates from the actions simplifies adding new kinds of rates or actions later.

Actions not tied to any rate are run by the controller as soon as the Benchmark resource is created. This is often how I/O workloads are instantiated, since they often use a long running process that generates I/O throughout the benchmark's duration.

Synchronizing workloads In many cases, I/O workloads require an initialization step such as loading data into a database or creating a working set of files. When there are multiple I/O workloads being run, some workloads can finish their initialization step faster than others and begin running their main workload earlier. This can cause misleading and inconsistent results. If the purpose of the benchmark is to evaluate a storage provider's performance under the concurrent load of ten read-heavy I/O workloads, then all ten should start at the same time.

To synchronize the I/O workloads, CNSBench leverages Kubernetes' *initialization containers* feature. Pods have a list of initialization containers which are executed in order, each one running to completion before the next one starts. The Pod's main containers do not run until all of the initialization containers have completed. CNSBench assumes that a workload's initialization step has been put into an initialization container, which is the responsibility of the I/O workload's author. Although this is usually a straightforward task, it is an example of why separating the I/O workload specifications from the rest of the Benchmark specifications is useful: it allows users to select existing workloads from the Workload Library and not worry about how their workload's initialization is implemented.

When the Benchmark controller instantiates the I/O workloads, it adds an additional *synchronization container* at the end of the list of initialization containers. This container runs a script that queries the

Kubernetes API server for the status of each instance of the I/O workload and checks to see if all of their initialization containers have completed (all except for the other synchronization containers). Once all of the non-synchronization initialization containers have completed, the script exits and the synchronization containers stop successfully, allowing Kubernetes to run each Pods' main container. Since all instances of the I/O workload have this synchronization container added, all instances begin running their main containers simultaneously.

Output collection As described in Section 4.1, I/O workload authors can specify which files to extract from a workload's Pods and provide a script to parse those files. Extracting these files from the workload Pods is difficult since there is no standard interface for doing so [24]. The approach used by the official Kubernetes command-line client `kubectl` involves running the `tar` utility inside the target container, and does not work after the container has finished running [26]. This means that output collection must be done while the workload runs, since there is no opportunity to collect the output afterwards.

To extract these files and run the parser, the controller adds an additional helper container to each workload Pod. This container is configured to share a process namespace with the workload container, so the workload process's file system is accessible via Linux's `/proc/$pid/root/` interface. The helper container uses `tail -f` to capture the contents of the workload's output file as it is written, since the file will be inaccessible after the workload completes. After the workload process completes, the parser script runs in the helper container and outputs the parsed results. The controller then reads the output from the helper container using the client-go [8] library.

Chapter 5

Evaluation

To demonstrate both the need for and the utility of CNSBench, we ran several benchmarks to look at different aspects of cloud native storage performance. We examine the performance of individual control operations, the impact that control operations have on I/O workloads, and the impact that different combinations of I/O workloads can have on overall performance.

5.1 Methodology

To evaluate our benchmark, we instantiated an 11-node Kubernetes cluster in an on-premises OpenStack cluster: one control plane node and 10 workers. Each worker node is a virtual machine with 4 vCPUs, 8GB of RAM, and 384GB of locally attached storage. The control plane node is a VM with 4 vCPUs, 12GB of RAM, and 100GB of local storage. The VM hosts were located in multiple racks, with racks connected via a 10Gbps network and individual hosts connected to the top of rack switch via 1Gbps links.

We used two storage providers: OpenEBS and Ceph.

OpenEBS [34] is one of a new storage providers built specifically to be cloud native. OpenEBS uses the *Container Attached Storage* paradigm [10], where controllers that provision volumes and manage features such as data replication, themselves run in containers. This provides storage with all of the advantages of the cloud native methodology, such as agility and flexibility. It also enables the storage to be managed like any other resource in a cloud native cluster.

Ceph [71] is a widely used file storage system that is built on top of the RADOS object store [72]. We used the Rook operator for Ceph [41], which handles the deployment and management of a Ceph cluster. The Rook management layer allows Ceph to be managed in a cloud native fashion, using Kubernetes objects and standard Kubernetes management tools.

Both Ceph and OpenEBS provide storage by aggregating the local storage attached to each cluster node. Volumes are provisioned from this combined storage pool and are formatted with Ext4 prior to being attached to a Pod. Ceph and OpenEBS both come with CSI drivers that interface with Kubernetes.

Both OpenEBS and Ceph also offer volume replication for high availability use cases. With volume replication, data written to a volume by a Pod is transparently copied across several volume replicas, which are ideally situated in different availability zones. This enables the cluster to tolerate the loss of one or more hosts—depending on the replication factor—without suffering any data loss. The trade-off is that volume replication often comes at a cost of increased I/O latencies and an increase in network and disk utilization.

Ceph has an additional high availability mechanism using erasure coding, which encodes data into chunks using a forward error-correction code and then replicates those chunks. The use of a forward error-correction code means that fewer replicas are needed to provide the same availability guarantees, and hence

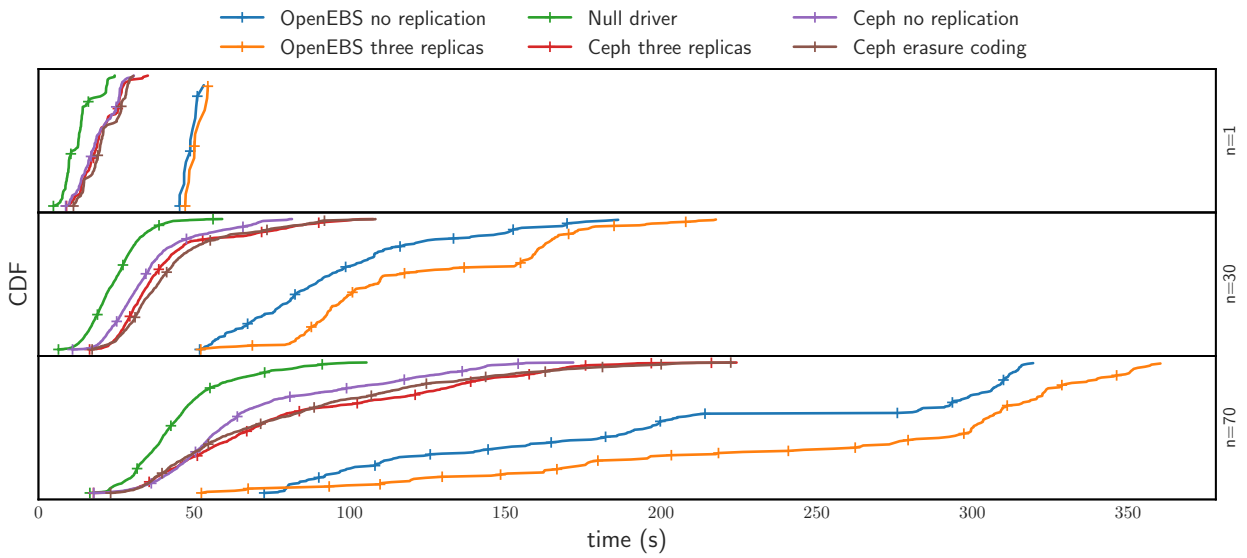

Figure 5.1: CDFs of time required to create and attach volumes for different storage provider configurations. n is the number of simultaneous volume creations. For all storage configurations, increasing the number of simultaneous volume operations increased the average time to create and attach an individual volume.

less disk space is needed overall. However, erasure coding uses more CPU and RAM than basic data replication.

In our experiments, we use Ceph and OpenEBS in three ways: without replication, in triple-replication mode, and Ceph (only) in erasure-coded mode (ec). In addition to Ceph and OpenEBS, in some evaluations we used a null storage provider that implements the CSI functions involved in provisioning and attaching volumes. The null driver simply returns success to most CSI functions without performing actual work. The null driver does, however, maintain a list of provisioned volumes so the *ListVolumes* CSI function returns an accurate result. We use the null driver as a baseline to show the maximum possible performance of the underlying Kubernetes cluster.

Each evaluation was conducted five times and unless otherwise noted has a standard deviation of less than 20%.

5.2 Performance of Control Operations

In Section 3.1 we described the importance of control operations in cloud native workflows. In this section, we demonstrate how the performance of these operations can vary across different storage providers and configurations. We looked at two common storage control operations: volume provisioning and attaching.

Our goal was to time how long it took each storage provider configuration to provision a volume and attach that volume to a Pod. To do so, we timed how long it took to create and run new Pods that were attached to volumes. The time to create and run a Pod with an attached volume includes the time taken by the storage provider to provision and then attach that volume. Any additional overhead related to running the Pod is constant across storage configurations.

We ran this test with 1, 10, 20, 30, 40, 50, 60, and 70 parallel Pod creations. Each test ran for five minutes, where we maintained a fixed parallelism level N by starting a new Pod whenever one Pod was created; there were always N Pods in the process of being created. The workload run by each Pod simply exited immediately, so Pods finished running as soon as they started.

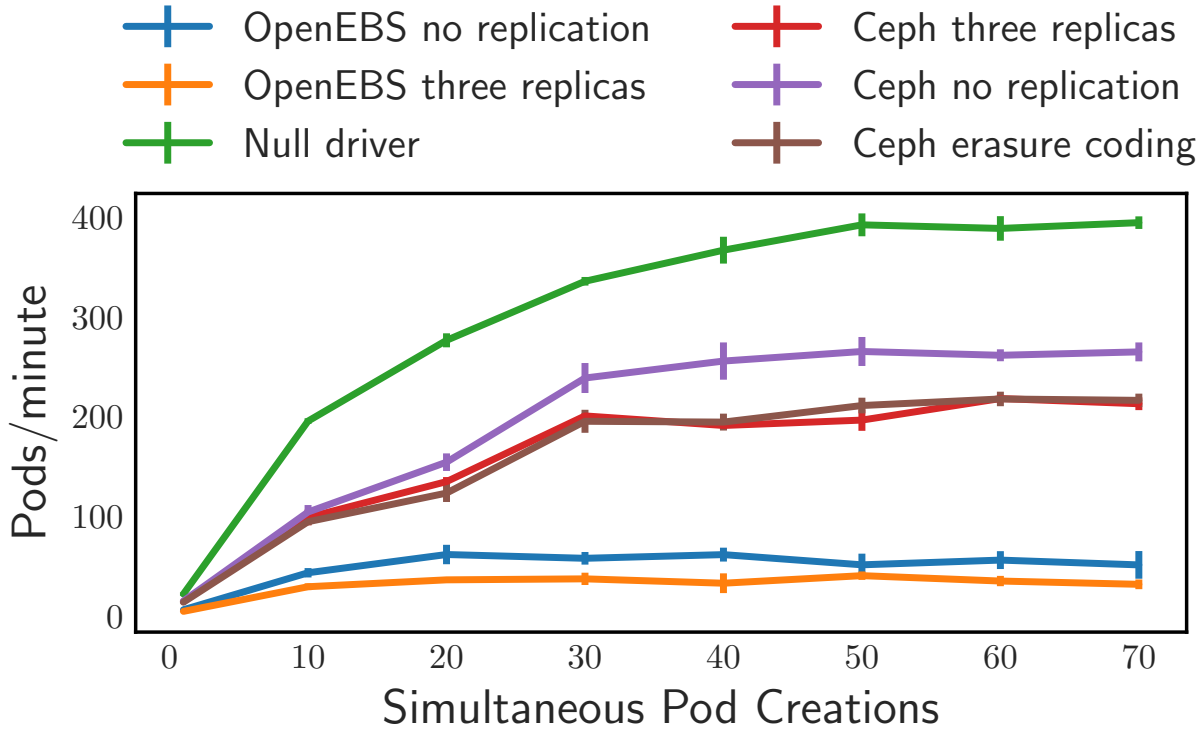

Figure 5.2: Volume creations and attachments per minute, for different numbers of simultaneous operations. The vertical lines at each point shows the standard deviation for volume creation and attachment rate at that point.

We repeated each run five times. Figure 5.1 shows CDFs for Pod start time across all of the Pods created during each of the five runs, for six storage provider configurations. We show CDFs only for three degrees of parallelism (1, 30, and 70) because the CDFs for the intermediate parallelism values follow the trends that are visible from these three. Figure 5.2 shows the overall volume creation and attachment rate per minute for different parallelism levels. These rates are averaged across each of the five runs and had a standard deviation under 11% of the mean, except for OpenEBS which had standard deviations of up to 30% of the mean. This higher standard deviation can be attributed to the polling architecture which is used throughout Kubernetes and OpenEBS [35], which causes some actions to take sometimes significantly different amounts of time depending on which side of the poll the resource becomes available.

As expected, Pod creation is fastest with the null storage provider. The storage provider configurations with no replication are slightly faster than their replicated counterparts. This is also expected, since volumes with replication require additional resources to be allocated during provisioning.

As the number of simultaneous Pod creations increases, we noticed that subsets of Pods took an increasingly long time to start (see Figure 5.2). Eventually, each of the six storage configurations reached a point where its Pod creation rate plateaus. Note that Pod creation goes through three states: initially it is in a "Pending" state before it can be assigned to a Node. Once the Kubernetes scheduler has assigned the Pod a Node to run on, it moves it to a "Creating" state where container images are downloaded and volumes are mounted. Then, the Pod enters the "Running" state.

As an initial investigation, we counted how many Pods were in each state to identify the bottleneck. We observed that for the null storage provider and the three Ceph configurations, the rate that Pods moved from "Pending" to "Creating" and then from "Creating" to "Running" equalizes when the number of simultaneous

Pod creations reaches around 50. At this point, increasing the number of simultaneous Pod creations only increased the number of Pods in the "Pending" state, and did not increase the overall Pod creation rate.

The situation is different for the two OpenEBS configurations. As shown in Figure 5.2, these configurations plateau at a lower rate of around 30 simultaneous Pod creations. When observing the Pod transitions for these configurations, we saw that the rate at which Pods moved from "Creating" to "Running" was low compared to the rate that Pods moved from "Pending" to "Creating" resulting in all Pods being in either "Creating" or "Running" states throughout the test. The Pods in the "Creating" state were all waiting for OpenEBS to finish provisioning and attaching a volume for the Pod. So, increasing the number of simultaneous Pod creations did not increase the overall Pod creation rate, since that rate was limited by how fast OpenEBS was able to provision and attach volumes.

From these experiments we see that although all three storage providers have scalability limits in terms of how many simultaneous Pod creations they support, the source of their limits appear to be different. Whereas the null storage provider and Ceph are limited by the scheduling stage of Pod creation, OpenEBS is limited by its own volume creation and attachment rate.

Overall, the experiment shows that there can be significant differences in the performance of control operations across different storage providers and configurations. This highlights the need to systematically benchmark these kinds of operations to understand their bottlenecks and improve upon them.

5.3 Impacts on I/O Workloads

In this section, we demonstrate the impact that control operations, in particular snapshotting a volume, can have on the I/O workload that uses the volume. As described in Section 3.1, control operations are executed far more often in cloud native environments than they are elsewhere. Snapshotting is especially common and users take frequent snapshots of their volumes for a number of reasons: periodically, during a long running task to checkpoint progress, prior to making some significant change so rollback to a known good point is possible, or to protect themselves against attacks such as ransomware.

Although previously these operations were executed too infrequently to have a noticeable effect on an I/O workload, this is no longer guaranteed to be the case in cloud native environments. Due to differences in the design and architecture of different storage providers, the degree to which these control operations impact an I/O workload can vary significantly.

To evaluate the impact of snapshotting operations, we used CNSBench to run three instances of MongoDB [31] with ten clients each. The clients ran YCSB Workload A [51] (consisting of a mix of reads and updates) for twenty minutes to reach steady state; the volumes holding the MongoDB databases were snapshotted every thirty seconds.

Figure 5.3 shows the per-client throughput in terms of operations per second for five storage provider configurations, with and without snapshotting. The throughput values are averaged across all thirty YCSB clients.

Overall the results show that snapshotting reduces the throughput across all configurations. The decrease in throughput is more noticeable for OpenEBS (27% and 38% for zero and three volume replica configurations, respectively) than for Ceph (up to 22% for three volume replicas but as low as 5% and 6% for erasure coding and zero replication configurations, respectively). We found that although the average throughput decreased with snapshotting across all OpenEBS YCSB clients, the decrease was more pronounced for some clients than others. For those clients, we observed that the maximum latency reported by YCSB was much higher than the average maximum latency. In addition, these clients reported extended periods (30+ seconds) when zero operations were executed.

One possible explanation is the fact that OpenEBS quiesces and suspends I/O while a snapshot operation is in progress [36]. During that time, any writes issued by Mongo cannot complete. Some of these periods of

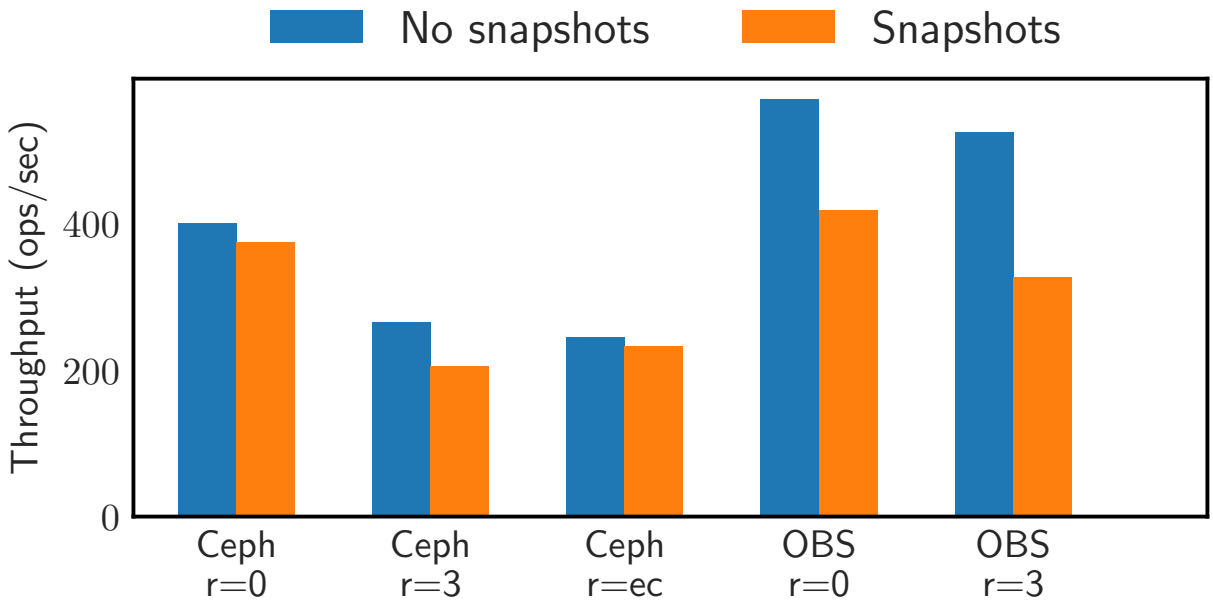

Figure 5.3: Effect of snapshotting on I/O workload. r=0 indicates zero volume replicas, r=3 indicates three volume replicas, and r=ec indicates erasure coding.

suspended I/O lasted several seconds, which could explain the periods when no operations could be executed by the clients and the reduction in overall throughput. We analyzed the distribution of throughputs for all clients and found a long tail with many clients timing out after several quiescing periods, then retrying.

Ceph does not quiesce [7] I/O during a snapshot and we did not observe the same spikes in maximum latency that we observed with OpenEBS. We observed that with Ceph, around four times as many objects were created in the underlying RADOS object pools with snapshotting—compared to OpenEBS. That would make sense since Ceph did not quiesce during snapshots, but OpenEBS did.

To create a new snapshot in Kubernetes, users create a Snapshot resource. This resource is created immediately. However, the underlying snapshot is not necessarily ready right away. Figure 5.4 shows a CDF of how long it took after creating a new Snapshot resource until the storage provider reported that the snapshot was actually ready to be used.

Both Ceph and OpenEBS implement copy-on-write snapshots, so it is expected that for most storage configurations, snapshots became available nearly as fast as the Snapshot resources were created. However, some configurations exhibited a long tail where snapshots took several minutes to become ready. For example, although the median time to become ready for snapshots on OpenEBS with three volume replicas was 12 seconds, 10% took longer than 310 seconds and 5% took longer than 702 seconds. The interface between Kubernetes and the storage provider's CSI driver is the Kubernetes Snapshot Controller [44]. When we analyzed the logs for this container, we found that the CreateSnapshot CSI calls for some snapshots were timing out. The controller has an initial timeout period of 500 milliseconds; it then uses an exponential backoff timeout interval mechanism and retries. We empirically noticed retry intervals as long as 30 minutes.

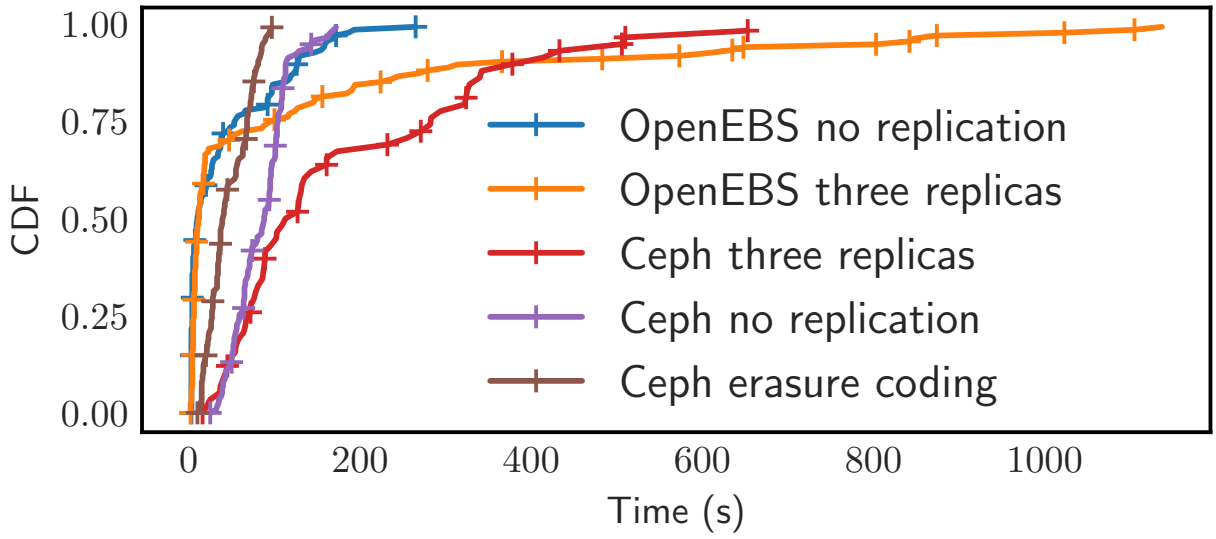

Figure 5.4: CDF of snapshot creation times for different storage provider configurations.

5.4 Orchestration

One of the core CNSBench capabilities is to make it easy to run various mixes of I/O workloads. This is needed since the alternative is to manually choose and assemble workloads together to form a representative combined workload; the diversity of workloads in cloud native environments makes this task infeasible.

One potential use case for this task is to determine which storage configuration is best suited for a particular set of workloads. Another might be to help influence scheduling decisions, such as which workloads to run simultaneously.

To demonstrate CNSBench's orchestration capabilities, we ran multiple instances of three different workloads: (1) MEGAHIT [59], a bioinformatics tool that processes genetic data; (2) fio [20] for generating an intense I/O workload of mixed random reads and writes; and (3) the PostgreSQL [40] database with a workload generated using its benchmark tool pgbench [38]. Each instance of the PostgreSQL workload ran a distinct pair of database and client. Out of each of the workloads, fio was the most I/O intensive, followed by pgbench. Both fio and pgbench spent most of their time waiting for I/O, whereas MEGAHIT was mostly CPU bound.

We tested four different workload ratios: a baseline with ten independent instances of each workload, and then three additional ratios with ten instances of two of the workloads and five of the third. For MEGAHIT and fio we measured the total time to run a fixed load; for pgbench we measured the average throughput after running for ten minutes. This was necessary since the different storage provider configurations performed significantly different, so it would be impractical to evaluate using a fixed amount of work.

Figure 5.5 shows the changes in runtime and throughput, normalized to the baseline values, for different workload ratios and storage providers. The baseline throughputs for pgbench are 5.2, 0.37, 0.38, 85, and 16 operations per second for Ceph (no volume replication), Ceph (three volume replicas), Ceph (erasure coding), OpenEBS (no volume replication), and OpenEBS (three volume replicas), respectively. MEGAHIT had baseline runtimes of 309, 910, 609, 185, and 324 seconds, and fio had baseline runtimes of 427, 816, 699, 923, and 2478 seconds, respectively.

The largest increase in performance of 3.2× is for pgbench when the number of fio instances is reduced. This makes sense: the Ceph storage configurations shows the largest increase in pgbench performance, since

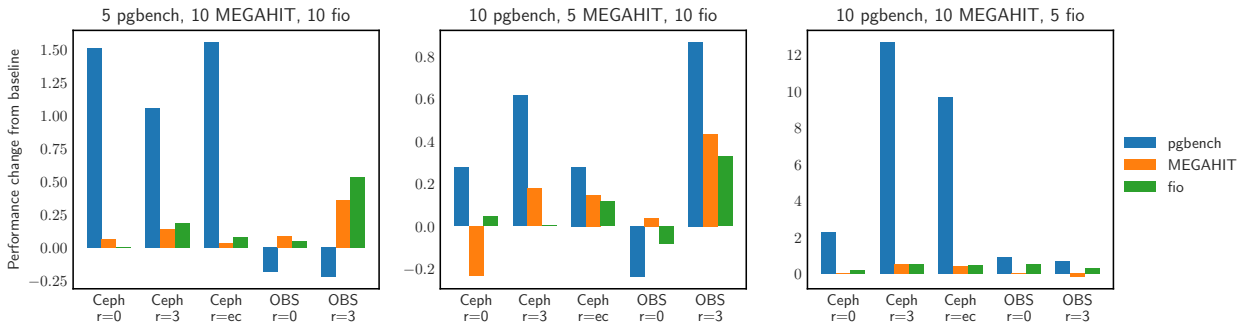

Figure 5.5: Change in performance compared to baseline, for three different ratios of I/O workload on five different storage configurations.

pgbench's baseline performance on Ceph is much worse than on OpenEBS so there is a larger potential for improvement. Also, pgbench and fio are both I/O-intense workloads, *i.e.*, reducing the number of fio instances would help pgbench, but not MEGAHIT.

The workload that had the overall smallest impact on performance is MEGAHIT. This is also expected as fio and pgbench are mainly I/O bound while MEGAHIT is mainly CPU bound and hence reducing the number of MEGAHIT instances does not free up significant I/O resources.

Surprisingly, for some configurations, reducing the number of MEGAHIT instances actually increased the average MEGAHIT runtime. We found that MEGAHIT was more likely to be scheduled on the same node as other fio and pgbench instances when there were five MEGAHIT instances rather than ten. Since there were no memory or CPU limits placed on any of the workloads, this colocation caused other workloads to interfere with MEGAHIT by reducing the number of cycles available to the MEGAHIT workload. Further experimentation showed that applying CPU and memory limits to each workload reduces interference from other workloads since workloads are spread more evenly across nodes. This is because without those limits, the Kubernetes scheduler assumes that each workload requires the same amount of resources and cannot distribute workloads based on their actual resource consumption. We used standard Kubernetes configuration fields to set resource limits for each workload. Determining the ideal resource limits for a workload is difficult, so we set limits based on observed resource usage that would roughly distribute resources fairly across the workloads: fio was limited to 1 CPU and 1.5 GB of memory, MEGAHIT was limited to 1 CPU and 1 GB of memory, and pgbench was limited to 2 CPU and 1.75 GB of memory.

Applying resource limits to workloads helps reduce interference but does not eliminate it entirely. Other sources of potential interference include resources not accounted for by Linux's resource management [11] and resource usage by processes not managed by Kubernetes [17]. Also, getting resource limits right is tricky: although measuring the minimum amount of memory needed to run a workload is straightforward, many workloads that require I/O will benefit from being able to use the page cache. Since page cache usage counts against a workload's memory limit, setting the limit too low can cause pages to be repeatedly moved in and then evicted from the cache as the workload does I/O. Setting the limit too high will allow pages to remain unnecessarily in the cache after they are no longer needed, wasting memory and reducing the node's workload capacity. CNSBench can be used to help explore different possibilities for resource limits and to understand how these limits impact each workloads' performance.

Overall, these results demonstrate the variability in storage provider performance, and the utility of being able to easily compose and run diverse sets of workloads at various ratios.

Benchmark	Lines
Volume creation and attachment § 5.2	19
YCSB and MongoDB, no snapshots § 5.3	35
YCSB and MongoDB, with snapshots § 5.3	54
Multiple workloads § 5.4	72–117

Table 5.1: Number of lines needed to specify CNSBench benchmarks used during evaluation.

5.5 Benchmark Usability

Requirement 4 in Section 3.2 states that CNSBench should be easy for users to configure and run. Although usability is often subjective, one metric that can be used to estimate ease of use is the number of lines necessary for specifying a workload. Table 5.1 shows the number of lines needed to specify each of the benchmarks used in this evaluation section.

Overall a user can specify the complex, distributed, and diverse workloads in just 19–117 lines of configuration. The workloads used in Section 5.4 require slightly longer specifications as they contain multiple instances of the same sub-workload, which currently results in duplication in the CNSBench's benchmark specification. We plan to eliminate such repetitions in future to make using CNSBench even simpler.

Chapter 6

Related Work

Classic storage benchmarks Storage benchmarking is an old and complex topic with many applicable techniques and intricate nuances [65]. It is not surprising, therefore, that the array of tools for benchmarking and corresponding studies is extensive. Filebench [67], fio [20], SPEC SFS [42], and IOZone [50] are just a few examples of popular file system benchmarks. For a comprehensive survey of file system and storage benchmarks we refer the reader to a study by Traeger *et al.* [70].

The majority of such benchmarks generate a single, stationary workload per run, which is not representative of cloud native environments. Few benchmarks have built-in mechanisms to dynamically increase the load, in order to discover the peak throughput where diminishing returns (*e.g.*, due to thrashing) begin to take over. For example, measuring NFS throughput via SPEC SFS [64] and process scheduling throughput using AIM7 [68].

Filebench [47, 67] comes with several canned configurations [62] and even has its own Workload Modeling Language (WML) [73]. It, however, is not distributed (cannot run in a coordinated manner across multiple containers) and, though WML is flexible for encoding stationary workloads, is still limited in creating dynamically changing workloads. In our experience, adding support for distributed and temporally varying workloads to Filebench's WML is a difficult task. Therefore, in CNSBench, we exploited the orchestration capabilities of cloud native environments and delegated these tasks to a higher level (*i.e.*, the CNSBench controller and the Kubernetes orchestrator itself). This further allowed us to support any existing benchmarks as canned I/O generators.

RocksDB [53] is a popular key-value store with canned, preconfigured workloads using a db_bench driver to create random/sequential reads/writes and mixes thereof. One can run these workloads in any order and configure their working-set size. However, that is still a manual process with little flexibility, and no support for control operations (which is true for the previously mentioned benchmarks as well).

Object storage benchmarks In recent years the need to test the performance of cloud storage has motivated academia and industry to develop several micro-benchmarks for that task such as YCSB [51] and COSBench [74]. YCSB is an extensible workload generator that evaluates the performance of different cloud-serving key-value stores. COSBench measures the performance of cloud object storage services and comes with plugins for different cloud providers. Unlike these benchmarks, CNSBench focuses on workloads that run in containers and require a file system interface.

Cloud native benchmarks TailBench [56] provides a set of interactive macro-benchmarks: web servers, speech recognition databases, and cloud based machine translation systems. Similarly, DeathStarBench [55] is a benchmark suite for microservices and their hardware-software implications for cloud and edge systems. Both TailBench and DeathStarBench target cloud applications and are not explicitly storage benchmarks.

Chapter 7

Conclusion

Although measuring storage performance was always an important topic, its relevance has escalated in recent years due to the increased demand to reliably move containerized applications across clouds. Furthermore, I/O patterns of applications have evolved, exhibiting higher density, diversity, dynamicity, and specialization than before. Perhaps most importantly, storage services now experience a high rate of *control operations* (*e.g.*, volume creation, formatting, snapshotting), which directly impact the performance of applications that call them and indirectly influence the I/O of other applications in a cluster. Existing storage benchmarks, however, are not able to model these new cloud native scenarios and workloads holistically and faithfully.

In this report we presented the design of CNSBench—a storage benchmarking framework that containerizes legacy I/O benchmarks, orchestrates their concurrent runs, and concurrently generates a stream of control operations. CNSBench is easy to configure and run, while still being versatile enough to express a high variety of real-world cloud native workloads. We used CNSBench to evaluate two cloud native storage backends—OpenEBS and Ceph—and found several differences. For example, our evaluation shows that the maximum rate of control operations varies significantly across storage technologies and configurations by a factor of up to 8.5×.

Future work We plan to work on extending the library of I/O workloads with I/O "kernels" that represent microservices, and also improve the benchmark specification language to make the syntax more concise and avoid having to duplicate sub-workloads. Further, we will work on collecting I/O and control operation traces from production environments, analyze them, and create corresponding profiles for CNSBench. Our longer term plans including finding and fixing performance bugs using CNSBench, and even developing our own efficient storage solution.

We hope our benchmark will be adopted by storage and cloud native communities, and look forward to contributions.

Chapter 8

Acknowledgements

This work was made possible in part thanks to Dell-EMC, NetApp, and IBM support; and NSF awards CCF-1918225, CNS-1900706, CNS-1729939, and CNS-1730726. Thanks also to Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Aneesh Joshi, Julie Lee, Lukas Rupprecht, Dimitris Skourtis, Abhiraj Smit, Yang Yang, and Erez Zadok for their contributions.

Bibliography

[1] Amazon s3. *https://aws.amazon.com/s3/*.

[2] Amazon web services (aws). *https://aws.amazon.com/*.

[3] Bloomberg: An early adopter's success with kubernetes at scale. *https://www.cncf.io/case-studies/bloomberg/*.

[4] Building applications with serverless architectures. *https://aws.amazon.com/lambda/serverless-architectures-learn-more/*.

[5] Building large clusters). *https://kubernetes.io/docs/setup/best-practices/cluster-large/*.

[6] Ceph. *https://ceph.io/*.

[7] Ceph snapshots. *https://docs.ceph.com/en/latest/rbd/rbd-snapshot/*.

[8] client-go. *https://github.com/kubernetes/client-go*.

[9] Cloud native computing foundation. *https://www.cncf.io/*.

[10] Container attached storage is cloud native storage (cas. *https://www.cncf.io/blog/2020/09/22/container-attached-storage-is-cloud-native-storage-cas/*.

[11] Container isolation gone wrong. *https://sysdig.com/blog/container-isolation-gone-wrong/*.

[12] Container object storage interface api. *https://github.com/kubernetes-sigs/container-object-storage-interface-api*.

[13] Container Storage Interface (CSI) Specification. *https://bit.ly/3bqQX4b*.

[14] Containerization. *https://www.ibm.com/cloud/learn/containerization*.

[15] Docker. *https://docker.com/*.

[16] Docker hub. *https://hub.docker.com/*.

[17] Don't let linux control groups run uncontrolled. *https://engineering.linkedin.com/blog/2016/08/don_t-let-linux-control-groups-uncontrolled*.

[18] Dynamic Provisioning and Storage Classes in Kubernetes. *https://bit.ly/2Uh3Qbw*.

[19] filesystems.org. *https://www.filesystems.org/*.

[20] fio. *https://github.com/axboe/fio*.

[21] Going cloud native: 6 essential things you need to know. *https://www.weave.works/technologies/going-cloud-native-6-essential-things-you-need-to-know/*.

[22] Google cloud). *https://cloud.google.com/*.

[23] Ibm cloud. *https://www.ibm.com/cloud*.

[24] Improve kubectl cp, so it doesn't require the tar binary in the container #58512. *https://github.com/kubernetes/kubernetes/issues/58512*.

[25] Kibana. *https://www.elastic.co/kibana*.

[26] kubectl cp to work on stopped/completed pods #454. *https://github.com/kubernetes/kubectl/issues/454*.

[27] Kubernetes. *https://kubernetes.io/*.

[28] Kubernetes object management. *https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/*.

[29] Kubernetes Storage. *https://kubernetes.io/docs/concepts/storage/*.

[30] Microsoft azure. *https://azure.microsoft.com/*.

[31] Mongodb. *https://www.mongodb.com/*.

[32] News uk keeps new content and capabilities coming fast with amazon eks and new relic. *https://blog.newrelic.com/product-news/news-uk-content-capabilities-amazon-eks-new-relic/*.

[33] Openbenchmarking.org. *https://openbenchmarking.org/*.

[34] OpenEBS. *https://openebs.io/*.

[35] Openebs cstor csi driver. *https://github.com/openebs/cstor-csi/blob/master/pkg/driver/controller_utils.go#L243*.

[36] Openebs replication.c. *https://github.com/openebs/istgt/blob/replication/src/replication.c#L1958*.

[37] Operator pattern. *https://kubernetes.io/docs/concepts/extend-kubernetes/operator/*.

[38] pgbench. *https://www.postgresql.org/docs/10/pgbench.html*.

[39] Portworx kubernetes snapshots and backups. *https://docs.portworx.com/portworx-install-with-kubernetes/storage-operations/kubernetes-storage-101/snapshots/*.

[40] Postgresql. *https://www.postgresql.org/*.

[41] Rook. *https://rook.io/*.

[42] SPEC SFS 2014. *https://www.spec.org/sfs2014/*.

[43] Sysdig 2019 Container Usage Report. *https://sysdig.com/blog/sysdig-2019-container-usage-report/*.

[44] Volume snapshot & restore - kubernetes csi developer documentation. *https://kubernetes-csi.github.io/docs/snapshot-restore-feature.html*.

[45] What is a container? *https://www.docker.com/resources/what-container*.

[46] A hybrid and multicloud strategy for system administrator. Technical Report #F21608_0220, Red Hat, 2020.

[47] George Amvrosiadis and Vasily Tarasov. Filebench github repository, 2016. *https://github.com/filebench/filebench/wiki*.

[48] Andrew Bartels, Dave Bartolett, John Rymer, Matthew Guarini, Charlie Dai, and Alyssa Danilow. The public cloud market outlook, 2019 to 2022: Public cloud growth continues to power tech spending. Technical report, Forrester, July 2019.

[49] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[50] Don Capps and Tom McNeal. Analyzing nfs client performance with iozone. NFS Industry Conference, 2002.

[51] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[52] Paul Dix. Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB Performance for InfluxDB. *https://bit.ly/365KSL2*, 2014.

[53] Facebook. RocksDB. *https://rocksdb.org/*, September 2019.

[54] *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, Santa Clara, CA, February 2020. USENIX.

[55] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An Open-source Benchmark Suite for Microservices and their Hardware-software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[56] Harshad Kasture and Daniel Sanchez. Tailbench: A Benchmark Suite and Evaluation Methodology for Latency-critical Applications. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[57] Sachin Katti, John Ousterhout, Guru Parulkar, Marcos Aguilera, and Curt Kolovson. Scalable control plane substrate).

[58] Stefan Kolb. *On the Portability of Applications in Platform as a Service*, volume 34. University of Bamberg Press, 2019.

[59] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. In *Bioinformatics*, 2015.

[60] Jack McElwee and Allan Krans. Public cloud benchmark: First calendar quarter 2020. Technical report, Technology Business Research, July 2020.

[61] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Lukas Rupprecht, Dimitris Skourtis, and Erez Zadok. The case for benchmarking control operations in cloud native storage. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[62] Filebench pre-defined personalities, 2016. http://filebench.sourceforge.net/wiki/index.php/Pre-defined_personalities.

[63] Frank Della Rosa. Implementation of microservices architecture hastens across industries. Technical Report #US46108319, IDC, 2020.

[64] SPEC SFS®2014. https://www.spec.org/sfs2014/.

[65] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.

[66] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login:*, 41(1), 2016.

[67] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *;login: The USENIX Magazine*, 41(1):6–12, March 2016.

[68] AIM Technology. AIM multiuser benchmark - suite VII version 1.1. http://sourceforge.net/projects/aimbench, 2001.

[69] Johannes Thönes. Microservices. *IEEE Software*, 32(1), 2015.

[70] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P Wright. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2), 2008.

[71] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 307–320, Seattle, WA, November 2006. ACM SIGOPS.

[72] Sage Weil, Andrew Leung, Scott Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW)*, 2007.

[73] Filebench workload model language (WML), 2016. https://github.com/filebench/filebench/wiki/Workload-Model-Language.

[74] Qing Zheng, Haopeng Chen, Yaguang Wang, Jian Zhang, and Jiangang Duan. COSBench: Cloud Object Storage Benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2013.