Micron® 9300 MAX NVMe™ SSDs + Red Hat® Ceph® Storage for 2nd Gen AMD EPYC™ Processors

Reference Architecture

John Mazzie, Principal Storage Solution Engineer
Tony Ansley, Principal Technical Marketing Engineer

AMD Red Hat Micron

Contents

Executive Summary

This document describes an example configuration of a performance-optimized Red Hat® Ceph® Storage (RHCS) cluster using Micron® NVMe™ SSDs, AMD EPYC™ 7002 x86 architecture rack-mount servers, and 100 Gb/E networking.

It details the hardware and software building blocks used to construct this reference architecture (including the Red Hat Enterprise Linux OS configuration, network switch configurations, and Ceph tuning parameters) and shows the performance test results and measurement techniques for a scalable 4-node RHCS architecture.

Optimized for block performance while also providing very high-performance object storage, this all-NVMe solution provides a rack-efficient design to enable:

Faster deployment: The configuration has been pre-validated, optimized, and documented to enable faster deployment and faster performance than using default instructions and configuration.

Balanced design: The right combination of NVMe SSDs, DRAM, processors, and networking ensures a balanced set of subsystems optimized for performance.

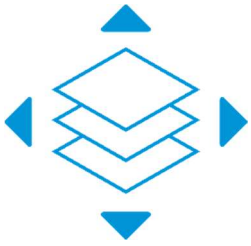
Broad use: Complete tuning and performance characterization is documented across multiple IO profiles for broad deployment across multiple uses.

Our testing illustrates exceptional performance results for 4 KiB random block workloads and 4 MiB object workloads (Tables 1a and 1b).

4KiB Random Block Performance		
IO Profile	IOPS	Avg. Latency
100% Read	3,161,146	1.31ms
70%/30% R/W	1,325,888	W: 6.2ms R: 1.82ms
100% Writes	645,655	6.44ms

4MiB Object Performance		
IO Profile	Throughput	Avg. Latency
100% Sequential Read	41.03 GiB/s	29.81ms
100% Random Read	43.77 GiB/s	27.92ms
100% Random Writes	19.15 GiB/s	38.11ms

Table 1a and Table 1b - Performance Summary



Micron Reference Architectures

Micron Reference Architectures are optimized, pre-engineered, enterprise-leading solution templates for platforms that are co-developed between Micron and industry-leading hardware and software companies.

Designed and tested at Micron's Storage Solutions Center, they provide end users, system builders, independent software vendors (ISVs), and OEMs with a proven template to build next-generation solutions with reduced time investment and risk.

Why Micron for this Solution

Storage (SSDs and DRAM) represent a large portion of the value of today's advanced server/storage solutions. Micron's storage expertise starts at memory technology research, innovation, and design and extends to collaborating with customers and software providers on total data solutions. Micron develops and manufactures the storage and memory products that go into the enterprise solutions described here.

Ceph Distributed Architecture Overview

A Ceph storage cluster (Figure 1) consists of multiple Ceph monitor nodes and data nodes for scalability, fault-tolerance, and performance. Ceph stores all data as objects, regardless of the client interface used. Each node is based on industry-standard hardware and uses intelligent Ceph daemons that communicate with each other to:

- Store, retrieve, and replicate data objects
- Monitor and report on cluster health
- Redistribute data objects dynamically
- Ensure data object integrity
- Detect and recover from faults and failures

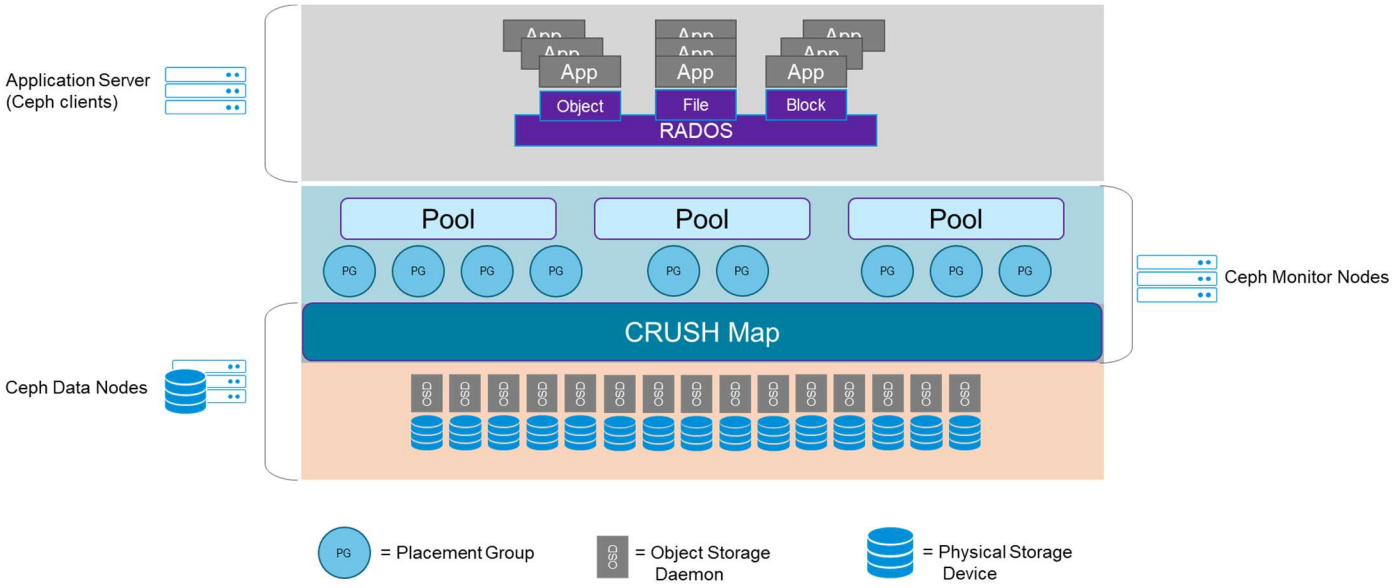


Figure 1 - Ceph Architecture

To the application servers (Ceph clients) that read and write data, a Ceph storage cluster looks like a simple pool storage resource for data; however, the storage cluster performs many complex operations in a manner that is completely transparent to the application server. Ceph clients and Ceph object storage daemons (Ceph OSDs or OSDs) both use the Controlled Replication Under Scalable Hashing (CRUSH) algorithm for storage and retrieval of objects.

For a Ceph client, the storage cluster is very simple. When a Ceph client reads or writes data (referred to as an I/O context), it connects to a logical storage pool in the Ceph cluster. The figure above illustrates the overall Ceph architecture, with concepts described in the sections that follow.

Clients write to Ceph storage pools while the CRUSH ruleset determines how placement groups get distributed across OSDs.

Pools: Ceph clients store data objects in logical, dynamic partitions called pools. Administrators can create pools for various reasons such as for particular data types, to separate block, file and object usage, application isolation, or to separate user groups (multitenant hosting). The Ceph pool configuration dictates the number of object replicas and the number of placement groups (PGs) within the pool. Ceph storage pools can be either replicated or erasure-coded, as appropriate, for the application and cost model. Additionally, pools can "take root" at any position in the CRUSH hierarchy, allowing placement on groups of servers with differing performance characteristics, which allows for the optimization of different storage workloads.

Object storage daemons: Object storage daemons (OSDs) store data and handle data replication, recovery, backfilling, and rebalancing. They also provide some cluster state information to Ceph monitor nodes by checking other Ceph OSDs with a heartbeat mechanism. A Ceph storage cluster, configured to keep three replicas of every object, requires a minimum of three OSDs; two of which need to be operational to process write requests successfully. Ceph OSDs roughly correspond to a file system on a physical hard disk drive.

Placement groups: Placement groups (PGs) are shards, or fragments, of a logical object pool that are composed of a group of Ceph OSDs that are in a peering relationship. PGs provide a means of creating replication or erasure coding groups of coarser granularities than on a per-object basis. A larger number of placement groups (e.g., 200 per OSD or more) leads to better balancing.

CRUSH map: The CRUSH algorithm determines how to store and retrieve information from data nodes. CRUSH enables clients to communicate directly with OSDs on data nodes rather than through an intermediary service. By doing so, this removes a single point of failure from the cluster. The CRUSH map consists of a list of all OSDs and their physical location. Upon initial connection to a Ceph-hosted storage resource, the client contacts a Ceph monitor node for a copy of the CRUSH map, which enables direct communication between the client and the target OSDs.

Ceph monitors (MONs): Before Ceph clients can read or write data, they must contact a Ceph MON to obtain the current CRUSH map. A Ceph storage cluster can operate with a single MON, but this introduces a single point of failure. For added reliability and fault tolerance, Ceph supports an odd number of monitors in a quorum (typically three or five for small to mid-sized clusters). Consensus among various MON instances ensures consistent knowledge about the state of the cluster.

Reference Architecture Overview

Micron designed this reference architecture on the AMD EPYC 7002 architecture using dual AMD 7742 processors (Figure 2). This processor architecture provides a performance-optimized server platform while yielding an open, cost-effective software-defined storage (SDS) platform suitable for a wide variety of use cases such as OpenStack™ cloud, video distribution, transcoding, and big data storage.

The Micron 9300 MAX NVMe SSD offers tremendous performance with low latencies. Capacity per rack unit (RU) is maximized with ten 12.8TB NVMe SSDs per 1U storage node. This reference architecture takes up six RUs consisting of one monitor node and four data nodes and one Ethernet switch. Using this reference architecture as a starting point, administrators can add additional data nodes 1RU and 128TB at a time.

Two Mellanox ConnectX®-5 100 Gb/E network cards per server handle data traffic—one for the client/public network traffic and a second for the internal Ceph replication network traffic. Mellanox ConnectX-4 50 Gb/E network cards are installed in both the clients and monitor nodes for connection to the storage networks.

Network Switches:
1x 100 Gb/E, 32x QSFP28 ports

Monitor Nodes
AMD EPYC single-socket x86 1U
1x AMD 7551P CPU
256GB RAM

Storage Nodes
AMD EPYC 2 dual-socket x86 1U
2x AMD 7742 CPUs
512GB RAM
10x 9300 MAX NVMe SSDs

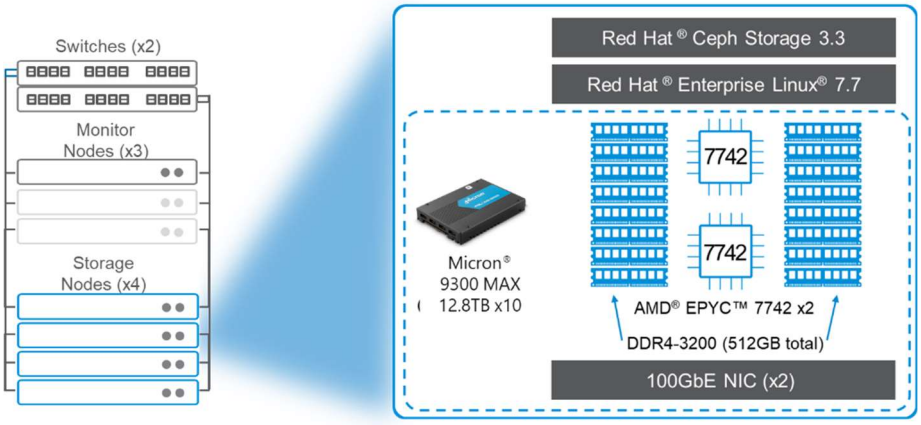


Figure 2 – Micron Ceph Reference Architecture

> **Note:** Micron performed all tests using a single monitor node. Production deployments should use at least three monitor nodes to provide adequate redundancy to the solution.

Software

This section details the software versions used in the reference architecture.

Red Hat Ceph Storage

Red Hat collaborates with the global open source Ceph community to develop new Ceph features, then packages changes into predictable, stable, enterprise-quality releases. Red Hat Ceph Storage uses the open-source Ceph Luminous version 12.2, to which Red Hat was a leading code contributor. This reference architecture uses version 3.3 of Red Hat Ceph Storage.

As a self-healing, self-managing, unified storage platform with no single point of failure, Red Hat Ceph Storage decouples software from hardware to support block, object, and file storage services on standard x86 servers, using either HDDs and/or SSDs, significantly lowering the cost of storing enterprise data. OpenStack® also uses Red Hat Ceph Storage along with services, including Nova, Cinder, Manila, Glance, Keystone, and Swift, and it offers user-driven storage lifecycle management. Ceph is a highly tunable,

extensible, and configurable architecture, well suited for archival, rich media, and cloud infrastructure environments.

Among many of its features, Red Hat Ceph Storage provides the following advantages to this reference architecture:

- Block storage integrated with OpenStack, Linux, and KVM hypervisor
- Data durability via erasure coding or replication
- Red Hat Ansible automation-based deployment
- Advanced monitoring and diagnostic information with an on-premise monitoring dashboard
- Availability of service-level agreement (SLA)-backed technical support
- Red Hat Enterprise Linux (included with subscription) and the backing of a global open source community

Red Hat Enterprise Linux

In need of a high-performance operating system environment, enterprises depend on Red Hat® Enterprise Linux® (RHEL) for scalable, fully supported, open-source solutions. Micron uses version 7.7 of Red Hat Enterprise Linux in this reference architecture due to its performance, reliability, and security, as well as its broad usage across many industries. Supported by leading hardware and software vendors, RHEL provides broad platform scalability (from workstations to servers to mainframes) and consistent application environment across physical, virtual, and cloud deployments.

Software by Node Type

Table 2 shows the software and version numbers used in the Ceph monitor and storage nodes.

Operating System	Red Hat Enterprise Linux	7.7
Storage Software	Red Hat Ceph Storage	3.3
NIC Driver	Mellanox® OFED Driver	4.7-1.0.0.0

Table 2 - Software Deployed on Ceph Data and Monitor Nodes

The software used on the load generation client is the same as that used on the Ceph data and monitor nodes. All block testing used the open-source FIO storage load generation tool, version 3.1.0, leveraging the `librbd` module.

Hardware by Node Type

Ceph Data Node

The Ceph data nodes in this RA host two or more OSDs per physical SSD. While this RA used a server product available for purchase from one vendor, this RA does not make any recommendations regarding any specific server vendor or implementation, focusing on the overall solution architecture built around AMD EPYC 7002 processors and architecture.

Mellanox ConnectX-5 network controller offers dual ports of 10/25/50/100 Gb/s Ethernet connectivity and advanced offload capabilities while delivering high bandwidth, low latency, and high computation efficiency for high performance, data-intensive, and scalable HPC, cloud, data analytics, database, and storage platforms.

Table 3 provides the details for the server architecture used for this Ceph data node role.

Server Type	AMD x86 (dual-socket) 1U with PCIe Gen 3/4
CPU (x2)	AMD EPYC 7742 (64 cores, 2.25GHz base)
DRAM (x16)	Micron 32GB DDR4-2666 MT/s, 512GB total per node
NVMe (x10)	Micron 9300 MAX NVMe SSDs, 12.8TB each
SATA (OS)	Micron 2200 BOOT (NVMe)
Network	2x Mellanox ConnectX-5 100 Gb/E dual-port (MCX516A-CCAT)

Table 3 - Storage Node Hardware Details

Ceph Monitor Node

The Ceph monitor node in this RA is an AMD EPYC 7001 architecture server. As with the Ceph data node, this RA does not make any recommendations regarding any specific server vendor or implementation, focusing on the overall solution architecture built around AMD processors and architecture.

ConnectX-4 EN network controller offers dual ports of 10/25/50/100 Gb/s Ethernet connectivity and advanced offload capabilities while delivering high bandwidth, low latency, and high computation efficiency for high performance, data-intensive, and scalable HPC, cloud, data analytics, database, and storage platforms.

Table 4 provides the details for the server architecture used for this Ceph monitor node role.

Server Type	AMD x86 (single-socket) 1U with PCIe Gen3
CPU (x1)	1x AMD EPYC 7551P (32 cores, 2.0GHz base)
DRAM (x8)	Micron 32GB DDR4-2400 MT/s, 256GB total per node
SATA (OS)	64GB SATA Disk on Motherboard
Network	1x Mellanox ConnectX-4 50 Gb/E single-port (MC4X413A-GCAT)

Table 4 -Monitor Node Hardware Details

Micron Components Used

Micron 9300 MAX NVMe SSDs

This RA uses the 9300 MAX 12.8TB NVMe SSD. The Micron 9300 series of NVMe SSDs is our flagship performance SSD family and our third generation of NVMe SSDs. The 9300 family has the right capacity for demanding workloads, with capacities from 3.2TB to 15.36TB in mixed-use or read-intensive designs. The 9300 is also the first offering from Micron to provide "no compromise" read and write performance by offering balanced 3500 MB/s throughput on certain models.[1]

Table 5 summarizes the 9300 MAX 12.8TB specifications

Model	9300 MAX	**Interface**	PCIe Gen 3 x4
Form Factor	U.2	**Capacity**	12.8TB
NAND	Micron® 3D TLC	**MTTF**	2M device hours
Sequential Read	3.5 GB/s	**Random Read**	850,000 IOPS
Sequential Write	3.5 GB/s	**Random Write**	310,000 IOPS
Endurance	144.8 PB	**Status**	Production

Note: GB/s measured using 128K transfers, IOPS measured using 4K transfers. All data is steady state. Complete MTTF information can be provided by your Micron sales associate.

Table 5 - 9300 MAX 12.8TB Specifications Summary

Network Switches

This RA uses one 100 Gb/E switch (32x QSFP28 ports each). For production purposes, Micron recommends using two switches for redundancy purposes, with each switch partitioned to support two network segments—one for the client data transfer and the second for the Ceph intercluster storage network. Switches used in this RA use the Broadcom Tomahawk® switch architecture (Table 6).

Model	Supermicro SSE-C3632SR
Software	Cumulus™ Linux 3.4.2

Table 6 - Network Switches (Hardware and Software)

The SSE-C3632S Layer 2/3 Ethernet Switch provides an open networking-compliant solution, providing the ability to maximize the efficient and flexible use of valuable data center resources while providing an ideal platform for managing and maintaining those resources in a manner in tune with the needs of an organization.

1. Offered on 9300 PRO 7.68TB and 15.36TB and 9300 MAX 6.4TB and 12.8TB models.

Planning Considerations

The following topics provide information to enhance the overall experience of the solution and ensure the solution is scalable while maximizing performance.

Number of Ceph Storage Nodes

At least three (3) storage nodes must be present in a Ceph cluster to become eligible for Red Hat technical support. While this RA uses four data nodes, additional nodes can provide scalability and redundancy. Four (4) storage nodes represent a suitable starting point as a building block for scaling up to larger deployments.

Number of Ceph Monitor Nodes

A Ceph storage cluster deployed for production workloads should have at least three (3) monitor nodes on separate hardware for added resiliency. These nodes do not require high-performance CPUs. They do benefit from having SSDs to store the monitor map data. For testing purposes, this solution uses a single monitor node..

Replication Factor

NVMe SSDs have high reliability, with high MTTF and low bit error rate. Micron recommends using a minimum replication factor of two in production when deploying OSDs on NVMe versus a replication factor of three, which is common with legacy HDD-based storage.

CPU Sizing

Ceph OSD processes can consume large amounts of CPU while doing small block operations. Consequently, a higher CPU core count results in higher performance for I/O-intensive workloads.

For throughput-intensive workloads characterized by large sequential object-based I/O, Ceph performance is more likely to be bound by the maximum network bandwidth of the cluster. CPU sizing is less impactful.

Ceph Configuration Tuning

Tuning Ceph for NVMe devices can be complex. The ceph.conf settings used in this reference architecture optimize the solution for small block random performance (see Appendix A).

Networking

A 25 Gb/E network enables the solution to leverage the maximum block performance benefits of a NVMe-based Ceph cluster. For throughput-intensive workloads, Micron recommends 50 Gb/E or faster throughput connections.

Number of OSDs per Drive

Write performance with two OSD per drive yields approximately 3% - 7% better IOPS than the performance seen when using four OSDs per drive, but tail latency (99.99%) is dramatically better when running four OSDs at higher thread count levels at queue depth of 32, as seen in Figure 3.

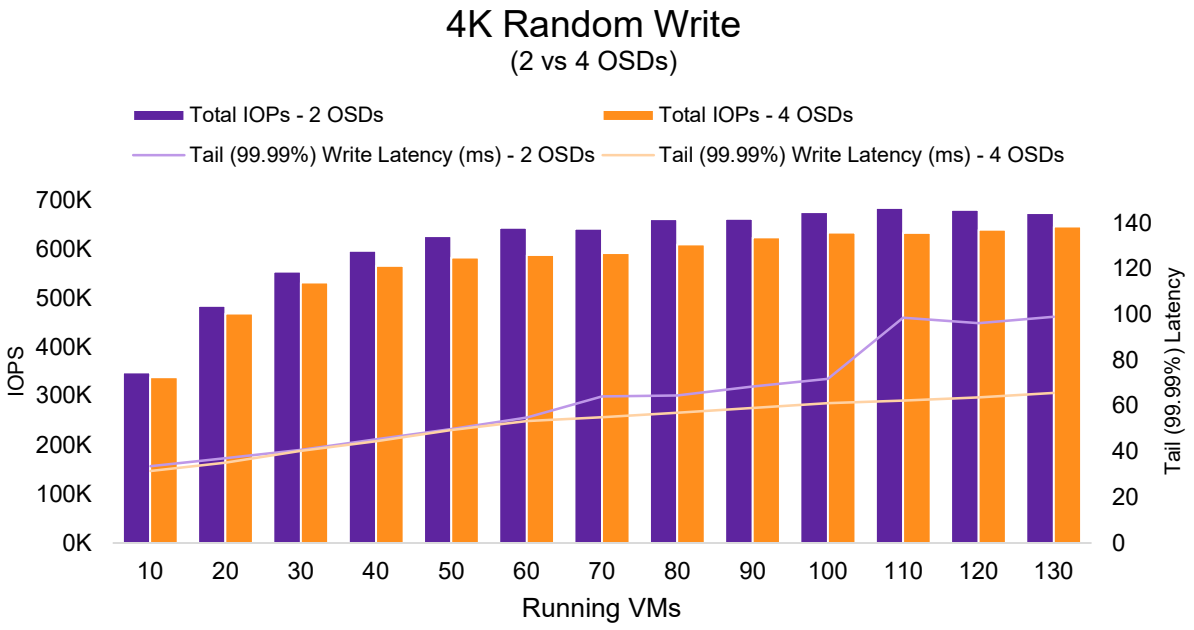


Figure 3 - Write Performance Comparison of 2 OSDs vs. 4 OSDs per SSD

Figure 4 shows that read performance is also similar, with a dramatic improvement in tail latency for 130 simultaneous execution threads as queue depth increases from 16 to 32.

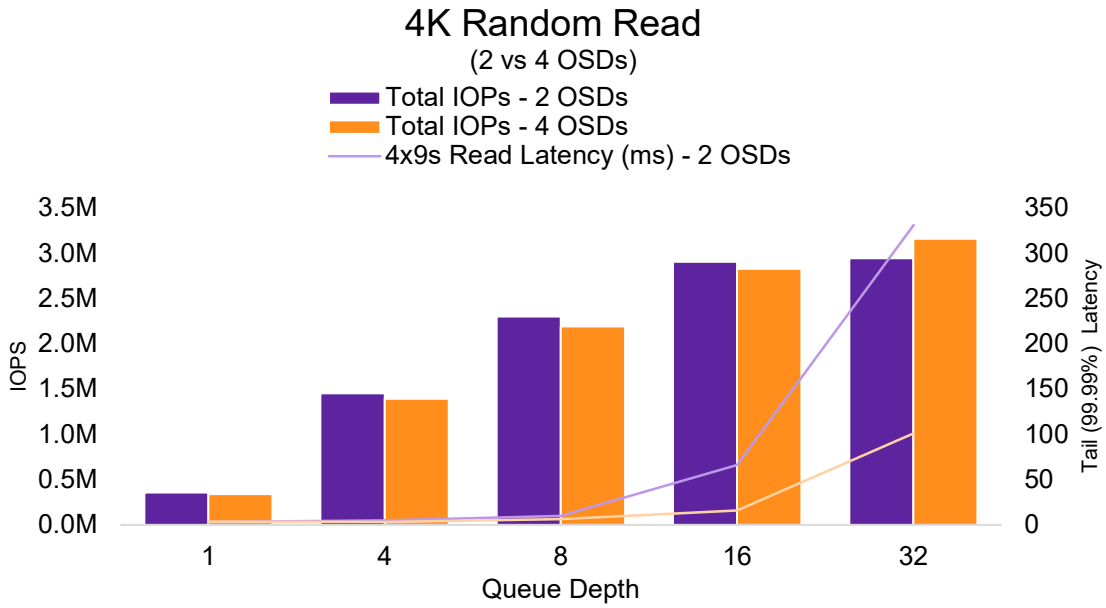


Figure 4 - Read Performance Comparison of 2 OSDs vs. 4 OSDs per SSD

OS Tuning/NUMA

This RA used Ceph-Ansible for OS tuning and applied the following OS settings:

```
disable_transparent_hugepage: true
kernel.pid_max, value: 4,194,303
fs.file-max, value: 26,234,859
vm.zone_reclaim_mode, value: 0
vm.swappiness, value: 1
vm.min_free_kbytes, value: 1,000,000
net.core.rmem_max, value: 268,435,456
net.core.wmem_max, value: 268,435,456
net.ipv4.tcp_rmem, value: 4096 87,380 134,217,728
net.ipv4.tcp_wmem, value: 4096 65,536 134,217,728
ceph_tcmalloc_max_total_thread_cache: 134,217,728
```

Due to the unbalanced nature of the servers concerning PCIe lane assignments (four NVMe devices and both NICs attach to CPU 1, while the other six NVMe devices attach CPU 2), this RA did not use any NUMA tuning during testing.

`Irqbalance` was active for all tests and did a reasonable job balancing across CPUs.

Measuring Performance

4 KiB Random Workloads

Small block testing used the FIO synthetic I/O generation tool and the Ceph RADOS Block Device (RBD) driver to generate 4 KiB random I/O workloads.

The test configuration consisted of initially creating 130 RBD images, resulting in each RBD image size of 75GB and a total of 9.75TB of data. Implementing a 2x replicated pool resulted in 19.5TB of total data stored within the cluster.

The four data nodes have a combined total of 2TB of DRAM (512GB per server), which is 10.2% of the dataset size.

Random write tests scaled the number of FIO clients running against the Ceph cluster at a fixed queue depth of 32. (A client is a single instance of FIO running on a load generation server.) Using a queue depth of 32 simulates an active RDB image consumer and allows tests to scale up to a high client count. The number of clients scale from 10 clients to 130 clients. The test used 10 load generation servers with an equal number of FIO instances on each load generation server.

Random reads and 70/30 read/write tests all used 130 FIO clients and their associated RBD images, scaling the queue depth per FIO client from 1 to 32 in base-2 increments. It is important to use all 130 FIO clients for these tests to ensure that tests access the entire 19.5TB dataset; otherwise, Linux filesystem caching can skew results, resulting in a false report of higher performance.

Three test iterations executed for 10 minutes, with a two-minute ramp-up time, for a total of 12-minute per test iteration or 36 minutes per pass. Before each iteration, the test script clears all Linux filesystem caches. The results reported are the mathematical average across all test runs.

4MB Object Workloads

Object testing utilizes the RADOS Bench tool provided as part of the Ceph package to measure object I/O performance. This benchmark reports throughput performance in GiB/s and represents the best-case object performance. Object I/O uses a RADOS gateway service operating on each load generation server. The configuration of RADOS gateway is beyond the scope of this RA.

To measure object write throughput, each test executed RADOS Bench with a "threads" value of 16 on a load generation server writing directly to a Ceph storage pool using 4MB objects. RADOS Bench executed on a varying number of load generation servers scaled between 2 to 20 in base-2 increments.

To measure object read throughput, 10 RADOS Bench instances executed 4MB object reads against the storage pool while scaling RADOS Bench thread count between one thread and 32 threads in base-2 increments.

Five test iterations executed for 10 minutes. Before each iteration, the test script cleared all Linux filesystem caches. The results reported are the mathematical average across all test runs.

Baseline Performance Test Methodology

Storage and network performance is baseline tested without Ceph software to determine the theoretical hardware performance maximums using FIO (storage) and iPerf (network) benchmark tools. Each storage test executes one locally run FIO instance per NVMe drive (10 total NVMe drives) simultaneously. Each network test executes four concurrent iperf3 instances from each data node and monitor node to each other and from each client to each data server. The results represent the expected maximum performance possible using the specific server and network components in the test environment.

10 Micron 9300 MAX SSDs deliver 1.1 million 4KB random read IOPS and 900,000 4KB random write IOPS in baseline FIO testing on a single storage server.

Storage Baseline Results

The baseline block storage test executed FIO across all 10 9300 MAX NVMe SSDs on each storage node. FIO instances executed 4 KiB random writes at a queue depth of 64 per FIO instance. Table 7 provides the average IOPS and latency for all storage baseline testing.

Storage Node	Write IOPS	Write Avg. Latency	Read IOPS	Read Avg. Latency
Node 1	879,759	2.99ms	1,143,551	2.44ms
Node 2	869,205	3.00ms	1,170,419	2.25ms
Node 3	892,080	2.95ms	1,115,564	2.31ms
Node 4	902,797	2.94ms	1,173,535	2.27ms

Table 7 - Baseline FIO 4 KiB Random Workloads

The baseline object storage test executed FIO across all 10 9300 MAX NVMe SSDs on each node. FIO instances executed 128 KiB sequential writes at a queue depth of eight per FIO instance. FIO instances executed 4 MiB sequential reads at a queue depth of eight per FIO instance. Table 8 provides the average throughput and latency results.

Storage Node	Write Throughput	Write Avg. Latency	Read Throughput	Read Avg. Latency
Node 1	11.196 GiB/s	13.98ms	14.100 GiB/s	11.08ms
Node 2	10.800 GiB/s	14.52ms	12.247 GiB/s	12.75ms
Node 3	11.018 GiB/s	14.22ms	13.615 GiB/s	11.47ms
Node 4	11.277 GiB/s	13.88ms	13.877 GiB/s	11.25ms

Table 8 - Baseline FIO 128 KiB Sequential Workloads

Network Baseline Results

Network connectivity tests used six concurrent iPerf3 instances running for one minute. Each iPerf3 instance on each server transmitted data to all other servers.

All storage nodes with 100 GbE NICs averaged 96+ Gb/s during testing. Monitor nodes and clients with 50 GbE NICs averaged 45+ Gb/s during testing.

Ceph Performance Results and Analysis

The results detailed below are based on a 2x replicated storage pool using version 3.3 of Red Hat Ceph Storage with 8192 placement groups.

Tests used 130 RBD images at 75GB each, providing 9.75TB of data on a 2x replicated pool (19.5TB of total data). Random write tests used a constant queue depth of 32, increasing the number of simultaneous clients from 10 to 130 in increments of 10. A queue depth of 32 simulated a reasonably active RDB image consumer and enabled tests to scale to a high number of clients.

Random read and 70/30 R/W tests executed an I/O load against all 130 RBD images, scaling up the queue depth per client from 1 to 32 in base-2 increments. Using130 clients for every test ensured that Ceph used the Linux filesystem cache equally on all tests.

For each I/O workload described below, five 10-minute tests executed with a five-minute ramp-up time for each test. The results reported in the sections below is the mathematical average of each five test pass.

Small Block Random Workload Testing

The following sections describe the resulting performance measured for random 4 KiB (4.0 x 210 byte) block read, write, and mixed read/write I/O tests.

4KiB 100% Random Write Workloads

Write performance reached a maximum of 645K 4 KiB IOPS. Average latency showed a linear increase as the number of clients increased, reaching a maximum average latency of 6.44ms at 130 clients. Tail (99.99%) latency increased smoothly across the entire 130 clients tested, reaching a maximum of 65.65ms at 130 clients (Figure 5).

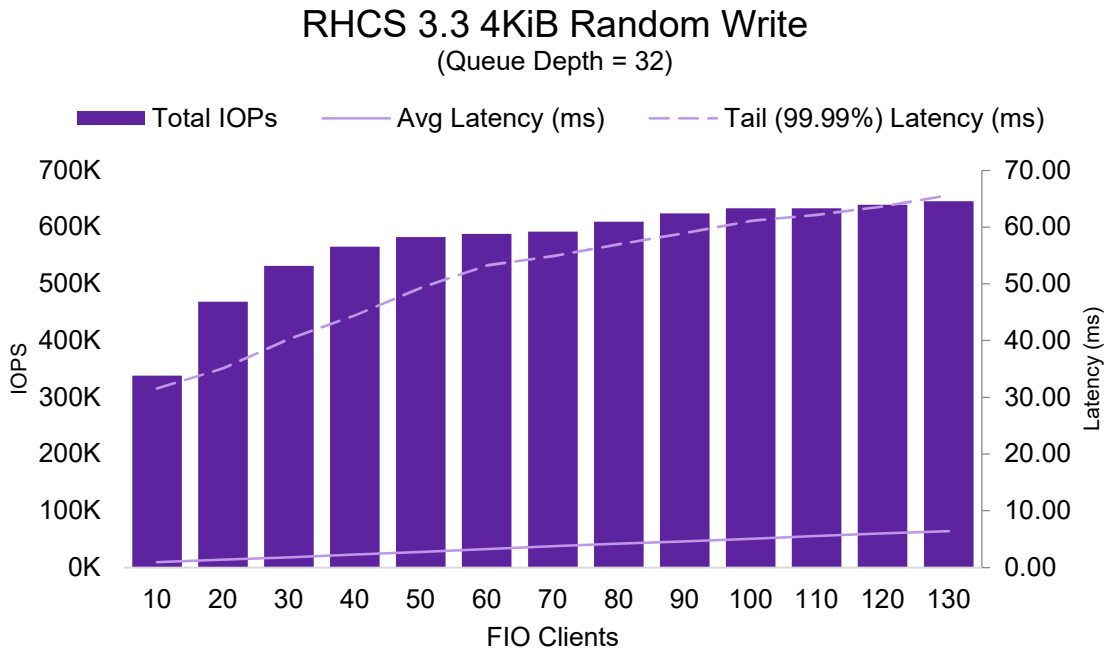


Figure 5 – 4 KiB Random Write IOPS vs Latency

Ceph data nodes depend heavily on CPU for performance. Low client load shows CPU utilization starting at 47% with 10 clients and increasing steadily to over 82% at a load of 130 clients (Figure 6).

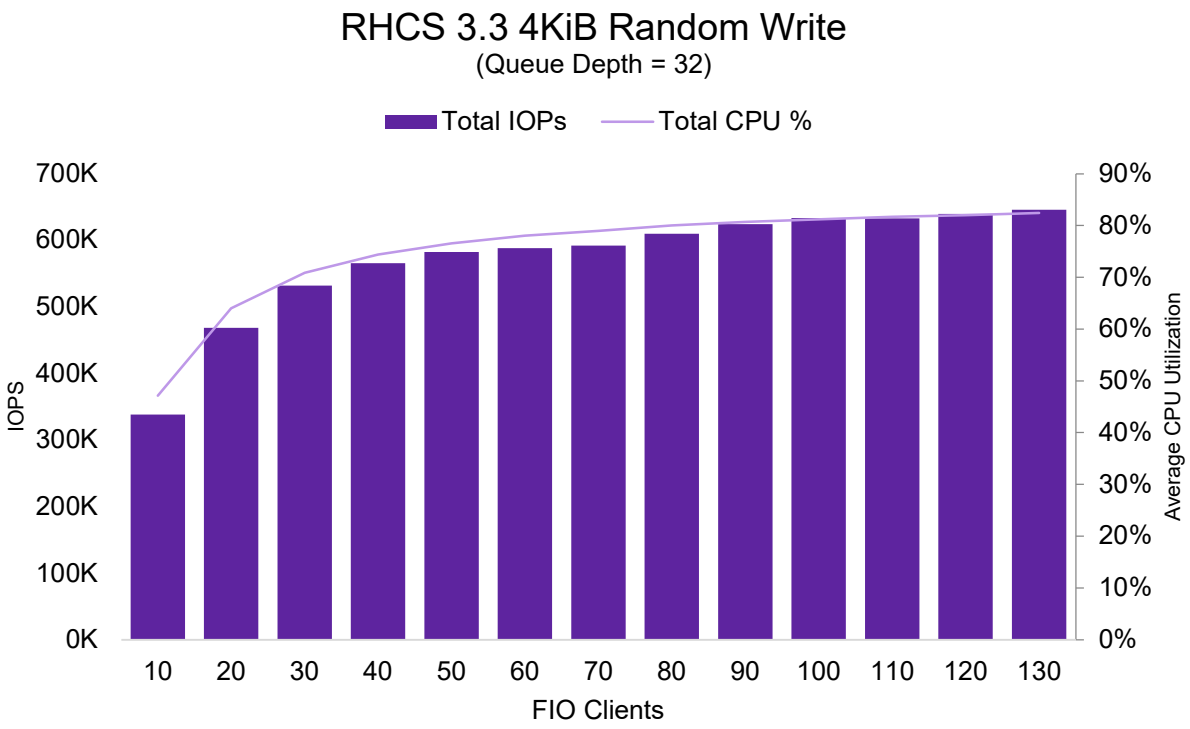


Figure 6 – 4 KiB Random Write IOPS vs. CPU Utilization

Table 9 summarize the solution's write performance. Based on the observed behavior, write-centric small-block workloads should focus on sizing for no more than 85% CPU utilization. The actual number of clients attained before reaching this level of utilization depends on the CPU model chosen. This RA's choice of an AMD EPYC 7742 CPU indicates this solution can scale to 130 clients.

FIO Clients	IOPS	Average Latency	95% Latency	99.99% Latency	Average CPU Utilization
10 Clients	338,374	0.94ms	1.21ms	31.58ms	47.16%
20 Clients	468,341	1.36ms	1.93ms	35.10ms	63.98%
30 Clients	531,714	1.80ms	2.83ms	40.31ms	70.84%
40 Clients	565,523	2.26ms	4.04ms	44.46ms	74.36%
50 Clients	582,308	2.74ms	5.38ms	49.32ms	76.51%
60 Clients	587,993	3.26ms	6.60ms	53.26ms	78.03%
70 Clients	591,883	3.78ms	7.79ms	54.91ms	78.96%
80 Clients	609,614	4.19ms	8.97ms	56.99ms	79.97%
90 Clients	624,014	4.61ms	10.04ms	58.98ms	80.66%
100 Clients	633,400	5.05ms	11.16ms	61.12ms	81.18%
110 Clients	632,990	5.55ms	12.33ms	62.20ms	81.66%
120 Clients	639,602	6.00ms	13.28ms	63.66ms	81.98%
130 Clients	645,655	6.44ms	14.33ms	65.65ms	82.42%

Table 9 – 4 KiB Random Write Results Summary

4 KiB Random Read Workload Analysis

Read performance of 130 FIO clients reached a maximum of 3.1 million 4 KiB IOPS. Average latency showed an increase as the queue depth increased, reaching a maximum average latency of only 1.3ms at queue depth 32. Tail (99.99%) latency increased steadily up to a queue depth of 16, then spiked upward at queue depth of 32, going from 15.8ms at queue depth of 16 to 101ms at queue depth of 32 — an increase of over 539% (Figure 7).

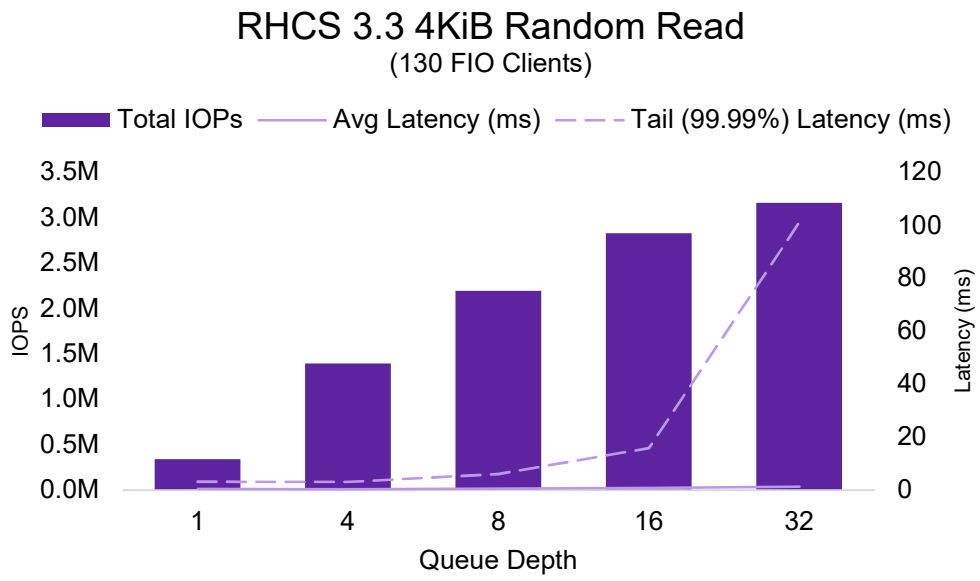


Figure 7- 4 KiB Random Read IOPS vs Latency

Low queue depth showed CPU utilization starting at 14% at queue depth of 1 and increasing steadily to over 89% at a queue depth of 32 (Figure 8).

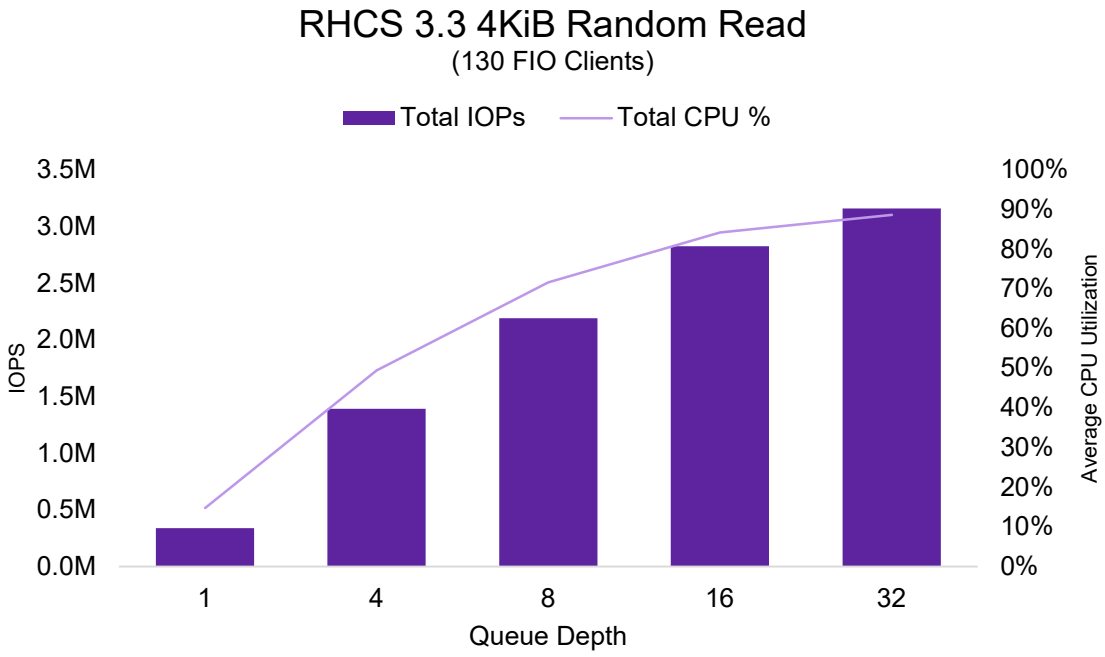


Figure 8 – 4 KiB Random Read IOPS vs. CPU Utilization

Table 10 summarize the solution's read performance. Based on the observed behavior, read-centric small-block workloads should focus on sizing for no more than 90% CPU utilization. The actual queue depth and client load attained before reaching this level of utilization will depend on the CPU model chosen. This RA's choice of an AMD EPYC 7742 CPU indicates the target sizing should be a queue depth of 16 for latency-sensitive use cases, and to a queue depth of 32 for use cases that are less latency sensitive.

FIO Clients	IOPS	Average Latency	95% Latency	99.99% Latency	Average CPU Utilization
QD 1	337,578	0.38ms	0.46ms	3.17ms	14.78%
QD 4	1,391,105	0.37s	0.48ms	3.07ms	49.43%
QD 8	2,190,725	0.47ms	0.68ms	6.04ms	71.63%
QD 16	2,827,109	0.73ms	1.29ms	15.79ms	84.28%
QD 32	3,161,146	1.31ms	3.19ms	101.07ms	88.72%

Table 10 – 4 KiB Random Read Results Summary

4KB Random 70% Read/30% Write Workload Analysis

Mixed read and write (70% read/30% write) performance of 130 FIO clients reached a maximum of 1.3 million 4 KiB IOPS. Both read and write average latency showed an increase as the queue depth increased, reaching a maximum average read latency of 1.82ms, and a maximum average write latency of 6.2ms at queue depth 32.

Tail (99.99%) latency for both reads and writes increased steadily as queue depth increased with read latency going from 6.81ms at queue depth of 1 to 45.4ms at queue depth of 32 — an increase of over 660% — and with write latency increasing from 29.6ms at queue depth 1 to 64.4ms at queue depth of 32 — an increase of 217% (Figure 9).

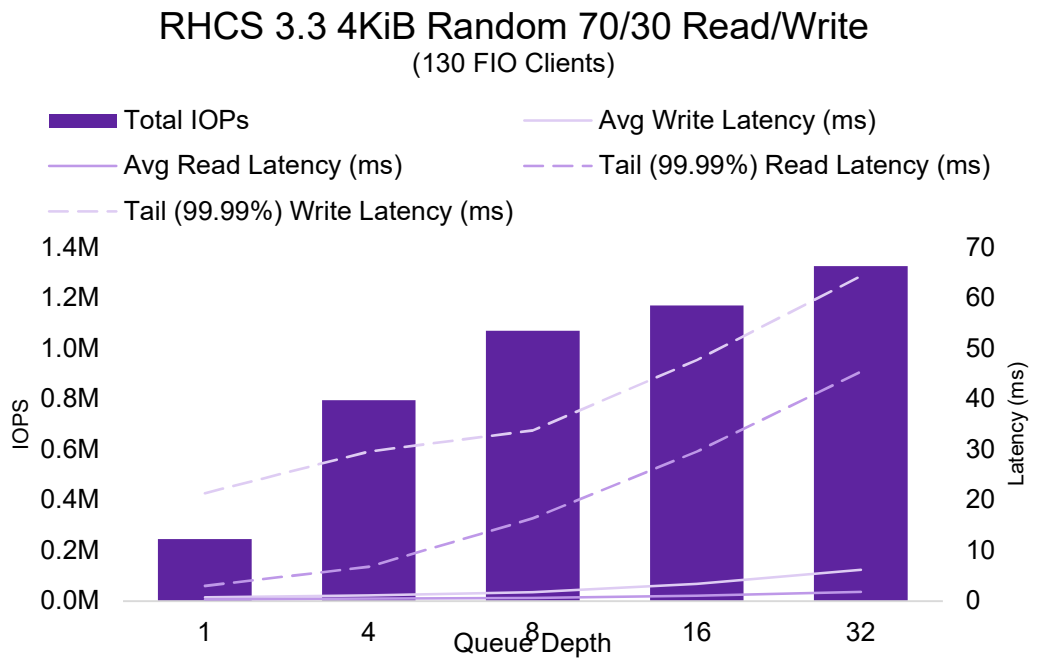


Figure 9 - 4KiB Random 70/30 Read/Write IOPS vs Latency

Low queue depth shows CPU utilization starting at 19.8% and increasing steadily to over 85.6% at a queue depth of 32 (Figure 10).

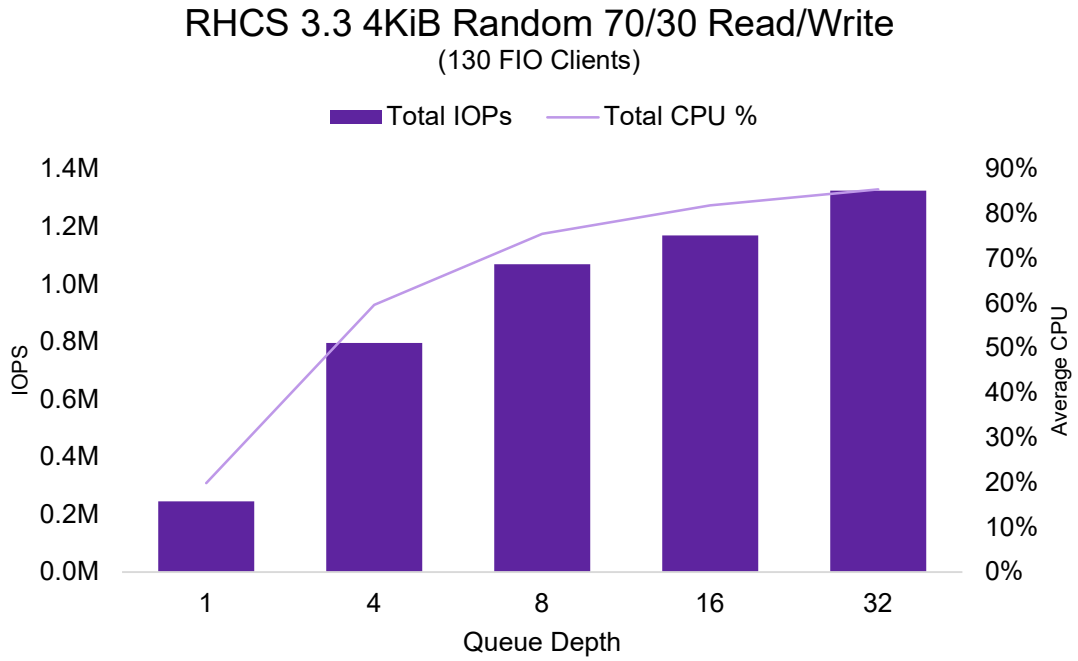


Figure 10 – 4 KiB Random 70/30 Read/Write IOPS vs CPU Utilization

Table 11 summarize the solution's 70%/30% read/write performance. Based on the observed behavior, mixed I/O small-block workloads should focus on sizing for no more than 85% CPU utilization. The actual queue depth and client load attained before reaching this level of utilization will depend on the CPU model chosen and the actual read/write ratio. This RA's choice of an AMD EPYC 7742 processor and 70%/30% read/write mix indicates the target sizing should be a queue depth of 32.

FIO Clients	IOPS	Avg. Read Latency	Avg. Write Latency	99.99% Read Latency	99.99% Write Latency	Avg. CPU Utilization
QD 1	245,801	0.79ms	0.41ms	21.36ms	3.02ms	19.88%
QD 4	795,608	1.09ms	0.46ms	29.61ms	6.81ms	59.67%
QD 8	1,070,094	1.75ms	0.63ms	33.78ms	16.4 ms	75.59%
QD 16	1,170,277	3.42ms	1.06ms	47.73ms	29.60ms	81.89%
QD 32	1,325,888	6.20ms	1.82ms	64.36ms	45.44ms	85.57%

Table 11 - 4 KiB Random 70/30 Read/Write Results Summary

4MB Object Workloads

The following sections describe the resulting performance measured for random, 4 MiB (4.0 x 220 byte) object read and write data in both sequential (read and write) and random (reads only) I/O scenarios.

Write tests measure performance using RADOS Bench workload instances consisting of a constant 16 threads per instance. The test increases load by instantiating additional RADOS Bench instances from two to 20 in base-2 increments.

Read tests executed measure performance by executing a fixed 10 RADOS Bench instances while increasing the number of threads from four to 32 in base-2 increments.

Object Write Workload Analysis

Object write performance reached a maximum throughput of 19.15 GiB/s with an average latency of 38.2ms at a workload level of 10 instances. Latency growth was consistent as workload increased to 10 instances, with a spike to 77ms at 20 instances—a nearly 2x increase—while overall throughput decreased (Figure 11).

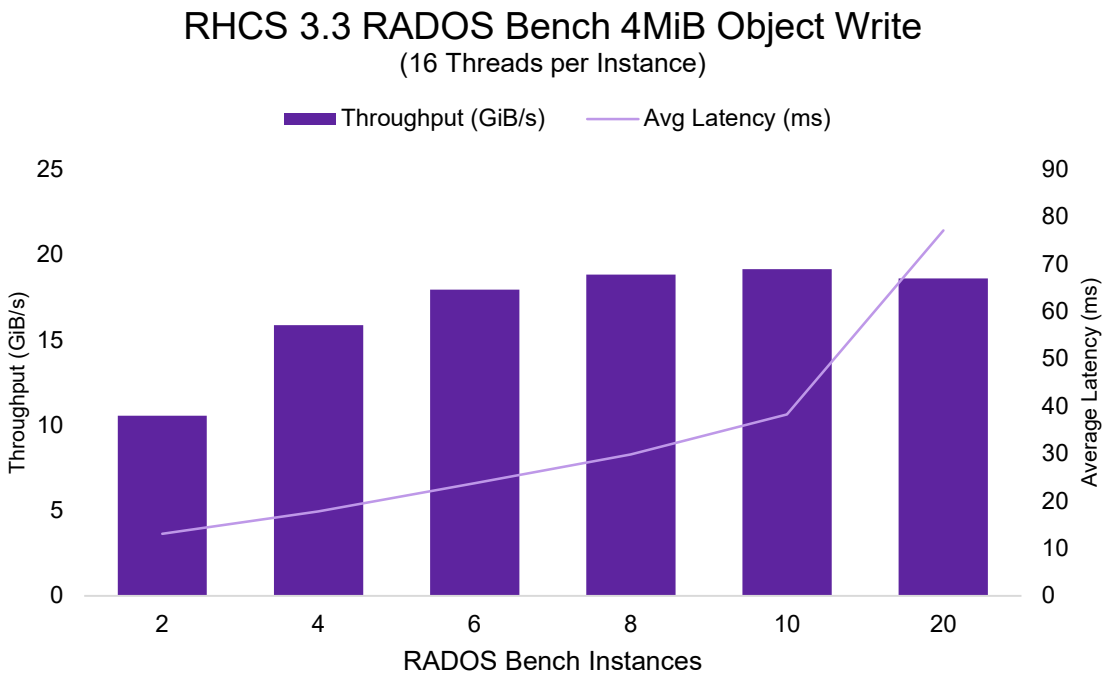


Figure 11 - 4MB Object Write Throughput vs. Average Latency

CPU utilization was extremely low for this test, indicating it may be possible to scope a lower power CPU for large-block, object-based use cases. Average CPU utilization for this RA never reached higher than 10% (Figure 12).

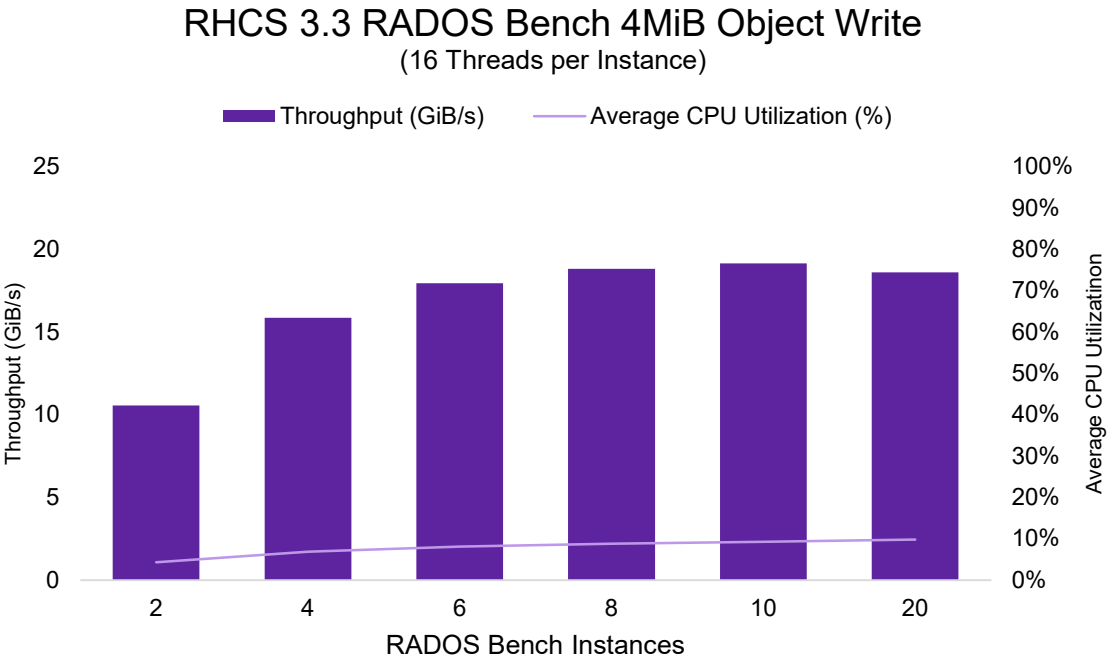


Figure 12 – 4 MiB Object Write Throughput vs. Average CPU Utilization

Table 12 summarizes the solution's object write performance. Based on the observed behavior, write-centric object workloads should focus on maximizing throughput. The actual queue depth and client load attained before reaching this level of throughput depends on the CPU model chosen. This RA's choice of an AMD EPYC 7742 CPU indicates the target sizing should be a in the range of 128 and 160 total threads for optimal performance.

Clients	Write Throughput	Average Latency	Average CPU Utilization
2 Instances	10.56 GiB/s	13.06ms	4.33%
4 Instances	15.87 GiB/s	17.78ms	6.88%
6 Instances	17.94 GiB/s	23.72ms	8.08%
8 Instances	18.83 GiB/s	29.82ms	8.81%
10 Instances	19.15 GiB/s	38.22ms	9.25%
20 Instances	18.61 GiB/s	77.11ms	9.80%

Table 12 - 4 MiB Object Write Results Summary

Object Read Workload Analysis

Object read performance reached a maximum sequential throughput of 41 GiB/s – 85% of the aggregated available bandwidth of the four-node cluster – with an average latency of 29.8ms attained at 32 threads, while maximum random throughput achieved 43.7 GB/s – 91% of the aggregated available bandwidth of the four-node cluster – with an average latency of 27.9ms at 32 threads. Sequential read performance was slightly lower at 16 threads (6% lower), but average latency was one-half the 29.8ms measured at 32 threads. Latency growth was consistent as workload increased, indicating that there were no apparent cases of resource constraints (Figure 13).

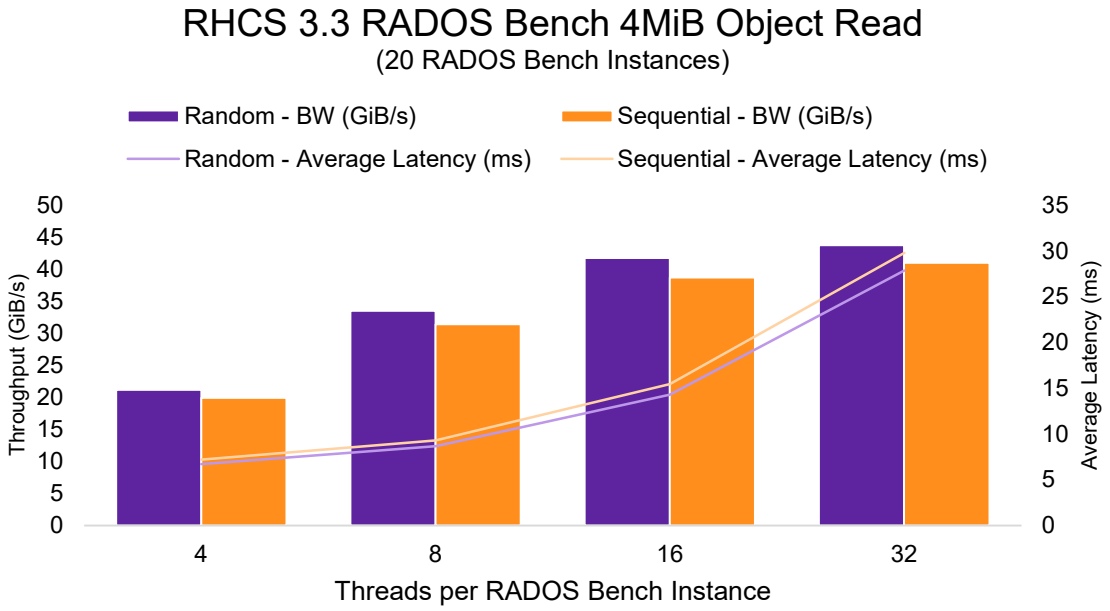


Figure 13 – 4 MiB Object Read Throughput vs Average Latency

CPU utilization was extremely low for this test. Average CPU utilization for this RA never exceeded higher than 14% (Figure 14).

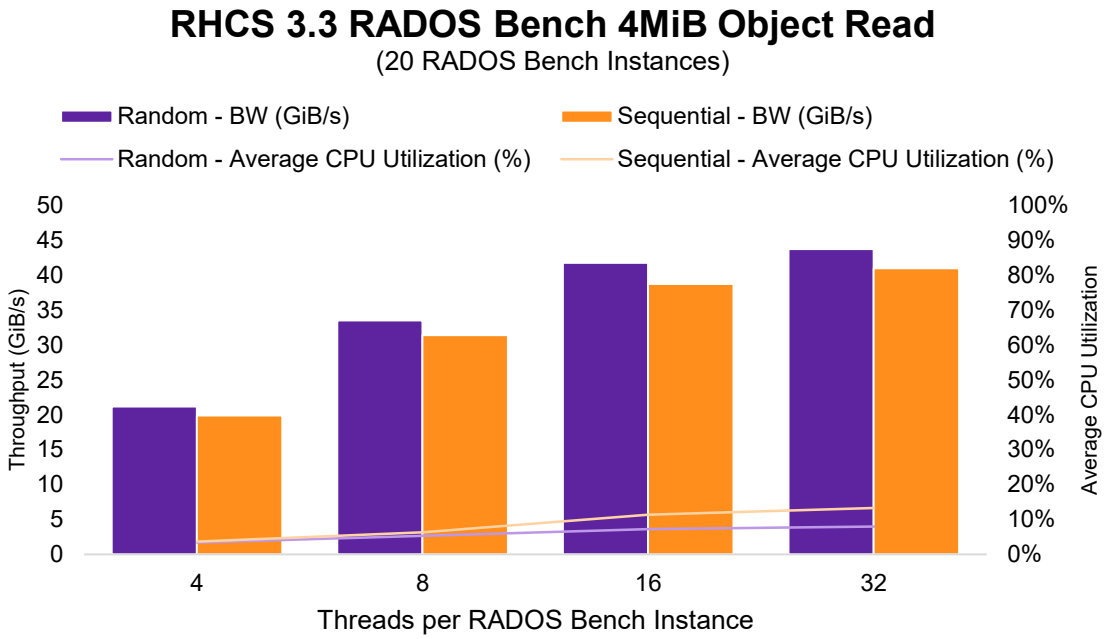


Figure 14 – 4 MiB Object Read Throughput vs. Average CPU Utilization

Table 13 summarizes the solution's object read performance. Based on the observed behavior, read-centric object workloads should focus on maximizing throughput. The actual queue depth and client load attained before reaching this level of throughput depends on the CPU model chosen. This RA's choice of an AMD EPYC 7742 CPU indicates the target sizing should be a in the range of 160 total threads for optimal performance. More threads may provide additional performance, but is beyond the scope of testing performed in this RA.

Threads per Instance	Random			Sequential		
	Throughput	Average Latency	Average CPU%	Throughput	Average Latency	Average CPU%
4	21.19 GiB/s	6.72ms	3.4%	19.92 GiB/s	7.21ms	3.7%
8	33.54 GiB/s	8.67ms	5.3%	31.41 GiB/s	9.30ms	6.3%
16	41.79 GiB/s	14.30ms	7.2%	38.73 GiB/s	15.47ms	11.3%
32	43.77 GiB/s	27.92ms	8.0%	41.03 GiB/s	29.81ms	13.3%

Table 13 – 4 MiB Object Read Results Summary

Summary

Micron designed this reference architecture for small block random workloads. With over 3 million 4 KiB random reads and 645,000 4 KiB random writes in a compact design based on four 1RU data nodes and one or more 1RU monitor nodes and 512GB of total storage, this is the most performant Ceph architecture we've tested to date.

While optimized for small-block workloads, this Ceph solution demonstrated excellent object performance as well as offering aggregated throughput of 90% of the available network bandwidth. Taking typical TCP/IP overhead into consideration, this solution as configured fully utilized the available throughput. Additional network interfaces installed in each storage node may support increased Ceph object throughput, though this hypothesis was not tested.

Micron's enterprise NVMe SSDs enable massive performance and provide a suitable solution for many different types of storage solutions, such as Red Hat Ceph Storage software-defined storage area networks. Whether you need to support general-purpose use cases or require ultra-fast responses for transactional workloads or large, fast data analytics solutions, Micron has taken the guesswork out of building the right solution.

Appendix A

Ceph.conf

```
[client]
rbd_cache = False
rbd_cache_writethrough_until_flush = False

# Please do not change this file directly since it is managed by Ansible and will be
overwritten
[global]
auth client required = none
auth cluster required = none
auth service required = none
auth supported = none
cephx require signatures = False
cephx sign messages = False
cluster network = 192.168.1.0/24
debug asok = 0/0
debug auth = 0/0
debug bluefs = 0/0
debug bluestore = 0/0
debug buffer = 0/0
debug client = 0/0
debug context = 0/0
debug crush = 0/0
debug filer = 0/0
debug filestore = 0/0
debug finisher = 0/0
debug hadoop = 0/0
debug heartbeatmap = 0/0
debug journal = 0/0
debug journaler = 0/0
debug lockdep = 0/0
debug log = 0
debug mds = 0/0
debug mds_balancer = 0/0
debug mds_locker = 0/0
debug mds_log = 0/0
debug mds_log_expire = 0/0
debug mds_migrator = 0/0
debug mon = 0/0
debug monc = 0/0
debug ms = 0/0
debug objclass = 0/0
debug objectcacher = 0/0
debug objecter = 0/0
debug optracker = 0/0
debug osd = 0/0
debug paxos = 0/0
debug perfcounter = 0/0
debug rados = 0/0
```

```
debug rbd = 0/0
debug rgw = 0/0
debug rocksdb = 0/0
debug throttle = 0/0
debug timer = 0/0
debug tp = 0/0
debug zs = 0/0
fsid = 36a9e9ee-a7b8-4c41-a3e5-0b575f289379
mon host = 192.168.0.203
mon pg warn max per osd = 800
mon_allow_pool_delete = True
mon_max_pg_per_osd = 800
ms type = async
ms_crc_data = False
ms_crc_header = True
osd objectstore = bluestore
osd_pool_default_size = 2
perf = True
public network = 192.168.0.0/24
rocksdb_perf = True

[mon]
mon_max_pool_pg_num = 166496
mon_osd_max_split_count = 10000

[osd]
bluestore_csum_type = none
bluestore_extent_map_shard_max_size = 200
bluestore_extent_map_shard_min_size = 50
bluestore_extent_map_shard_target_size = 100
osd memory target = 9465613516
osd_max_pg_log_entries = 10
osd_min_pg_log_entries = 10
osd_pg_log_dups_tracked = 10
osd_pg_log_trim_min = 10
```

Ceph-Ansible Configuration

All.yml

```
---
dummy:
fetch_directory: ~/ceph-ansible-keys
mon_group_name: mons
osd_group_name: osds
client_group_name: clients
mgr_group_name: mgrs
configure_firewall: False
ceph_repository_type: cdn
ceph_origin: repository
ceph_repository: rhcs
ceph_rhcs_version: 3
fsid: "36a9e9ee-a7b8-4c41-a3e5-0b575f289379"
generate_fsid: false
cephx: false
rbd_cache: "false"
rbd_cache_writethrough_until_flush: "false"
monitor_interface: enp99s0f1.501
public_network: 192.168.0.0/24
cluster_network: 192.168.1.0/24
osd_mkfs_type: xfs
osd_mkfs_options_xfs: -f -i size=2048
osd_mount_options_xfs: noatime,largeio,inode64,swalloc
osd_objectstore: bluestore
ceph_conf_overrides:
  global:
    auth client required: none
    auth cluster required: none
    auth service required: none
    auth supported: none
    osd objectstore: bluestore
    cephx require signatures: False
    cephx sign messages: False
    mon_allow_pool_delete: True
    mon_max_pg_per_osd: 800
    mon pg warn max per osd: 800
    ms_crc_header: True
    ms_crc_data: False
    ms type: async
    perf: True
    rocksdb_perf: True
    osd_pool_default_size: 2
    debug asok: 0/0
    debug auth: 0/0
    debug bluefs: 0/0
    debug bluestore: 0/0
```

```
    debug buffer: 0/0
    debug client: 0/0
    debug context: 0/0
    debug crush: 0/0
    debug filer: 0/0
    debug filestore: 0/0
    debug finisher: 0/0
    debug hadoop: 0/0
    debug heartbeatmap: 0/0
    debug journal: 0/0
    debug journaler: 0/0
    debug lockdep: 0/0
    debug log: 0
    debug mds: 0/0
    debug mds_balancer: 0/0
    debug mds_locker: 0/0
    debug mds_log: 0/0
    debug mds_log_expire: 0/0
    debug mds_migrator: 0/0
    debug mon: 0/0
    debug monc: 0/0
    debug ms: 0/0
    debug objclass: 0/0
    debug objectcacher: 0/0
    debug objecter: 0/0
    debug optracker: 0/0
    debug osd: 0/0
    debug paxos: 0/0
    debug perfcounter: 0/0
    debug rados: 0/0
    debug rbd: 0/0
    debug rgw: 0/0
    debug rocksdb: 0/0
    debug throttle: 0/0
    debug timer: 0/0
    debug tp: 0/0
    debug zs: 0/0
  mon:
    mon_max_pool_pg_num: 166496
    mon_osd_max_split_count: 10000
  client:
    rbd_cache: false
    rbd_cache_writethrough_until_flush: false
  osd:
    osd_min_pg_log_entries: 10
    osd_max_pg_log_entries: 10
    osd_pg_log_dups_tracked: 10
    osd_pg_log_trim_min: 10
    bluestore_csum_type: none
    bluestore_extent_map_shard_min_size: 50
```

```
    bluestore_extent_map_shard_max_size: 200
    bluestore_extent_map_shard_target_size: 100
disable_transparent_hugepage: "{{ false if osd_objectstore == 'bluestore' else true }}"
os_tuning_params:
  - { name: kernel.pid_max, value: 4194303 }
  - { name: fs.file-max, value: 26234859 }
  - { name: vm.zone_reclaim_mode, value: 0 }
  - { name: vm.swappiness, value: 1 }
  - { name: vm.min_free_kbytes, value: 1000000 }
  - { name: net.core.rmem_max, value: 268435456 }
  - { name: net.core.wmem_max, value: 268435456 }
  - { name: net.ipv4.tcp_rmem, value: 4096 87380 134217728 }
  - { name: net.ipv4.tcp_wmem, value: 4096 65536 134217728 }
ceph_tcmalloc_max_total_thread_cache: 134217728
ceph_docker_image: "rhceph/rhceph-3-rhel7"
ceph_docker_image_tag: "latest"
ceph_docker_registry: "registry.access.redhat.com"
containerized_deployment: False
```

Osds.yml

```
---
dummy:

osd_scenario: lvm

lvm_volumes:
  - data: data-lv1
    data_vg: vg_nvme0n1
    db: db-lv1
    db_vg: vg_nvme0n2
  - data: data-lv1
    data_vg: vg_nvme0n3
    db: db-lv1
    db_vg: vg_nvme0n4
  - data: data-lv1
    data_vg: vg_nvme1n1
    db: db-lv1
    db_vg: vg_nvme1n2
  - data: data-lv1
    data_vg: vg_nvme1n3
    db: db-lv1
    db_vg: vg_nvme1n4
  - data: data-lv1
    data_vg: vg_nvme3n1
    db: db-lv1
    db_vg: vg_nvme3n2
  - data: data-lv1
    data_vg: vg_nvme3n3
```

```
    db: db-lv1
    db_vg: vg_nvme3n4
  - data: data-lv1
    data_vg: vg_nvme4n1
    db: db-lv1
    db_vg: vg_nvme4n2
  - data: data-lv1
    data_vg: vg_nvme4n3
    db: db-lv1
    db_vg: vg_nvme4n4
  - data: data-lv1
    data_vg: vg_nvme5n1
    db: db-lv1
    db_vg: vg_nvme5n2
  - data: data-lv1
    data_vg: vg_nvme5n3
    db: db-lv1
    db_vg: vg_nvme5n4
  - data: data-lv1
    data_vg: vg_nvme6n1
    db: db-lv1
    db_vg: vg_nvme6n2
  - data: data-lv1
    data_vg: vg_nvme6n3
    db: db-lv1
    db_vg: vg_nvme6n4
  - data: data-lv1
    data_vg: vg_nvme7n1
    db: db-lv1
    db_vg: vg_nvme7n2
  - data: data-lv1
    data_vg: vg_nvme7n3
    db: db-lv1
    db_vg: vg_nvme7n4
  - data: data-lv1
    data_vg: vg_nvme8n1
    db: db-lv1
    db_vg: vg_nvme8n2
  - data: data-lv1
    data_vg: vg_nvme8n3
    db: db-lv1
    db_vg: vg_nvme8n4
  - data: data-lv1
    data_vg: vg_nvme9n1
    db: db-lv1
    db_vg: vg_nvme9n2
  - data: data-lv1
    data_vg: vg_nvme9n3
    db: db-lv1
    db_vg: vg_nvme9n4
```

```
  - data: data-lv1
    data_vg: vg_nvme10n1
    db: db-lv1
    db_vg: vg_nvme10n2
  - data: data-lv1
    data_vg: vg_nvme10n3
    db: db-lv1
    db_vg: vg_nvme10n4
  - data: data-lv1
    data_vg: vg_nvme0n5
    db: db-lv1
    db_vg: vg_nvme0n6
  - data: data-lv1
    data_vg: vg_nvme0n7
    db: db-lv1
    db_vg: vg_nvme0n8
  - data: data-lv1
    data_vg: vg_nvme1n5
    db: db-lv1
    db_vg: vg_nvme1n6
  - data: data-lv1
    data_vg: vg_nvme1n7
    db: db-lv1
    db_vg: vg_nvme1n8
  - data: data-lv1
    data_vg: vg_nvme3n5
    db: db-lv1
    db_vg: vg_nvme3n6
  - data: data-lv1
    data_vg: vg_nvme3n7
    db: db-lv1
    db_vg: vg_nvme3n8
  - data: data-lv1
    data_vg: vg_nvme4n5
    db: db-lv1
    db_vg: vg_nvme4n6
  - data: data-lv1
    data_vg: vg_nvme4n7
    db: db-lv1
    db_vg: vg_nvme4n8
  - data: data-lv1
    data_vg: vg_nvme5n5
    db: db-lv1
    db_vg: vg_nvme5n6
  - data: data-lv1
    data_vg: vg_nvme5n7
    db: db-lv1
    db_vg: vg_nvme5n8
  - data: data-lv1
    data_vg: vg_nvme6n5
```

```
    db: db-lv1
    db_vg: vg_nvme6n6
  - data: data-lv1
    data_vg: vg_nvme6n7
    db: db-lv1
    db_vg: vg_nvme6n8
  - data: data-lv1
    data_vg: vg_nvme7n5
    db: db-lv1
    db_vg: vg_nvme7n6
  - data: data-lv1
    data_vg: vg_nvme7n7
    db: db-lv1
    db_vg: vg_nvme7n8
  - data: data-lv1
    data_vg: vg_nvme8n5
    db: db-lv1
    db_vg: vg_nvme8n6
  - data: data-lv1
    data_vg: vg_nvme8n7
    db: db-lv1
    db_vg: vg_nvme8n8
  - data: data-lv1
    data_vg: vg_nvme9n5
    db: db-lv1
    db_vg: vg_nvme9n6
  - data: data-lv1
    data_vg: vg_nvme9n7
    db: db-lv1
    db_vg: vg_nvme9n8
  - data: data-lv1
    data_vg: vg_nvme10n5
    db: db-lv1
    db_vg: vg_nvme10n6
  - data: data-lv1
    data_vg: vg_nvme10n7
    db: db-lv1
    db_vg: vg_nvme10n8
```

Creating Multiple Namespaces

The Ceph recommendation for the volume storing the OSD database is no less than 4% of the size of the OSD data volume, which is the number that we used.

```
python3.6 create_namespaces.py -ns 6001169368 250048724 6001169368 250048724
 6001169368 250048724 6001169368 250048724 -ls 512 -d /dev/nvme0 /dev/nvme1 /dev/nvme3
/dev/nvme4 /dev/nvme5 /dev/nvme6 /dev/nvme7 /dev/nvme8 /dev/nvme9 /dev/nvme10
```

create_namespaces.py

```
import argparse
import time
from subprocess import Popen, PIPE


def parse_arguments():
    parser = argparse.ArgumentParser(description='This file creates namespaces across
NVMe devices')
    parser.add_argument('-ns', '--namespace-size', nargs='+', required=True, type=str,
                        help='List of size of each namespace in number of LBAs.
Specifying more than one will create '
                             'multiple namespaces on each device.')
    parser.add_argument('-ls', '--lba-size', default='512', choices=['512', '4096'],
required=False, type=str,
                        help='Size of LBA in bytes. Valid options are 512 and 4096
(Default: 512)')
    parser.add_argument('-d', '--devices', nargs='+', required=True, type=str,
                        help='List of data devices to create OSDs on.')

    return {k: v for k, v in vars(parser.parse_args()).items()}


def execute_command(cmd):
    process = Popen(cmd, stdout=PIPE, stderr=PIPE)
    stdout, stderr = process.communicate()
    stdout = stdout.decode()
    stderr = stderr.decode()

    if stderr not in ('', None):
        print(stdout)
        raise Exception(stderr)
    else:
        return stdout


def remove_namespaces(devices, **_):
    for dev in devices:
        cmd = ['nvme', 'list-ns', dev]
        namespaces = [int(value[value.find('[') + 1:-value.find(']')].strip())
                      for value in execute_command(cmd=cmd).strip().split('\n')]
```

```
        for namespace in namespaces:
            cmd = ['nvme', 'detach-ns', dev, '-n', str(namespace+1), '-c', '1']
            print(execute_command(cmd=cmd))

            cmd = ['nvme', 'delete-ns', dev, '-n', str(namespace+1)]
            print(execute_command(cmd=cmd))
            time.sleep(0.1)


def create_namespaces(devices, namespace_size, lba_size):
    lba_key = '2' if lba_size == '4096' else '0'
    for dev in devices:
        for size in namespace_size:
            cmd = ['nvme', 'create-ns', dev, '-f', lba_key, '-s', size, '-c', size]
            print(execute_command(cmd=cmd))

    attach_namespaces(devices=devices, num_namespaces=len(namespace_size))


def attach_namespaces(devices, num_namespaces):
    for namespace in range(1, num_namespaces+1):
        for dev in devices:
            cmd = ['nvme', 'attach-ns', dev, '-n', str(namespace), '-c', '1']
            print(execute_command(cmd=cmd))
            time.sleep(1)


def run_test():
    arguments = parse_arguments()

    remove_namespaces(**arguments)
    create_namespaces(**arguments)


if __name__ == '__main__':
    run_test()
```

Partitioning Drives for OSDs

```
python3.6 create_ceph_osd_partitions.py -o 1 -d /dev/nvme0n1 /dev/nvme1n1 /dev/nvme3n1
/dev/nvme4n1 /dev/nvme5n1 /dev/nvme6n1 /dev/nvme7n1 /dev/nvme8n1 /dev/nvme9n1
/dev/nvme10n1 /dev/nvme0n3 /dev/nvme1n3 /dev/nvme3n3 /dev/nvme4n3 /dev/nvme5n3
/dev/nvme6n3 /dev/nvme7n3 /dev/nvme8n3 /dev/nvme9n3 /dev/nvme10n3 /dev/nvme0n5
/dev/nvme1n5 /dev/nvme3n5 /dev/nvme4n5 /dev/nvme5n5 /dev/nvme6n5 /dev/nvme7n5
/dev/nvme8n5 /dev/nvme9n5 /dev/nvme10n5 /dev/nvme0n7 /dev/nvme1n7 /dev/nvme3n7
/dev/nvme4n7 /dev/nvme5n7 /dev/nvme6n7 /dev/nvme7n7 /dev/nvme8n7 /dev/nvme9n7
/dev/nvme10n7 -c /dev/nvme0n2 /dev/nvme1n2 /dev/nvme3n2 /dev/nvme4n2 /dev/nvme5n2
/dev/nvme6n2 /dev/nvme7n2 /dev/nvme8n2 /dev/nvme9n2 /dev/nvme10n2 /dev/nvme0n4
/dev/nvme1n4 /dev/nvme3n4 /dev/nvme4n4 /dev/nvme5n4 /dev/nvme6n4 /dev/nvme7n4
/dev/nvme8n4 /dev/nvme9n4 /dev/nvme10n4 /dev/nvme0n6 /dev/nvme1n6 /dev/nvme3n6
/dev/nvme4n6 /dev/nvme5n6 /dev/nvme6n6 /dev/nvme7n6 /dev/nvme8n6 /dev/nvme9n6
/dev/nvme10n6 /dev/nvme0n8 /dev/nvme1n8 /dev/nvme3n8 /dev/nvme4n8 /dev/nvme5n8
/dev/nvme6n8 /dev/nvme7n8 /dev/nvme8n8 /dev/nvme9n8 /dev/nvme10n8
The value of 1 was used for osds-per-device because we used multiple namespaces on the
same physical device. Each namespace only has 1 OSD.
create_ceph_osd_partitions.py
import argparse
import os
from subprocess import Popen, PIPE


class NoVGError(Exception):
    pass


class NoPVError(Exception):
    pass


def parse_arguments():
    parser = argparse.ArgumentParser(description='This file partitions devices for ceph
storage deployment')
    parser.add_argument('-o', '--osds-per-device', required=True, type=int, help='Number
of OSDs per data device')
    parser.add_argument('-d', '--data-devices', nargs='+', required=True, type=str,
                        help='List of data devices to create OSDs on.')
    parser.add_argument('-c', '--cache-devices', nargs='+', required=False, type=str,
                        help='Cache devices to store BlueStore RocksDB and write-ahead
log')
    parser.add_argument('-ws', '--wal-sz', required=False, type=int,
                        help='Size of each write-ahead log on specified cache devices in
GiB')
    parser.add_argument('-dnr', '--do-not-remove', action='store_true',
                        help='Do not remove old volumes (Disabled by default)')
    parser.add_argument('-dnc', '--do-not-create', action='store_true',
                        help='Do not create new volumes (Disabled by default)')
```

```
    return {k: v for k, v in vars(parser.parse_args()).items()}


def execute_command(cmd):
    process = Popen(cmd, stdout=PIPE, stderr=PIPE)
    stdout, stderr = process.communicate()

    if stderr not in ('', None, b''):
        print(stdout.decode())
        if b'Volume group' in stderr and b'not found' in stderr:
            raise NoVGError(stderr.decode())
        elif b'No PV found on device' in stderr:
            raise NoPVError(stderr.decode())
        else:
            raise Exception(stderr.decode())
    else:
        return stdout.decode()


def remove_lvm_volumes(data_devices, cache_devices, **_):
    dev_path = '/dev/'

    if cache_devices:
        device_list = data_devices + cache_devices
    else:
        device_list = data_devices

    vg_list = [('vg_{}'.format(device[len(dev_path):]), device) for device in
device_list]

    for vg, device in vg_list:
        vg_path = os.path.join(dev_path, vg)

        # Remove Logical Volumes
        try:
            for item in os.listdir(vg_path):
                cmd = ['lvremove', '-y', os.path.join(vg_path, item)]
                print(execute_command(cmd=cmd))
        except OSError as e:
            if e.errno == 2:
                pass
            else:
                raise e
        try:
            # Remove Volume Group
            cmd = ['vgremove', '-y', vg]
            print(f'Attempting to remove volume group {vg} for device {device}')
            print(execute_command(cmd=cmd))
        except NoVGError:
```

```
            print(f'No volume group found for device {device}')
            pass

        try:
            # Remove Physical Volume
            cmd = ['pvremove', '-y', device]
            print(f'Attempting to remove physical volume for device {device}')
            print(execute_command(cmd=cmd))
        except NoPVError:
            print(f'No physical volume found for device {device}')
            pass

        # Wipe FS
        cmd = ['nvme', 'format', device, '-s', '1']
        print(f'Secure erasing device {device}')
        print(execute_command(cmd=cmd))
        #
        # # Create GPT
        # cmd = ['sudo', 'parted', device, '-s', 'mklabel', 'gpt']
        # print(f'Creating GPT label on device {device}')
        # print(execute_command(cmd=cmd))


def create_partitions(data_devices, osds_per_device, cache_devices, wal_sz, **_):
    # Create cache partitions
    if cache_devices:
        print('Creating cache device partitions')
        db_partitions = len(data_devices) * osds_per_device // len(cache_devices)
        create_cache_device_volumes(cache_devices=cache_devices, wal_sz=wal_sz,
db_partitions=db_partitions)

    # Create data partitions
    print('Creating data partitions')
    create_data_device_volumes(data_devices=data_devices,
osds_per_device=osds_per_device)


def create_cache_device_volumes(cache_devices, wal_sz, db_partitions):
    for dev in cache_devices:
        cmd = ['pvcreate', dev]
        print(execute_command(cmd=cmd))

        vg_name = 'vg_{}'.format(os.path.basename(dev))
        cmd = ['vgcreate', vg_name, dev]
        print(execute_command(cmd=cmd))

        gb_total = get_total_size(vg_name=vg_name)

        # If WAL was given
        if not wal_sz:
            wal_sz = 0
```

```
        sz_per_db = (gb_total // db_partitions) - wal_sz

        for i in range(1, db_partitions+1):
            cmd = ['lvcreate', '-y', '--name', 'db-lv{}'.format(i), '--size',
'{}G'.format(sz_per_db), vg_name]
            print(execute_command(cmd=cmd))
            if wal_sz:
                cmd = ['lvcreate', '-y', '--name', 'wal-lv{}'.format(i), '--size',
'{}G'.format(wal_sz), vg_name]
                print(execute_command(cmd=cmd))


def create_data_device_volumes(data_devices, osds_per_device):
    for dev in data_devices:
        cmd = ['pvcreate', dev]
        print(f'Creating LVM physical volume on device {dev}')
        print(execute_command(cmd=cmd))

        vg_name = 'vg_{}'.format(os.path.basename(dev))
        cmd = ['vgcreate', vg_name, dev]
        print(f'Creating LVM volume group {vg_name} on {dev}')
        print(execute_command(cmd=cmd))

        gb_total = get_total_size(vg_name=vg_name)

        sz_per_osd = gb_total // osds_per_device

        for i in range(1, osds_per_device+1):
            cmd = ['lvcreate', '-y', '--name', f'data-lv{i}', '--size',
f'{sz_per_osd}G', vg_name]
            print(f'Creating {sz_per_osd}G LVM logical volume data-lv{i} on volume group
{vg_name}')
            print(execute_command(cmd=cmd))


def get_total_size(vg_name):
    cmd = ['vgdisplay', vg_name]
    stdout = execute_command(cmd=cmd)

    total_pe = 0
    pe_size = 0

    for line in stdout.split('\n'):
        if 'Total PE' in line:
            total_pe = int(line.split()[2])
        elif 'PE Size' in line:
            pe_size = int(float(line.split()[2]))

    gb_total = total_pe * pe_size // 1024
```

```
    if gb_total != 0:
        return gb_total
    else:
        raise ValueError(f'Issue found when displaying volume groups. Total
PE:{total_pe}\tPE Size: {pe_size}')


def run_test():
    arguments = parse_arguments()

    if not arguments['do_not_remove']:
        # Remove All Old LVM Volumes
        remove_lvm_volumes(**arguments)

    if not arguments['do_not_create']:
        create_partitions(**arguments)


if __name__ == '__main__':
    run_test()
```

About Micron

Micron Technology (Nasdaq: MU) is a world leader in innovative memory solutions. Through our global brands — Micron, Crucial® and Ballistix® — our broad portfolio of high-performance memory technologies, including DRAM, NAND, NOR Flash and 3D XPoint™ memory, is transforming how the world uses information to enrich life. Backed by 40 years of technology leadership, our memory and storage solutions enable disruptive trends, including artificial intelligence, machine learning and autonomous vehicles, in key market segments like data center, networking, automotive, industrial, mobile, graphics and client. Our common stock trades on the Nasdaq under the symbol MU.

About Red Hat

Red Hat is the world's leading provider of open source software solutions, using a community-powered approach to provide reliable and high-performing cloud, Linux, middleware, storage, and virtualization technologies. Red Hat also offers award-winning support, training, and consulting services. As a connective hub in a global network of enterprises, partners, and open source communities, Red Hat helps create relevant, innovative technologies that liberate resources for growth and prepare customers for the future of IT.

About Ceph Storage

Ceph is an open source distributed object store and file system designed to provide excellent performance, reliability, and scalability. It can:

- Free you from the expensive lock-in of proprietary, hardware-based storage solutions.
- Consolidate labor and storage costs into one versatile solution.
- Introduce cost-effective scalability on self-healing clusters based on standard servers and disks

Rev. A 12/19, CCM004-676576390-11401