A PERFORMANCE AND STORAGE EVALUATION OF LIGHTWEIGHT CONTAINERIZATION WITH NIXOS

By

Matthew Kenigsberg

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Computer Science

May 14, 2021

Nashville, Tennessee

Approved:

Aniruddha Gokhale, Ph.D.

Yogesh Barve, Ph.D.

ACKNOWLEDGMENTS

Thank you to my advisor, Aniruddha Gokhale, for walking me through every step of writing this thesis.

Thank you to Yogesh Barve for being my second reader.

Thank you to Matthew Leon for indulging my never-ending desire to talk about containerization and for being willing to work with me. I miss you.

Thank you to Waldek Kozaczuk for helping me so much with OSv.

Thank you to everyone who suggested improvements and edits to this paper.

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The philosophy of containerizing software could be summarized as follows: put all the tools ("dependencies") you might need to run a piece of software in a box ("container"). Make as many copies of that box as needed. Send that box anywhere anyone wants to run the software ("cloud computing" or "HPC").

This approach lacks elegance. Sometimes unnecessary dependencies are included in a container, or two different containers include the same dependency. Even in a best case scenario, containers duplicate tools most operating systems already have installed. Every time a container with extra dependencies is copied, resources are wasted.

Recent research has presented unikernels as a lightweight alternative to containerization, but unikernels suffer from some of the same weaknesses as containers. Unikernels attempt to eliminate overhead by only supporting the functions required by a single application, which makes them extremely small. Although this does eliminate overhead when compared to traditional virtual machines (VMs), unikernels compound the overhead induced by containers; unikernels include not only dependencies but also the kernel needed by an application. Although unikernels are more lightweight than VMs, they waste even more resources than containers.

Containerization and unikernels do solve some of the primary challenges of cloud computing and high performance computing (HPC). In both of these fields, software must be decoupled from hardware, allowing it to run on whatever hardware is avail-

able. Containers and unikernels both enable such portability, because containers and unikernel images can easily be copied and can be run on any platform that supports containerization or virtualization. Since containers and unikernels provide all the dependencies an application needs to run, behavior on any server will be the same. As already stated, however, both approaches sacrifice storage efficiency in order to allow portability.

This paper examines an experimental form of containerization based on the Nix package manager and NixOS operating system. NixOS containerization provides scalability and dependability just like normal containerization and unikernels, but it eliminates the duplication inherent in those approaches. More specifically, this paper shows the following. NixOS containerization:

- satisfies the requirements of cloud computing and HPC for portability and reproducibility
- guarantees the smallest possible container image size
- matches or exceeds performance of containerization and unikernels
- reduces deployment size for commonly used applications by 30% to 90%

CHAPTER 2

Problem Requirements

2.1 Use Cases

This paper focuses on deployment requirements for cloud computing and HPC, which are both driven by a need for portability and reproducibility. Prior research comparing unikernels to containerization has focused on cloud computing, but since containerization is commonly used in HPC, HPC would benefit from an improvement to containerization architecture as well. Many current containerization solutions are tailored to fit either cloud computing or HPC, but other solutions have begun to cater to both fields. For example, Podman, the containerization platform used in this paper, was originally designed for use in the cloud, but it has become more popular in HPC due to its support for rootless containers and Message Passing Interface [1]. Ideally, a good software deployment solution would provide the features needed for both use cases.

Cloud computing and HPC do have slightly different requirements, which are as follows:

- Cloud: An administrator runs software, typically microservices, on abstracted hardware. The hardware could be a combination of servers in different geographical locations, and computing power must be elastic, allowing rapidly increasing or decreasing usage of physical resources. Both cloud providers and consumers must be able to monitor the amount of resources consumed [2].

Scalability is achieved through orchestration.

- HPC: An untrusted user needs access to servers to run a workload [3]. Applications and user data must be portable so they can be migrated to different hardware, and applications must run in isolation from other untrusted applications [1]. The amount of resources given to the user must be controlled.

Although cloud and HPC have differing requirements, different solutions are unnecessary. Regarding node-specific operating system requirements, the primary difference between cloud and HPC containerization is whether or not a trusted user starts the container. HPC containerization solutions, such as Singularity, give this as one of the main reasons for their development [3]. Cloud computing solutions like Podman, however, now support rootless containers, which would make having a separate HPC solution unnecessary [1].

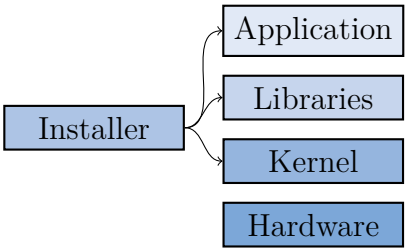
Note that this analysis is restricted to the requirements for running a program on a worker node; this paper does not address larger architectural requirements for networking and coordination. For example, cloud computing needs an orchestrator like Kubernetes, and HPC needs a scheduler such as SLURM. Assuming a node supports networking and remote administration, it should be able to support orchestration or workload management.

At the node level, cloud computing and HPC have very similar requirements: they both need to easily run an application on any hardware, while monitoring its resource usage and isolating it from other applications. Cloud computing and HPC would both benefit from a lightweight architecture that meets these needs.

2.2 Minimal Multitasking Architecture

The software requirements of cloud computing and HPC must be met by a platform compatible with current computer architecture. A high-level view of the most minimal architecture possible with a multitasking kernel is shown in Figure 2.1, and this will be referred to as minimal architecture.

Figure 2.1: Minimal Multitasking Architecture



Minimal architecture has only five components:

- The application is ultimately the goal of software deployment.
- Libraries provide functions needed by the application.
- A kernel provides resource usage monitoring and isolation by managing shared usage of the same hardware by multiple applications.
- The installer both installs and starts the application.
- Hardware is needed to run any software at all.

Installer is used in a very general sense, and it refers to the entirety of the component running the application; for example, it could refer to an entire OS that includes a package manager to install applications and an init system to start them.

In cloud computing, the installer role is fulfilled by the node agent. For Kubernetes, this would be kubelet, which spawns and manages containers. Currently, in most cases, the kernel installer is separate from the library and program installer.

Although there could be scenarios in cloud computing or HPC where a multitasking kernel would not be necessary, truly elastic scalability would allow subdividing a server between multiple applications. For this reason, all the solutions considered in this paper assume a single physical server should be able to run multiple isolated applications. Running a single application per physical server is considered in Section 6.3.

As already mentioned, while this analysis does not address networking, it is compatible with networking. For example, the installer component can be compared to components of orchestration. In Kubernetes, kubelet is the node agent responsible for communicating with the orchestrator, and it starts containers on the local node. This corresponds to the installer in Figure 2.1. The only other additional component on a node in Kubernetes is k-proxy, which is responsible for networking [4]. So long as the kernel supports networking, a node using the architecture of Figure 2.1 would be compatible with orchestration.

A lightweight architecture for cloud computing and HPC would provide portability without adding components other than those necessary for a multitasking server: the application, libraries, kernel, installer, and hardware.

CHAPTER 3

Solutions

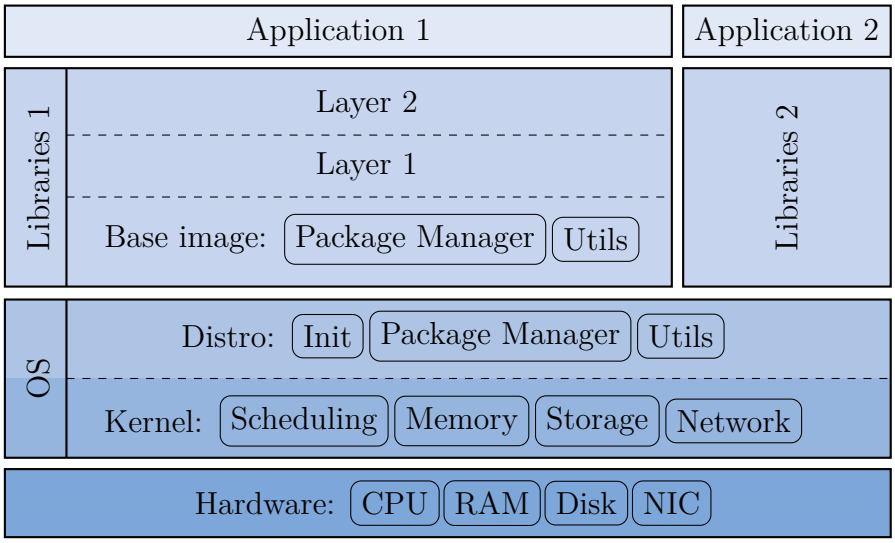
Although current solutions enable portable software deployment using the components of the minimal architecture of Figure 2.1, they add unnecessary components. NixOS containerization allows the same portability but uses only the minimal components and guarantees the smallest possible application image size.

3.1 Layer-based Containerization: Podman

The most common current deployment approach for microservices in the cloud is layer-based containerization. Each component of the software stack is provided in a layer, where each layer provides an all-purpose foundation for any possible software layered on top as shown in Figure 3.1. Most of the layers have a one-to-one correspondence to components in minimal architecture, although the OS layer provides both the kernel and the installer. Layers are shared between different containers whenever possible, but they can only be shared when multiple containers need the exact same layer.

Layer-based containerization adds extra components to minimal architecture in three possible ways. First, unneeded libraries can be included in the library layer. Because container layers can contain multiple libraries, it is possible to use a container layer to provide one dependency and indirectly include other libraries. Second, libraries can be duplicated across the library layer for multiple containers when those containers contain the same package. Even if both containers use the exact same

Figure 3.1: Layer-based Containerization Approach



version of a dependency, the dependency could be provided by two different layers, because those layers might contain other non-identical information. Third, libraries can be duplicated between the library and installer components. Any shared dependencies between containers and the host OS are duplicated, because layer-based containerization does not share layers with the host OS. As shown in Figure 3.1, at a minimum this will result in having a copy of a package manager and utilities in both the library component and the installer. Containerization does not manage libraries efficiently, duplicating or even adding completely unneeded libraries.

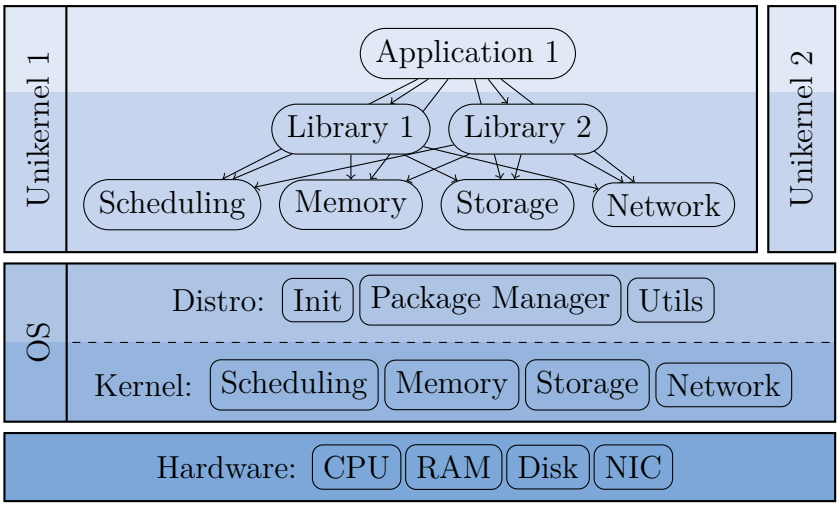
For the experiments in this paper, Podman was chosen as the containerization platform because it is daemonless and allows rootless containers. Docker enabled rootless mode only recently (December of 2020) and still requires a daemon [5]. Eliminating a daemon allows a more lightweight architecture, and rootless mode allows running as an untrusted user, which is needed for HPC. For these reasons, Podman

better satisfies the goal of a lightweight architecture for both cloud computing and HPC.

3.2 Unikernel: OSv

Another solution in recent research is unikernels. Unikernels are minimalist kernels that get packaged with an application, which provide the isolation and scalability of VMs without the overhead of running a large guest kernel such as Linux. Unikernel architecture is shown in Figure 3.2.

Figure 3.2: Unikernel Architecture



Compared to minimal architecture, unikernels have multiple kernel components instead of one. Although unikernels eliminate the bloat of a second full-featured kernel, they still introduce duplication since every application must start another kernel. Hypothetically, a unikernel could be written to run on bare-metal, but current stable implementations require running on top of a hypervisor, which must perform kernel features such as scheduling and memory management.

Unikernels fit into two categories [6]. The first is library operating systems, where, for example, the operating system is a C++ library that can be included in a C++ file and compiled into the resulting program. Since they are compiled with a program, library operating systems are language specific, and some examples are IncludeOS (C/C++), MirageOS (OCaml), and HaLVM (Haskell). The second type of unikernels are much more generalized and can run any ELF executable. Some examples include RumpRun, Nanos, and OSv. Supporting any ELF executable eliminates the need to compile the unikernel into the program being run, so these unikernels are not language specific.

For the experiments in this paper, OSv was chosen as the unikernel platform. Library operating systems can be difficult to use, and OSv had all of the applications used for benchmarking prepackaged [6]. OSv was chosen over Rumprun because of superior performance in recent research [7].

OSv runs programs in kernelspace, which eliminates the need for context switching. It is single process, but supports threading, and it only includes drivers for virtual hardware. Both of these characteristics cut down on image size, as IPC support and many drivers are not required in the kernel.

3.3 Graph-based Containerization: Nix

The final solution benchmarked in this paper is NixOS containerization. Nix is a functional package manager that ensures reproducibility by running software with exact versions of its dependencies, unlike most software installations, which allow system state to change their operation [8]. For instance, software on most Linux distributions would use a different version of a library depending on what was already

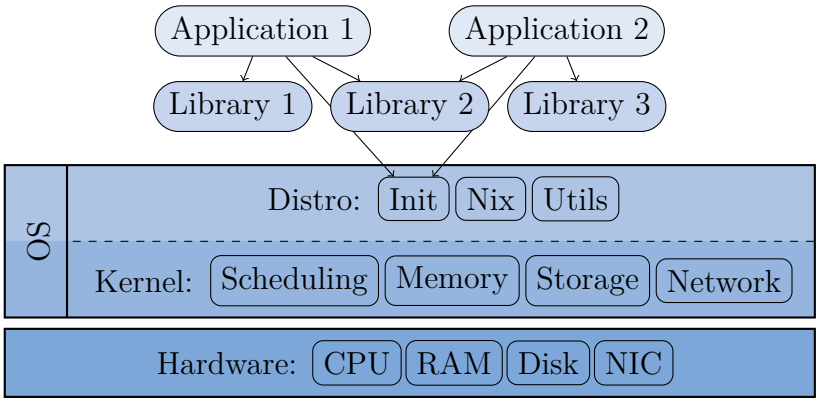
installed, or a program might completely crash if a library was not installed. Nix eliminates the indeterminacy of system state, and it applies this to all three steps of the software lifecycle: development, testing, and deployment. NixOS is an operating system that uses Nix to manage software, generating the entirety of the system from configuration files [9]. This declarative approach allows managing system state in the same way as code with techniques like version control.

NixOS provides an experimental form of containerization based on `systemd-nspawn`. A known weakness is that NixOS containerization exposes all software installed on the system to running containers, although it is exposed read only. Although Nix itself has been evaluated for use in HPC, Nix containerization was not used [10]. From here on, NixOS containerization will be referred to simply as Nix.

The key difference between Nix and layer-based containerization is that Nix manages dependencies using graphs, and it shares dependencies not only between containers, but also between containers and the host operating system. For example, two Nix containers and NixOS itself could all share the same version of libc. Nix architecture is shown in Figure 3.3.

Nix has no more components than those in minimal architecture, although it does run multiple instances of the same init program. A dependency graph contains exactly the dependencies an application needs to run, so having fewer libraries is not possible. Although Nix does not technically use a container image, the size of the application itself and its libraries can be thought of as the size of a container image, so this will be referred to as image size. Dependencies are shared between containers and the host operating system, so there is no duplication. Nix uses exactly

Figure 3.3: NixOS Containerization Architecture



one installer and one kernel. NixOS, which performs the role of installer, could be made more lightweight; it could, for example, include fewer utilities. Nevertheless, it is still a single installer component, even if it is more complex than necessary. Nix containers run an extra init process, which is a component of the installer, but the init process is the same as used by the host OS, so this does not cause overhead in storage. Removing this overhead is discussed in Section 6.1. Because Nix manages libraries with dependency graphs, it guarantees the smallest possible container size, and it does so using exactly the components of minimal architecture.

CHAPTER 4

Experiment

Podman, OSv, and Nix were compared by measuring throughput and container or image size for three common pieces of server software: Nginx, Redis, and MySQL. Nginx is a webserver, targeting networking performance. Redis is an in-memory key value store, more heavily testing memory. MySQL is a database, stressing disk and processing.

All of the benchmarks used a single server and a single client machine, which were identical servers with specifications shown in Table 4.1. The servers were deployed using Openstack on Chameleon Cloud, which is described in [11]. Nixops was used to declaratively manage the deployment of each piece of software needed on both the server and client, and the Nixops configuration as well as code to run the experiments is available at [12].

Server	PowerEdge R740
Processor	2x Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz
RAM	192 GiB
Storage	THNSF8240CCSE 240 GB SSD
Network Adapter	BCM57412 NetXtreme-E 10Gb RDMA

Table 4.1: Server Specifications

For every benchmark, both the server and client were running NixOS 20.09 and Linux kernel 5.4.72. NixOS was used as the OS for all three platforms in order to

isolate the performance of the deployment solution used. Kernel tuning parameters were set using the values in Table 4.2, the values given for network stack optimization in [13].

Setting	Parameter	Value
boot.kernel.sysctl	fs.file-max	20000
	net.core.somaxconn	1024
	net.ipv4.ip_local_port_range	1024 65535
	net.ipv4.tcp_tw_reuse	1
	net.ipv4.tcp_keepalive_time	60
	net.ipv4.tcp_keepalive_intvl	60
security.pam.loginLimits	nofile soft	20000
	nofile hard	20000

Table 4.2: Linux Kernel Tuning Parameters

The OSv kernel was likewise tuned with the parameters in Table 4.3, which were also taken from [13].

4.1 Performance Procedure

Tests were run using standard benchmarking tools for Nginx, Redis, and MySQL: `weighttp`, `redis-benchmark`, and `sysbench`, respectively. Each benchmarking tool was run with different parameters for number of threads (T), clients (C), and requests (R), which are shown in Table 4.4. Every possible combination of parameters was used, although for Nginx, the number of clients had to be greater than or equal to the number of threads. Each set of parameters was used for four consecutive iterations, and the results presented are averages.

File	Parameter	Value
bsd/sys/netinet/in.h	IPPORT_EPHEMERALFIRST	1024
	IPPORT_HIFIRSTAUTO	1024
bsd/sys/netinet/tcp_timer.h	TCPTV_KEEP_INIT	60*hz
	TCPTV_KEEP_INIT	60*hz
bsd/sys/sys/socket.h	SOMAXCONN	1024
include/api/sys/socket.h	SOMAXCONN	1024
include/osv/file.h	FDMAX	0x30D40
libc/libc.cc	RLIMIT_NOFILE	20000

Table 4.3: OSv Kernel Tuning Parameters

Additionally, for Nginx running on Podman and Nix, benchmarks were run once with multiple processes and once with a single process. Nginx is specifically designed to take advantage of using multiple processes, but OSv only runs a single process [14]. For this reason, a fair comparison between platforms required limiting Podman and Nix to only using a single worker process. This was done by leaving the default value of `worker_processes` = 1 in the Nginx configuration for one set of trials, although trials were also run with `worker_processes` = *auto* to compare Podman and Nix when more optimally configured. Since the server had 32 CPU cores and 64 threads, this allowed much higher performance.

4.2 Performance Results

4.2.1 Nginx

Performance for Nginx is shown in Figure 4.1. For each platform, the results are shown for only the fastest set of tester parameters. Both Nix and Podman achieved

	Nginx	Redis	MySQL
Threads	1, 32, 64, 128	1, 32, 64, 128	1, 2, 3, 4, 32, 64, 128
Clients	64, 128, 512, 1024	16, 64, 512	NA
Requests	1,000; 100,000; 1,000,000	1,000; 100,000	1,000; 100,000; 1,000,000

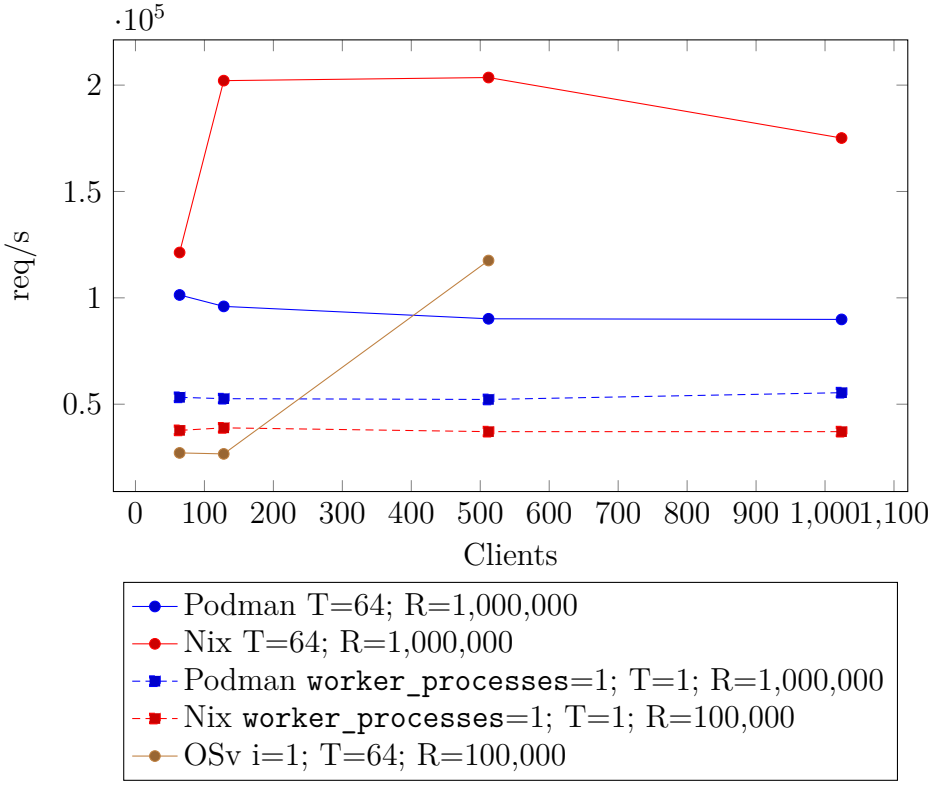
Table 4.4: Test Framework Parameters

maximum throughput with `weighttp` making 1,000,000 requests and running 64 threads. When configured with `worker_processes` = 1, both Podman and Nix performed better when the client was running a single thread. OSv performance dropped significantly after 100,000 requests, so Figure 4.1 shows OSv performance for a single iteration ($i = 1$) rather than averaging performance for four iterations.

OSv was unstable when the number of client threads was more than 512. For most numbers of tester threads, performance decreased drastically for the second iteration when the number of requests was 100,000. Figure 4.2 shows OSv performance remained consistent for small numbers of clients, but with 512 clients, performance was always low on the second iteration. This indicates OSv cannot handle high numbers of requests when many requests are sent in parallel. Although an initial investigation was made into why performance drops, tracing the exact cause would likely require further kernel debugging, which was outside the scope of this experiment. OSv did have unusually high performance for the first 100,000 requests - with a single process, it outperformed Podman configured with `worker_processes` = *auto*. Likewise, finding an explanation for this was judged beyond the scope of this experiment.

Overall, Nginx performed best when run using Nix. For both values of `worker_processes`, Nix outperformed Podman. OSv on average had the worst performance, although it

Figure 4.1: Nginx Performance

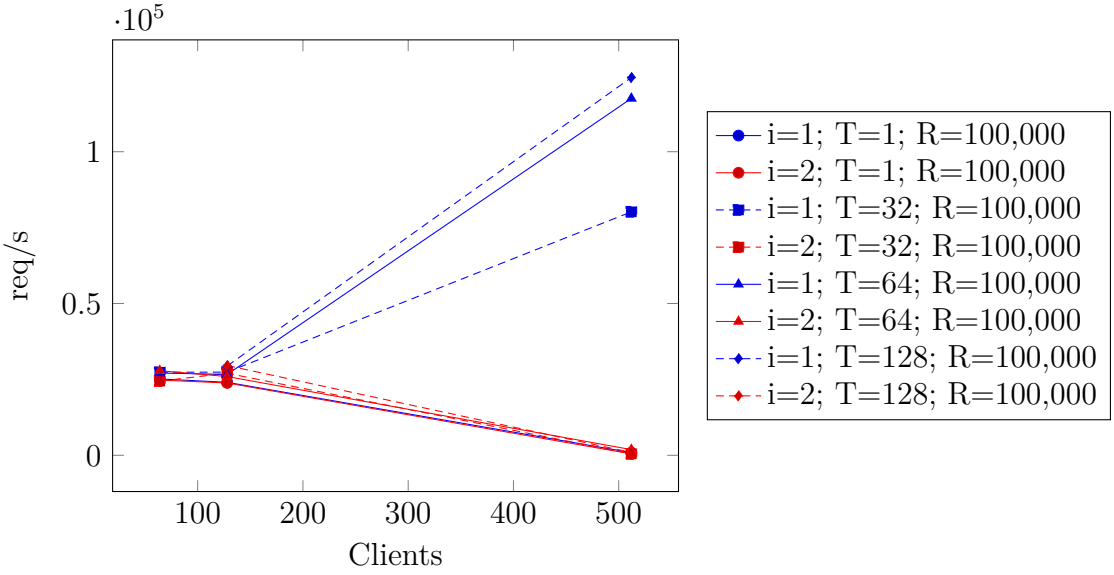


did have high performance for the first 100,000 requests using 512 clients.

4.2.2 Redis

Redis performance with fastest tester parameters is shown in Figure 4.3. `redis-benchmark` reports speed for completing a number of different operations, all of which are shown. Nix and Podman had the best performance for different test parameters: Nix performed best with 32 tester threads and 64 clients, while Podman performed best at 1 thread and 16 clients. Results for both platforms are included with each set of parameters. OSv had its best performance with 32 threads and 16 clients. Once

Figure 4.2: OSv Performance Dropping when C=512 after 100,000 Requests



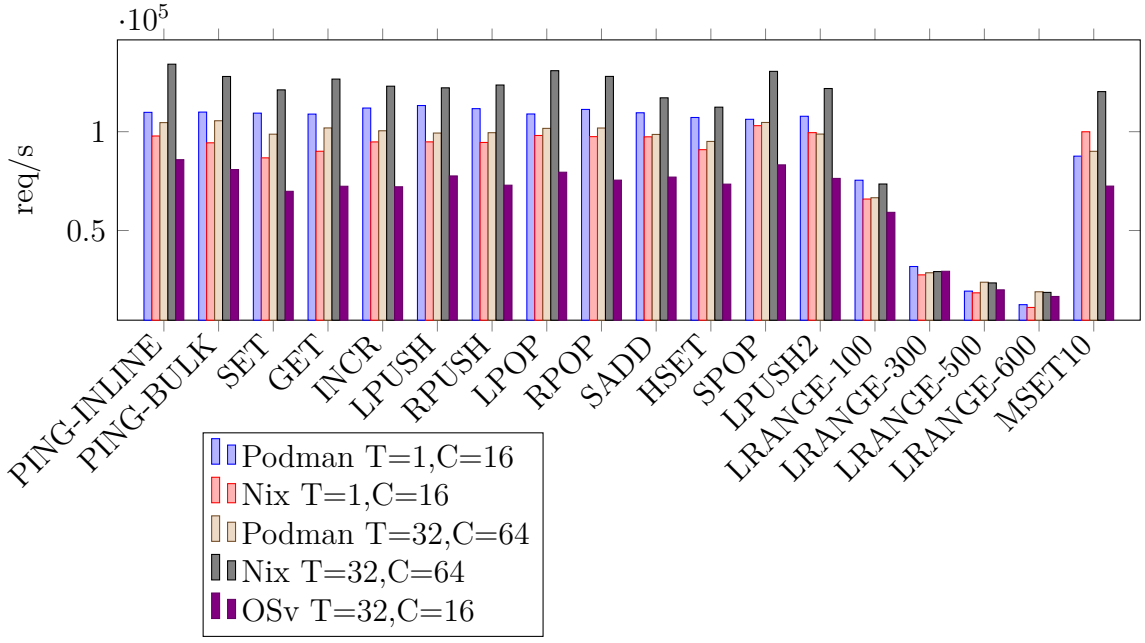
again, OSv performance was unstable for a high number of clients. With 64 or more clients, throughput for consecutive iterations differed by more than 1000% in some cases. At other times, the server completely crashed.

For almost every Redis operation benchmarked, Nix had the highest performance, although with 1 tester thread and 16 tester clients, Podman was faster. OSv had the slowest performance for almost every benchmark.

4.2.3 MySQL

OSv could not support more than two client thread connections at the same time, so MySQL performance is split into Figure 4.5 for low numbers of tester threads and Figure 4.4 for all numbers of tester threads. Nix and Podman had highest performance with R=1,000,000, while OSv performed better with R=100,000.

Figure 4.3: Redis Performance



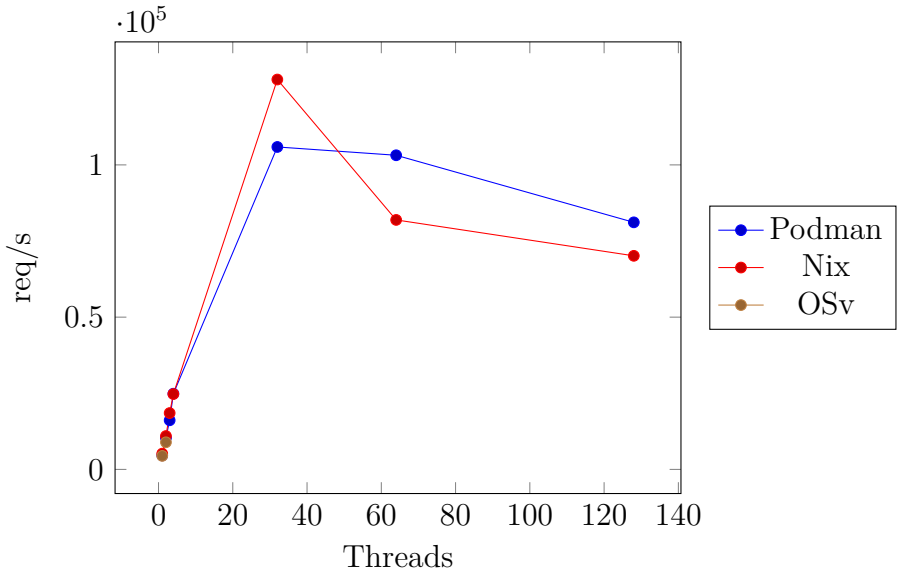
Just as for Redis, Nix had maximum performance, but Podman had better performance for certain tester parameters ($T \geq 64$). OSv had the slowest performance for all numbers of supported threads.

4.3 Performance Analysis

Containers clearly outperformed OSv, but Podman and Nix performed comparably. Although Nix did have the best performance running Nginx, either Podman or Nix performed better for Redis and MySQL depending on tester parameters.

It is not clear whether OSv's inferior performance stemmed from architectural differences or instability. For test parameters OSv ran successfully, the difference in performance between OSv and the slower container platform was similar to the

Figure 4.4: MySQL Performance



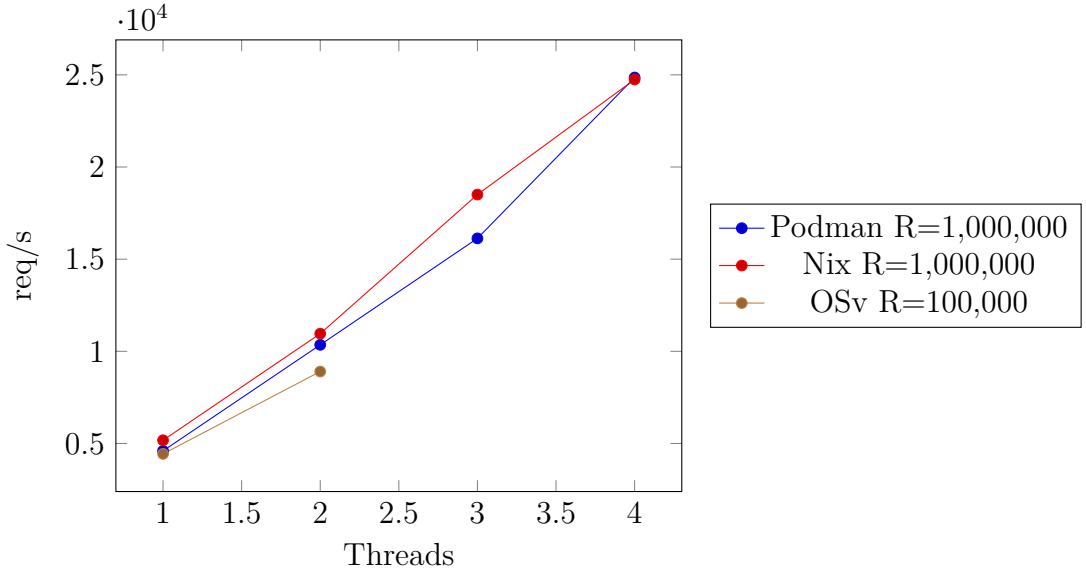
difference in performance between the two container platforms. This suggests OSv may eventually be able to compete with containerization, even if it can never outperform it. For pieces of software designed like Nginx to take advantage of multiple cores, however, using OSv does not make sense. Containers far exceeded OSv's Nginx performance when using multiple cores.

Based on these results, Podman and Nix have similar performance in most cases. OSv tends to be slower but could still reasonably be used for applications that do not require multiple CPUs and expect low numbers of concurrent clients.

4.4 Storage Procedure

To compare storage efficiency, the deployment sizes of Nginx, Redis, and MySQL were measured on each platform. Storage for Podman was analyzed using `podman images`

Figure 4.5: MySQL Performance Few Threads



and `podman history`, which allow finding both the total size of the container image and the size of the application specific layers added on top of a base image. OSv was analyzed by using `du` to find the size of the virtual disk image. The disk image was then mounted, and `du` was used to find the size of the entire filesystem. Nix was analyzed by using the `nix-store` command to query all dependencies needed by a container and summing the size of all dependencies with `du`. The size of unique dependencies of the container was then found by using `nix-store` to find which dependencies were required by the operating system and excluding those from the total.

4.5 Storage Results

Podman, OSv, and Nix manage dependencies very differently, and this was reflected in the amount of storage each platform used. Storage consumption for each platform is shown in Table 4.5.

	Podman		OSv		Nix	
	Total	Base Image	Image	Mounted	Total	Unique
Nginx	137	72.5	8.3	1.9	728.3	12.6
Redis	109	72.5	9.9	4.7	728	12.3
MySQL	551	72.5	44.6	199.6	1087.5	365.8

Table 4.5: Storage Consumption (MB)

Additionally, Table 4.6 compares the storage cost of OSv and Nix to two different components of Podman storage consumption. First, OSv and Nix deployment size is shown as a percentage of total Podman image size. Second, the size of the base image used by Podman is subtracted from the size of the entire container image, giving the size of each application and extra libraries that application installed. OSv and Nix usage is then shown as a percentage of just the size of the application and extra libraries.

	OSv		Nix	
	% of Total	% of App+Libs	% of Total	% of App+Libs
Nginx	6	12.8	9.2	19.5
Redis	9.1	27.1	11.2	33.6
MySQL	8.1	9.3	66.4	76.5

Table 4.6: Comparison to Podman Image Size

4.6 Storage Analysis

Podman has storage overhead in all three of the ways identified in Section 3.1. First, the container base image includes unused libraries. All three applications use `buster-slim` as their base image, which contains 84 packages, 49 of which are libraries. Not all of the libraries included are used by Nginx, Redis, or MySQL. Second, dependencies are not shared between containers. Nginx, Redis, and MySQL all install the package `ca-certificates`, and Redis and MySQL both install `gnupg` and `wget`. These packages are embedded in container layers that also contain other dependencies, so they cannot be shared across the three different containers. Third, packages with the same function are duplicated between the host OS and the containers. `buster-slim` includes libraries like `libstdc++`, the package manager `apt`, and `coreutils`. NixOS already contains `libstdc++`, the package manager Nix, and `coreutils`, so this is unnecessary duplication.

OSv has the smallest storage consumption, but this may not necessarily be caused by better dependency management. First, OSv disk images are in a format that can either contain more or less data than indicated by the disk image size. This is due to the fact that virtual disk images can be compressed, which would save space on the host, but they can also contain free space within the image, which would waste space on the host. Additionally, OSv applications are compiled in a way that minimizes space but has fewer features. For example, Nginx compilation has a number of flags to control what features are supported. OSv builds Nginx with only three of these options enabled, while Nix, depending on a few parameters, enables around twenty to thirty. Because OSv uses a different storage format than containerization and OSv

applications have fewer features than containerized applications, it is difficult to draw conclusions about whether OSv has better storage efficiency than containerization.

Nix consumes far more total space than Podman, but the size of unique dependencies is much smaller than the size of the application and library layers for Podman, ranging from only 20% to 77% of Podman size. In other words, the operating system already contains many of the dependencies needed to run each container, so the size of additional dependencies installed is very low. This eliminates all three types of overhead caused by Podman.

When starting a piece of software on an already running server, the storage cost of deployment is the size of every piece of additional software that must be installed on the server. For this reason, although Nix has the largest total size, it has a much lower deployment cost than Podman once shared dependencies with the host OS are taken into account. OSv also consumes significantly less space than Podman, but this is not due to better dependency management; OSv attains smaller image size by compiling applications with options that minimize size. Considering deployment size for a single application, OSv consumes as little as 6% of the space needed by Podman, and Nix consumes as little as 9.2%.

CHAPTER 5

Related Work

In general, prior work has drawn similar conclusions when comparing containerization to unikernels. In 2014, one of the first papers to introduce OSv described it as "a more suitable operating system for virtual machines in the cloud than are traditional operating systems such as Linux" [15]. This original claim has been confirmed by benchmarking comparing unikernels and VMs, and it is an accurate description of the purpose of unikernels: replacements for full-featured guest VMs with kernels like Linux. Since the introduction of unikernels, many papers have compared them to containers instead of VMs, and most have found that containers outperform unikernels as would be expected.

Prior work on comparing performance of containerization and unikernels has primarily taken two approaches. One approach is running benchmarking tools to stress specific computing components. Examples of this include running Netperf to test networking or running Memcached to test memory. The second approach is to benchmark real world applications, such as webservers or databases.

In addition to performance benchmarking, research has been done on more specific topics, ranging from provisioning time [16], to PHP applications [17], to Software-Defined Security [18]. The focus of this paper, however, is on performance and deployment size, so a summary of papers focused on benchmarking is given in Table 5.1. Some of these papers also address memory consumption, which is beyond the scope of this paper, so it is not presented.

Platforms	Benchmark	Results	Paper	Year
Docker, LXC, OSv, VM	noploop, Linpack, STREAM, Netperf	Containers fastest	Morabito [19]	2015
Docker, OSv, VM	Netperf, Memcached, Mutilate	Containers fastest	Enberg [20]	2016
Docker, LXD, OSv, Rumprun, MirageOS, VM	Webserver, Redis	Unikernels fastest on hypervisor	Plauth [13]	2017
Docker, OSv	REST service, heavy workload	Unikernels faster for REST service	Goethals [21]	2018
Docker, rkt, OSv, Rumprun, VM	SysBench, STREAM, Iperf	Containers fastest	Acharya [22]	2018
Docker, LinuxKit, OSv, IncludeOS, Rumprun	Webserver, MySQL	Containers fastest throughput, unikernels lowest latency	Mavridis [7]	2019

Table 5.1: Related Work

Across all papers, the containerization platforms considered are Docker, LXC, LXD, LinuxKit, and rkt; and the unikernels considered are OSv, Rumprun, MirageOS, and IncludeOS. The final platform is a full-fledged Linux guest VM running on KVM. In most cases, the unikernels are run on KVM, although some papers also use Xen.

Most papers found that containers outperformed unikernels, with only two exceptions. First, Plauth et al. found, “Regarding application throughput, most unikernels

performed at least equally well as or even better than containers" [13]. Based on data presented in the paper, it appears this claim is based on comparing containers running on a KVM Linux guest with unikernels running on KVM. Results from the paper about bare-metal containers indicate comparable or better performance than unikernels on KVM.

Second, Goethals et al. claimed lack of context switching gives unikernels an advantage over containers, but this does not make sense since the host OS must still perform context switches [21]. The benchmarks presented by Goethals et al. found unikernels performed better than containers for a REST benchmark, from which it is argued, "Unikernel performance for the REST service stress test can be explained by the fact that this test relied heavily on kernel functions and thus context switches from user space to kernel space. These do not exist in a unikernel, giving unikernels a large advantage over containers in situations where context switches happen very often." Although it is true that unikernels eliminate a context switch when comparing Linux VMs to unikernels, the REST benchmark compares bare-metal containers to unikernels. The explanation given by Goethals et al. is flawed, because unikernels running on top of KVM must make context switches just like a container.

KVM makes these context switches for both memory management and I/O. It allocates memory just like a normal userspace process with malloc and mmap [23]. For I/O, KVM uses virtio, which either requires context switching, or a packet forwarding framework or device passthrough [24]. The same techniques to avoid context switching are available for containers [25][26]. Since hypervisors must make context switches when running VMs, context switching does not give unikernels an advan-

tage when compared to bare-metal containers. Benchmarks in most papers confirm this claim, and so do the results found in this paper comparing Podman and OSv.

Overall, current research suggests that kernels without context switches outperform kernels with context switches, but total number of running kernels and total number of context-switching kernels both impact performance. Containers outperform unikernels because they have fewer total kernels, while unikernels can outperform VMs because they run fewer context-switching kernels.

CHAPTER 6

Future Work

Although using Nix would eliminate some of the overhead of layer-based containerization, there is still room to optimize in many other areas.

6.1 Lightweight Init

Minimizing the architectural overhead of containerization requires viewing it as a core part of the operating system, not as an additional component added on top of the operating system. At a high level, microservices are just like any other system service, except they are run in a permissions sandbox. One area shifting towards this mindset would be an improvement is the init system. Podman has already begun to encourage this shift, because it is daemonless. Docker runs the Docker daemon to manage container startup, while Podman relies on `systemd`, which is already present on most Linux systems, to start containers [27].

Nevertheless, Nix runs an additional init process inside every container, and Podman starts an additional init process if `systemd` features are desired [27]. Running an init process for every container increases startup time and memory consumption. A more optimized solution would be to let the host OS init system control the startup for the container. This would make startup for single application containers extremely fast, while still supporting more complicated containers with multiple services. Although it is not currently possible to replace a container's init process with the init process of the host OS, if containerization could be improved to allow this,

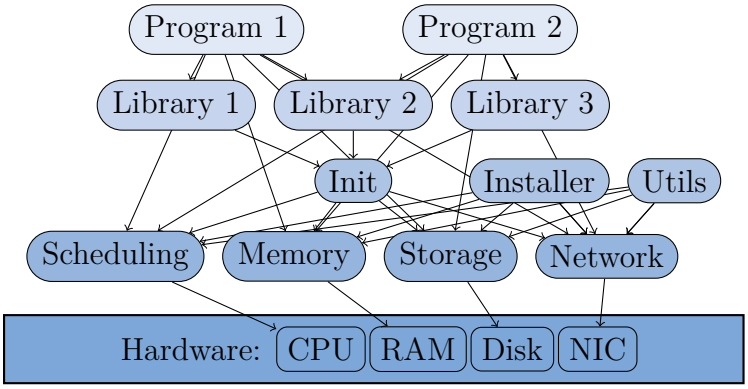
containers would operate more like sandboxed system services and less like guest operating systems with unnecessary overhead.

6.2 Lightweight Kernel

An advantage of unikernels is how much smaller they are than Linux. One of the primary reasons Linux is so much larger is how much hardware Linux supports. While Linux must provide drivers for many different types of hardware, unikernels only have to support a few virtual devices. Many of the drivers Linux provides are not necessary for cloud computing.

Applying the graph-based dependency approach of Nix to kernel modules could break up the kernel into dependencies as shown in Figure 6.1 and lead to much smaller kernels.

Figure 6.1: Light Kernel



Hardware could be viewed as having kernel dependencies, so that only kernel modules for installed hardware were included. Alternatively, software could have kernel dependencies, so that only needed functions would be provided by the kernel.

Most microservices, for example, would only need drivers for processing, memory, storage, and network, and all other I/O drivers could be excluded.

While current NixOS containerization eliminates layering for libraries and base images, the kernel is still provided as a layer. Using graph-based kernel module dependencies would eliminate another layer in architecture, leaving only hardware as a layer that cannot be broken into individual dependencies.

6.3 Baremetal Unikernel

Both Linux and hypervisors allow sharing resources between different processes, which is necessary for running multiple isolated processes on the same computer. For any application where multiple users must be supported this is necessary, and in HPC or cloud computing this is necessary for running multiple workloads on the same node. This is not necessary, however, in any case where a single service needs at least all the resources of a single server, in which case it would make sense to run a single application per physical server. Running a single application per server would likely be much faster if a bare-metal unikernel was used.

Bare-metal provisioning allows remote installation of an operating system and would make using bare-metal unikernels practical for running single applications. For example, the experiments for this paper used Openstack Ironic, which has support for many different types of hardware [28]. The bare-metal provisioner could fulfill the role of the installer in minimal architecture.

Some microservices, like Redis, are designed to require multiple instances, so running a bare-metal unikernel would not make sense. Others, like Nginx, are designed to take advantage of multiple processes on the same server [14]. Unlike most current

unikernels, this would require a bare-metal unikernel to support multiple processes.

Although a bare-metal unikernel would run an application in kernelspace, it could still support running applications for untrusted users like needed for HPC. Instead of the kernel performing sandboxing, the bare-metal provisioner would be responsible for supervising nodes. An untrusted user could not permanently gain control of a node because as soon as a certain time limit was reached, the provisioner would completely erase the operating system on the node.

Since platforms with one context switching kernel perform better than platforms with two, it is likely that having zero context switches would significantly boost performance. Deploying bare-metal unikernels would cause some overhead since the kernel would have to be reinstalled each time a program was updated. Currently OS provisioning takes significantly longer than starting a container, but if the OS was a unikernel, this would be much faster. Alternatively, updates could be performed by the kernel, in which case a server would function similar to a current cloud node, but the microservice would run in kernelspace. Currently, it does not appear that stable software exists to use bare-metal unikernels in either of these ways, but eliminating a context switch might boost performance enough to make bare-metal unikernels worth developing.

CHAPTER 7

Conclusion

Containerization is a Linux kernel technology that allows isolating processes from each other, which some have even explained as, "Containers are Linux" [29]. Despite this fact, current containerization approaches do not treat containerized software like Linux treats software. Linux operating systems share libraries between processes, and they share libraries between applications and the operating system. Popular container images for Nginx, Redis, and MySQL do not share libraries in either of these ways, while also adding unnecessary libraries to container images. NixOS containerization, on the other hand, allows sharing libraries like Linux.

By sharing dependencies whenever possible, NixOS containerization has the most minimal architecture that satisfies the requirements of cloud computing and HPC for software deployment, which leads to significant savings in software deployment size. At the same time, it performs just as well as current containerization solutions in benchmarking tests. Although unikernels also have small image size, they do not perform as well as containers in benchmarks. For this reason, unikernels may be a good alternative to VMs, but containers should be used when application performance is desired. Nix achieves both small image size and high performance by providing efficient dependency management for containers. Nix provides exactly the libraries needed by an application, and it shares libraries not only between containers but also between containers and the host OS. By doing so, Nix guarantees the smallest possible size for a container.

References

[1] Y. Fisher, *Podman paves the road to running containerized hpc applications on exascale supercomputers*, 2020. [Online]. Available: https://www.redhat.com/en/blog/podman-paves-road-running-containerized-hpc-applications-exascale-supercomputers.

[2] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," Gaithersburg, MD, USA, Tech. Rep., 2011.

[3] *About singularity*, Mar. 2021. [Online]. Available: https://singularity.lbl.gov/about.

[4] *Kubernetes components*, Aug. 2020. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components.

[5] *Docker engine release notes*, Feb. 2021. [Online]. Available: https://docs.docker.com/engine/release-notes/.

[6] M. Leon, *The dark side of unikernels for machine learning*, 2020. arXiv: 2004.13081 [cs.DC].

[7] I. Mavridis and H. Karatza, "Lightweight virtualization approaches for software-defined systems and cloud computing: An evaluation of unikernels and containers," in *2019 Sixth International Conference on Software Defined Systems (SDS)*, 2019, pp. 171–178. DOI: 10.1109/SDS.2019.8768586.

[8] E. Dolstra, *The purely functional software deployment model*. Utrecht University, 2006.

[9] E. Dolstra and A. Löh, "Nixos: A purely functional linux distribution," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '08, Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 367–378, ISBN: 9781595939197. DOI: 10.1145/1411204.1411255. [Online]. Available: https://doi.org/10.1145/1411204.1411255.

[10] B. Bzeznik, O. Henriot, V. Reis, O. Richard, and L. Tavard, "Nix as hpc package management system," in *Proceedings of the Fourth International Workshop on HPC User Support Tools*, ser. HUST'17, Denver, CO, USA: Association for Computing Machinery, 2017, ISBN: 9781450351300. DOI: 10.1145/3152493.3152556. [Online]. Available: https://doi.org/10.1145/3152493.3152556.

[11] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, USENIX Association, Jul. 2020.

[12] M. Kenigsberg. [Online]. Available: https://github.com/mkenigs/research-deploy.

[13] M. Plauth, L. Feinbube, and A. Polze, "A performance survey of lightweight virtualization techniques," Sep. 2017, pp. 34–48, ISBN: 978-3-319-67261-8. DOI: 10.1007/978-3-319-67262-5_3.

[14] O. Garrett, *Inside nginx: How we designed for performance & scale*, Jun. 2015. [Online]. Available: https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale.

[15] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv—optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72, ISBN: 978-1-931971-10-2. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[16] B. Xavier, T. Ferreto, and L. Jersak, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 277–280. DOI: 10.1109/CCGrid.2016.86.

[17] T. Pasquier, D. Eyers, and J. Bacon, "Php2uni: Building unikernels using scripting language transpilation," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, 2017, pp. 197–203. DOI: 10.1109/IC2E.2017.13.

[18] M. Compastié, R. Badonnel, O. Festor, R. He, and M. Kassi-Lahlou, "Unikernel-based approach for software-defined security in cloud infrastructures," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–7. DOI: 10.1109/NOMS.2018.8406155.

[19] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[20] P. Enberg, "A performance evaluation of hypervisor, unikernel, and container network i/o virtualization," 2016.

[21] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications," Nov. 2018, pp. 1–8. DOI: 10.1109/SC2.2018.00008.

[22] A. Acharya, J. Fanguède, M. Paolino, and D. Raho, "A performance benchmarking analysis of hypervisors containers and unikernels on armv8 and x86 cpus," in *2018 European Conference on Networks and Communications (EuCNC)*, 2018, pp. 282–9. DOI: 10.1109/EuCNC.2018.8443248.

[23] Feb. 2010. [Online]. Available: https://www.linux-kvm.org/page/Memory.

[24] E. P. Martín, *Virtio devices and drivers overview: The headjack and the phone*, Jun. 2020. [Online]. Available: https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone.

[25] *Virtio_user for container networking*. [Online]. Available: https://doc.dpdk.org/guides/howto/virtio_user_for_container_networking.html.

[26] *About single root i/o virtualization (sr-iov) hardware networks*, 2021. [Online]. Available: https://docs.openshift.com/container-platform/4.7/networking/hardware_networks/about-sriov.html.

[27] V. Roth and D. Walsh, *Improved systemd integration with podman 2.0*, 2020. [Online]. Available: https://www.redhat.com/sysadmin/improved-systemd-podman.

[28] *Drivers, hardware types and hardware interfaces*, Oct. 2020. [Online]. Available: https://docs.openstack.org/ironic/latest/admin/drivers.html.

[29] J. Fernandes, *Containers are linux*, Apr. 2017. [Online]. Available: https://www.openshift.com/blog/containers-are-linux.

Appendix A

Results for Additional Tester Parameters

A.1 Nginx

Figures A.1, A.2, and A.3 show average Nginx throughput over four iterations for every set of tester parameters.

Figure A.1: Podman Performance

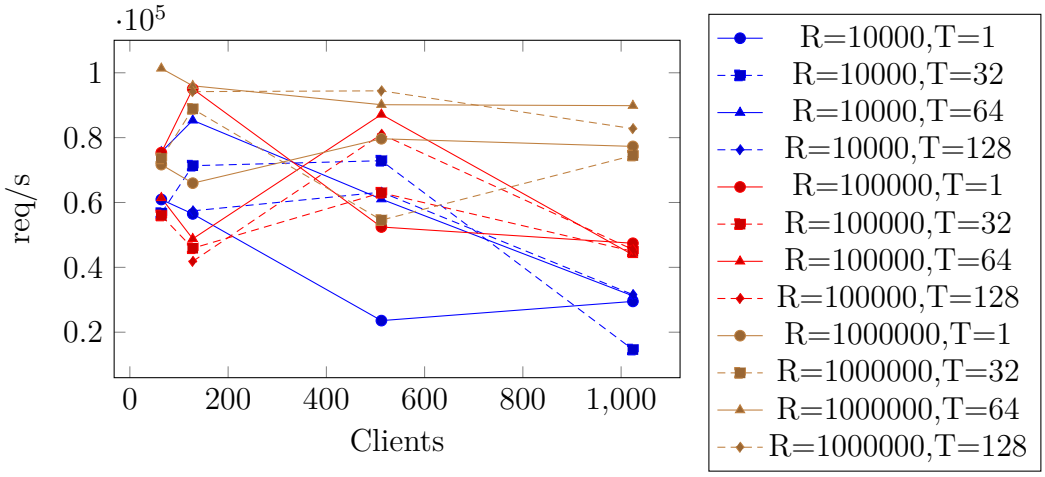


Figure A.2: OSv Performance

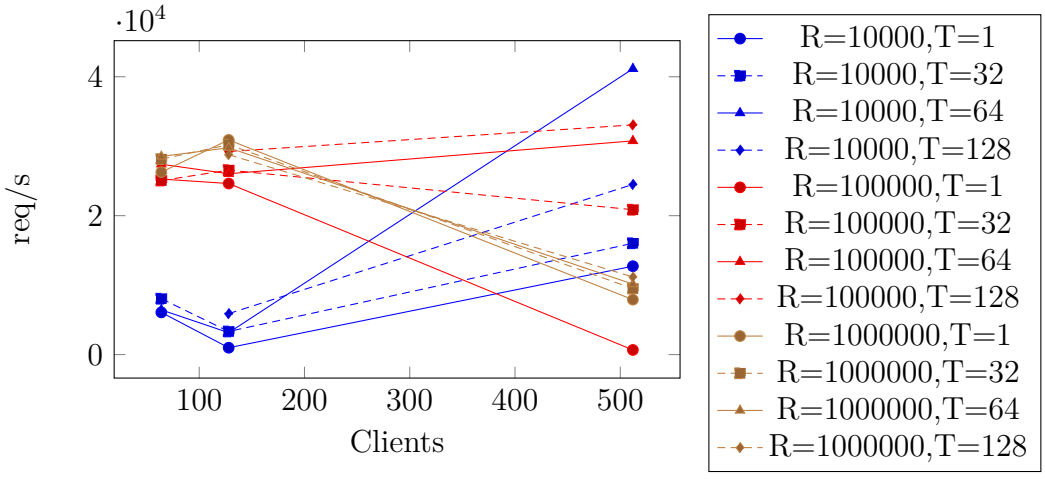
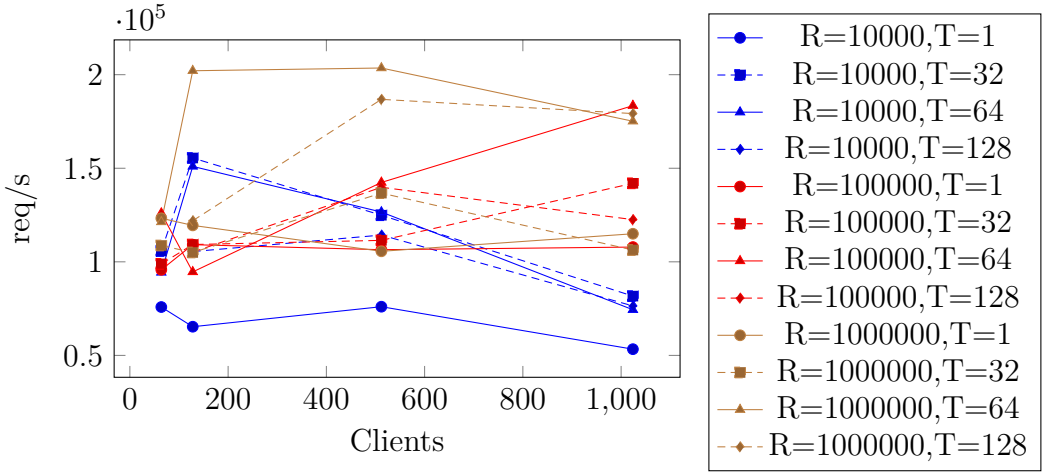


Figure A.3: Nix Performance



A.2 Redis

Figures A.4 and A.6 show Redis performance for each value of tester threads with whatever number of clients resulted in maximum performance. Figure A.5 also only shows results for the fastest number of clients, but OSv was only stable for $T = 1$ and $T = 32$, so only those results are shown.

Figure A.4: Podman Comparison of Best Performance for Each T-value

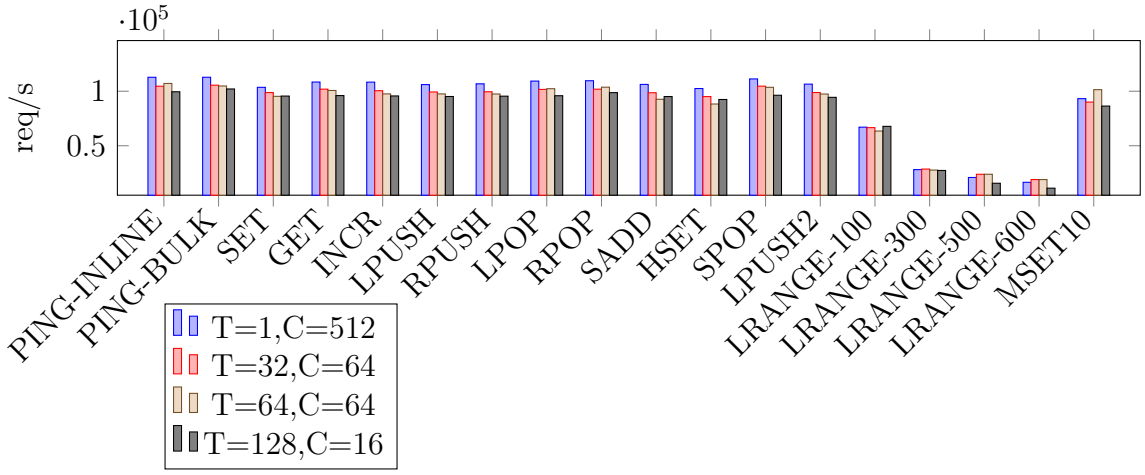


Figure A.5: OSv Comparison of Best Performance for Each Stable T-value

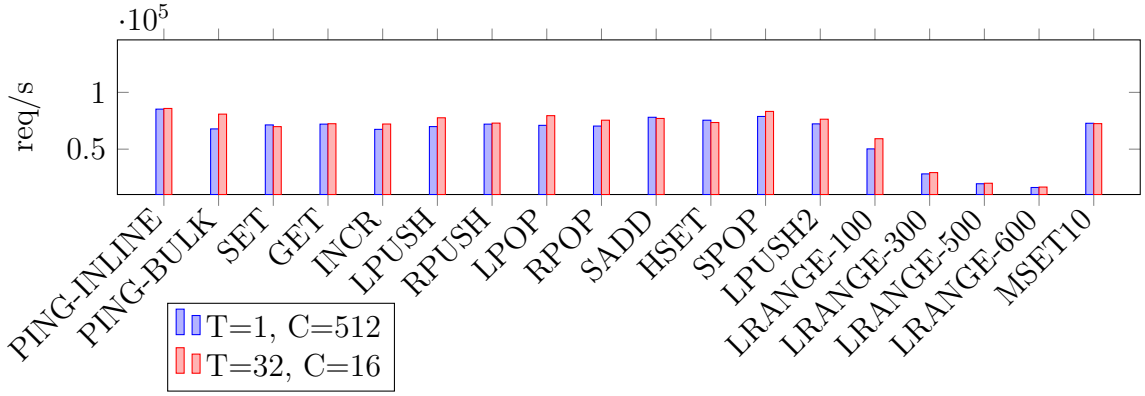
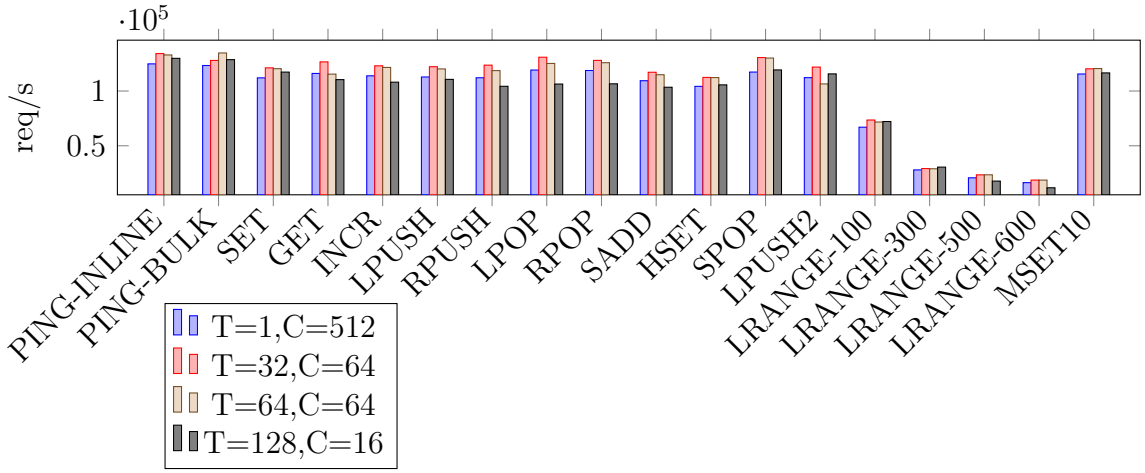


Figure A.6: Nix Comparison of Best Performance for Each T-value



A.3 MySQL

Figures A.7, A.8, and A.9 show average MySQL throughput over four iterations for every set of tester parameters.

Figure A.7: Podman Performance

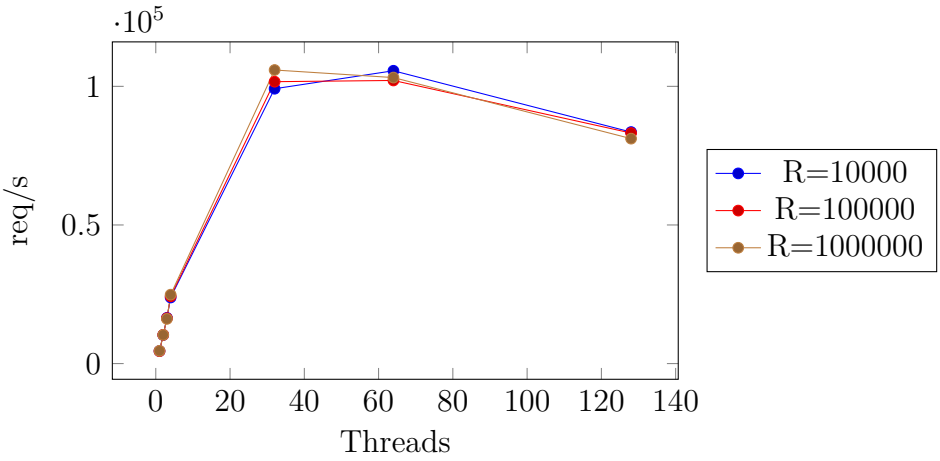


Figure A.8: OSv Performance

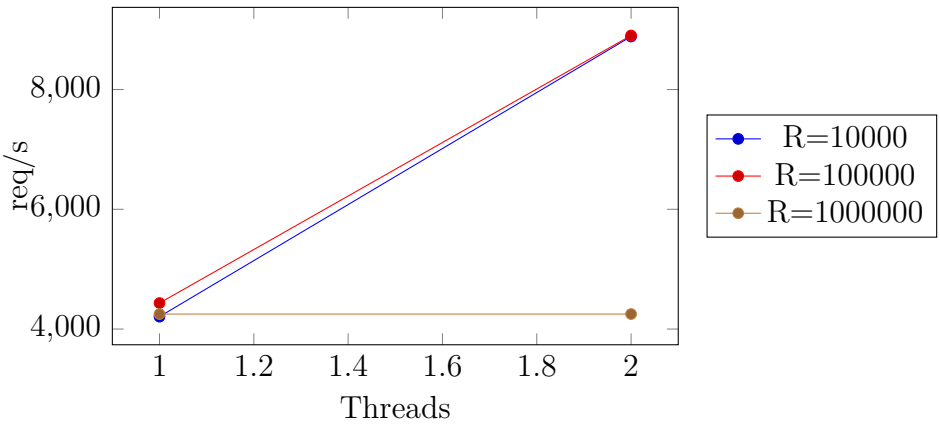


Figure A.9: Nix Performance