NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DIVISION OF COMPUTER SCIENCE

Efficient file sharing between host and unikernel

DIPLOMA THESIS

Fotios Zafeiris M. Xenakis

Supervisor: Nectarios Koziris
Professor, NTUA

Athens, October 2020

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Efficient file sharing between host and unikernel

DIPLOMA THESIS

Fotios Zafeiris M. Xenakis

Supervisor: Nectarios Koziris
Professor, NTUA

Approved by the three-member examining committee on October 30, 2020.

.....	.....	.....
Nectarios Koziris	Georgios Goumas	Dionisios Pnevmatikatos
Professor, NTUA	Assistant professor, NTUA	Professor, NTUA

Athens, October 2020

.....
Fotios Zafeiris M. Xenakis
Dipl. in Electrical and Computer Engineering, NTUA

Abstract

Cloud computing is the dominant approach to compute infrastructure, established on the technology of virtualization. As the cloud expands, efficient utilization of its compute resources by software becomes imperative. One solution towards that are *unikernels*, operating system kernels specialized to run a single application, sparing resources compared to a general-purpose kernel. Efficient access from virtualized guests to the underlying host's resources is a substantial challenge in virtualization. In this aspect, *virtio* has been a significant contribution, as a specification of paravirtual devices enabling efficient usage of the host's resources. For host-guest file sharing, *virtio-fs* has been proposed, as a virtio device offering guest access to a file system directory on the host, providing high performance and local file system semantics.

This thesis is concerned with the implementation and evaluation of virtio-fs in the context of the OSvunikernel. We demonstrate that combining the two offers great benefits, both with regard to performance achieved, which is comparable to local file systems, and the operational aspect in a cloud context. Moreover, the above are carried out fully within the open-source project behind the unikernel we based our work on. This way, the resulting product gains practical value, being a useful contribution to the project, thus achieving a pivotal, non-technical goal. Furthermore, we explore how open-source software projects and the communities around them work, as we become active members of one.

Keywords

virtualization, cloud, file system, unikernel, virtio, OSv, virtio-fs, QEMU

Acknowledgements

For this work, signalling the completion of a long course, I would like to thank the members of the computing systems laboratory, under whose auspices it was carried out. Most of all, I want to thank them, as well as other members of the ECE school, for their teaching, their genuine interest and for cultivating the spirit of an engineer in me.

Moreover, I owe a big thank you to the people in the OSv and virtio-fs communities for their support, their guidance and their time, but more importantly for their openness, their spirit and work which sparked this contribution.

Finally, those I am most grateful for are my family and friends, who always stand on my side, bear with and support me and without whom nothing could be accomplished.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Why unikernels

This past decade, we have witnessed the domination of the cloud computing model. The latter has become a cornerstone in many application fields, enabling previously impractical architectures, for a growing number of users. Its broad adoption has also led to a need for managing compute infrastructure of unprecedented scale. The technology behind cloud in its current form is that of virtualization: "curving" multiple, virtual, machines out of a single host system, at will.

Despite the changes brought about by the advent of cloud computing, the conventional slices of the software stack (the operating system being a stark example) have been minimally affected by it. Thus, this part of infrastructure is dominated by general-purpose OS's, like Linux, carrying decades-old design decisions and legacy. This is undoubtedly an indication of the difficulty involved in their implementation, as well as the value contained in current systems, accumulated over a long series of improvements.

A combination of factors makes it obvious that general-purpose operating systems are not the optimal solution for modern application requirements. These factors include the transition from physical to virtual machines, invalidating the assumption of exclusive ownership over the underlying hardware resources. Another contributing element is the rapid bridging of the gap between I/O and processing performance, rendering prohibitive the involvement of the OS in a high-performance application's data path, as backed by the growing popularity of frameworks like [3, 9]. Finally, the large scale of the infrastructure at hand, as mentioned above, amplifies the need for optimized efficiency, since sub-optimum operation bears huge costs.

Unikernels are a notable proposal for substituting conventional operating systems in guests, when those are used to run only a single application. Those result from

merging an application with all supporting elements it requires (typically offered by an OS), in the form of libraries, in a common address space. The executable images produced are run as virtual machines, achieving higher efficiency due to their reduced size (thus faster transport and startup [32], as well as lower storage space requirements), lower memory requirements and more performant system operations, e.g. due to the elimination of mode switches and a simplified security model.

1.1.2 Why shared file system and virtio-fs

Typically, file systems are local, implemented over block devices, inside an operating system kernel. There are cases though which benefit from sharing a common file system across several machines. One such case is that of virtualization, where host and guest share a file system (usually one pre-existing on the host). One example of how useful this can be are virtual machines that get reconfigured by the host, while another one is found in short-lived virtual machines which read input or configuration data and write output data to a common file system.

Virtio-fs is a recent effort in the field of shared file systems, the first one built to target virtualization environments exclusively. This specialization allows it to have no dependency on network protocols or (virtual) network infrastructure, resulting in lower requirements for a guest utilizing it, in addition to improved performance and local file system semantics [54]. What plays an important role in achieving these is the use of a shared memory area between the host and the guest, where file contents are mapped, the so called DAX (Direct Access) window.

1.2 Goals

This work aims to explore the possibilities offered by virtio-fs in a unikernel context. Specifically, we choose OSv and extend its existing, elementary virtio-fs implementation by adding read-only support for the DAX window as well as support for booting off of a virtio-fs file system. These render its use practical across a multitude of cases. Moreover, we evaluate our implementation's performance compared to that of an array of other file systems, in various representative scenarios.

Our goals though extend to non-technical ones, having to do with the free / open source model of software development. Since both projects involved use this model, we consider it an opportunity and a moral obligation for all of our work on OSv to be contributed back to the project. This way, other users may benefit from it, extending its value beyond the academic plain, while the respective project communities gain a new, active member.

1.3 Related work

In the years since the introduction of MirageOS [33], a plethora of unikernel frameworks have made their appearance, forming a diverse ecosystem. Some of them, beyond OSv are RumpRun [47], IncludeOS [6], HermitCore [29], and HermiTux [37], ukl [46], Toro [10] and Nanos [7].

The main pre-existing alternative to virtio-fs is VirtFS [26] which is based on the 9P network protocol [1]. Virtio-fs is still a young project, under active development, so there are relatively few implementations available. On the host side, apart from its main implementation in QEMU [53], there exists a complete implementation of it in cloud-hypervisor [19], an alternative virtiofsd implementation named memfsd from the nabla containers community [36] and an implementation for firecracker [14] which has not been accepted by the project for now. On the guest side, apart from the reference implementation in Linux [52], there is an implementation in Toro (a Pascal unikernel) [10], as well as another for Windows [55].

1.4 Thesis structure

A thorough description of the technical background of this thesis, from cloud use cases to the supporting technologies of virtio-fs is included in the second chapter. The third chapter concerns the specifics of the implementation of our extensions to OSv, followed by its evaluation, describing the methodology and examining the results in the fourth chapter. Finally, the fifth chapter contains a brief review in addition to directions for future work.

Chapter 2

Background

2.1 Cloud computing

Cloud computing is a model of providing computing resources as a service, which is now very well established. At its core, it makes concentrated, shared compute resources available on demand, via the network. These resources are owned and controlled by a cloud service provider and made available remotely to users - clients who thus achieve lower up front cost computing infrastructure, flexibility, scalability, reduced deployment times and simplified infrastructure administration [57].

We can distinguish three primary cloud service models, through which the underlying resources are offered, as shown in fig. 2.1 [35]:

Infrastructure-as-a-Service (IaaS) which provides compute, storage and networking resources, such as virtual machines, storage drives, virtual networks and load balancers. This model offers the lowest level of abstraction but the greatest flexibility to the users.

Platform-as-a-Service (PaaS) where the users are able to develop their applications, assuming the "environment" in which those will be executed. Fundamentally, this model provides a higher level service, in the sense that they are not burdened with managing components such as the operating system or runtime system, allowing them to focus on the application instead.

Software-as-a-Service (SaaS) presenting the highest level of abstraction from the infrastructure. According to this model, the users are granted access to applications which are fully managed by the provider, from the underlying infrastructure to the configuration of the application itself. Examples of this are email services, project management and office suite applications such as document and spreadsheet editors.

One model which emerged more recently, attracting a lot of user attention is the so called serverless or Function-as-a-Service (FaaS). This resembles PaaS (it

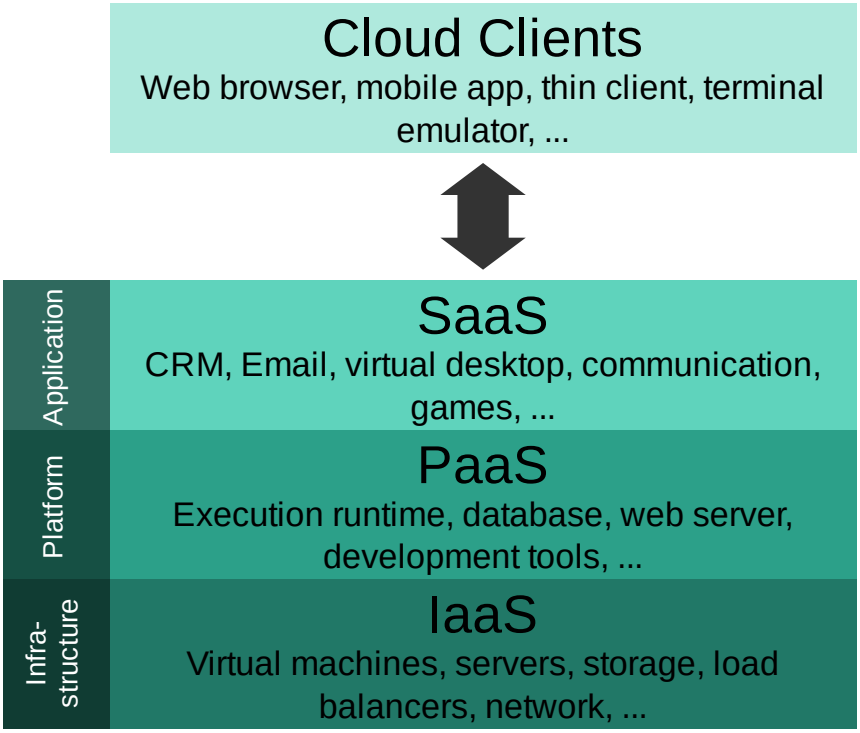


Figure 2.1: Cloud service models. Source Wikimedia Commons.

could be considered part of it, or an evolved form), since as the name suggests, it completely takes away the duty of server management from the users. The emphasized technical aspects include automatic scalability from zero as well as instant reaction to load fluctuations thanks to rapid startup and termination of application instances [5]. One of the drawbacks of serverless is that only some applications fit this model, allowing them to benefit from the above [18, 20].

2.2 Virtualization

The technology at the foundation of the cloud is that of virtualization. This adds an abstraction layer over the physical compute infrastructure, enabling the mapping of one (typically large) physical machine (host) to more, smaller virtual machines (guests), created, modified, moved and deleted dynamically through software [58].

Regarding the requirements for software running as guest, we discern two cases:

Full virtualization where the guest is not aware of running inside a virtual machine. This way, no changes are required for software written for physical machines to execute in virtual ones.

Paravirtualization where the guest is modified specifically for execution in a virtual machine.

Virtualization has been around for decades prior to the advent of the cloud. One decisive point for the realization of the latter was the addition of hardware support for it in the x86 architecture, which led to its efficient use, without high demands from software [13]. Unto that point, virtualization on this highly popular architecture relied upon paravirtualization techniques.

The responsibility for accomplishing virtualization lies with the hypervisor or virtual machine monitor (VMM). This is usually software running on the host, with elevated privileges. Its tasks include starting the virtual machines and emulating the operations not permitted to them, mainly interacting with the system's devices. The latter is normally implemented through the "trap and emulate" technique, where some instructions during the virtual machine's execution cause CPU traps, transferring control to the hypervisor code. That in turn checks and executes the guest's operation, eventually returning control back where it had stopped. This mechanism grants the hypervisor full control over the (virtual) device model of the virtual machine [59].

Hypervisors are classified in two primary categories (fig. 2.2) [45]:

Type 1 which execute directly on top of the physical machine, assuming complete control over it, in addition to managing the virtual machines. Usually these hypervisors depend on a special-purpose guest, running with elevated privileges, to manage the machine's devices, having the necessary drivers. The best known, free software such hypervisor is Xen [16].

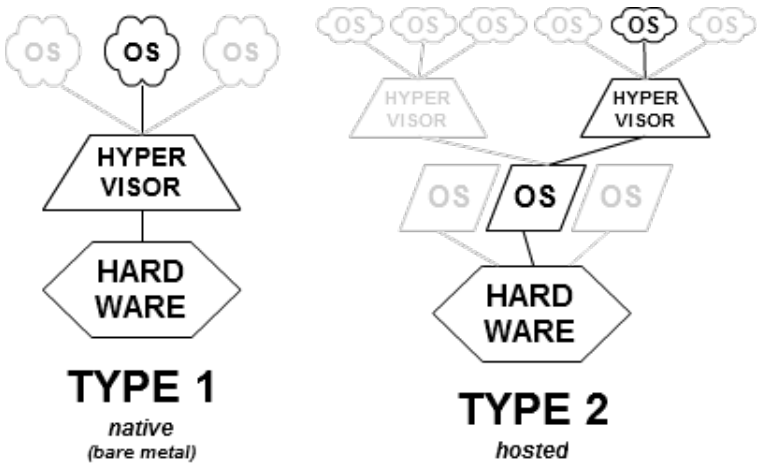


Figure 2.2: Hypervisor types. Source Wikimedia Commons.

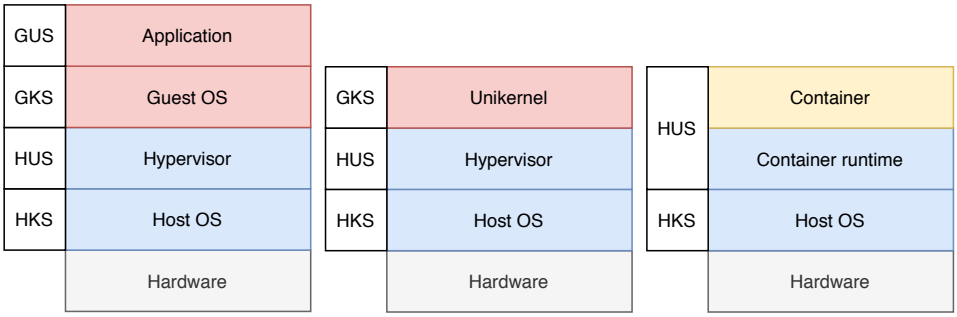
Type 2 which execute as part of a typical operating system, only in charge of starting and monitoring the virtual machines, while the host is managed by the operating system as usual. In this category it is customary for every guest to be a regular process from the host's perspective. The most popular free software representative of this category is QEMU [17] (usually in combination with KVM [27] to provide hardware acceleration).

2.3 Unikernels

In the common virtualization use case, the software supporting an application can be viewed as (see fig. 2.3a):

- On the host (kernel space) runs a general-purpose operating system with direct access to the physical machine's resources and assuming responsibility over them.
- On the host (typically user space) also runs the hypervisor, assuming the duty of managing the virtual machines which correspond to the system.
- On the guest (kernel space) again runs an instance of a general-purpose operating system. This is responsible for the resources allocated to the virtual machine by the hypervisor.
- On the guest (user space) runs the application, frequently in conjunction with components like third-party libraries and runtime systems.

Ths software stack contains duplication and high complexity, two elements commonly at fault for problems such as sub-optimal resource usage, reduced performance due



(a) Typical VM (b) Unikernel (c) Container

Figure 2.3: Comparison between "classic" virtual machine arrangement, unikernel and container, with a type 2 hypervisor. Where HKS=Host Kernel Space, HUS=Host User Space, GKS=Guest Kernel Space and GUS=Guest User Space.

to overheads (e.g. mode switches) in the overall system operation and security hazards due to the software's size, of which a good part is unnecessary (e.g. guest drivers).

A modern attempt at rectifying the above problem are unikernels [33]. These are executable machine images comprising of an application bundled with all necessary dependencies for its execution, from the likes of libraries and runtime systems to network stacks and device drivers. All of these are situated in a common, flat address space, as seen in fig. 2.3b. Unikernels are thus special-purpose kernels, built to run a single application in a virtual machine, considering that virtual machines are regularly employed to serve just a single application.

Unikernels are established on the older idea of library operating systems, which were introduced with the innovative at the time and very relevant today exokernels [21]. In a library operating system the majority of the functionalities traditionally offered as part of a monolithic kernel (e.g. file systems and networking) are extracted from it, becoming independent elements in the form of libraries which accompany an application. This rearrangement grants the application the liberty of choosing which of these elements in requires, as well as which implementation of each it prefers. Therefore, the application can optimize its operation through said choices, but at the same time is burdened with this extra duty.

A big hurdle in implementing unikernels is the diversity of devices and consequently drivers necessary to operate them. This is overcome thanks to the hypervisors' device model, encompassing few, common and well documented devices. Another obstacle is the complexity associated with matching the various components together to create the final executable image. There are numerous solutions proposed to this, each forming a "unikernel framework". Virtually all of those are free software projects, each providing a set of libraries for inclusion by the user applications, as well as the tools necessary for building the images. Each

of these frameworks typically takes one of two principal approaches:

- Clean-slate, providing custom APIs. In this case, both library and application code is written in the same programming language. A major downside to this approach is that the applications need to be written from scratch in the language and using the API of the respective framework.
- Compatible, providing standard interfaces (e.g. POSIX), which allows running existing applications with minimum to no modifications, independently of any programming language (binary compatibility). In this case, the biggest tradeoff is the commitment to legacy interfaces, often hindering taking full advantage of the unikernel potential.

For example, in the first category one finds MirageOS [33] and IncludeOS [6], whereas in the second we note RumpRun [47], OSv [28] and HermiTux [37]. Overall, there are several unikernel frameworks, varying in popularity (with none prevailing), maintenance status (from active to practically abandoned) and origins (academic, corporate-backed or personal projects).

2.3.1 OSv

OSv is a unikernel framework supporting applications written for Linux, while offering its own API which new applications can opt to use [28]. It has been designed with the cloud in mind, in order to accommodate modern applications most often encountered there. The project was started by Cloudius Systems (later ScyllaDB) but lately it has been maintained and enhanced by a small team of volunteers. It is made available under the terms of the BSD software license. Its community makes use of a mailing list[1] for its development: submitting patches, code reviews and general discussion.

Although it is mostly written in C++, also incorporating C code from other projects, this imposes no restriction on the applications it supports, as long as those don't make use of operations that by nature are not applicable in unikernels: system calls in the family of `fork()` and `exec()`. Moreover, it is supported on several hypervisors, including QEMU/KVM, Xen and firecracker [14]. As a general note, it is one of the most advanced unikernels in terms of features, and more heavy-weight as a result of that.

Regarding file systems, OSv offers many choices. First of all, it boasts a complete virtual file system (VFS), based on that of Prex [8]. Beneath that are implementations of various file systems, which can be grouped in two categories:

- Pseudo-file systems, corresponding to those of Linux: devfs, procfs, sysfs and ramfs.

[1] `https://groups.google.com/forum/#!forum/osv-dev`

- Conventional file systems: ZFS (based on FreeBSD's implementation), rofs (custom, read-only file system with simple caching functionality) and NFS (powered by libnfs [30]).

2.3.2 Alternatives

As expected, there have been alternative approaches to the problem of bloat in the virtualized software stack. By far the most popular at this point is that of containers. These belong to the broad class of OS-level virtualization [60] (see fig. 2.3c).

There exist multiple techniques for implementing containers, most of which rely on Linux kernel mechanisms, like cgroups and namespaces, to achieve isolation between separate containers. What's common among all container technologies is that applications running in all containers on the same host share the same kernel instance. One could say that in this case there is a single kernel but many user spaces.

Probably the chief advantage of containers is that they are a lot "lighter" than virtual machines: much lower startup times (compared to a VM with a general-purpose guest), greater flexibility and lower resource usage, resulting in higher density (instances running concurrently on the same physical machine). Their weakest point is that they offer much weaker isolation than virtual machines, due to the common kernel serving them [34].

2.4 Shared file systems

"Classic" file systems are implemented over block devices ("disks") connected via a local system bus. The software comprising them is usually part of an operating system kernel which assumes it is the exclusive user of the file system for as long as that is mounted. This more often than not constitutes a restriction with attempts to lift it since the advent of computer networking, in order to allow concurrent use of a file system by more computers, rendering it shared. The most prevalent and among the oldest representatives of shared file systems is NFS (Network File System) [48], with this class of file systems gaining many new members with the proliferation of distributed systems in recent years.

Virtualization can be viewed as a special case of multiple, interconnected machines: the host and the virtual machines running on it. The demand for a file system shared between them is thus present and in fact very common due to their coexistence on the same physical machine. After all, the host's file system is yet another one of its resources, so it is expected for it to be accessible by the guests.

Most solutions for host-guest shared file systems rely on the existing, network file systems, not differentiating between this case and that of separate hosts. This approach builds of course on the network connection between the parts, fully reusing preexisting solutions. A popular example of shared file system in the context of

virtualization is VirtFS [26] which depends on the 9P protocol [1]. Although the latter is a network protocol, VirtFS does not use the network as its transport layer, granting it improved performance.

It is worth mentioning that in the case of containers access to a file system shared with the host is also in high demand. One way this is accomplished in Docker [2] (by far the most prevalent container runtime) is through bind mounts [11], taking advantage of the common kernel to mount a host directory in the container's file system.

2.5 virtio-fs

The shared memory underlying both host and guests is an element not fully taken advantage of by current shared file systems. Virtio-fs [54] sets out with the goal of changing that. Its differentiating factor is that it is not dependent on the network at the transport layer (using virtio instead), neither at the protocol layer (where it opts for FUSE). These two choices allow for better performance as well as local file system semantics, which manifests e.g. in coherence issues.

The existing virtio-fs implementation in QEMU is distinctive in its architecture. Whereas device implementation is usually fully contained inside QEMU, here a significant part is split in the so-called virtiofsd (virtio-fs daemon), a separate process handling all file system operations on the host, while QEMU is left with the task of implementing the virtual device. On one hand, has the merit of higher security, since virtiofsd can be sandboxed with the host's mechanisms (namespaces, seccomp). On the other hand, it is modular, facilitating easy switching between different virtiofsd implementations (e.g. with one backed by a distributed file system instead of the host's local file system). Splitting the implementation in separate processes is made possible by the host-user protocol [12], which evolves the idea of vhost in QEMU [23]. Pushing devices out of the VMM has many advantages and is expected to be favored in the future [24].

2.5.1 Virtio

Virtio is a standard specifying devices exclusively for virtualized environments [51]. It enjoys wide support and is considered the de facto solution to the problem of fragmentation in virtual device models. Offering an efficient and extendable mechanism, it builds on physical device terminology and mechanisms, thus assisting in new support for it and enabling reuse of existing driver code in guests.

The standard consists mainly of two notions:

The transport layer which defines a generic way for device discovery, configuration and normal operation (bidirectional data transfer). The latter is achieved via so-called "virtqueues", relying on the shared memory between guest and hypervisor. The transport layer, initially given an abstract specification, is

specialized by three implementations: PCI bus, memory-mapped I/O (MMIO) and channel I/O (used in cases of the IBM S/390 architecture).

The set of devices which builds on the transport layer to specify for each device the available configuration fields, the number of virtqueues and the messages exchanged over those for its operation.

2.5.2 FUSE

FUSE (Filesystem in Userspace) is a protocol introduced in Linux to enable file system implementations in user space instead of inside the kernel [4]. As made clear in fig. 2.4, the file system operations are carried out by a user space daemon communicating with the kernel according to the protocol via a special character device (`/dev/fuse`). When the kernel receives VFS requests corresponding to a FUSE file system, it forwards them to the appropriate daemon which acts as a backend.

In the context of virtio-fs, FUSE is utilized for communicating the file system requests to the device. Hence its architecture resembles the classic FUSE architecture, with the difference that the client is the guest instead of the kernel, while communication is performed through virtio rather than the FUSE character device. In practice though, virtio-fs and regular FUSE are incompatible because virtio-fs incorporates modifications and extensions to the FUSE protocol. Moreover, the security model differs (it is essentially inverted), with the client (the guest) being untrusted in virtio-fs, whereas in FUSE it is the other way round (by default there is trust in the kernel the daemon is running on). The latter translates in safer request handling in a virtio-fs device backend than in a FUSE backend.

2.5.3 DAX window

For the file read and write operations, FUSE utilizes the FUSE_READ and FUSE_WRITE requests respectively. These involve copying the data and in the case of virtio-fs ensue VM exits. Since these are operations at the core of the data path, it is crucial for them to be optimized. To that end, virtio-fs introduces the DAX window: a "window" of memory shared between the host and the guest where file contents are mapped, providing the guest with direct access on them. This results in skipping the copying (and bypassing the guest page cache in the case of Linux guests), while not every read or write operation implies suspending the virtual machine's execution.

For realizing the DAX window functionality, initially the FUSE protocol is extended with messages for establishing and removing mappings. The guest dispatches these messages to virtiofsd, which carries out all necessary checks and instructs QEMU to eventually perform the actual operation (establishing or removing a mapping). See also fig. 2.5.

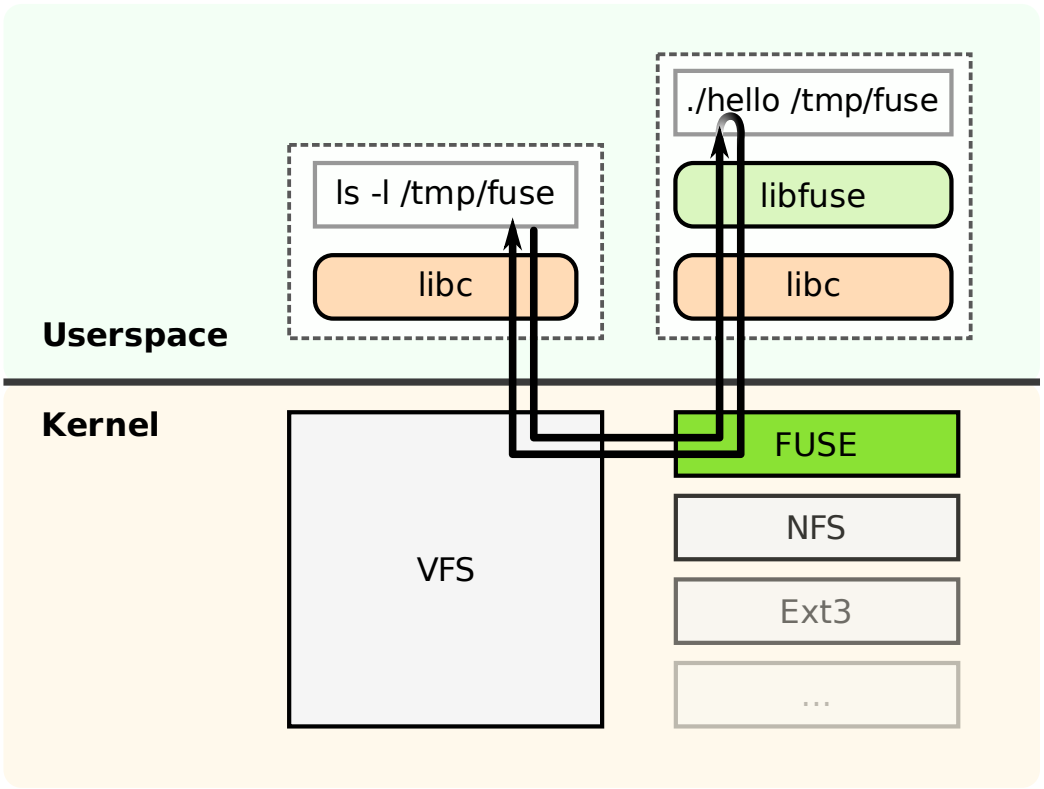


Figure 2.4: FUSE architecture in Linux. By Sven (`https://commons.wikimedia.org/wiki/User:Sven`) licensed under CC-BY-SA 3.0 (`https://creativecommons.org/licenses/by-sa/3.0/legalcode`). Source `https://commons.wikimedia.org/wiki/File:FUSE_structure.svg`.

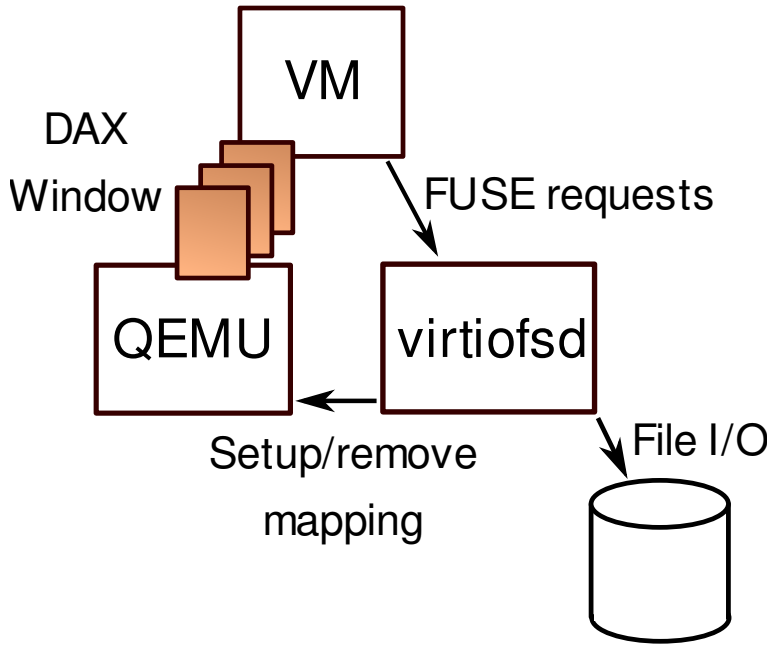


Figure 2.5: DAX window architecture in virtio-fs. By Stefan Hajnoczi (`https://vmsplice.net/`) licensed under CC-BY-SA 4.0 (`https://creativecommons.org/licenses/by-sa/4.0/legalcode`). Source `https://gitlab.com/virtio-fs/virtio-fs.gitlab.io/-/blob/master/architecture.svg`.

Chapter 3

Implementation

A skeleton implementation of virtio-fs existed already in OSv. That included support (via FUSE_READ) for reading files (but not directories). In coordination with the members of the OSv community, we set out to enhance it feature-wise. The main focus of this effort was to support reading through the virtio-fs DAX window, a feature designed and implemented (with "experimental" status) by the virtio-fs community with dramatic performance improvement in mind. Besides that, we had the opportunity to add support for directory reading as well as booting OSv from a virtio-fs root file system. Over the course of these, other structural changes, fixes and enhancements to the existing code were necessary.

This chapter details the most significant of these contributions: reading files using the DAX window and booting off of virtio-fs.

3.1 DAX window in virtio-fs

OSv's internal code structure largely resembles that of most monolithic kernels, separating the notion of a driver and a file system. Drivers (in drivers/) are responsible for adhering to an interface that's common across all members of a particular device family (e.g. block storage or network), allowing the device to be used by the other kernel subsystems. File systems (in fs/) also implement a common interface defined by OSv's virtual file system. This interface encompasses operations such as opening, closing, reading files etc. Most file systems are implemented over a block device, utilizing its interface for "translating" from high-level file system operations to low-level device ones. Naturally, there are exceptions to this, like NFS which is backed by the network subsystem instead.

Virtio-fs is unique in that, as a file system it depends on the corresponding device which is not part of broader family like block devices. This is of course FUSE heritage. There is therefore strong, explicit dependence of the virtio-fs file system from the virtio-fs device driver (fig. 3.1), as is the case on Linux, where the two are implemented together (in fs/fuse/virtio_fs.c). It is then clear that the distinction

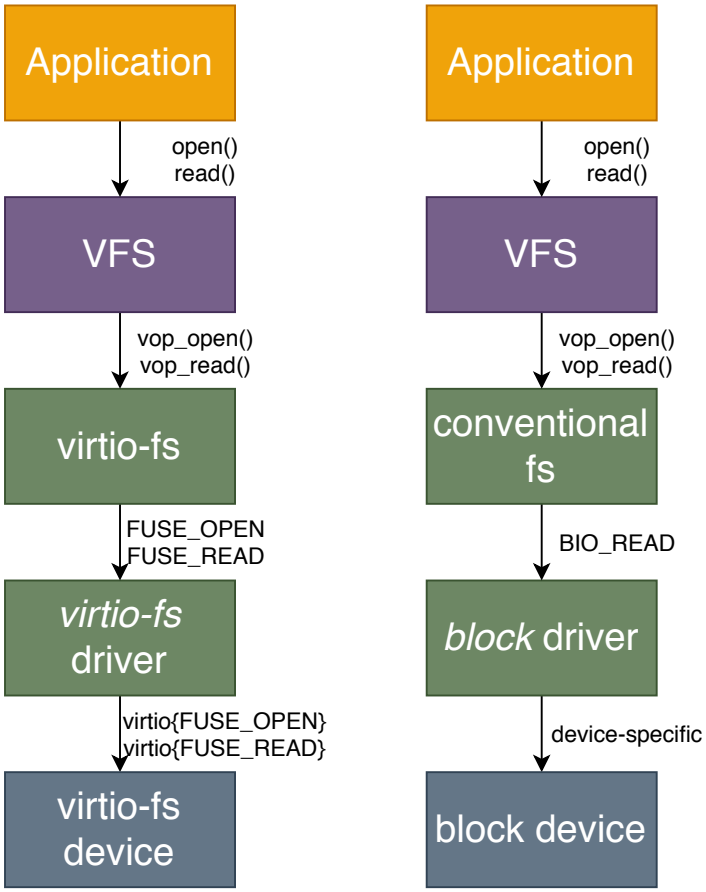


Figure 3.1: Components and dependencies in the guest: virtio-fs compared to conventional local file systems.

between virtio-fs driver and file system (i.e. partitioning of related functionality) is rather malleable. Nonetheless, the ensuing presentation is made through the lenses of this distinction.

3.1.1 Driver

The virtio-fs driver is in charge of direct communication and low-level handling of the device. Apart from initialization (virtqueue discovery, feature negotiation) [51], its main task is transferring the requests and responses to and from the device using the virtqueues. To this end it offers an asynchronous interface for submitting requests encapsulating FUSE messages. The driver remains agnostic to FUSE, handling requests as opaque.

The initial virtio-fs implementation in QEMU was in the form of a PCI device. According to the virtio specification [51], the DAX window is the shared memory

region with identifier ("shmid") 0. In the case of virtio PCI transport, shared memory regions are realized as PCI BARs (base address registers), each of which is exposed as a VIRTIO_PCI_CAP_SHARED_MEMORY_CFG PCI capability, having a unique identifier [51, 56, 38].

The first step towards DAX window support was thus to discover it, if offered by the device. This was almost effortless thanks to OSv's thorough and well-modeled PCI subsystem. Specifically, it consisted of the following changes:

1. Enabling the PCI subsystem to discover multiple capabilities of the same type (in drivers/pci-function.cc). This was necessary as, previously the discovery functions looked only for the first capability of a certain type. This could prove insufficient in the case of multiple shared memory regions, which as mentioned above correspond to a PCI capability each (although in virtio-fs, currently there is only one).

2. Extending the virtio PCI device model (in drivers/virtio-pci-device.cc) so as to discover all the shared memory regions of each device, searching for the corresponding capabilities and recording the attributes of the BARs those designate.

3. Extending the virtio-fs device driver (in drivers/virtio-fs.cc) so as to discover the shared memory region with id 0 (the DAX window) and directly expose it in its interface, as a device MMIO (memory-mapped I/O) region. We note that the mapping of the region to the virtual address space has already been performed by the device model at this point.

3.1.2 File system

The virtio-fs file system assumes a mechanism for communicating FUSE requests (provided by the driver), building its functionality on top of it. Therefore it is responsible for the content of the messages the driver simply delivers.

3.1.2.1 File mapping

Mounting a virtio-fs file system consists of opening a FUSE session by sending a FUSE_INIT request [51]. This carries out the negotiation of the session parameters, including one concerning the operation of the DAX window: the map alignment. The first modification required in the file system was thus to extend the mount process to notify the device of DAX window support in the OS and collect the parameter's value from the response.

The aforementioned parameter restricts file mappings in the DAX window as per the following: both the offset of a mapping's start within the file and within the DAX window must be aligned according to the map alignment. In practice, this is dictated by the usage of `mmap()` [31] on the host, in the current device

```
#define FUSE_SETUPMAPPING_FLAG_WRITE (1ull << 0)
struct fuse_setupmapping_in {
    /* An already open handle */
    uint64_t        fh;
    /* Offset into the file to start the mapping */
    uint64_t        foffset;
    /* Length of mapping required */
    uint64_t        len;
    /* Flags, FUSE_SETUPMAPPING_FLAG_* */
    uint64_t        flags;
    /* memory offset in to dax window */
    uint64_t        moffset;
};

struct fuse_removemapping_in {
    /* number of fuse_removemapping_one follows */
    uint32_t        count;
};

struct fuse_removemapping_one {
    /* Offset into the dax to start the unmapping */
    uint64_t        moffset;
    /* Length of mapping required */
    uint64_t        len;
};
```

Listing 3.1: FUSE definitions for the mapping operations. See appendix A for copyright notice.

implementation (this is also the reason why the map alignment usually matches the host's page size).

As defined by the specification, providing the DAX window by the device as well as using it by the guest are both optional. Moreover, DAX window usage is independent of regular FUSE_READ usage (which is always available). Our implementation strives to satisfy all read requests through the DAX window, but in case that is not available or using it fails, it dynamically switches to the FUSE_READ fallback, as seen in fig. 3.3.

For establishing a new mapping, virtio-fs extends the FUSE protocol, introducing the FUSE_SETUPMAPPING operation. A request of this type is described by a fuse_setupmapping_in struct (the definition can be found in listing 3.1). For canceling existing mappings, FUSE_REMOVEMAPPING has been introduced,

its body consisting of a struct fuse_removemapping_in, followed by the indicated number of fuse_removemapping_one structs.

Finally, we note that according to the specification, a new mapping can be requested which overlaps an existing one. If the overlap is complete, the existing is replaced by the new one, whereas if it is only partially overlapped, it is split into smaller mappings. A request for a new mapping may fail in case of device resource exhaustion, at which point it is recommended to retry the request after releasing resources by removing existing mappings. Therefore, in our implementation FUSE_REMOVEMAPPING is only employed under such exceptional conditions.

3.1.2.2 DAX window manager

From the fine-grained control over the DAX window mappings arises the problem of how to manage it optimally. There are a lot in common between this and the more general problem of memory management in the context of an operating system with paging, due to the restrictions posed by the map alignment. To tackle this we came up with a DAX window manager on the file system level, with all read operations happening through this manager, allowing it to apply the policy described below.

The management policy was designed with these considerations in mind:

- In our case, the file system is read-only.
- The time to fulfill a FUSE_SETUPMAPPING request is independent of the mapping's size.
- A simpler policy translates to a simpler implementation, which is easier to get right and to be performant.

According to them, we arrived at a management scheme summarized by the subsequent characteristics (illustrated in part in fig. 3.2):

- The DAX window is split into fixed-size "chunks" (1 MiB by default). These comprise the building blocks in the sense that all mappings are made in terms of these chunks. Every new mapping starts at an offset both within the file and within the DAX window, which is a multiple of the chunk size and contains an integer amount of chunks. It is then apparent that requiring the chunk size to be compatible with the map alignment automatically satisfies all the alignment requirements placed by the virtio-fs device. This was decided upon, on the one hand for simplicity and on the other hand to serve as the unit of prefetching (with the right choice of chunk size).
- A new mapping always starts from the lowest available address within the DAX window. If there is not enough room (the window is full), then it overlaps the mappings in the highest addresses, only as much as necessary to fit the new one. Hence, a LIFO (Last In - First Out) model is applied, with the DAX window resembling a stack. This was selected in the name of

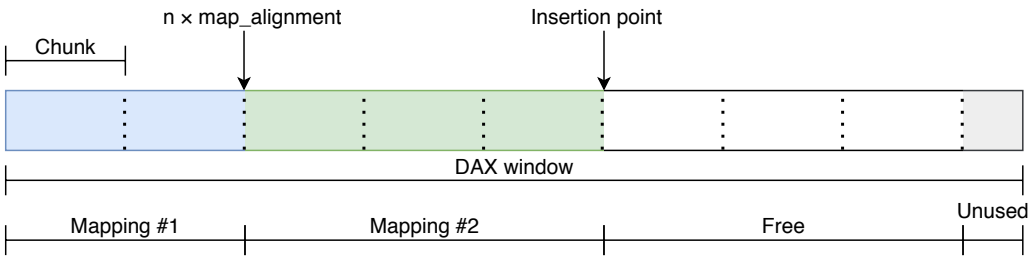


Figure 3.2: Indicative view of the DAX window under the manager.

simplicity as well as to accommodate an application reading a file serially in consecutive requests, by having consecutive chunks in the file be consecutive in the window too.

- No prefetching is done beyond one chunk in order to avoid potential waste of space in the DAX window. Nonetheless, we note that the implementation support a mode (disabled by default) of "aggressive" prefetching, whereby the whole file is mapped upon the first access to it.

The whole read process as pertains to the virtio-fs implementation is depicted in the control flow diagram in fig. 3.3.

3.2 Boot from virtio-fs

Adding support for booting off of virtio-fs in OSv comprised two main components: the main one concerning the kernel in addition to a supplementary one having to do with the automation tools for building and running the images. Both were guided by previous similar extensions, since OSv was already able to use either ZFS or rofs (or even ramfs, as a special case) for its root file system.

In brief, the points involving the root file system in the lifecycle of an OSv unikernel before our modifications were:

1. Building the image (mainly in scripts/build): at this stage the root file system type is selected by the user and the root file system is built with the contents defined by the application. Moreover, here begins the determination of the unikernel's command line, containing options for the kernel itself in addition to the command line of the application itself. This is stored in a temporary file, where the next stage picks it up [39].

2. Executing the image (mainly in scripts/run.py): here the command line is optionally complemented with ephemeral options, e.g. the verbosity level of the kernel messages. Subsequently the finalized command line is written to the image, before that is executed.

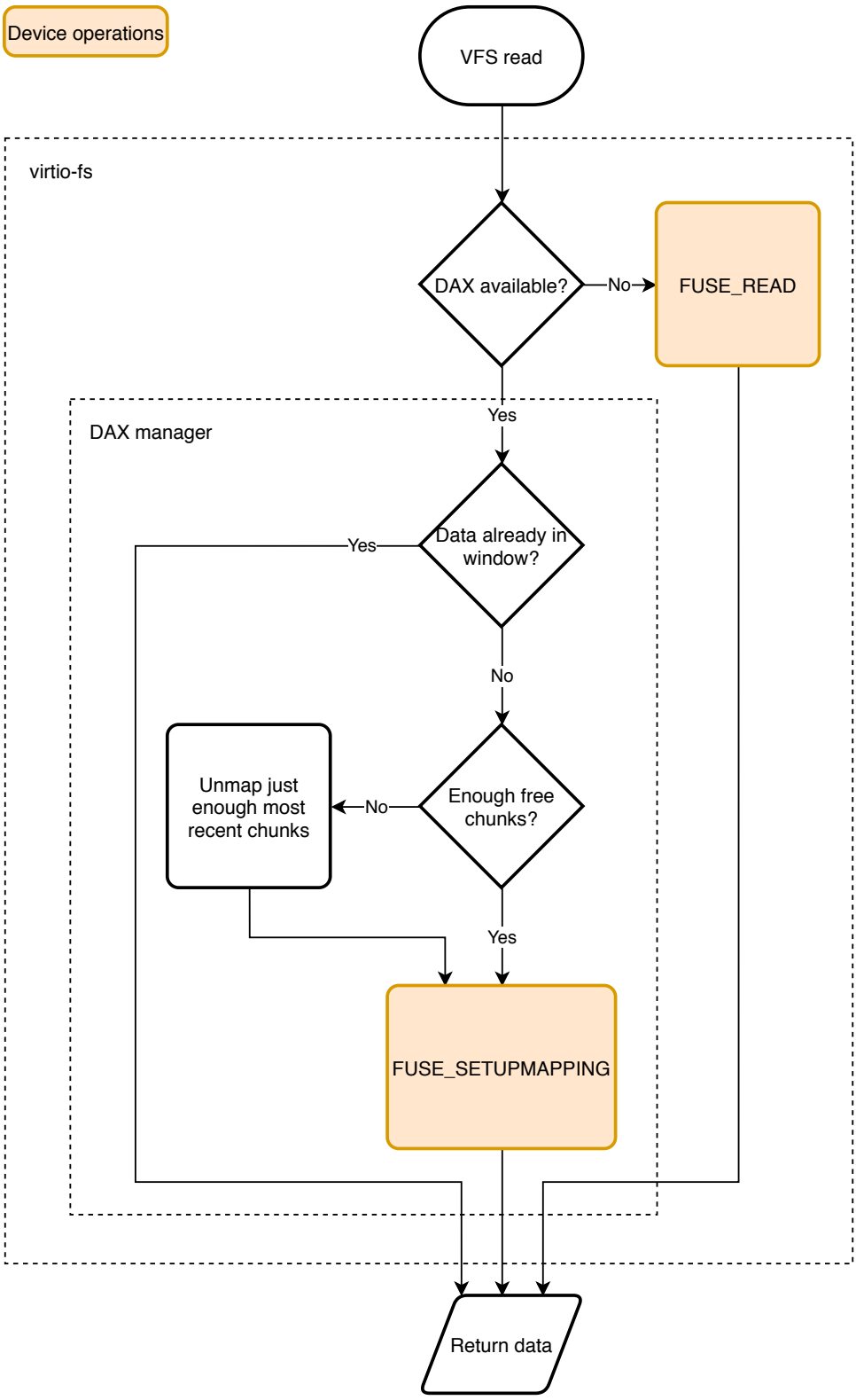


Figure 3.3: Process of reading from virtio-fs under the DAX window manager.

3. Loading the kernel (mainly in loader.cc): following, among others, the initialization of the kernel, the devices and drivers and reading the command line, OSv attempts to transition from its initial, embedded file system (ramfs) to the root file system [42]. This process consists of mounting the file system at some point of the existing, initial ramfs, followed by pivoting the virtual file system to it. These are made simple thanks to the virtual file system, after all prerequisites for its operation are completed.

The determination of which file system to use as root during loading was previously dynamic: there was an attempt to mount the first block device, first as rofs and, if that failed, as ZFS. If both failed, execution continued with the initial ramfs file system. Despite the above process being easy to extend to include virtio-fs, it was clear that there were many silent assumptions, without allowing much user control. Hence, we decided to go the extra mile and grant more control over the process: we added a kernel command line option, `--rootfs`, which allows explicitly specifying the root file system type (between ZFS, rofs, virtio-fs and ramfs). As seen in diagram 3.4, this option is optional and, if it is not specified, the older, dynamic process is adopted, for compatibility. Last, we made the necessary modifications to the build process (scripts/build) for it set this option depending on the file system type specified by the user at build time. Moreover, this way we achieved to make the build process and the available root file system types more predictable and better documented.

Finally, as far as virtio-fs itself is concerned, using it as the root file system did not have any notable challenge in store: if selected via `--rootfs`, it was mounted from the first virtio-fs device (in OSv, in contrast to Linux, those are exposed, serially numbered in the devfs, instead of being identified by their virtio-fs tag). In order to automatically populate a directory on the host with the appropriate contents for each application and Subsequently use that as the virtio-fs shared directory, one can use the `export` and `export_dir` options to scripts/build.

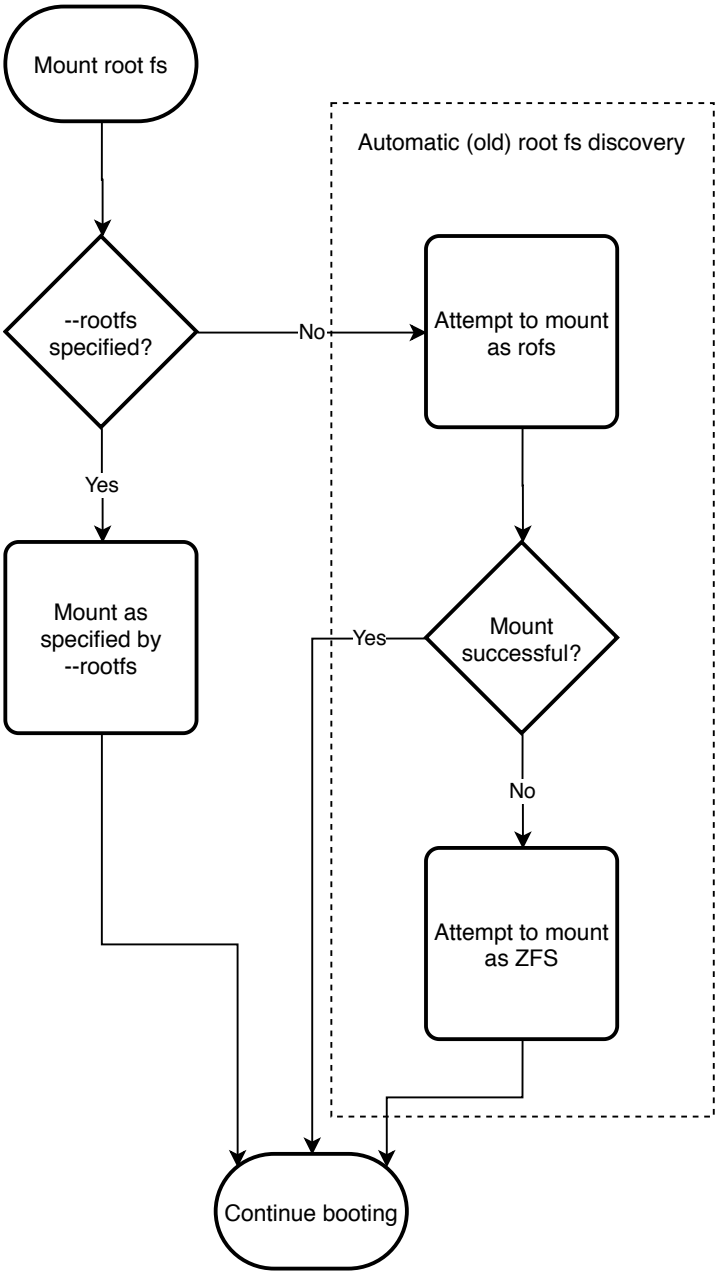


Figure 3.4: Root file system mounting process.

Chapter 4

Evaluation

To evaluate the new file system on OSv, we conducted tests comparing it with the other available file systems, as well as with virtio-fs on Linux, when that was applicable. The tests include three scenarios, as detailed in this chapter: a synthetic benchmark or microbenchmark, a startup time test and an application benchmark.

4.1 Methodology

All the tests were performed on a personal computer with the specifications in table 4.1, while table 4.2 includes the specifications of the OSv and Linux guests. As for their execution, we note the following:

- All tests used a temporary file system (tmpfs) on the host, for the OSv images as well as for the shared directories in the cases of virtio-fs and NFS. This was decided upon so that performance was independent of the host's storage devices.
- Dynamic CPU frequency scaling was disabled, utilizing the "performance" CPU scaling governor on the host.
- The QEMU process was isolated from the rest (virtiofsd, perf, vegeta) in terms of CPU execution (CPU pinning). Specifically, QEMU was dedicated as many CPU cores as the guest's CPUs, whereas the other aforementioned processes were restricted to the rest of the cores available. The reason behind this was to minimize interference between them which could affect the results.
- For measuring CPU usage we chose perf [44] (version 5.7.g3d77e6a8804a), the `perf stat` command with the "task-clock" perf event to be precise. We used this to acquire measurements for the QEMU process, virtiofsd, the vhost kernel thread [23] and the NFS kernel threads, each where applicable.

CPU	Intel Core i7-6700 @3.4GHz
Memory	2×8 GiB @2666MHz
Swap	No
Linux kernel	5.8.13-arch1-1
QEMU	5.1.50 @ c37a890d12e57a3d28c3c7ff50ba6b877f6fc2cc [53]

Table 4.1: Specifications of the host.

CPUs	4
Memory	4 GiB
DAX window	4 GiB
OSv	5372a230ce0abf0dc72e92ec1116208145e595c5 [43]
Linux kernel	5.8.0-rc4-33261-gfaa931f16f27 [52]

Table 4.2: Specifications of the guests.

- All tests were repeated 10 times, of which we later present the mean value and standard deviation of the measurements. In OSv's case every repetition consisted of a fresh VM instantiation, whereas in Linux all repetitions were performed in a common virtual machine execution.
- Virtiofsd was run with its cache mode disabled (`cache=none`) and a single thread (`--thread-pool-size=1`). The latter was selected because it led to slightly more consistent measurements, though without improved performance, as was mentioned in a virtio-fs mailing list discussion[1].
- For running OSv we made use of the helper script (scripts/run.py) it provides, after modifying it in order to orchestrate execution of all other tools (virtiofsd, perf, vegeta). As for the virtual machine's networking, QEMU's tap backend was used, with vhost enabled, with a static IP address granted to the VM.
- For the NFS server, the respective Linux implementation was used on the host. All tests were carried out with NFS version 3, since version 4 was not functional on the OSv side. Readahead on the NFS client (libnfs), which is equivalent to the chunk size in our implementation of the DAX window manager, was set to 2 MiB.

[1] `https://www.redhat.com/archives/virtio-fs/2020-September/msg00068.html`

4.2 Microbenchmark

4.2.1 Description

For measuring the file system's performance we used the flexible I/O tester (fio) [15] in its 3.23 version. In order for it to run on OSv, two modifications were necessary[2] in addition to the appropriate arguments to fio's configure script, to disable features not provided by OSv. We note that the exact same executable (with the above disabled) was used in Linux as well.

The comparison includes ZFS, rofs, ramfs, virtio-fs (with and without the DAX window and with a ramfs root file system) and NFS on OSv, virtio-fs (with and without the DAX window and with an ext4 root file system) on Linux, as well as tmpfs on the host, which we include as a baseline. Specifically for the testing process:

- In all cases except ramfs, the fio test files have been layed out in advance and subsequently placed in the appropriate location (virtual disk or shared directory). We chose this approach despite fio being able of laying out the files dynamically at run time since some of the file systems are read-only. Because ramfs is restricted as to the size of individual files in its image, the test files are generated at run time in its case.
- We examine two cases as to the test files:
  - A single, large file (1 GiB).
  - Multiple (10) smaller files (80-100 MiB). We note that in all tests the files are identical (each respective file is of the same size).

  In both cases the overall file size does not exceed 1 GiB, in part due to this being limited by the guest's memory for rofs and ramfs, as well as the DAX window for virtio-fs with DAX.
- We examine two cases as to the file reading pattern: serial (`rw=read`) and random (`rw=randread`).
- In all cases there is a single thread reading (`numjobs=1`).
- On Linux, for the automation of the tests we relied on Vivek Goyal's relevant helper[3] [22]. This invalidates the page, inode and dentry caches using a sysctl (/proc/sys/vm/drop_caches) before each fio run.
- In the case of Linux (guest and host), there was no CPU usage measurement, since that was deemed unworthy, given the difference in nature from OSv (general-purpose OS vs unikernel).

[2] All modifications can be found in the "fio-3.23-osv" tag of the git repository at `https://github.com/foxeng/fio`.

[3] commit hash 8c99f50c878cb39db76abec7e0882fd83c99f4b3

pattern	virtio-fs	NFS	virtio-fs DAX
serial	1.0	3.1	6.5
random	1.0	0.4	5.7

Table 4.3: Normalized fio throughput, virtio-fs and NFS on OSv.

4.2.2 Results

As we can see in fig. 4.1 through 4.4, the highest throughput is achieved by ramfs on OSv, even higher than that of tmpfs on the host. This is so because, with the data in memory the virtualization overhead is minimized, while the simpler file system implementation and lack of mode switches due to system calls on OSv seem to make the difference. Virtio-fs with DAX offers the next best performance on OSv, in all cases, also having the lowest impact in terms of processing resources consumed, again after ramfs.

Focussing on virtio-fs and comparing between OSv and Linux, we discern consistent behavior in all cases: in both operating systems the DAX window outperforms virtio-fs without it, by a large margin ($> 6\times$ throughput), while on OSv we notice $20 - 30\%$ better performance than Linux. The latter is expected, in part due to the simpler, hence more efficient OSv implementation, supporting a lot less features and in part due to the relative advantages of a unikernel.

We owe a mention to the comparison between virtio-fs and NFS, the only shared file systems on OSv. Here virtio-fs with the DAX window leads in performance, with NFS following and virtio-fs without DAX trailing, in the serial read tests. As seen in table 4.3, the pattern is different in the random read tests, with NFS performing the worst, both in terms of throughput and of CPU usage. It seems thus to have been much more adversely impacted than virtio-fs by the change in the access pattern which adds pressure to the speculation and caching mechanisms of all file systems.

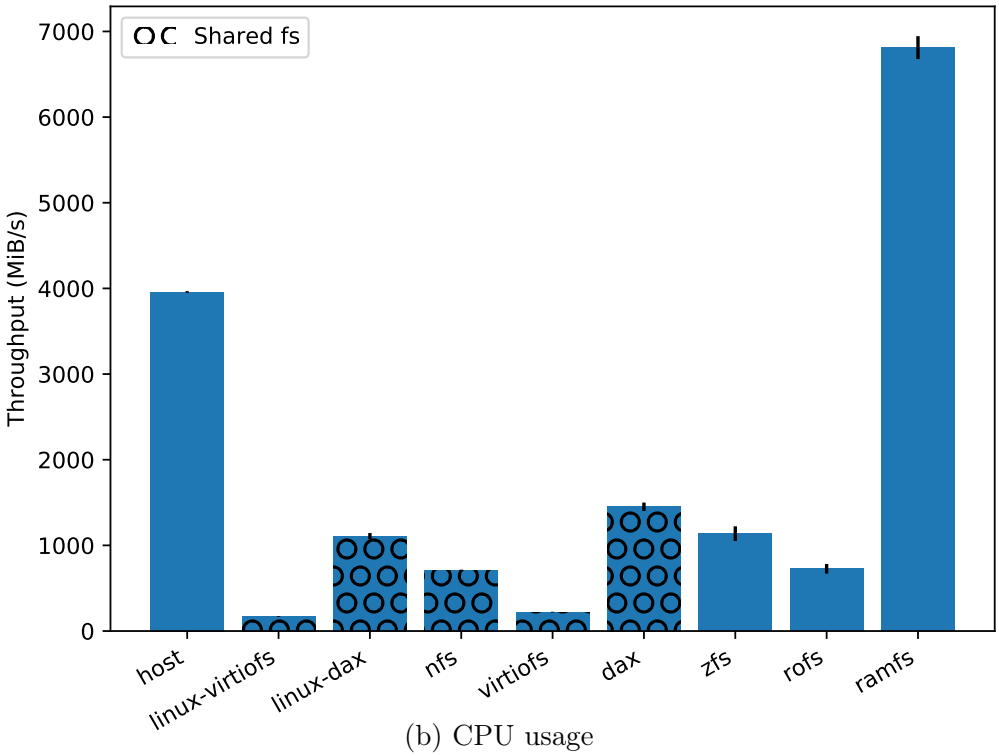
4.3 Startup time

4.3.1 Description

In order to evaluate startup time on virtio-fs we chose an application of those already ported to OSv as examples [40]. Specifically, we selected spring-boot-example, a simple web application from [25], built using the spring boot framework, in its 2.3.4 version.[4] As for the Java runtime, we chose openjdk8-zulu-full, again from OSv's application repository. This application was qualified as representative of stateless web applications, prime examples of the class suitable for deployment

[4]All changes made for our tests can be found in the "virtio-fs-tests" tag of the git repository at `https://github.com/foxeng/osv-apps`.

(a) Throughput



(b) CPU usage

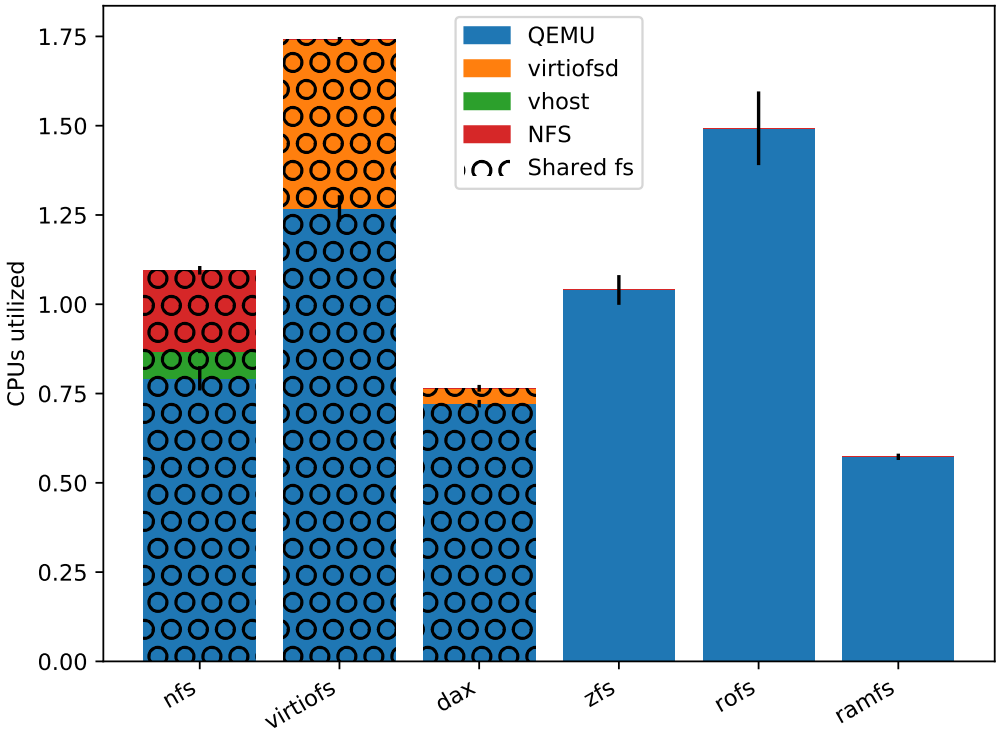
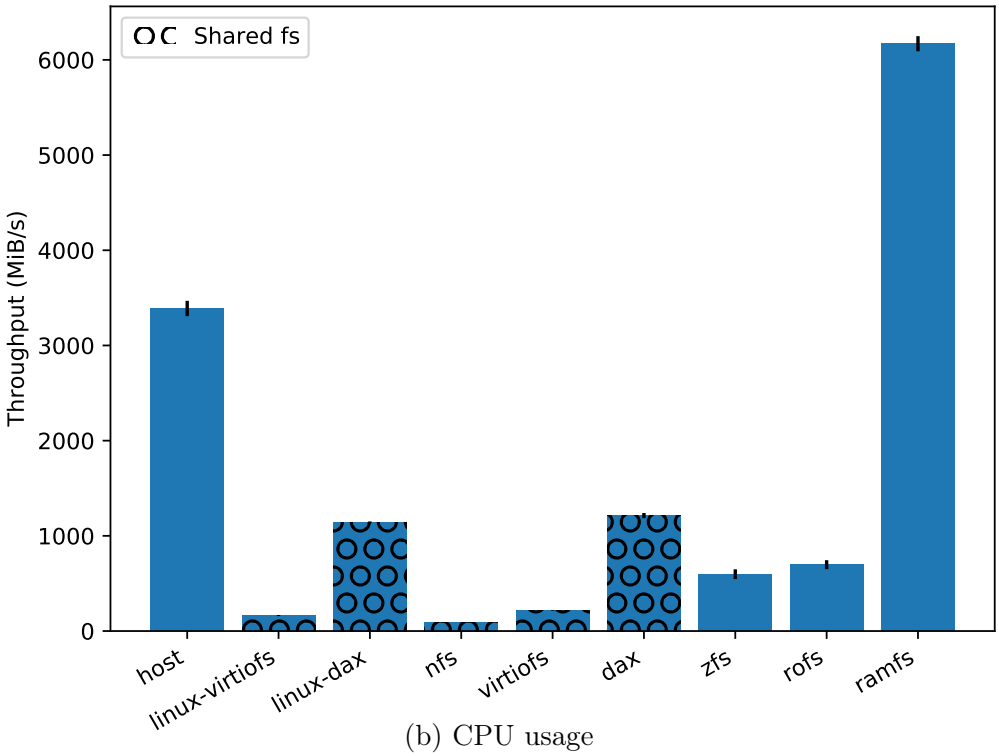


Figure 4.1: fio, single file, serial reading

(a) Throughput



(b) CPU usage

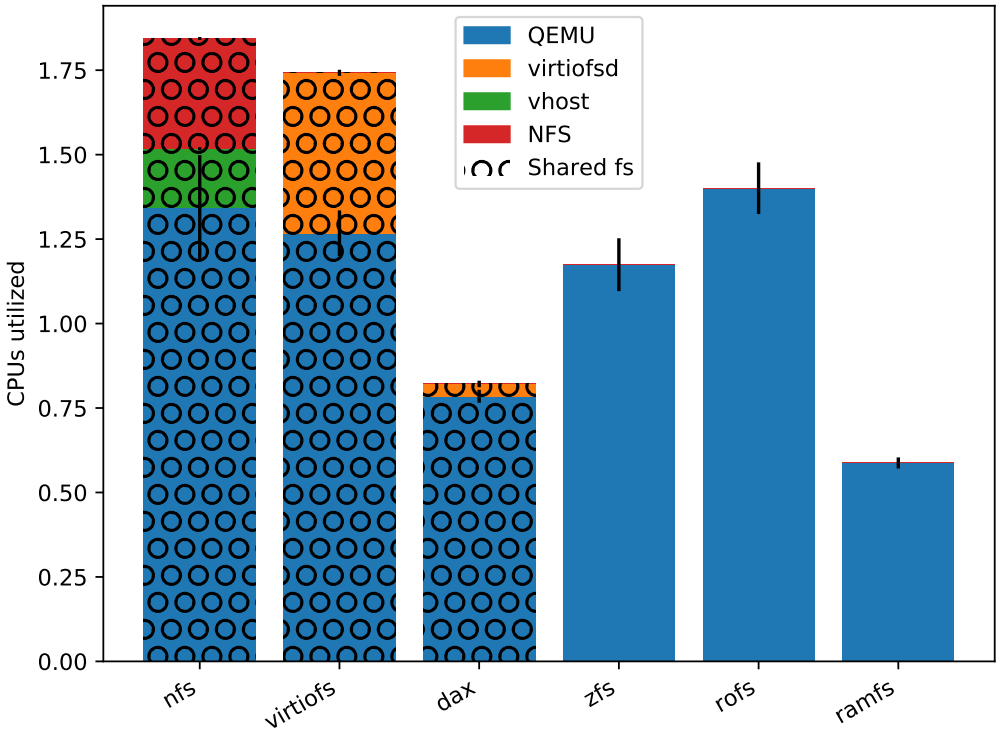
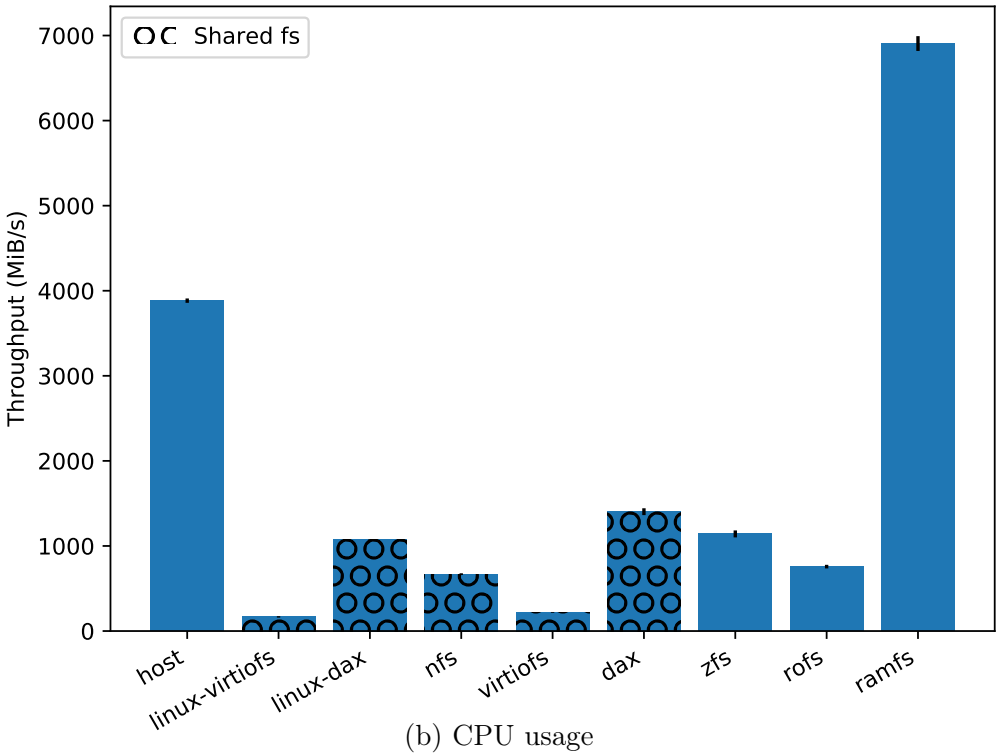


Figure 4.2: fio, single file, random reading

(a) Throughput



(b) CPU usage

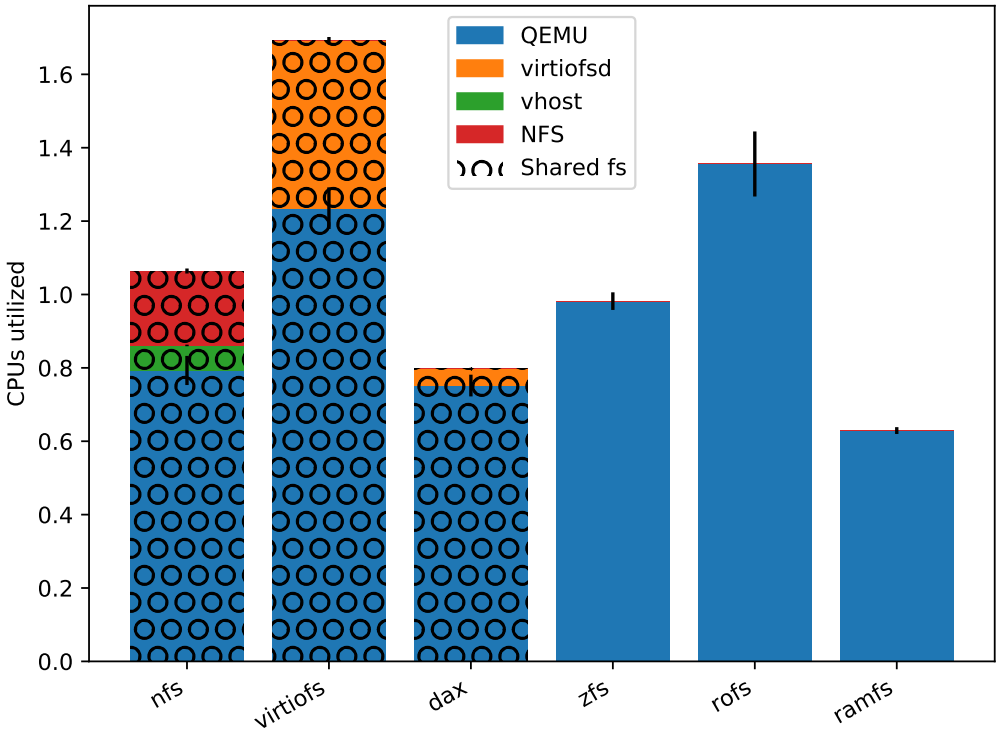
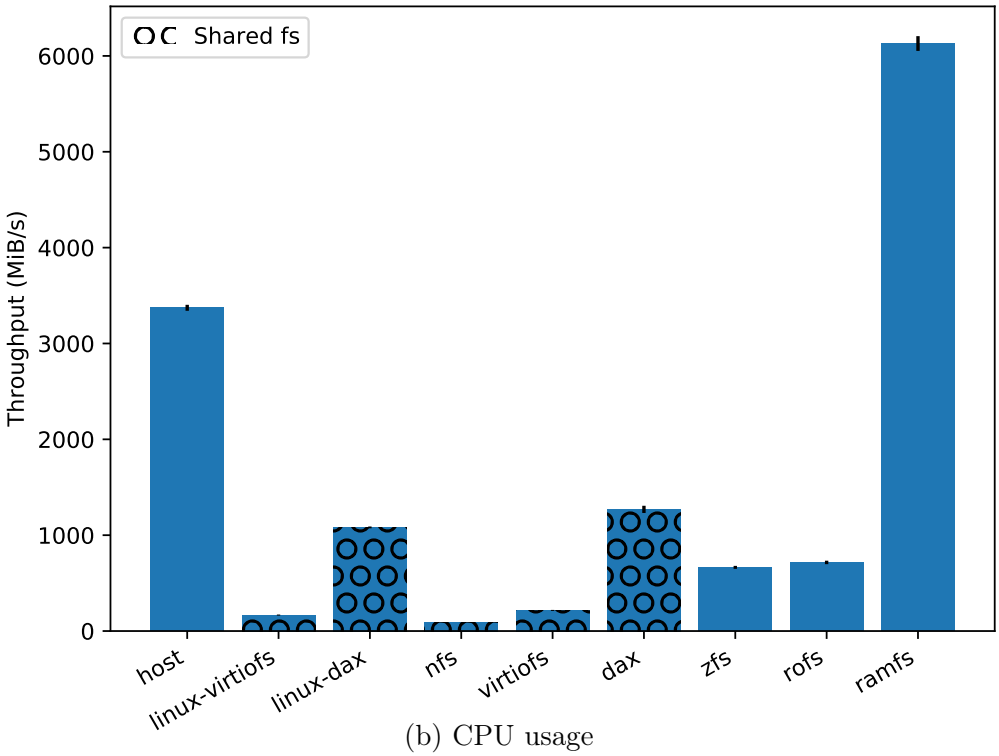


Figure 4.3: fio, multiple files, serial reading

(a) Throughput



(b) CPU usage

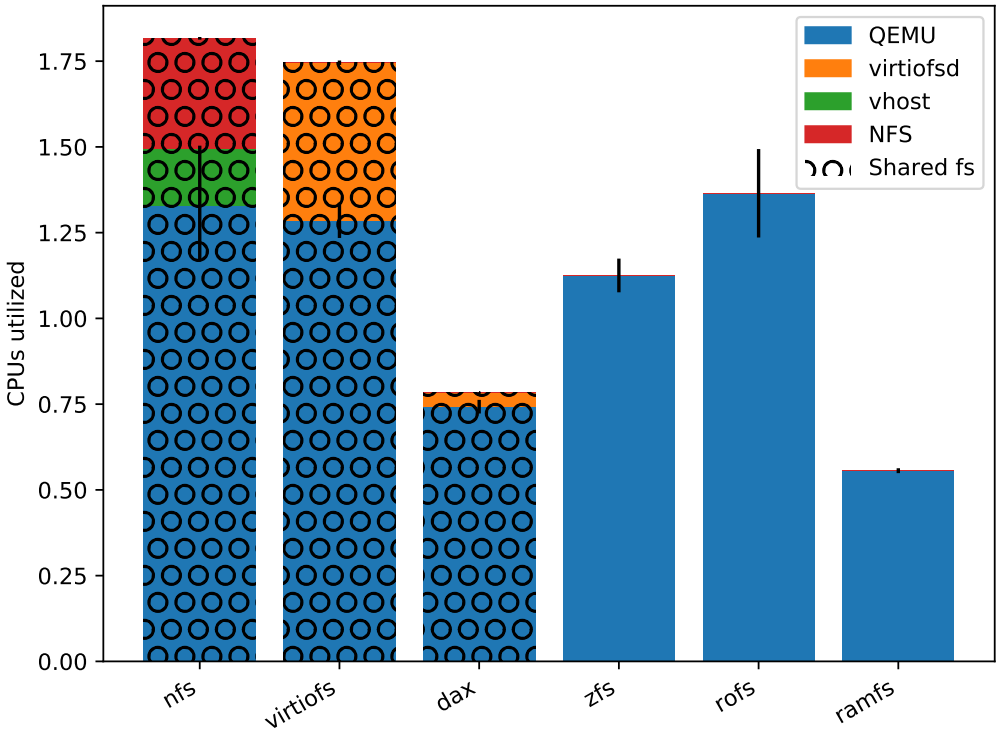


Figure 4.4: fio, multiple files, random reading

ZFS	rofs	ramfs	virtio-fs	virtio-fs DAX
1.13	1.00	1.08	1.37	1.00

Table 4.4: Normalized total spring-boot-example startup time on OSv.

as unikernels in the cloud.

All OSv file systems usable as root file systems participated in this set of tests: ZFS, rofs, ramfs and virtio-fs. The testing process consisted of booting OSv with each file system as the root file system. This was then left to run for a reasonable amount of time (a few seconds), during which the application reached its fully initialized state and subsequently terminated by the test orchestration mechanism (script), by terminating the QEMU process.

4.3.2 Results

Figure 4.5 depicts a breakdown of the total startup time:

OSv boot is the system's boot time, i.e. the application-independent part.

Root fs mount involves the root file system mount time and the time to pivot "root" (/) to it. We note that this is also a stage of the above, but here it is extracted from it and examined separately.

Application encompasses the time taken for the application's initialization, as reported by it.

As made clearer in table 4.4, in terms of total startup time, virtio-fs without the DAX window is inferior to the rest, with virtio-fs with the DAX window being the best performer along rofs (ramfs and ZFS are slightly slower).

As far as mount time is concerned, it is virtually negligible ($< 2\%$ of OSv's boot time) for all file systems with the exception of ZFS. In its case, mount time takes up roughly 14% of the system's boot time, as expected given the relative complexity of this file system, which translates to a longer initialization procedure.

Finally, it is worth mentioning the noticeably increased OSv boot time with ramfs. This can be justified, considering that in the case of ramfs, the root file system matches the boot file system (in OSv terms, what Linux calls initramfs). This is part of the ELF object containing the kernel, which is uncompressed and loaded during the early boot stages [41], in a process with throughput limited due to the restricted initial environment when booting from BIOS on the x86 architecture. Hence, when in the case of ramfs, this ELF object is up to an order of magnitude larger than in the rest, this is responsible for the increase in boot time. Of course, here we have abused ramfs, which is not destined for such large images.

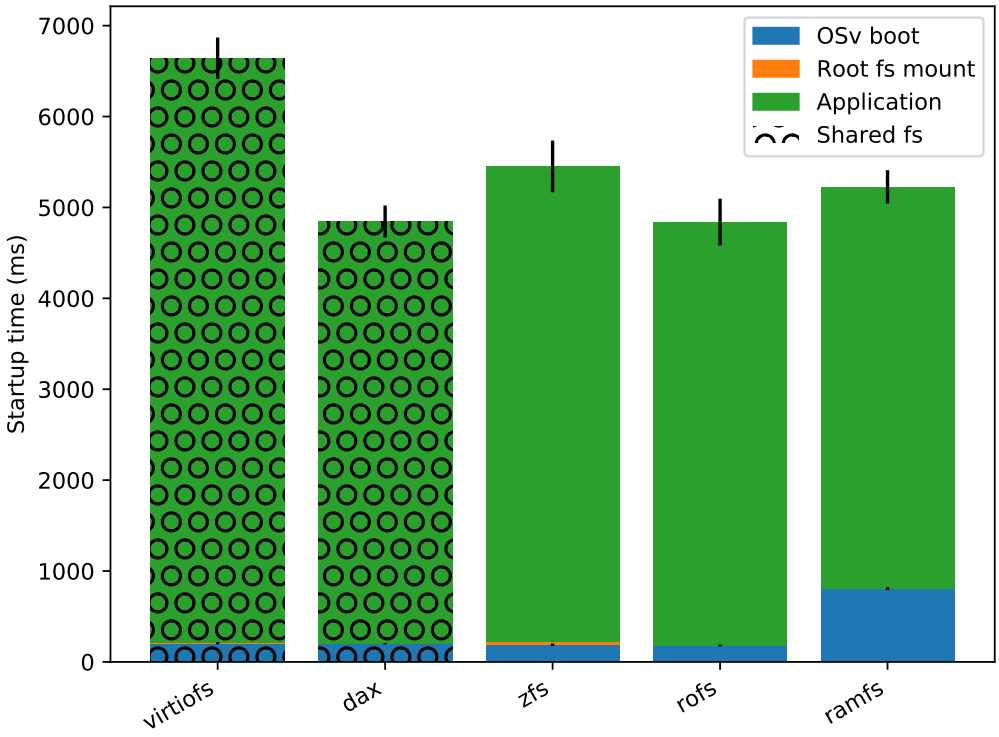


Figure 4.5: Spring boot example, startup times.

4.4 Application benchmark

4.4.1 Description

To evaluate virtio-fs in terms of a complete, real-world application we again turned to the field of stateless applications in the context of the cloud, choosing a static web server scenario. Specifically, we used nginx [50] (in its 1.19.2 version), one of the most popular free - open source web servers, also available in the collection of OSv applications.[5]

This set of tests included all OSv file systems (in the case of virtio-fs, ramfs was used as the root file system), except NFS which led to test failure. Specifically, while the server was handling the first request, after sending few initial response data, the whole process stopped and the guest seemed to "freeze". This was found to not be caused by nginx, since the same behavior was displayed using lighttpd (also included in the "osv-apps" collection) instead. Further investigation is warranted to pinpoint and possibly fix the cause, which from experience seems like a deadlock, possibly in the NFS implementation or OSv's network stack.

The files exposed by the web server were 10 in total, ranging in size from roughly 500 KiB to 12 MiB. Each respective file's size was constant throughout all tests.

For conducting the tests, in the form of HTTP load tests, we used vegeta [49] on the client side, a popular tool for this purpose, in its 12.8.3 version. The testing process (automated by the orchestration script) was:

1. First of all the OSv guest was started, given an one second time frame to complete initialization of the nginx server.

2. Vegeta was launched on the host, configured to generate HTTP 1.1 GET requests for all of the server's files, with the maximum rate possible, using 20 "workers" and up to 4 CPUs, with TCP connection keepalive on.

3. After ten seconds vegeta concluded its work and terminated, at which point the automation mechanism also terminated the guest through the QEMU process.

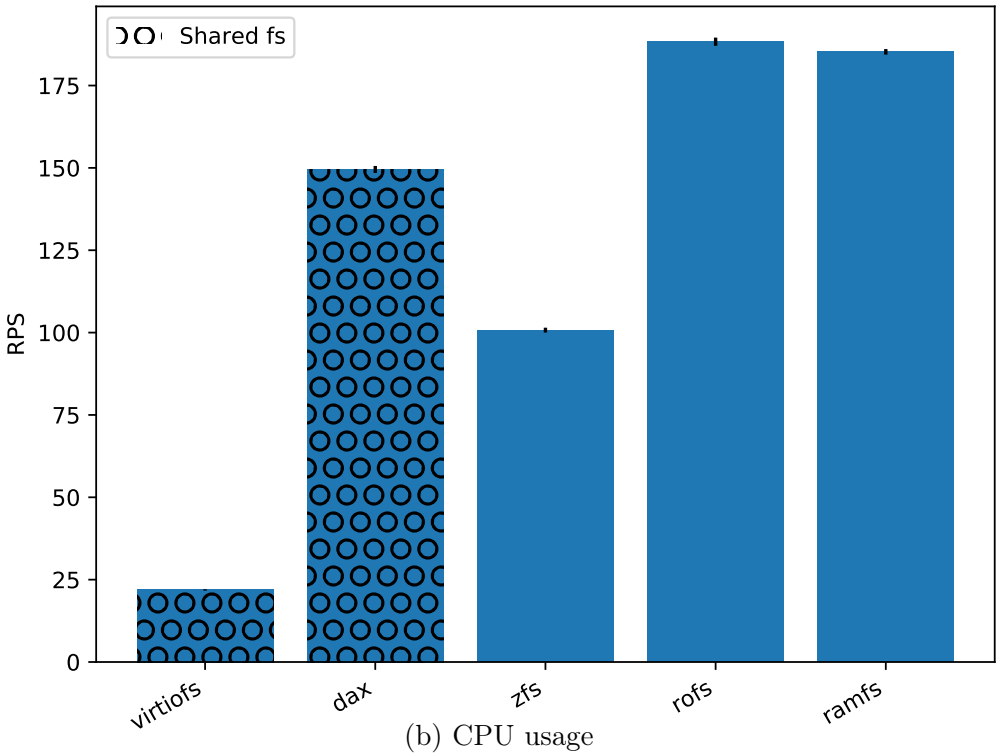
4.4.2 Results

As we see in fig. 4.6a, virtio-fs with the DAX window falls short of rofs and ramfs (which lead) by $\sim 20\%$ in terms of throughput (requests serviced per second). Given its CPU usage is marginally lower than that of the other two, this difference calls for further future investigation through profiling, with focus on the virtio-fs with DAX read datapath on OSv as the primary suspect.

As one could expect in this use case, virtio-fs without DAX trails behind all other file systems by a large margin. This is down to it not making use of any form

[5] All changes made for our tests can be found in the "virtio-fs-tests" tag of the git repository at `https://github.com/foxeng/osv-apps`.

(a) Requests per second



(b) CPU usage

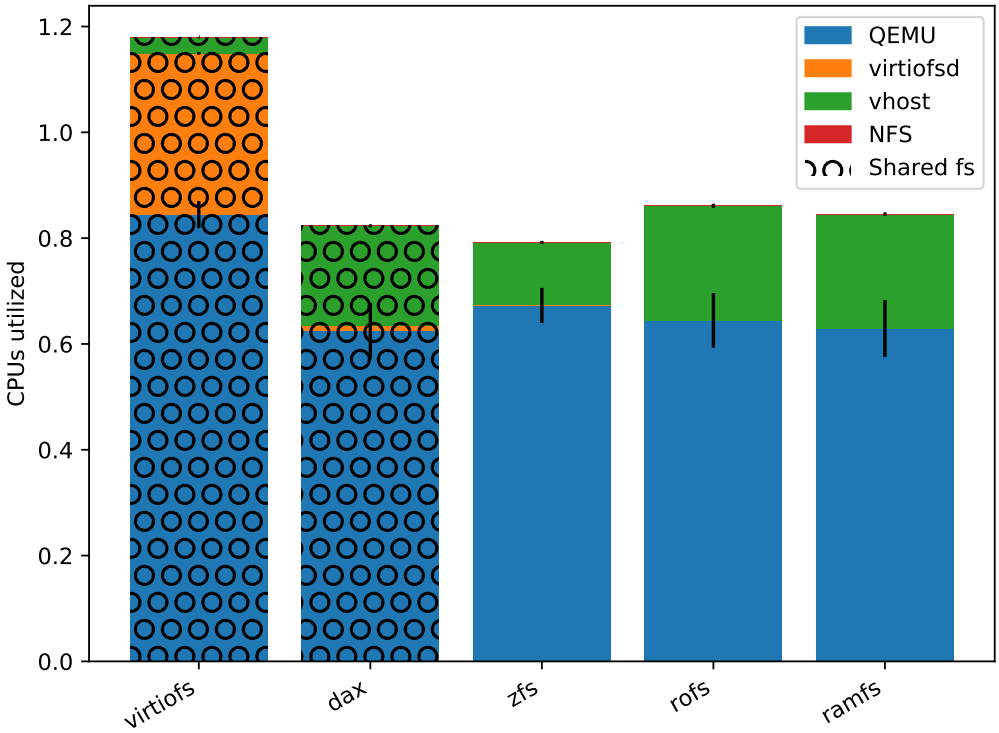


Figure 4.6: nginx HTTP load test

of caching, such that each read operation involves exiting and processing in the host (indicated by the high CPU usage). These operations are very frequent, amplifying the effects of this overhead.

Appendix A

FUSE copyright notice

Βιβλιογραφία

[1] *9P website*. http://9p.cat-v.org/, accessed 30/10/2020.

[2] *Docker website*. https://www.docker.com/, accessed 30/10/2020.

[3] *DPDK website*. https://www.dpdk.org/, accessed 30/10/2020.

[4] *FUSE*. https://www.kernel.org/doc/html/latest/filesystems/fuse.html, accessed 30/10/2020.

[5] *How are serverless computing and Platform-as-a-Service different?* https://www.cloudflare.com/learning/serverless/glossary/serverless-vs-paas/, accessed 30/10/2020.

[6] *IncludeOS website*. https://www.includeos.org/, accessed 30/10/2020.

[7] *Nanos website*. https://nanos.org/, accessed 30/10/2020.

[8] *Prex website*. http://prex.sourceforge.net/, accessed 30/10/2020.

[9] *SPDK website*. https://spdk.io/, accessed 30/10/2020.

[10] *Toro Kernel website*. https://torokernel.io/, accessed 30/10/2020.

[11] *Use bind mounts*. https://docs.docker.com/storage/bind-mounts/, accessed 30/10/2020.

[12] *Vhost-user Protocol*. https://www.qemu.org/docs/master/interop/vhost-user.html, accessed 30/10/2020.

[13] Adams, Keith and Ole Agesen: *A comparison of software and hardware techniques for x86 virtualization*. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 2–13, New York, NY, USA, 2006. Association for Computing Machinery, ISBN 1595934510. https://doi.org/10.1145/1168857.1168860.

[14] Agache, Alexandru, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana Maria Popa: *Firecracker: Lightweight*

virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association, ISBN 978-1-939133-13-7. https://www.usenix.org/conference/nsdi20/presentation/agache.

[15] Axboe, Jens and fio contributors: *Flexible I/O tester repository*. https://github.com/axboe/fio.

[16] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield: *Xen and the art of virtualization*. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery, ISBN 1581137575. https://doi.org/10.1145/945445.945462.

[17] Bellard, F.: *Qemu, a fast and portable dynamic translator*. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[18] Castro, Paul, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski: *The rise of serverless computing*. Commun. ACM, 62(12):44–54, November 2019, ISSN 0001-0782. https://doi.org/10.1145/3368454.

[19] Cloud Hypervisor contributors: *Cloud Hypervisor repository*. https://github.com/cloud-hypervisor/cloud-hypervisor.

[20] Eismann, Simon, Joel Scheuner, Erwin Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup: *A review of serverless use cases and their characteristics spec rg cloud working group*. Technical report, May 2020.

[21] Engler, D. R., M. F. Kaashoek, and J. O'Toole: *Exokernel: An operating system architecture for application-level resource management*. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery, ISBN 0897917154. https://doi.org/10.1145/224056.224076.

[22] Goyal, Vivek: *virtiofs-tests repository*. https://github.com/rhvgoyal/virtiofs-tests.

[23] Hajnoczi, Stefan: *QEMU Internals: vhost architecture*. https://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html, accessed 24/10/2020.

[24] Hajnoczi, Stefan: *Requirements for out-of-process device emulation*. https://blog.vmsplice.net/2020/10/requirements-for-out-of-process-device.html, accessed 30/10/2020.

[25] in28minutes contributors: *spring-boot-examples repository.* https://github.com/in28minutes/spring-boot-examples.

[26] Jujjuri, Venkateswararao, Eric Van Hensbergen, Anthony Liguori, and Badari Pulavarty: *Virtfs - a virtualization aware file system pass-through.* In *Proceedings of the Linux Symposium*, pages 109–120, 2010.

[27] Kivity, Avi, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori: *kvm: the linux virtual machine monitor.* In *Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, pages 225–230, 2007.

[28] Kivity, Avi, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov: *Osv: Optimizing the operating system for virtual machines.* In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 61–72, USA, 2014. USENIX Association, ISBN 9781931971102.

[29] Lankes, Stefan, Simon Pickartz, and Jens Breitbart: *Hermitcore: A unikernel for extreme scale computing.* In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450343879. https://doi.org/10.1145/2931088.2931093.

[30] libnfs contributors: *libnfs repository.* https://github.com/sahlberg/libnfs.

[31] The Linux man-pages project: *mmap(2) Linux Programmer's Manual*, 5.08 edition. https://man7.org/linux/man-pages/man2/mmap.2.html.

[32] Madhavapeddy, Anil, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie: *Jitsu: Just-in-time summoning of unikernels.* In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, May 2015. USENIX Association, ISBN 978-1-931971-218. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy.

[33] Madhavapeddy, Anil, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft: *Unikernels: Library operating systems for the cloud.* In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery, ISBN 9781450318709. https://doi.org/10.1145/2451116.2451167.

[34] Manco, Filipe, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici: *My vm is lighter (and safer) than your container.* In *Proceedings of the 26th Symposium on*

Operating Systems Principles, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450350853. https://doi.org/10.1145/3132747.3132763.

[35] Mell, Peter M. and Timothy Grance: *Sp 800-145. the nist definition of cloud computing.* Technical report, Gaithersburg, MD, USA, 2011.

[36] Nabla Containers contributors: *memfsd repository.* https://github.com/nabla-containers/qemu-virtiofs.

[37] Olivier, Pierre, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran: *A binary-compatible unikernel.* In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 59–73, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450360203. https://doi.org/10.1145/3313808.3313817.

[38] OSDev contributors: *PCI.* https://wiki.osdev.org/index.php?title=PCI&oldid=25084, accessed 24/10/2020.

[39] OSv contributors: *Components of OSv.* https://github.com/cloudius-systems/osv/wiki/Components-of-OSv, accessed 24/10/2020.

[40] OSv contributors: *osv-apps repository.* https://github.com/cloudius-systems/osv-apps.

[41] OSv contributors: *OSv early boot (MBR).* https://github.com/cloudius-systems/osv/wiki/OSv-early-boot-(MBR), accessed 18/10/2020.

[42] OSv contributors: *OSv lzloader and early loader.* https://github.com/cloudius-systems/osv/wiki/OSv-lzloader-and-early-loader, accessed 24/10/2020.

[43] OSv contributors: *OSv repository.* https://github.com/cloudius-systems/osv.

[44] perf contributors: *Perf Wiki.* https://perf.wiki.kernel.org, accessed 18/10/2020.

[45] Popek, Gerald J. and Robert P. Goldberg: *Formal requirements for virtualizable third generation architectures.* Commun. ACM, 17(7):412–421, July 1974, ISSN 0001-0782. https://doi.org/10.1145/361011.361073.

[46] Raza, Ali, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman: *Unikernels: The next stage of linux's dominance.* In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 7–13. ACM, 2019.

[47] RumpRun contributors: *RumpRun repository*. https://github.com/rumpkernel/rumprun.

[48] Sandberg, Russel, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon: *Design and implementation or the sun network filesystem*, 1985.

[49] Senart, Tomás and vegeta contributors: *vegeta repository*. https://github.com/tsenart/vegeta.

[50] Sysoev, Igor and nginx contributors: *nginx*. http://nginx.org.

[51] Tsirkin, Michael S. and Cornelia Huck: *Virtual I/O Device (VIRTIO) working draft*. https://github.com/oasis-tcs/virtio-spec/commit/af6b93bfd9a0ea1b19147e5357d4434b5075b3c8.

[52] virtio-fs contributors: *virtio-fs linux repository*. https://gitlab.com/virtio-fs/linux.

[53] virtio-fs contributors: *virtio-fs qemu repository*. https://gitlab.com/virtio-fs/qemu.

[54] virtio-fs contributors: *virtio-fs website*. https://virtio-fs.gitlab.io/, accessed 30/10/2020.

[55] virtio-win contributors: *virtio-win repository*. https://github.com/virtio-win/kvm-guest-drivers-windows.

[56] Wikipedia contributors: *PCI configuration space — Wikipedia, the free encyclopedia*. https://en.wikipedia.org/w/index.php?title=PCI_configuration_space&oldid=976450463, accessed 24/10/2020.

[57] Wikipedia contributors: *Cloud computing — Wikipedia, the free encyclopedia*, 2020. https://en.wikipedia.org/w/index.php?title=Cloud_computing&oldid=985719602, accessed 30/10/2020.

[58] Wikipedia contributors: *Hardware virtualization — Wikipedia, the free encyclopedia*, 2020. https://en.wikipedia.org/w/index.php?title=Hardware_virtualization&oldid=964209293, accessed 30/10/2020.

[59] Wikipedia contributors: *Hypervisor — Wikipedia, the free encyclopedia*, 2020. https://en.wikipedia.org/w/index.php?title=Hypervisor&oldid=985571176, accessed 30/10/2020.

[60] Wikipedia contributors: *Os-level virtualization — Wikipedia, the free encyclopedia*, 2020. https://en.wikipedia.org/w/index.php?title=OS-level_virtualization&oldid=983612845, accessed 30/10/2020.