Scalable and Fault-Tolerant Storage and File System Services with Non-Blocking Synchronization for Private Clouds

Mincheol Sung
Virginia Tech
Blacksburg, VA, USA
mincheol@vt.edu

Ruslan Nikolaev
The Pennsylvania State University
University Park, PA, USA
rnikola@psu.edu

Binoy Ravindran
Virginia Tech
Blacksburg, VA, USA
binoy@vt.edu

Abstract

We present two system services – the storage service and the file system service designed for private cloud environments to facilitate file sharing across different virtual machines (VMs). Our services are scalable, fault-tolerant, and deliver excellent performance. These system servers are implemented as unikernels running atop of the Xen hypervisor. Additionally, our storage service can leverage NetBSD code, enabling support for a wide range of both legacy and modern storage devices, such as NVMe. Furthermore, the storage service addresses the challenge of transparent fault recovery for storage, a complex task for stateful subsystems, without incurring significant overhead – a well-known challenge in storage systems. Our file system service is designed to be copy-free, enhancing overall performance. We have also designed an inter-VM communication (IVMC) mechanism that fosters scalability and reliability by leveraging lock-free concurrent ring buffers. Since this mechanism is lock-free, our system services communicate with application VMs in a more scalable manner compared to traditional ring buffers used in hypervisors such as Xen. Our lock-free design also aids in restoring storage states during the fault recovery process of the storage server. Our evaluation results demonstrate that our system services achieve performance comparable to that of Linux.

CCS Concepts

• **Software and its engineering → Operating systems**.

Keywords

Xen, file system, storage, microkernel, rumpkernel, unikernel

ACM Reference Format:
Mincheol Sung, Ruslan Nikolaev, and Binoy Ravindran. 2025. Scalable and Fault-Tolerant Storage and File System Services with Non-Blocking Synchronization for Private Clouds. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772235

1 Introduction

In certain enterprise systems,[1] a private cloud is used to run multiple applications in separate VMs. The main idea, also implemented in well-known open-source systems such as Qubes OS [59], is to isolate individual applications in separate VMs and run them atop of a hypervisor, e.g., Xen [9], in the environment that is controlled by the enterprise. Applications may belong to the same or different tenants, and the user and application activity can be monitored by built-in sensors to detect malicious behavior in the private cloud.

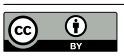
This private cloud does not necessarily run on dedicated servers, it can use bare metal instances [4] or dedicated hosts [5] of public clouds such as Amazon EC2, effectively turning the private cloud into a subcloud. Furthermore, application VMs can be built from unikernels [29], a single-application library OS, due to their lightweightness and memory efficiency. Alternatively, VMs can be a mixture of unikernel and Linux OS instances [47].

There are two fundamental problems that need to be addressed within the private cloud: isolation and data sharing. These problems are in conflict with each other – applications need to run in separate VMs for better isolation, but VMs typically provide very limited mechanisms to share data across different VMs. Furthermore, the mechanisms to share data in this (private cloud) environment should also run in isolation to prevent any interference from malicious actors.

One way to address these conflicting problems is to use "driver domains," special VMs that run drivers and allow sharing I/O across different VMs. This is exactly what systems such as Qubes OS [59] are currently doing: they run special Xen driver domains for storage and networking in dedicated Linux VMs. Furthermore, recent works proposed to use unikernels [47] for the same purpose. However, the granularity of such data sharing is often unacceptable for an enterprise user, especially for storage, where each VM would still use a virtual partition with its *own* file system. In other words, data can only be shared at a very coarse granularity, which would inhibit normal user experience in the private cloud. For example, if every application hosted on a cloud runs in a separate VM, a user will have to take roundabout ways to do trivial tasks such as downloading a PDF file in a browser and then opening it in a separate application.

To overcome present inter-VM sharing limitations, one can use a network-based file system (e.g., NFS). A dedicated NFS server can run as a separate VM, and each application VM would need to mount the NFS access point. Unfortunately, NFS introduces unnecessary performance overheads (e.g., using the network layer even though all VMs are still running locally on the same host) and complicates the setup. Moreover, the NFS-based file sharing mechanism will still not appear as "native" to the user as NFS does not support various extensions available to local file systems (e.g, extended attributes). NFS also has usability issues: it will appear as a "remote" rather than "local" file system. Reliability and recoverability in this setup

[1] Paper authors were previously involved in building a similar system, SAVIOR [49].



SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772235

is also somewhat limited, as the NFS server will run atop a virtual storage device, for which Xen implements no recoverability.

In this sense, we ask a question: Can we build a new mechanism for storage and file system sharing which enables much more transparent integration into different VMs? Moreover, can we build the mechanism which recovers from failures (e.g., an application can be restarted and reconnected to the file system server)?

Another concern is to make inter-VM communication (IVMC) *scalable* to avoid bottlenecks in multi-core systems. As we show in Section 5, typical approaches used in present hypervisors do not scale well as the number of cores increases. This task is very challenging since, for correctness, we also need to ensure *linearizability* [26] of concurrent requests, which inherently limits parallelism.

We present a storage server, which is responsible for low-level block I/O, and a file system server, which is responsible for higher-level file I/O, in a private cloud. They address issues of isolation, performance, and failure recovery. The storage server supports a wide range of legacy and modern storage devices (e.g., NVMe, which is also used by public clouds such as Amazon EC2). The storage server can be transparently substituted with hardware virtualization techniques such as SR-IOV that enable more direct access to hardware (or multiple virtual devices in Amazon EC2). For the virtualization of system component states and our IVMC design, discussed below, we add an abstraction layer between a system server and applications.

In our design, the storage server can be restarted. Applications can thus recover from faults occurring in the storage server. Recovering storage devices requires the entire state (prior to the fault) to be recovered. The challenge here is to make this process without incurring significant overheads, e.g., avoiding duplication of requests or additional IVMCs, a known problem with past systems with server recovery [23].

For the transparent failure recovery, we suggest techniques such as *state virtualization* and *flying data recovery*. For the state virtualization, we separate storage states into two classes: one that is invariant and the other one that can be restored afterwards. Invariant states such as the device node that clients mount (or open) are virtualized and preserved after failure recovery. Other states such as the IVMC channel with pending I/O requests (in the shared client-server memory) are recovered through a special procedure.

Our IVMC design is very scalable due to its use of state-of-the-art multiple-producer multiple-consumer lock-free ring buffers [51]. By adopting these recent advances in the concurrent data structure design, we achieve better scalability compared to existing methods, e.g., I/O ring buffers used in the Xen hypervisor. We present a method for reconstructing IVMC and ring buffers during failures without sacrificing performance and scalability.

The contributions of this paper include: (1) **Two servers:** a storage server (for low-level block I/O) and a file system server for applications (for high-level file I/O) running in separate isolated VMs. Applications typically talk to the file system server; the file system server, in turn, talks to the storage server. (2) A transparent failure recovery mechanism for the storage server with state virtualization and flying data recovery techniques. (3) A special inter-VM communication (IVMC) mechanism that is scalable and recoverable in the event of failures.

Our evaluation in Section 5 shows that our servers allow to attain performance that is on par with Linux.

2 Background

2.1 Private Cloud

Private cloud computing represents a distinct and influential model within the broader cloud computing spectrum. It is characterized by a dedicated cloud infrastructure that is used exclusively by a single organization or entity. In contrast to public clouds, private clouds offer heightened control, customization, and security, making them a preferred choice [6] for organizations with stringent data privacy, compliance, and security requirements. This exclusive control enables organizations to configure the cloud environment to meet their specific needs, ensuring that data and applications are housed in a more controlled and sometimes on-premises setting. Private clouds are further categorized into *on-premises* private clouds, where the cloud infrastructure is hosted within the organization's data centers, and *hosted* private clouds, which are provided by a third-party service provider. The utilization of private clouds has gained prominence in sectors such as finance, healthcare, and government, where sensitive and confidential data handling is paramount. This paper seeks to delve deeper into the nuances of private cloud computing, especially as it relates to file system and storage organization.

2.2 Unikernels

Cloud providers deploy customer VMs with fully functioning OSs, such as Linux, even though most customers only run a single application within the VMs [43]. Such OSs have numerous device drivers and features. Most of them are not used by the customers but can potentially lead to a large attack surface, poor performance, or long boot times [42].

Unikernels [43] can be viewed as a specialized form of library OSs. A unikernel instance includes a single application that is statically compiled together with the necessary kernel components and executes in a single address space. Since traditional system calls are replaced with regular function calls [13, 55], mode switch overheads [62] are avoided, thereby improving performance. Unikernels also typically optimize various OS layers to avoid unnecessary overheads. Since each unikernel instance houses one application, the resulting code base and the attack surface are significantly smaller than that of monolithic OSs. Unikernels are usually executed in a virtualized environment, atop a hypervisor, which provides strong isolation – usually via hardware virtualization – between different unikernel instances, further improving security. Given these benefits, they have recently gained traction in a number of domains including cloud/edge computing [10, 33, 34, 44, 61], server applications [34, 42, 43, 61, 73], NFV [14, 43–45], IoT [14, 18], HPC [36], VM introspection and malware analysis [72], and regular desktop applications [56, 70].

Linux-based Unikernels. UKL [57] and Lupine [35] implement unikernel-like systems[2] from Linux, but they are experimental, not officially maintained by the Linux community, and currently

[2]Lupine is not a unikernel in a strict sense but is partially inspired by unikernels; Lupine's authors called it "a Linux in unikernel clothing."

lack flexibility offered by other unikernels. Moreover, as discussed in [54], wide adoption of Linux-based unikernels in general is problematic due to licensing issues which will only manifest when using the unikernel model.

Rumprun. NetBSD's device drivers can be factored out from its monolithic kernel by using "rump kernels," which are officially maintained by the NetBSD community [29]. Rumprun [7] takes one step further to create a unikernel from rump kernels by using additional layers and running everything on top of the Xen hypervisor, KVM, or a bare-metal machine. LibrettOS [54] has previously demonstrated how a simple network server can be created from rumprun.

We opted to continue moving in this direction by building a more advanced storage server. Since most NetBSD's device drivers can be reused, we reuse most legacy and state-of-the-art (e.g., NVMe) drivers directly from NetBSD.

Hardware isolation is another advantage that comes with the rumprun unikernel. Similar to LibrettOS, we use the Xen hypervisor and its hardware-assisted virtualization mode (HVM) to isolate different instances of clients and servers.

2.3 Xen Hypervisor

Xen is a type-1 hypervisor that was first introduced and developed by the Computer Laboratory at the University of Cambridge. HVM guests are widely supported by Xen. HVM uses CPU virtualization (e.g., Intel VT or AMD-V) [71]. Unlike KVM [31], Xen's *hypervisor* itself is not using any general-purpose OS (e.g., Linux) and is relatively small, which makes it a strong candidate for private clouds, where security and small trusting computing base are crucial.

There are two mechanisms to control interactions between Xen and guest VMs (also known as *domains* in Xen). The first one is through hypercalls, which are synchronous calls from the domains to Xen. The second one is through event channels, which are asynchronous notifications delivered from the domains to Xen [9].

A hypercall is a software trap from guest domains into the hypervisor to perform a privileged operation, similar to system calls in a conventional OS. The event channel is a signaling mechanism that is used in a path from the Xen hypervisor to guest domains. Events can also be sent from one domain to another through virtual interrupts (VIRQs).

Grant tables provide a generic mechanism for memory sharing between domains. Each domain has its own grant table that is shared with Xen. Domains can notify Xen what permissions other domains have on their pages through the grant table. Grant references are integers and are used by the domains to map the pages of the granting domains. Shared pages via grant tables are used by the back-end/front-end drivers for block and network I/O in Xen.

2.4 VirtIO and Xen I/O Drivers

Xen's back-/front-end drivers and (similar) KVM's [31] virtio drivers are well-known and widely used. The key problem with these drivers is that they work at coarse (disk, partition, etc.) granularity, making it impossible to share individual files. Their recoverability and scalability are also very limited. More specifically, if the storage back-end driver fails, a corresponding guest OS will be unable to recover, especially if the virtual storage device was used for critical guest OS files or mounted as a root partition.

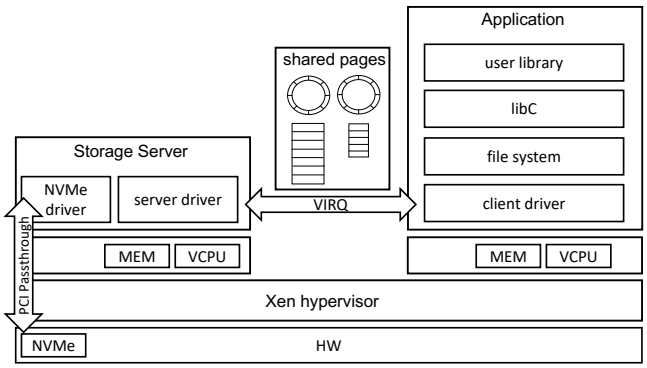


Figure 1: Architecture of storage server.

2.5 Scalable Circular Queue

Scalable Circular Queue (SCQ) is a lock-free multiple-producer and multiple-consumer FIFO ring buffer which is fairly scalable [51]. SCQ consists of two circular queues (`fq` and `aq`), and one array (data buffer). `fq` (*free queue*) maintains indices of unallocated entries of the array. `aq` (*allocate queue*) maintains allocated indices to be consumed. A producer dequeues an index from `fq` and writes data on the corresponding array entry. Then, it enqueues the index into `aq`. A consumer dequeues the index from `aq` and reads data from the array. Lastly, it releases the index by inserting it into `fq`.

SCQ's advantage is that it uses specialized hardware instructions such as fetch-and-add[3] which scale much better as the contention grows than more traditional compare-and-swap implementations [48, 51] while still being fully linearizable unlike other fetch-and-add ring buffers that often incorrectly [39] claim that they are "lock-free."

Although SCQ is not a direct contribution, its integration with real-life systems (including non-trivial recovery paths) and macrobenchmark evaluation have not yet been done.

3 Design

We design servers and applications built upon unikernels, which benefits our design in multiple ways. For example, a unikernel's small image size allows for fast booting, enabling the storage server to restart in a few seconds to recover from faults. Additionally, its small image size exposes a limited attack surface compared to full-fledged OSs. Finally, unikernels are VMs that are specialized for a single application. Therefore, we are able to provide servers and applications with strong isolation, which comes from virtualization.

We do not violate main isolation principles: we use separate ring buffers for each application, the file system server only shares specific memory regions, etc. Although the recovery is dependent on the client, it only affects client-related pieces.

[3]While not all present architectures implement fetch-and-add directly, x86-64 and the most recent versions of ARM64 do, making this instruction widely available.

3.1 Client-Server Driver Design

We use a split-driver model. A client driver is linked as a library in the application's address space so that the application can directly use it. On the server side, a server driver is linked as a library along with other device drivers such as the NVMe driver. The client and server drivers communicate with each other through IVMC that consists of ring buffers. Applications issue block I/O requests through the client driver. Through this IVMC channel, I/O requests are transferred to the server driver. The server driver issues the I/O requests to the actual storage (e.g., NVMe) device driver. As shown in Figure 1, the device driver exclusively accesses the hardware resources due to Xen's PCI passthrough feature.

An application interacts with the client driver through a virtual block device node that the client creates. Data blocks that the application requests are packed in block I/O buffers and transferred to the client driver by the file system which is on the upper layer. The client driver forwards the buffers to the server driver through the IVMC. The IVMC consists of ring buffers and virtual interrupts. The ring buffers are constructed and initialized by the client driver in its shared memory. The server driver maps the memory to its address space. With this design, the data in the ring buffers are safe even if the server crashes and restarts.

3.2 Inter-VM Communication (IVMC)

The IVMC channel between the server and client drivers consists of ring buffers and virtual interrupts.

Ring Buffers. The ring buffers are a key component of our IVMC design. We design a multiple-producer and multiple-consumer ring buffer by leveraging a lock-free ring buffer implementation from [51]. There are two *data* buffers, two *free* rings and four *allocate* rings in the IVMC design. A *small* data buffer is used to contain small data such as write responses and read requests. A *large* data buffer, on the other hand, is used for large data such as write requests and read responses. Each of the free rings maintains the available indices of the small and large data buffers, respectively (i.e., available data slots in the small data buffer are indexed by indices in the small free buffer, and vice versa for the large data buffer). Ring buffers are located in the client driver's memory that is shared with the server driver.

A sender dequeues an index of an unallocated slot of the data buffer and writes data into the slot. For example, when the client driver issues a write request, it dequeues an index from *large free ring* (Figure 2). Using this index, the client driver copies data into the large data buffer slot. Finally, the index is enqueued into *write request allocate ring*. The server driver retrieves the index from the *write request allocate ring* and reads the data of the write request from the large data buffer using the index. It issues a block I/O to the storage device driver and releases the index of the data buffer by putting it back to the *large free ring*.

Once the storage responds to the I/O request, the server driver also forwards it to the client driver. As the response of the write operation has small data such as return value and error number, the server driver uses the small data buffer. Therefore, it dequeues a free index of the small data buffer from *small free ring*. The response is copied into the empty data slot and the server driver enqueues the index to *write response allocate ring*. Finally, the client receives

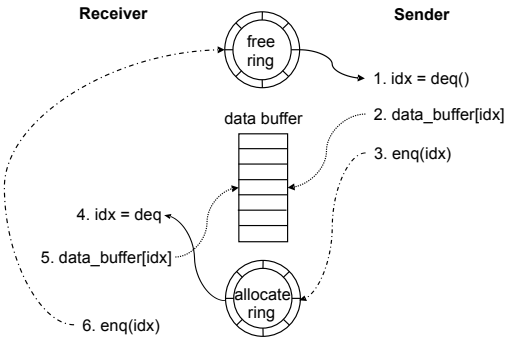


Figure 2: Ring buffer: (1) A producer dequeues an index from the free queue and (2) writes data to the corresponding data buffer entry. Then, (3) inserts the index into the allocate queue. (4) A consumer dequeues the index from the allocate queue and (5) accesses data from the data buffer. Lastly, (6) it releases the index by enqueueing back into the free queue.

the response with the index dequeued from the allocate ring and releases the slot by putting the index back to the *small free ring*.

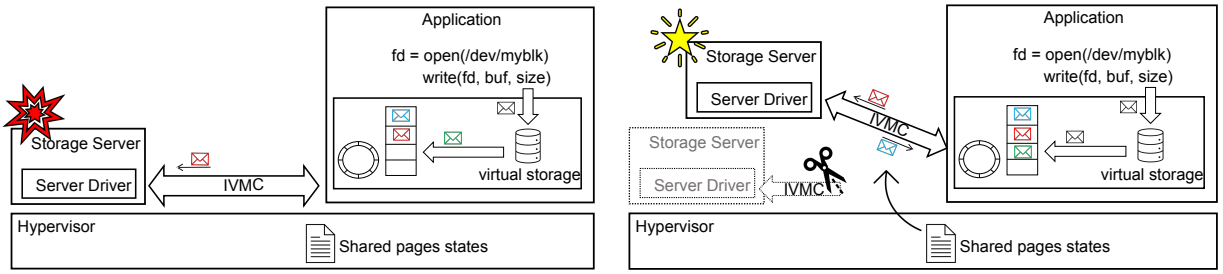
For a read request, the client driver uses the small data buffer instead of the large data buffer. Likewise, the large data buffer is used to transfer a read response because it brings data read from the storage.

Atomic Variables and Virtual Interrupt. When there is no entry to be consumed from the ring buffer, the receiver (consumer) thread goes to sleep to save CPU cycles. In this design, the sender needs to know the status of the receiver for the case when the sender needs to wake up the receiver. We introduce status variables to synchronize the sender with the receiver. Each ring buffer has an atomic variable that represents the status of the receiver. By reading this atomic variable, the sender does not send unnecessary interrupts to the other side if the receiver is active. Otherwise, it sends an interrupt to wake up the other end when the atomic variable represents that the receiver is sleeping.

The interrupt mechanism is another key component for efficient IVMC. We design virtual interrupts between the server and client drivers in multiple places. The main usage of the virtual interrupts is for the sender to notify the receiver to consume entries in a ring buffer. Besides that the sender notifies the receiver, a virtual interrupt is received by the sender. When there is no available index (i.e., the free ring is empty), the sender goes to sleep and waits for some index to be released by the receiver. The receiver sends a virtual interrupt to the sleeping sender when it enqueues an index into the free ring so that the sender can use that index.

3.3 Failure Recovery

The storage server can recover from faults such as memory access violations, deadlocks, and interrupt handling procedure failures. Faults occurring in the storage stack are isolated to the storage server and do not affect the rest of the system. With strong isolation, the storage server can restart to recover from faults. However, challenges come from state recovery. Storage and connection states have to be consistent before and after the storage server restarts,



(a) Failure of storage server

(b) Recovery of storage server

Figure 3: Failure recovery: the client driver virtualizes storage states by creating a virtual storage and device node, allowing the application to mount and open them. Since these states reside within the application's address space, they are preserved across failures. When the storage server fails and restarts, the shared memory and virtual interrupts are rebuilt using the states stored in the hypervisor's memory. Once the new storage server and IVMC are ready, the client driver resends block I/O requests. Each request in the data buffer is associated with a flag, enabling the client driver to identify and resend only those requests for which responses were not received.

and flying requests (i.e., requests are sent, but responses are not yet received) must not be lost to make the recovery transparent to applications.

To address these challenges, we separate the relevant states into two categories: (1) states that must remain invariant, and (2) states that can be reinitialized. We keep the invariant states in the client driver and assume that the application domain does not fail. The client driver virtualizes the states, for example, by creating a virtual device node that the application interfaces and communicates with. The virtual device node is preserved during failure recovery of the storage server and abstracts all block I/O transfers through the IVMC channel.

On the server side, the driver maintains the actual states of the storage stack and reinitializes them upon restart. Since the virtual states (e.g., virtual device node) maintained by the client are needed only by the application and are preserved during failure recovery, the storage server can restart and recover from faults transparently. Our storage server design makes it feasible to separate, virtualize, and safely keep part of the states in the client driver.

During failure recovery, the storage server must rebuild the IVMC with the applications after restarting. As shown in Figure 3, minimal state information is preserved in the hypervisor memory, enabling the restarted storage server to rebuild the IVMC channel following a crash.

Recovery of flying requests is also required for transparent failure recovery. In our design, the ring buffers are allocated in the client driver's memory and shared with the server driver. As a result, the data in the buffer is preserved even when the storage server restarts.

However, the client driver is unaware of which I/O requests are processed by the server driver when the storage server crashes and restarts unless responses for the corresponding requests are received from the server driver. To recover from failures of the storage server, the client driver preserves slots of the data buffers until receiving the corresponding responses from the server driver. Also, the client driver maintains flags in the data buffer and they are used to mark whether the response for the request is received.

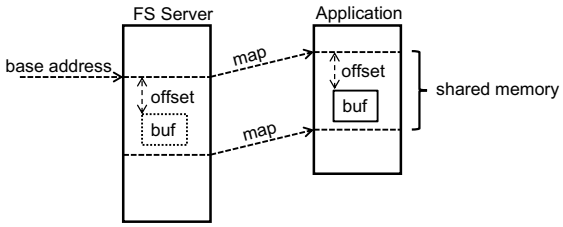


Figure 4: File system server directly accesses file I/O data buffer with base address and offset.

This array of flags is used as a reference when the client driver resends I/O requests to the restarted server driver. In a conservative way, the client driver resends I/O requests whose responses are not received from the server driver even though the requests are processed by the storage device.

3.4 File System Server and Copy-Free IVMC

Our file system (FS) server with the lock-free IVMC is designed to be memory copy-free. Instead of copying the file I/O data buffer to the IVMC layer, an application sends a pointer to the buffer directly to the FS server. For the FS server to directly access the buffer, the application shares its memory area of file I/O data buffers with the FS server. The FS server maps the shared memory in its address space and stores its base address. Since memory addresses from the application differ in the FS server's address space, the server calculates the correct memory locations using the base address and the buffer offsets (Figure 4).

File descriptors (FD) are important for doing file I/O. As an FD table is created and managed by each process, we create an FD mapping table in the client driver. The table maintains the FS server's FD and application's FD mappings. When an application allocates a new FD to open a file, it registers this FD in the client driver's FD table. On the other hand, when an application creates new FDs for sockets or devices (e.g., stdout), the client driver marks these FDs

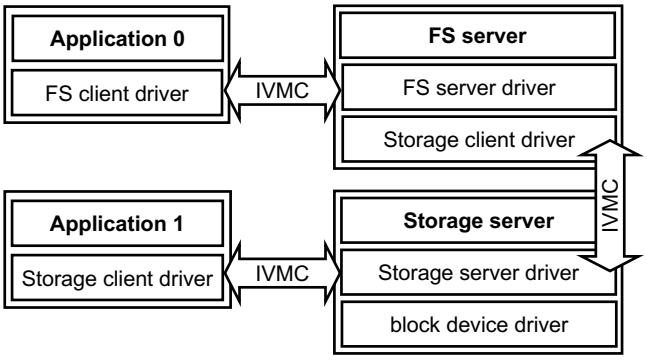


Figure 5: This diagram illustrates how two application VMs achieve isolation and data sharing with two service servers. Application VM 0, which requires file system semantics (FS-level access), communicates with the central FS server VM via IVMC. The FS server manages metadata, locking, and synchronization for shared files, and communicates with the storage server VM when block I/Os are issued. Meanwhile, Application VM 1, which requires raw block-device access (e.g., for a database), bypasses the FS server and communicates directly with the storage server VM over a separate IVMC channel.

accordingly. File operations on such FDs are handled locally by the application, rather than being forwarded to the FS server.

3.5 System Architecture Overview

Figure 5 illustrates the architecture of the proposed system, where applications interact with the file system server through an IVMC channel. The file system server, running as an isolated unikernel domain, not only communicates with the applications but also forwards high-level file I/O requests to the storage server through another IVMC channel. The storage server, also executed as a separate unikernel instance, performs low-level block I/O operations by directly accessing the NVMe device via PCI passthrough. All domains are managed by the Xen hypervisor, which enforces strong isolation while allowing controlled shared memory channels for IVMC. Hardware components such as the NVMe storage and network interface are exclusively attached to the respective service domains, enabling direct device access without involving the application VM. The bidirectional IVMC channels enable low-latency, lock-free communication between the application, the file system server, and the storage server.

4 Implementation

Our storage server and applications are built from the multi-core HVM version of rumprun [54] running on top of Xen. Using hardware (VM) virtualization, the storage server and applications remain strongly isolated. The rumprun unikernel that we use for our prototype is based on NetBSD 9.0's kernel code. As our storage server is implemented using rumprun, it leverages NetBSD's code and a wide range of device drivers (e.g., NVMe). Also, since our prototype runs on top of Xen, it uses Xen's hypercalls, grant table, and event channels for initialization, memory sharing, and virtual interrupts.

We modify the Xen (v4.14) hypervisor by introducing a new hypercall for server and client initialization. Our new hypercall consists of two operations: (1) registration of the server and client drivers; (2) fetching the information of each driver. The registration operation simply stores the driver information (e.g., domain ID, port numbers, and grant references) in Xen's memory. On the other hand, the fetching operation copies this information from Xen's memory to each driver's memory. Once the server and client drivers are connected through their communication channel (i.e., IVMC), our custom hypercall is no longer used.

4.1 Client Driver

The client driver bridges applications with the storage server. It is a library which is linked to the application address space. Client driver memory pages are shared with the server driver through the Xen grant table, and ring buffers are created within the shared pages. Also, the atomic variables for synchronization are attached to the ring buffers.

After the application domain is launched, the client driver queries the server driver information via a hypercall. It then allocates memory and creates six rings, along with two data buffers, in the contiguous page range. Several pages are additionally allocated to pass the grant references of the shared pages to the server driver. The grant references of these pages are stored in Xen's memory through the hypercall. Later, these pages are mapped by the server driver, allowing it to obtain the grant references for the shared pages.

After the ring buffers and the atomic variables are initialized, the client driver allocates the event channel ports and registers them via a hypercall. Finally, a VIRQ is sent to a doorbell port (we call it *welcome port*) of the server driver.

The virtual block device node is an interface through which applications interact with the client driver. We implement driver code that creates a virtual block device node in the `/dev` directory. Also, the driver code defines function wrappers such as open, read, write, and ioctl. File operations called by the application are routed to the functions in the driver code.

Block I/O data is transferred via our ring buffers. When the application requests a block I/O, a pointer of the I/O buffer is passed to the client driver. The client driver extracts the data, size of data, and block number. Then it copies the data, size, block number, and pointer of the block I/O buffer to the ring buffer. It has to track the pointer of the block I/O buffer in order to close the I/O request with `biodone`. Finally, it sends VIRQ to the server driver if it is asleep.

Consumer threads that dequeue entries from the rings go to sleep when the rings are empty. They first set the atomic variable to a negative value. As the atomic variables are shared, the server driver knows whether the threads are asleep. Later, they are awakened by the VIRQs from the server driver. If the rings are not empty and the threads are working on the rings, the atomic value is set to 1 so that the server driver does not send the VIRQs.

The client driver calls `biodone` which is a NetBSD kernel API to close the block I/O request. The response from the server driver contains a pointer of the block I/O buffer, the size of data written/read, and the error code. The client driver calculates the size of the remaining data to be processed and stores it in the block I/O

Processor	2 x Intel Xeon Silver 4114, 2.20 GHz
Number of cores	10 per processor, per NUMA node
HyperThreading	OFF (2 per core)
TurboBoost	OFF
L1/L2 cache	64 KB / 1024 KB per core
L3 cache	14080 KB
Main Memory	96 GB
Network	Intel x520-2 10GbE (82599ES)
Storage	Samsung 970 EVO Plus 500 GB NVMe

Table 1: Experimental setup.

buffer. Eventually, the client calls `biodone` with the block I/O buffer containing all the metadata. At this point, the block I/O ends.

4.2 Server Driver

The server driver resides in the storage server domain. Its role is to receive I/O requests from the client, perform I/O, and send responses back. As the storage server domain contains the device driver, the server driver can directly send the requests to the device drivers. Similar to the client driver, the server driver is built as a library in the storage server domain.

The server driver is initialized before an application is launched. It first allocates the welcome port and registers its domain ID and the welcome port number through a hypercall. After the client driver finishes its initialization, it sends a VIRQ to the welcome port and invokes the server driver to begin building the IVMC. The grant references for the shared pages and event channel ports are stored in Xen's memory. The server driver maps the shared pages and establishes the event channels. Also, a receiver thread is created when the server driver connects with the client. The thread sleeps when the rings are empty but wakes up upon receiving a VIRQ from the client driver.

The server driver creates an empty block I/O buffer using the NetBSD kernel API. It then populates the metadata, including the size of the data, block number, device number of the block device, a function pointer to the callback function, and a dedicated variable for private use. We use the variable to store an address of the block I/O buffer in the client to track the I/O requested by the application. Also, flags are marked as busy to indicate that the buffer is in use, along with the type of operation (e.g., write or read). Finally, the server driver can issue block I/O requests to the device driver with the block I/O buffer filled with the metadata and data.

4.3 Inter-VM Communication

Ring buffers use shared pages. To share pages between domains, the memory owner (client driver) grants access to the other end, which then maps the pages. The client driver initializes the grant table in its early stage. Then, it sends the grant references to the server driver. The server driver maps the pages by calling the corresponding hypercall. The client driver preallocates a big chunk of shared memory during its initialization. Unlike existing `blkfront` drivers, this design allows to avoid hypercalls for every single I/O request.

The size of the data block and the total number of entries in the rings significantly affect the performance of the ring buffers. We are able to set up 8192 entries (we set 1024 entries in the current implementation) for each ring, and any number beyond that is not allowed due to limited memory. Also, we set the data block size to 64 KB which is equivalent to the maximum block size that the NetBSD kernel can use.

We use the Xen event channel to implement virtual interrupts between the drivers. Hypercalls allocate an event channel and port number. Once Xen allocates the port number, the other end must know the number in order to bind. For this purpose, hypercalls are called with the corresponding port number.

Sending a virtual interrupt inevitably triggers a hypercall. That is, excessive virtual interrupts in the IVMC cause significant performance degradation. To avoid unnecessary interrupts, the drivers maintain atomic variables to synchronize each status. The atomic variables are attached to the ring buffers. With the atomic variables, the sender can know the receiver's status such that it does not send a virtual interrupt if the receiver is consuming the ring buffer.

5 Evaluation

We evaluate our storage server and file system server using a suite of micro- and macrobenchmarks, and compare them against rumprun and Linux (Ubuntu 22.04 with Linux 5.15). We chose these two as relevant baselines because the storage server is built on the rumprun unikernel, and Linux is a popular server OS. For a fair comparison, we considered multiserver OSs. However, to the best of our knowledge, none of the multiserver OSs, such as MINIX 3 [23], support NVMe devices. In our evaluation, most benchmarks focus on a single client-VM at a time. This approach is chosen intentionally to isolate the performance of a given application and precisely measure the overheads introduced by our IVMC mechanism under various loads and I/O sizes.

Table 1 presents our experimental setup. The host machine is running Xen 4.14.0 and Ubuntu 20.04 with Linux 5.4.0 as Xen's Dom0, which is used for system initialization and management. The rumprun unikernels used in our experiment are the versions maintained in the repository [63] and come with NetBSD 9.0 code. We set the I/O block size to 64 KB for the best performance. In addition to our storage server, we evaluate ordinary rumprun instances that enable direct access to storage devices. This baseline can be useful when comparing multiserver vs. library OS modes (for selected applications with dedicated NVMe partitions). We used Samsung 970 Evo Plus 500 GB NVMe device and Intel x520-2 10GbE NIC for storage and network devices, respectively. They are exclusively granted to virtual machines through the PCI passthrough feature.

5.1 Storage Server

5.1.1 Microbenchmark. We implemented our own test that repeatedly performs sequential reads and writes, rather than using `fio` [8], because `fio` has not yet been ported to rumprun, and its build system relies on Linux-specific system interfaces that are unavailable in our unikernel environment. Our test opens the virtual block device (`/dev/myblk`) as a single file. We varied the data size from 64 KB to 4 MB and measured the elapsed time to read/write a total of 56 GB of data to the NVMe storage. We assumed that 56 GB would

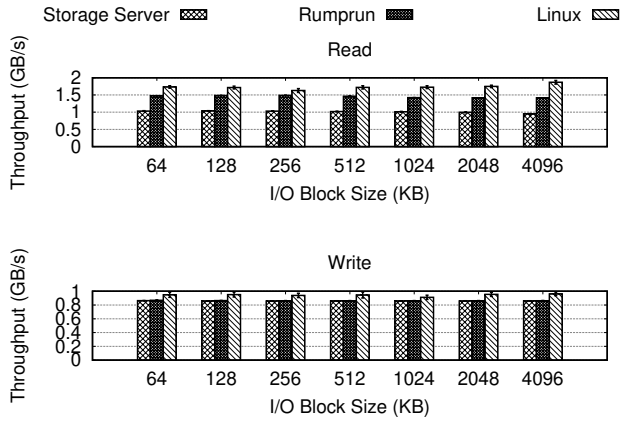


Figure 6: Storage server microbenchmark.

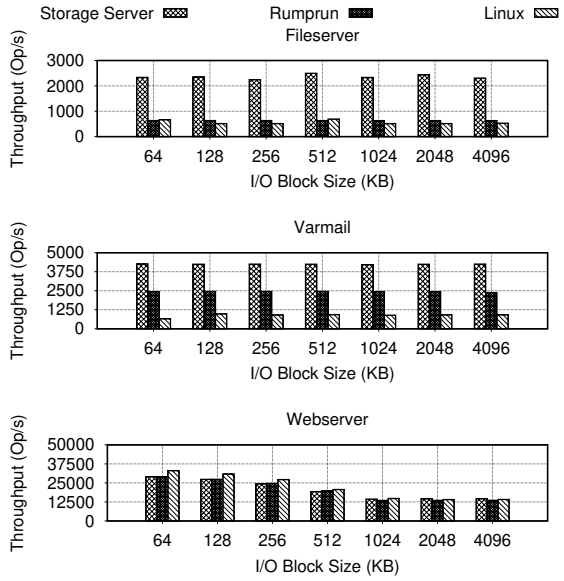


Figure 7: Storage server macrobenchmarks.

be enough to avoid the buffer cache, since each VM has 4 GB of memory. We ran 10 iterations of the benchmark and calculated the average throughput for the read and write operations. The results are shown in Figure 6.

For reads, both Linux and rumprun outperform the storage server. This is because the storage server introduces the IVMC data-copying overhead, which neither Linux nor rumprun have.[4] Rumprun is slower than Linux due to differences in I/O stacks (rumprun uses NetBSD code). On the other hand, the write throughputs are saturated at the same level. We attribute it to the overhead imposed by the NVMe controller's internal mechanism (e.g., wear leveling, garbage collecting).

[4]Unlike the file system server, the storage server has copying overheads. This is primarily a deliberate design choice for security reasons, though future work could explore further optimizations such as zero-copying where feasible without compromising inter-VM security guarantees.

5.1.2 Macrobenchmarks. To evaluate real applications, we used Filebench [68]. Since extensive system dependencies prevented building Filebench as a rumprun unikernel, we used an alternative approach: running an NFS server (v3) inside a virtual machine to act as the server for all three configurations (our storage server, rumprun, and Linux). On the server machine, an NVMe partition was formatted with the ext3 file system[5] and exported via NFS. On a separate client machine, the NFS share was mounted to run the Filebench workload. The client and server were connected together using 10GbE NICs (Intel x520-2), with the network configured for optimal performance (MTU of 9000 [37]). This setup allowed us to execute the *fileserver*, *varmail*, and *webserver* workloads, each involving 20,000 files of 1 MB (20 GB in total).

To isolate the impact of our IVMC-based storage server design from the underlying file system and device stack, we evaluated three server-side configurations: (A) a Linux-based NFS server where I/O requests traverse the Linux kernel block I/O stack and directly reach the NVMe device via PCI passthrough, (B) a rumprun-based NFS server where I/O follows the NetBSD block I/O path inside rumprun and also directly reaches the NVMe device via PCI passthrough, and (C) our design, where the NFS server runs inside a regular application VM that does not own the NVMe device. In configuration C, the NFS server issues requests through its client driver, which forwards them over our IVMC channel to the storage server VM. The storage server then processes the requests using the same NetBSD block I/O stack as rumprun and finally accesses the NVMe device via PCI passthrough. In all setups, the NFS server asynchronously issues requests to ext3, which then dispatches asynchronous operations to the block device. Therefore, the only difference between the three configurations lies in how the block requests are transported to the NVMe device – either directly via passthrough (Linux or rumprun) or through IVMC and our storage server. We explicitly include rumprun in the comparison because both rumprun and our storage server rely on the NetBSD block I/O implementation, allowing a fair evaluation of whether our IVMC-based design can match or exceed the performance of direct passthrough while enabling isolation and transparent recovery.

Fileserver. Fileserver has 50 threads that perform a sequence of creates, deletes, appends, write-whole-file, and read-whole-file. We chose fileserver because it intensively writes compared to other workloads. From the results in Figure 7, our storage server outperforms rumprun and Linux for all I/O sizes. With the storage server, the application can asynchronously batch write requests by inserting a bunch of requests into the ring buffers without waiting for the responses. The storage server provides some sort of parallelism, thus avoiding internal scheduling overhead. The storage server architecture parallelizes the I/O process: the application continues sending requests while the storage server simultaneously performs I/O operations on the NVMe device.

Varmail. Varmail emulates a multi-threaded mail server that represents the workload of the `/var/mail` directory on traditional UNIX systems [68]. Our storage server and rumprun outperform Linux due to Varmail's frequent system calls, which are known to cause non-negligible performance overheads [62]. The rumprun unikernel avoids the system call layer altogether. In addition, the

[5]We selected ext3 because NetBSD/rumprun does not support ext4.

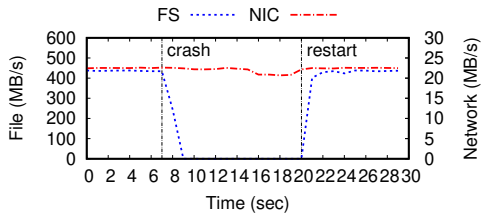


Figure 8: Failure recovery evaluation.

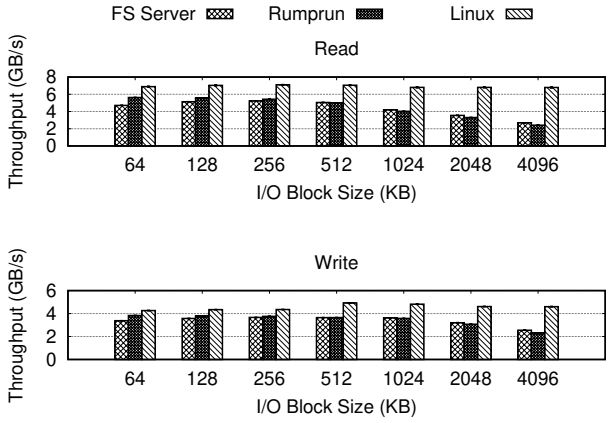


Figure 9: File system server microbenchmark.

storage server outperforms rumprun by leveraging the storage I/O parallelism.

Webserver. Webserver is a multi-threaded read-intensive workload. As a result, this workload is highly affected by the latency of the read operation. We observe similar results overall. Among the three, Linux has a smaller latency (Figure 7), which is visible for smaller data sizes. As the data sizes increase, the difference completely disappears due to the saturation of the NFS network throughput.

5.2 Storage Server Failure Recovery

We evaluated the failure recovery capability of our storage server. Our model assumes that faults occurring in the storage server such as memory access violations, deadlocks, and interrupt handling procedure failures, can be recovered by restarting the server. Since the storage server is strongly isolated by hardware virtualization, failures are contained within the server and do not propagate elsewhere. An application can fail, but we rely on the fault-recovery features of the file system (e.g., journaling) that reside on the client side. We designed an experiment to demonstrate the failure recovery of our storage server. We assumed a case where there are two applications: we implemented a simple program that performs a storage I/O task; the other application is the Nginx HTTP server. The program performing storage I/O uses the storage server, while the Nginx HTTP server uses NIC hardware for networking. To avoid the effects of the page cache, the storage I/O program reads and writes distinct storage blocks.

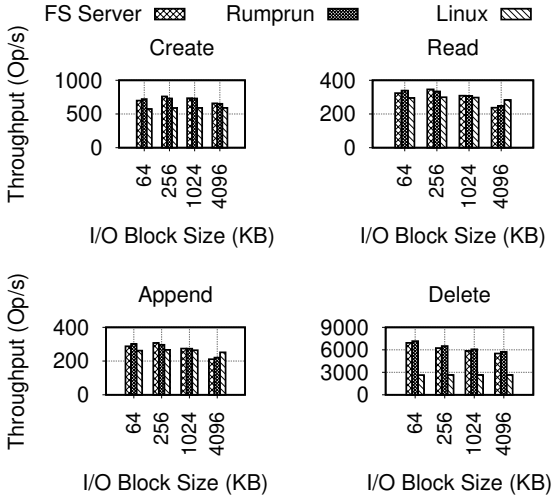


Figure 10: PostMark benchmark.

When the storage server fails, the program's file system data transfer is stopped, and we report an outage that is observed by the client. Figure 8 shows the corresponding file system and network transfer speed. At the 7-second mark, the storage server fails, causing the file system transfer speed to drop to zero. After the storage server restarts at the 20-second mark, the transfer speed is restored. During the failure, the network transfer speed of Nginx is not affected.

5.3 File System Server

5.3.1 Microbenchmark. We evaluated the file system server, rumprun, and Linux using a microbenchmark that reads and writes various data sizes on a ramdisk. The microbenchmark read and wrote a total of 14 GB of data, measuring the elapsed time to calculate average throughput. We measured 10 iterations. As Figure 9 shows, Linux outperforms both the file system server and rumprun, thanks to its optimized I/O stack compared to NetBSD. The file system server is slightly slower than rumprun due to IVMC overheads such as interrupt handling and scheduling overhead. However, the file system server outperforms rumprun with larger data sizes, as IVMC overheads become negligible and the file system server enhances parallelism by detaching the file I/O stack from the application.

5.3.2 Macrobenchmarks. We measured performance of real-life applications. For Nginx, we used a client machine (see Section 5.1.2) connected via 10GbE.

PostMark. We ran PostMark [30] to demonstrate the file system server performance. It creates, reads, appends, and deletes a pool of random text files with varying sizes [1]. We configured it to perform 2,000 transactions on 100 files in `tmpfs`, with block sizes varying from 64 KB to 4 MB. The file system server and rumprun outperform Linux since PostMark triggers numerous system calls. In particular, the performance of `delete` demonstrates that the system call latency of the file system server and rumprun is much smaller than that of Linux because `sys_unlink` only deletes a name from the file system [2], thereby representing a system call latency.

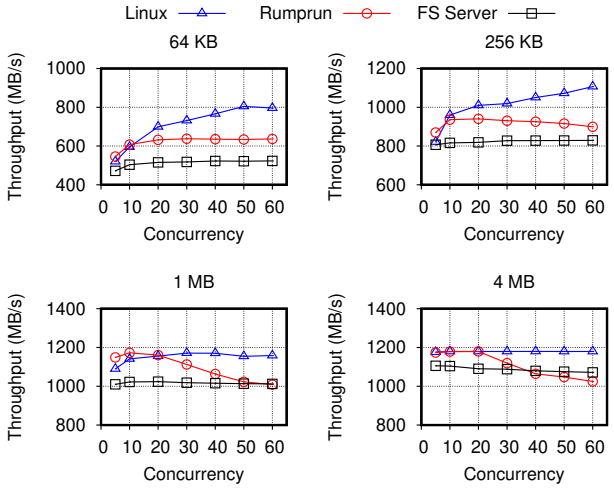


Figure 11: Nginx HTTP Server.

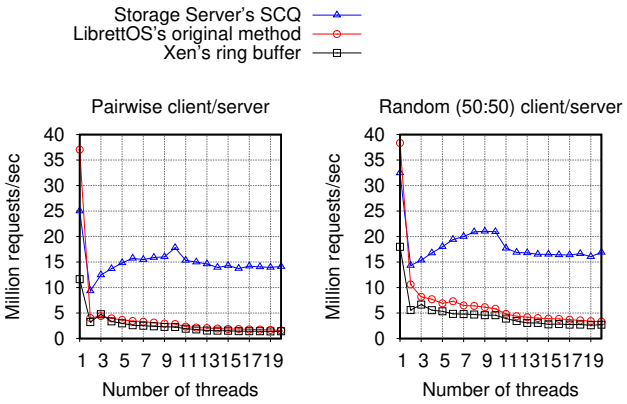


Figure 12: Null-call IVMCs: Our SCQ-based method, LibrettOS's original method, and IVMC with traditional (e.g., Xen) ring buffers. In the pairwise experiment, enqueues and dequeues are in a tight loop. In the random experiment, enqueues (50%) and dequeues (50%) are in random order.

However, the throughput of the file system server and rumprun does not always scale with increasing block size. This is partially attributed to rumprun's internal locking mechanism within its NetBSD glue layer, which can limit the degree of parallelism for I/O-intensive requests.

Nginx HTTP Server. We chose a popular Nginx server (v1.8.0) [50] because its workload combines file I/O and networking. We configured Apache Benchmark [69] to send 10,000 requests and ran it with various number of concurrent requests on the client machine. Figure 11 shows throughputs of different file sizes that the client downloads from the web server. Linux outperforms the file system server and rumprun in most experiments. This is due to the highly optimized I/O stack (e.g., zero-copy implementation) and NIC device driver in Linux. With the file size of 64 KB and 256 KB, the IVMC overhead contributes to the throughput gap, shown in Figure 11. With large file sizes (1 MB and 4 MB) and high level of concurrency, the gap narrows down as the IVMC overhead becomes negligible when processing large data.

5.4 IVMC Evaluation

In Figure 12, we evaluate our IVMC that uses state-of-the-art SCQ algorithm [51], LibrettOS's method with a simpler lock-free ring buffer [54], and IVMC through traditional I/O ring buffers such as those found in the Xen hypervisor. Each experiment has 10,000,000 operations repeated 10 times, and we calculate throughput using the mean elapsed time. SCQ's advantage is that it uses specialized hardware instructions, such as fetch-and-add, which scale better. Although more traditional I/O ring buffers such as those found in Xen have independent enqueuers and dequeuers, those still need to be synchronized across multiple threads on each side. Though one-thread case is optimized, there are still overheads which make it slower than SCQ even for one thread.

We present IVMC results for null-calls to measure the maximum number of requests per second. Our IVMC scales very well as the number of threads increases. With more than 10 threads, over-socket contention due to NUMA (non-uniform memory access) slows down the throughput but it still remains much higher than in other approaches. The gap increases from **2x** to **6x** as the number of threads grows.

6 Related Work

To improve OS reliability, a fundamental design principle studied in the OS literature is isolation, and numerous researchers have proposed methods to provide isolation in systems. Microkernels such as [3, 16, 21, 23, 27, 28, 40] provide isolation of system software components in separate address spaces. In particular, microkernels [20, 24, 32] enhance security and reliability by placing only essential system components within the kernel. L4 [40] is a family of microkernels, and members of this family are used for various purposes. In particular, seL4 [32] is known as a formally verified microkernel. Multiserver OSs are a type of microkernel, where OS components run in separate user processes called system servers (e.g., network server, storage server, device driver server). Examples of multiserver OSs include MINIX 3 [23], GNU Hurd [11], Mach-US [64], and SawMill [19].

Failure Recovery of Stateful Servers. Iguana [38] is a suite of OS services running on top of the L4 microkernel. While it lacks advanced failure recovery support, stateless services can still be restarted. However, failures in stateful services are more difficult to recover from. L4Linux [22] implements Linux services as user-mode servers, but its reliability is no better than that of the standard monolithic Linux. Chorus OS [58] executes system services in the privileged mode and they share the same address space with the microkernel. As a result, an error occurring in a server can potentially corrupt server states stored in persistent memory. EROS [60] is a capability-based OS that periodically takes snapshots of the entire system state and saves them to disk. These snapshots are used for recovery in case of failure. However, if the error that caused the failure is included in the snapshot, the system will crash again when it is restored. Additionally, creating snapshots of large systems introduces significant memory and performance overhead.

CuriOS [15] stores client states on the client side, rather than on servers. To prevent security problems where the client might modify its own state, the states are placed in memory that is mapped into the server's address space, making it inaccessible to the client. This isolates client states such that errors cannot propagate across all clients. The authors also point out some intuitions for the transparent failure recovery in microkernels. Our storage server addresses all the intuitions that the paper covers: (1) the storage server does not affect the application at all as it is strongly isolated by virtualization; (2) the client driver does not send requests during the server restart; (3) client states are located on the client side. Furthermore, our storage server has a performance benefit by eliminating expensive system call cost, whereas CuriOS introduces high context switch overhead.

Herder et al. proposed fault-resilience mechanisms for MINIX 3 [25]. They locate a filter driver between the file system server and the storage driver. The filter driver uses checksumming and mirroring to detect storage failure and guarantee data integrity. However, this design is not scalable and introduces high performance overhead.

Researchers tried to employ certain aspects of the multiserver design using existing monolithic OSs. For example, VirtuOS [53] employs a fault-tolerant multiserver design by leveraging virtualization to isolate the Linux kernel components. VirtuOS runs storage and networking servers as service domains on top of Xen. However, VirtuOS cannot recover storage states during failures. Snap [46] presents a network server in Linux to improve performance and simplify system upgrades. Snap uses its own protocol and is therefore incompatible with existing applications. Moreover, its recovery mechanism is unsuitable for (stateful) storage.

Swift et al. introduced Nooks [67] to isolate device drivers from the rest of the kernel with separated address spaces. Nooks' successor [66] extended its failure detection capabilities and implemented shadow drivers for transparent failure recovery. The shadow driver implements only the services needed for recovery instead of replicating the real driver. When a device driver fails, Nooks detects the failure, and the shadow driver impersonates the failed driver by handling requests until a new driver is restarted. Nooks restarts the failed driver, restores states which are collected from the configuration logs, and resubmits pending requests. However, shadow drivers cannot ensure exactly-once behavior for driver requests and depend upon higher-level protocols to maintain data integrity.

Fault-Tolerant File Systems. Chidambaram et al. introduced a crash-consistent file system, OptFS [12]. They decouple ordering and durability in the file system to reduce unnecessary flushing, assuming that a crash does not occur, thereby improving performance. However, since blocks may be re-ordered without flushes, they introduce additional techniques, such as checksums and *Asynchronous Durability Notifications*, to maintain consistency. With these techniques, OptFS provides consistency while trading freshness for better performance. However, OptFS relies on journaling file systems, such as Ext3, while our storage server ensures consistency regardless of the file system implementation.

IceFS [41] physically disentangles a file system using an abstraction called a cube. With the disentanglement and isolation, it introduces failure isolation, localized recovery, and specialized journaling to achieve improved reliability and performance. Membrane [65], on the other hand, provides an OS with restartable file systems support.

Hybrid Microkernel-Exokernel OS. LibrettOS [54] introduced a fusion of multiserver OS and library OS. A default mode of LibrettOS is a multiserver OS that runs system services in an isolated manner. LibrettOS introduced a network server as an example of system servers and demonstrated its failure recovery. For selected applications requiring high performance, LibrettOS acts as a library OS and grants the application exclusive access to hardware resources. Furthermore, LibrettOS has the ability to dynamically switch between two modes: applications can switch between multiserver OS and library OS mode during runtime with no interruption. This hybrid design allows users to exploit respective strengths of each model as well as simultaneously addresses issues of isolation, performance, and recoverability. However, LibrettOS lacks a storage server, which is one of the essential components in OSs. In addition, LibrettOS's network server is a stateless component and its failure recovery relies on the TCP/IP reconnect process.

System Services for Unikernels. Kite [47] operates Xen driver domains without the need for a full-fledged OS, making use of the rumprun design. This approach yields several advantages, including a reduction in system calls, smaller image sizes, and faster boot times when compared to Linux-based driver domains. Notably, Kite achieves these benefits without relying on heavyweight Linux tools such as xen-tools. However, Kite's biggest downside is coarse-grained granularity of sharing, a problem that we address in this paper by creating a file system server. Moreover, unlike Kite, our design supports full failure recovery.

Shared-Memory Communication. The use of multiple-producer multiple-consumer shared-memory mechanisms, similar to the presented IVMC mechanisms in this paper, has been also advocated in [17, 52, 54]. This paper specifically focuses on implementing non-trivial *recovery paths* while using advanced non-blocking data structures in the context of file systems and storage.

7 Conclusion

We presented two servers for a private cloud: a storage server (for low-level block I/O) and a file system server (for high-level file I/O). Both servers are complementing each other, e.g., the file system server talks to the storage server, and the storage server talks to the actual hardware. Our design uses modified rumprun unikernels, where strong isolation is provided by hardware virtualization. Furthermore, our servers leverage NetBSD code, i.e., a wide range of file system drivers[6] and a wide range of legacy and modern storage drivers[7] are available.

Aside from strong isolation, our storage server transparently recovers from faults without incurring significant overheads, which is a well-known and complex task for stateful subsystems. We present *state virtualization* and *flying data recovery* for our transparent fault recovery. For state virtualization, we separate storage states into two classes: one that is invariant and the other one that can be restored afterwards. The invariant states are preserved on the

[6]We evaluated ext3, but more file systems are available.
[7]We evaluated NVMe, but many other storage devices are supported.

application side and readily available during fault recovery, while all other states can be recovered from the hypervisor's memory.

Our storage server's and file system server's IVMC builds on recent advances in concurrent data structures by using state-of-the-art ring buffers, which make our storage server IVMC more scalable than those that rely on traditional ring buffers found in typical hypervisors (e.g., Xen). Our special IVMC design also helps to restore storage states during fault recovery without sacrificing performance in the critical data path.

The evaluation results demonstrate that the file system server and the storage server exhibit scalable IVMC, comparable performance with that of Linux, and transparent failure recovery.

Availability

The code is available at https://github.com/ssrg-vt/rumprun-servers.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions, which helped greatly improve this paper.

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

This research is also based upon work supported by the Office of Naval Research (ONR) under grants N00014-16-1-2104, N00014-16-1-2711, N00014-18-1-2022, and N00014-19-1-2493.

References

[1] 2004. PostMark Results. https://www.usenix.org/legacy/publications/library/proceedings/fast04/tech/full_papers/radkov/radkov_html/node17.html.

[2] 2025. unlink(2) — Linux manual page. https://man7.org/linux/man-pages/man2/unlink.2.html.

[3] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A new kernel foundation for UNIX development. (1986).

[4] Amazon. 2017. Amazon EC2 Bare Metal Instances with Direct Access to Hardware. https://aws.amazon.com/blogs/aws/new-amazon-ec2-bare-metal-instances-with-direct-access-to-hardware/.

[5] Amazon. 2025. Amazon EC2 Dedicated Hosts. https://aws.amazon.com/ec2/dedicated-hosts/.

[6] Amazon. 2025. What are the benefits of a private cloud? https://aws.amazon.com/what-is/private-cloud/.

[7] Antti Kantee. 2020. Rumprun. https://github.com/rumpkernel/rumprun.

[8] Jens Axboe. 2017. Welcome to FIO's documentation! https://fio.readthedocs.io/en/latest/index.html.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. 164–177. doi:10.1145/945445.945462

[10] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, 250–257. doi:10.1109/CloudCom.2015.89

[11] T. Bushnell. 1996. Towards a new strategy for OS design. http://www.gnu.org/software/hurd/hurd-paper.html.

[12] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 228–243. doi:10.1145/2517349.2522726

[13] Glauber Costa and Don Marti. 2014. Redis On OSv. http://blog.osv.io/blog/2014/08/14/redis-memonly/.

[14] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. FADES: Fine-Grained Edge Offloading with Unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet '17)*. ACM, 36–41. doi:10.1145/3094405.3094412

[15] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. 2008. CuriOS: Improving Reliability Through Operating System Structure. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI '08)*. 59–72.

[16] Martin Decky. 2017. HelenOS: Operating System Built of Microservices. http://www.nic.cz/files/nic/IT_17/Prezentace/Martin_Decky.pdf.

[17] Yanlin Du and Ruslan Nikolaev. 2025. Joyride: Rethinking Linux's network stack design for better performance, security, and reliability. In *Proceedings of the 3rd Workshop on Kernel Isolation, Safety and Verification* (Seoul, Republic of Korea) *(KISV '25)*. Association for Computing Machinery, New York, NY, USA, 25–31. doi:10.1145/3765889.3767045

[18] Bob Duncan, Andreas Happe, and Alfred Bratterud. 2016. Enterprise IoT security and scalability: how unikernels can improve the status Quo. In *IEEE/ACM 9th International Conference on Utility and Cloud Computing (UUC 2016)*. IEEE, 292–297.

[19] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill multiserver approach. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System* (Kolding, Denmark) *(EW 9)*. Association for Computing Machinery, New York, NY, USA, 109–114. doi:10.1145/566726.566751

[20] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security '12)*.

[21] Per Brinch Hansen. 1970. The nucleus of a multiprogramming system. *Commun. ACM* 13, 4 (1970), 238–241.

[22] Hermann Härtig, Michael Hohmuth, and Jean Wolter. 1998. Taming linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*. Citeseer, 49–56.

[23] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. Reorganizing UNIX for Reliability. In *Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture* (Shanghai, China) *(ACSAC '06)*. 81–94. doi:10.1007/11859802_8

[24] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. 2009. Fault isolation for device drivers. In *Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP Int. Conference*. 33–42.

[25] Jorrit N. Herder, David C. van Moolenbroek, Raja Appuswamy, Bingzheng Wu, Ben Gras, and Andrew S. Tanenbaum. 2009. Dealing with Driver Failures in the Storage Stack. In *2009 4th Latin-American Symposium on Dependable Computing*. 119–126. doi:10.1109/LADC.2009.12

[26] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*.

[27] Dan Hildebrand. 1992. An Architectural Overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*. 113–126.

[28] Galen C. Hunt, James R. Larus, David Tarditi, and Ted Wobber. 2005. Broad New OS Research: Challenges and Opportunities. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10* (Santa Fe, NM) *(HotOS '05)*. 15–15.

[29] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* (2014).

[30] Jeffrey Katcher. 1997. *Postmark: A new file system benchmark*. Technical Report. Technical Report TR3022, Network Appliance.

[31] Avi Kivity. 2007. KVM: the Linux virtual machine monitor. In *2007 Ottawa Linux Symposium*. 225–230.

[32] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. 207–220. doi:10.1145/1629575.1629596

[33] Michał Król and Ioannis Psaras. 2017. NFaaS: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking* (Berlin, Germany) *(ICN '17)*. Association for Computing Machinery, New York, NY, USA, 134–144. doi:10.1145/3125719.3125727

[34] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE*

'17). ACM, 15–29. doi:10.1145/3050748.3050757

[35] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the 15th European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 15 pages. doi:10.1145/3342195.3387526

[36] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016)*. ACM. doi:10.1145/2931088.2931093

[37] Breno Henrique Leitao. 2009. Tuning 10Gb network cards on Linux. In *2009 Ottawa Linux Symsposium*. 169–184.

[38] Ben Leslie, Carl Van Schaik, and Gernot Heiser. 2005. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux. Conf. Au, Canberra*, Vol. 20.

[39] liblfds. 2016. Ringbuffer disappointment, 2016-04-29. https://www.liblfds.org/slblog/2016-04.html.

[40] Jochen Liedtke. 1996. Toward Real Microkernels. *Commun. ACM* 39, 9 (1996), 70–77.

[41] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI '14)*. USENIX Association, USA, 81–96.

[42] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. 559–573.

[43] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 461–472. doi:10.1145/2451116.2451167

[44] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 218–233. doi:10.1145/3132747.3132763

[45] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI '14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[46] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. 399–413. doi:10.1145/3341301.3359657

[47] A K M Fazla Mehrab, Ruslan Nikolaev, and Binoy Ravindran. 2022. Kite: Lightweight Critical Service Domains. In *Proceedings of the 17th European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 384–401. doi:10.1145/3492321.3519586

[48] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 103–112. doi:10.1145/2442516.2442527

[49] Next Century. 2019. SAVIOR (Secure Applications in Virtual Instantiations of Roles). https://github.com/NextCenturyCorporation/VirtUE.

[50] NGINX Contributors. 2025. Nginx: High Performance Load Balancer, Web Server, Reverse Proxy. http://nginx.org/.

[51] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *33rd International Symposium on Distributed Computing (DISC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 146)*, Jukka Suomela (Ed.). European Association for Theoretical Computer Science (EATCS), Dagstuhl, Germany, 28:1–28:16. doi:10.4230/LIPIcs.DISC.2019.28

[52] Ruslan Nikolaev. 2025. Parsec: Fast, Scalable, and Secure Design with Wait-Free Parallelism. In *Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea) *(APSys '25)*. Association for Computing Machinery, New York, NY, USA, 202–208. doi:10.1145/3725783.3764410

[53] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: an operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 116–132. doi:10.1145/2517349.2522719

[54] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. 2020. LibrettOS: A Dynamically Adaptable Multiserver-Library OS. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 114–128. doi:10.1145/3381052.3381316

[55] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*. doi:10.1145/3313808.3313817

[56] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. doi:10.1145/1950365.1950399

[57] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. 2023. Unikernel Linux (UKL). In *Proceedings of the 18th European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 590–605. doi:10.1145/3552326.3587458

[58] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. 1992. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*. 39–70.

[59] Joanna Rutkowska and Rafal Wojtczuk. 2010. Qubes OS architecture. *Invisible Things Lab Tech Rep* 54 (2010).

[60] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 170–185. doi:10.1145/319151.319163

[61] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 Workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox 2016)*. ACM, 44–49. doi:10.1145/2940147.2940149

[62] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI '10)*. USENIX Association, Berkeley, CA, USA, 33–46. http://dl.acm.org/citation.cfm?id=1924943.1924946

[63] SSRG-VT. 2020. Rumprun-smp. https://github.com/ssrg-vt/rumprun-smp/tree/c166f324f57456dedc7b79711f16ed09fe0e71b4.

[64] J. Mark Stevenson and Daniel P. Julin. 1995. Mach-US: UNIX on generic OS object servers. In *Proceedings of the USENIX 1995 Technical Conference (TCON'95)*. 119–130.

[65] Swaminathan Sundararaman, Sriram Subramanian, and Michael M. Swift. 2010. Membrane: Operating System Support for Restartable File Systems. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast-10/membrane-operating-system-support-restartable-file-systems

[66] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. 2004. Recovering Device Drivers. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI '04)*. USENIX Association, USA, 1–16.

[67] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. 207–222. doi:10.1145/945445.945466

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

[69] The Apache Software Foundation. 2025. ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[70] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 9th European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. doi:10.1145/2592798.2592812

[71] Xen Project. 2014. Understanding the Virtualization Spectrum. https://wiki.xenproject.org/wiki/Understanding_the_Virtualization_Spectrum.

[72] Xen Website. 2018. Google Summer of Code Project, TinyVMI: Porting LibVMI to Mini-OS. https://blog.xenproject.org/2018/09/05/tinyvmi-porting-libvmi-to-mini-os-on-xen-project-hypervisor/.

[73] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*.