LibrettOS: A Dynamically Adaptable Multiserver-Library OS[*]

Ruslan Nikolaev, Mincheol Sung, Binoy Ravindran

Bradley Department of Electrical and Computer Engineering, Virginia Tech
$\{rnikola, mincheol, binoy\}$@vt.edu

Abstract

We present LibrettOS, an OS design that fuses two paradigms to simultaneously address issues of isolation, performance, compatibility, failure recoverability, and run-time upgrades. LibrettOS acts as a microkernel OS that runs servers in an isolated manner. LibrettOS can also act as a library OS when, for better performance, selected applications are granted exclusive access to virtual hardware resources such as storage and networking. Furthermore, applications can switch between the two OS modes with no interruption at run-time. LibrettOS has a uniquely distinguishing advantage in that, the two paradigms seamlessly coexist in the same OS, enabling users to simultaneously exploit their respective strengths (i.e., greater isolation, high performance). Systems code, such as device drivers, network stacks, and file systems remain identical in the two modes, enabling dynamic mode switching and reducing development and maintenance costs.

To illustrate these design principles, we implemented a prototype of LibrettOS using rump kernels, allowing us to reuse existent, hardened NetBSD device drivers and a large ecosystem of POSIX/BSD-compatible applications. We use hardware (VM) virtualization to strongly isolate different rump kernel instances from each other. Because the original rumprun unikernel targeted a much simpler model for uniprocessor systems, we redesigned it to support multicore systems. Unlike kernel-bypass libraries such as DPDK, applications need not be modified to benefit from direct hardware access. LibrettOS also supports indirect access through a network server that we have developed. Instances of the TCP/IP stack always run directly inside the address space of applications. Unlike the original rumprun or monolithic OSes, applications remain uninterrupted even when network components fail or need to be upgraded. Finally, to efficiently use hardware resources, applications can dynamically switch between the indirect and direct modes based on their I/O load at run-time. We evaluate LibrettOS with 10GbE and NVMe using Nginx, NFS, memcached, Redis, and other applications. LibrettOS's performance typically exceeds that of NetBSD, especially when using direct access.

***Keywords*:** operating system, microkernel, multiserver, network server, virtualization, Xen, IOMMU, SR-IOV, isolation

1 Introduction

Core components and drivers of a general purpose monolithic operating system (OS) such as Linux or NetBSD typically run in privileged mode. However, this design is often inadequate for modern systems [13, 30, 43, 50, 62]. On the one hand, a diverse and ever growing kernel ecosystem requires better isolation of individual drivers and other system components to localize security threats due to the increasingly large attack surface of OS kernel code. Better isolation also helps with tolerating component failures and thereby increases reliability. Microkernels achieve this goal, specifically in multiserver OS designs [13, 30, 33].[1] On the other hand, to achieve better device throughput and resource utilization, some applications need to bypass the system call and other layers so that they can obtain exclusive access to device resources such as network adapter's (NIC) Tx/Rx queues. This is particularly useful in recent hardware with SR-IOV support [63], which can create virtual PCIe functions: NICs or NVMe storage partitions. Library OSes and kernel-bypass libraries [77, 84] achieve this goal. Multiserver-inspired designs, too, can outperform traditional OSes on recent hardware [54, 56].

Microkernels, though initially slow in adoption, have gained more traction in recent years. Google's Fuchsia OS [24] uses the Zircon microkernel. Intel Management Engine [37] uses MINIX 3 [30] since 2015. A multiserver-like networking design was recently revisited by Google [56] to improve performance and upgradability when using their private

[1]As microkernels are defined broadly in the literature, we clarify that we consider multiserver OSes as those implementing *servers* to isolate core OS components, e.g., MINIX 3 [30].

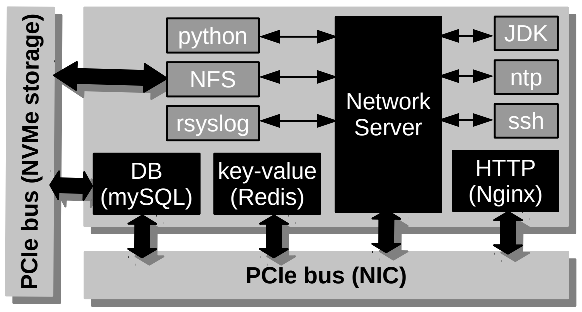


Figure 1. Server ecosystem example.

(non-TCP) messaging protocol. However, general-purpose application and device driver support for microkernels is limited, especially for high-end hardware such as 10GbE+.

Kernel-bypass techniques, e.g., DPDK [84] and SPDK [77], are also increasingly gaining traction, as they eliminate OS kernels from the critical data path, thereby improving performance. Unfortunately, these techniques lack standardized high-level APIs and require massive engineering effort to use, especially to adapt to existing applications [89]. Additionally, driver support in kernel-bypass libraries such as DPDK [84] is great only for high-end NICs from certain vendors. Re-implementing drivers and high-level OS stacks from scratch in user space involves significant development effort.

Oftentimes, it is overlooked that "no one size fits all." In other words, no single OS model is ideal for all use cases. Depending upon the application, security or reliability requirements, it is desirable to employ multiple OS paradigms in the *same* OS. In addition, applications may need to switch between different OS modes based on their I/O loads. In Figure 1, we illustrate an ecosystem of a web-driven server running on the same physical or virtual host. The server uses tools for logging (rsyslog), clock synchronization (ntp), NFS shares, and SSH for remote access. The server also runs python and Java applications. None of these applications are performance-critical, but due to the complexity of the network and other stacks, it is important to recover from temporary failures or bugs without rebooting, which is impossible in monolithic OSes. One way to meet this goal is to have a network server as in the multiserver paradigm, which runs system components in separate user processes for better isolation and failure recoverability. This approach is also convenient when network components need to be upgraded and restarted at run-time [56] by triggering an artificial fault.

Core applications such as an HTTP server, database, and key-value store are more I/O performance-critical. When having a NIC and NVMe storage with SR-IOV support (or multiple devices), selected applications can access hardware resources directly as in library OSes or kernel-bypass libraries – i.e., by bypassing the system call layer or inter-process communication (IPC). Due to finite resources, PCIe devices restrict the number of SR-IOV interfaces – e.g., the Intel 82599 adapter [38] supports up to 16 virtual NICs, 4 Tx/Rx queues each; for other adapters, this number can be even smaller. Thus, it is important to manage available hardware I/O resources efficiently. Since I/O load is usually non-constant and changes for each application based on external circumstances (e.g., the number of clients connected to an HTTP server during peak and normal hours), conservative resource management is often desired: use network server(s) until I/O load increases substantially, at which point, migrate to the library OS mode (for direct access) at run-time with no interruption. This is especially useful for recent bare metal cloud systems – e.g., when one Amazon EC2 bare metal instance is shared by several users.

In this paper, we present a new OS design – LibrettOS – that not only reconciles the library and multiserver OS paradigms while retaining their individual benefits (of better isolation, failure recoverability, and performance), but also overcomes their downsides (of driver and application incompatibility). Moreover, LibrettOS enables applications to switch between these two paradigms at run-time. While high performance can be obtained with specialized APIs, which can also be adopted in LibrettOS, they incur high engineering effort. In contrast, with LibrettOS, existing applications can already benefit from more direct access to hardware while still using POSIX.

We present a realization of the LibrettOS design through a prototype implementation. Our prototype leverages rump kernels [42] and reuses a fairly large NetBSD driver collection. Since the user space ecosystem is also inherited from NetBSD, the prototype retains excellent compatibility with existing POSIX and BSD applications as well. Moreover, in the two modes of operation (i.e., library OS mode and multiserver OS mode), we use an identical set of drivers and software. In our prototype, we focus only on networking and storage. However, the LibrettOS design principles are more general, as rump kernels can potentially support other subsystems – e.g., NetBSD's sound drivers [18] can be reused. The prototype builds on rumprun instances, which execute rump kernels atop a hypervisor. Since the original rumprun did not support multiple cores, we redesigned it to support symmetric multiprocessing (SMP) systems. We also added 10GbE and NVMe drivers, and made other improvements to rumprun. As we show in Section 5, the prototype outperforms the original rumprun and NetBSD, especially when employing direct hardware access. In some tests, the prototype also outperforms Linux, which is often better optimized for performance than NetBSD.

The paper's **research contribution** is the proposed OS design and its prototype. Specifically, LibrettOS is the first OS design that *simultaneously* supports the multiserver and library OS paradigms; it is also the first design that can *dynamically switch* between these two OS modes at run-time with no interruption to applications. Our network server, designed for the multiserver mode, accommodates both paradigms by using fast L2 frame forwarding from applications. Finally, LibrettOS's design requires only *one set of device drivers* for both modes and mostly retains POSIX/BSD compatibility.

Our work also confirms an existent hypothesis that kernel bypass is faster than a monolithic OS approach. However, unlike prior works [7, 56, 64], we show this without resorting to optimized (non-POSIX) APIs or specialized device drivers.

2 Background

For greater clarity and completeness, in this section, we discuss various OS designs, microkernels, and hypervisors. We also provide background information on rump kernels and their practical use for hypervisors and bare metal machines.

2.1 Multiserver and Library OSes

Traditionally, monolithic OSes run all critical components in a privileged CPU mode within a single kernel address space that is isolated from user applications running in an unprivileged mode. The isolation of systems software components such as device drivers in separate address spaces is the fundamental principle behind the microkernel model [1, 27, 30, 52], which provides stronger security and reliability [22, 31, 46] guarantees to applications compared to the classical monolithic kernel designs. To support existing applications and drivers, microkernel systems either implement emulation layers [26] or run a multiserver OS [13, 20, 29, 30, 33] on top of the microkernel. In the multiserver design, servers are typically created for specific parts of the system, such as device drivers and network and storage stacks. Applications communicate with the servers using IPC to access hardware resources.

The *system call* separation layer between user applications and critical components in monolithic OSes is somewhat arbitrary, chosen for historical reasons such as rough correspondence to the POSIX standard. However, more kernel components can be moved into the application itself, bringing it closer to the hardware. This concept was initially proposed by Tom Anderson [3], later implemented in the exokernel model [15], and named the "library OS." This model is often advocated for performance reasons and OS flexibility [6]. Subsequent works pointed out the security benefits of the library OS model [66, 85, 86] due to the strong isolation provided by the reduced interactions between the application and the privileged layer. Specialized library OSes such as unikernels [10, 43, 45, 50, 55, 90] do not isolate OS components from applications and are designed to run just a single application in virtualized environments typical of cloud infrastructures. As unikernels are application-oriented, they may provide a significant reduction in software stacks compared to full-fledged OSes.

2.2 NetBSD, rump kernels, and rumprun

Rump kernels are a concept introduced and popularized by Antti Kantee [42] and the larger NetBSD [59] community. NetBSD is a well known monolithic OS; along with FreeBSD and OpenBSD, it is one of the most popular BSD systems in use today. NetBSD provides a fairly large collection of device drivers, and its latest 8.0 version supports state-of-the-art 10GbE networking, NVMe storage, and XHCI/USB 3.0. Unlike in other OSes, NetBSD's code was largely redesigned to factor out its device drivers and core components into *anykernel* components. As shown in Figure 2, a special rump kernel glue layer enables execution of anykernels outside of the monolithic NetBSD kernel. For example, using POSIX threads (*pthreads*), rump kernels can run anykernels in user space on top of existing OSes. In fact, NetBSD already uses rump kernels to facilitate its own device driver development and testing process, which is done much easier in user space.

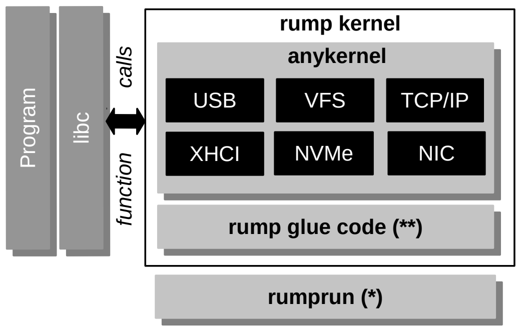


Figure 2. Rump software stack, (*) indicates components substantially and (**) partially modified in LibrettOS.

Rump kernels may serve as a foundation for new OSes, as in the case of the *rumprun* unikernel. As shown in Figure 2, systems built from rump kernels and rumprun are, effectively, library OSes. In contrast to NetBSD, rumprun-based systems substitute system calls with ordinary function calls from *libc*. The original rumprun, unfortunately, lacks SMP support.

Rump kernels were also used to build servers as in the rump file system server [41]. However, prior solutions simply rely on user-mode *pthreads*, targeting monolithic OSes. The design of such servers is very different from a more low-level, microkernel-based architecture proposed in this paper.

2.3 Linux-based Library OSes

We also considered Linux Kernel Library (LKL) [67], which is based on the Linux source. However, LKL is not as flexible as rump kernels yet. Additionally, rump kernels have been upstreamed into the official NetBSD repository, whereas LKL is still an unofficial fork of the Linux source branch. rump kernels also support far more applications at the moment.

System	API	Generality	TCP Stack	Driver Base	Paradigms	Dynamic Switch	Direct Access	Failure Recovery
DPDK [84]	Low-level	Network	3rd party	Medium	N/A	No	Yes	No
SPDK [77]	Low-level	Storage	N/A	NVMe	N/A	No	Yes	No
IX [7]	Non-standard	Network	App	DPDK	Library OS	No	Almost	No
Arrakis [64]	POSIX/Custom	Network	App	Limited	Library OS	No	Yes	No
Snap [56]	Non-standard	Network	unavailable	Limited	Multiserver	No	No	Yes
VirtuOS [62]	POSIX/Linux	Universal	Server	Large	Multiserver	No	No	Yes
MINIX 3 [30]	POSIX	Universal	Server	Limited	Multiserver	No	No	Yes
HelenOS [13]	POSIX	Universal	Servers	Limited	Multiserver	No	No	Yes
Linux	POSIX/Linux	Universal	Kernel	Large	Monolithic	No	No	No
NetBSD [59]	POSIX/BSD	Universal	Kernel	Large	Monolithic	No	No	No
LibrettOS	POSIX/BSD	Universal	App	Large (NetBSD)	Multiserver, Library OS	**Yes**	**Yes**	**Yes**

Table 1. Comparison of LibrettOS with libraries and frameworks as well as monolithic, multiserver, and library OSes.

Unikernel Linux (UKL) [69] uses the unikernel model for Linux by removing CPU mode transitions. UKL is in an early stage of development; its stated goals are similar to that of rumprun. However, it is unclear whether it will eventually match rump kernels' flexibility.

Aside from technical issues, due to incompatibility, Linux's non-standard GPLv2-only license may create legal issues[2] when linking proprietary[3] or even GPLv3+[4] open-source applications. Since licensing terms cannot be adapted easily without explicit permission from *all* numerous Linux contributors, this can create serious hurdles in the wide adoption of Linux-based library OS systems. In contrast, rump kernels and NetBSD use a permissive 2-BSD license.

2.4 Hypervisors and Microkernels

Hypervisors are typically used to isolate entire guest OSes in separate *Virtual Machines* (VMs). In contrast, *microkernels* are typically used to reduce the *Trusted Computing Base* (TCB) of a single OS and provide component isolation using *process* address space separation. Although hypervisors and microkernels have different design goals, they are inter-related and sometimes compared to each other [26, 28]. Moreover, hypervisors can be used for driver isolation in VMs [17], whereas microkernels can be used as virtual machine monitors [16].

The original uniprocessor rumprun executes as a paravirtualized (PV) [88] guest OS instance on top of the Xen hypervisor [4] or KVM [44]. Recently, rumprun was also ported [14] to seL4 [46], a formally verified microkernel, where rumprun instances execute as ordinary microkernel processes. The seL4 version of rumprun still lacks SMP support, as it only changes its platform abstraction layer.

2.5 PCI Passthrough and Input-Output Memory Management Unit (IOMMU)

VMs can get direct access to physical devices using their hypervisor's PCI passthrough capabilities. IOMMU [2, 39] hardware support makes this process safer by remapping interrupts and I/O addresses used by devices. When IOMMU is enabled, faulty devices are unable to inject unassigned interrupts or perform out-of-bound DMA operations. Similarly, IOMMU prevents VMs from specifying out-of-bound addresses, which is especially important when VM applications are given direct hardware access. We use both PCI passthrough and IOMMU in the LibrettOS prototype to access physical devices.

2.6 Single-Root I/O Virtualization (SR-IOV)

To facilitate device sharing across isolated entities such as VMs or processes, self-virtualizing hardware [68] can be used. SR-IOV [63] is an industry standard that allows splitting one physical PCIe device into logically separate *virtual functions* (VFs). Typically, a host OS uses the device's *physical function* (PF) to create and delete VFs, but the PF need not be involved in the data path of the VF users. SR-IOV is already widely used for high-end network adapters, where VFs represent logically separate devices with their own MAC/IP addresses and Tx/Rx hardware buffers. SR-IOV is also an emerging technology for NVMe storage, where VFs represent storage partitions.

VFs are currently used by VMs for better network performance, as each VM gets its own slice of the physical device. Alternatively, VFs (or even the PF) can be used by the DPDK

[2]This is just an opinion of this paper's authors and must not be interpreted as an authoritative legal advice or guidance.

[3]Due to imprecise language, Linux's syscall exception falls into a gray legal area when the system call layer is no longer present or clearly defined.

[4]Without the syscall exception, GPLv3+ is incompatible with GPLv2 (GPLv2+ only) [83]. Moreover, most user-space code has moved to GPLv3+.

library [84], which implements lower layers of the network stack (L2/L3) for certain NIC adapters. High-throughput network applications use DPDK for direct hardware access.

2.7 POSIX and Compatibility

Although high-throughput network applications can benefit from custom APIs [7], POSIX compatibility remains critical for many legacy applications. Moreover, modern server applications require asynchronous I/O beyond POSIX, such as Linux's epoll_wait(2) or BSD's kqueue(2) equivalent. Typically, library OSes implement their own APIs [7, 64]. Some approaches [34, 85] support POSIX but still run on top of a monolithic OS with a large TCB. LibrettOS provides full POSIX and BSD compatibility (except *fork(2)* and *pipe(2)* as discussed in Section 4.2), while avoiding a large TCB in the underlying kernel.

2.8 Summary

Table 1 summarizes OS design features and compares LibrettOS against existing monolithic, library, and multiserver OSes. Since LibrettOS builds on NetBSD's anykernels, it inherits a very large driver code base. Applications do not generally need to be modified as the user space environment is mostly compatible with that of NetBSD, except for scenarios described in Section 4.2. LibrettOS supports two paradigms simultaneously: applications with high performance requirements can directly access hardware resources as with library OSes (or DPDK [84] and SPDK [77]), while other applications can use servers that manage corresponding resources as with multiserver OSes. Similar to MINIX 3 [30], HelenOS [13], VirtuOS [62], and Snap [56], LibrettOS supports failure recovery of servers, which is an important feature of the multiserver OS paradigm. Additionally, LibrettOS's applications can dynamically switch between the library and multiserver OS modes at run-time.

In Section 6, we provide a detailed discussion of the past and related work and contrast them with LibrettOS.

3 Design

One of the defining features of LibrettOS is its ability to use the same software stacks and device drivers irrespective of whether an application accesses hardware devices directly or via servers. In this section, we describe LibrettOS's architecture, its network server, and dynamic switching capability.

3.1 LibrettOS's Architecture

Figure 3 presents LibrettOS's architecture. In this illustration, Application 1 runs in library OS mode, i.e., it accesses hardware devices such as NVMe and NIC directly by running corresponding drivers in its own address space.

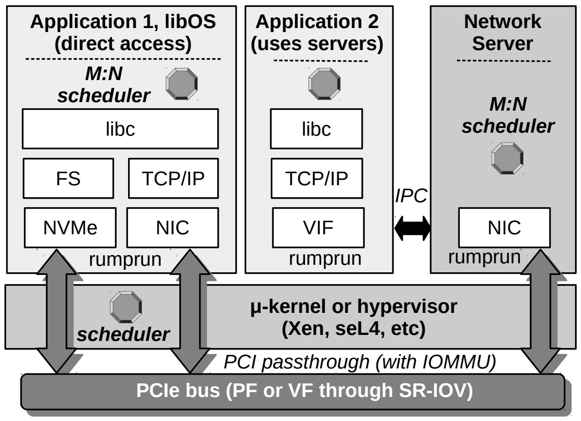

Application 1, libOS (direct access)
M:N scheduler
libc
FS
TCP/IP
NVMe
NIC
rumprun
Application 2 (uses servers)
libc
TCP/IP
VIF
rumprun
IPC
Network Server
M:N scheduler
NIC
rumprun
scheduler
μ-kernel or hypervisor (Xen, seL4, etc)
PCI passthrough (with IOMMU)
PCIe bus (PF or VF through SR-IOV)

Figure 3. LibrettOS's architecture.

Since an OS typically runs several applications, hardware resources need to be shared. In this model, devices are shared using hardware capabilities such as SR-IOV which splits one NVMe device into partitions and one NIC into logically separate NICs with dedicated Tx/Rx buffers. We assume that the NIC firmware does not have fatal security vulnerabilities – i.e., it prevents applications from adversely affecting one another. (VF isolation is already widely used by guest VMs where similar security concerns exist.)

While some high-end NICs expose VFs using SR-IOV, the total number of VFs is typically limited, and only a small number of applications that need faster network access will benefit from this hardware capability. When there are no available VFs for applications, or their direct exposure to applications is undesirable due to additional security requirements, we use network servers. As in library OS mode, Application 2's data (see Figure 3) goes through libc sockets and the entire network stack to generate L2 (data link layer) network frames. However, unlike Application 1, the network frames are not passed to/from the NIC driver directly, but rather relayed through a *virtual interface* (VIF). The VIF creates an IPC channel with a network server where frames are additionally verified and rerouted to/from the NIC driver.

LibrettOS is mostly agnostic to the kind of hypervisor or microkernel used by rumprun. For easier prototyping while preserving rich IPC capabilities and microkernel-like architecture, we used Xen, which runs rumprun instances in VMs. Unlike KVM, Xen has a microkernel-style design and does not use any existing monolithic OS for the hypervisor implementation. Since rumprun has already been ported to the seL4 microkernel [14], and our core contributions are mostly orthogonal to the underlying hypervisor, the LibrettOS design principles can be adopted in seL4 as well.

Scheduling in LibrettOS happens at both the hypervisor and user application levels. The hypervisor schedules per-VM *virtual CPUs* (VCPUs) as usual. Whereas the original rumprun implemented a simple non-preemptive $N : 1$ uniprocessor scheduler, we had to replace it entirely to support SMP – i.e., multiple VCPUs per instance. Each rumprun instance in LibrettOS runs its own $M : N$ scheduler that executes M threads on top of N VCPUs.

LibrettOS does not have any specific restrictions on the type of hardware used for direct access; we tested and evaluated both networking and storage devices. We only implemented a network server to demonstrate indirect access. However, storage devices and file systems can also be shared using an NFS server. The NFS rumprun instance runs on top of an NVMe device initialized with the ext3 file system and exports file system shares accessible through network. In the long term, an existing rump kernel-based approach [41] for running file systems in the user space of monolithic OSes can be adopted in the rumprun environment as well.

LibrettOS does not currently support anything outside of the POSIX/BSD APIs. Admittedly, POSIX has its downsides – e.g., copying overheads which are undesirable for high-performance networking. Nonetheless, LibrettOS already eliminates the system call layer, which can open the way for further optimizations by extending the API in the future.

3.2 Network Server

Due to flexibility of rump kernels, their components can be chosen case by case when building applications. Unlike more traditional OSes, LibrettOS runs the TCP/IP stack directly in the application address space. As we discuss below, this does not compromise security and reliability. In fact, compared to typical multiserver and monolithic OSes, where buggy network stacks affect all applications, LibrettOS's network stack issues are localized to one application.

The network server IPC channel consists of Tx/Rx ring buffers which relay L2 network frames in *mbufs* through a *virtual network interface* (VIF) as shown in Figure 4. We use a lock-free ring buffer implementation from [61]. When relaying mbufs, the receiving side, if inactive, must be notified using a corresponding notification mechanism. For the Xen hypervisor, this is achieved using a virtual interrupt (VIRQ).

Since a stream of packets is generally transmitted without delays, the number of such notifications is typically small. The sending side detects that the receiving side is inactive by reading a corresponding shared (read-only) status variable denoted as *threads*, which indicates how many threads are processing requests. When this variable is zero, a VIRQ is sent to wake a worker thread up.

The network server side has a per-application *handler* which forwards mbufs to the NIC driver. Frames do not need to be translated, as applications and servers are built from the same rump kernel source base.

An important advantage of this design is that an application can recover from any network server failures, as TCP state is preserved inside the application. The network server operates at the very bottom of the network stack and simply routes inbound and outbound mbufs. However, routing across different applications requires some knowledge of the higher-level protocols. We support per-application

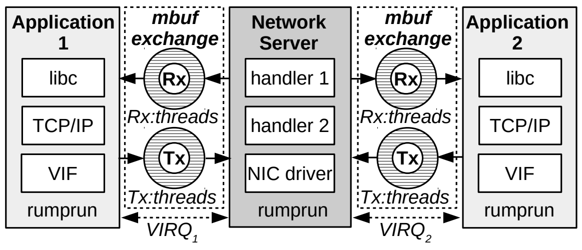


Figure 4. Network server.

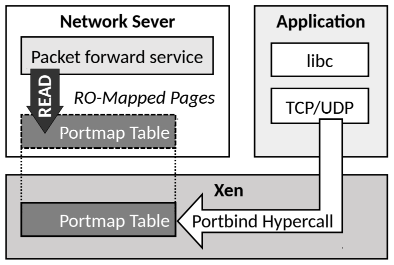


Figure 5. Portmap table.

routing based on TCP/UDP ports. When applications request socket bindings, the network server allocates corresponding ports (both static and dynamic) by issuing a special portbind hypercall as shown in Figure 5 which updates a per-server 64K port-VM (portmap) table in the hypervisor. We store the portmap in the hypervisor to avoid its loss when the network server is restarted due to failures. While this mechanism is specific to the network server, the added code to the hypervisor is very small and is justified for omnipresent high-speed networking. Alternatively, the portmap table can be moved to a dedicated VM.

Since packet routing requires frequent reading of the portmap entries, we read-only map the portmap pages into the network server address space, so that the network server can directly access these entries.

The network server does not trust applications for security and reliability reasons. We enforce extra metadata verification while manipulating ring buffers. Since applications may still transmit incorrect data (e.g., spoof MAC/IP source addresses), either automatic hardware spoofing detection must be enabled, or address verification must be enforced by verifying MAC/IP addresses in outbound frames. Similarly, ports need to be verified by the server.

Although, unlike POSIX, invalid packets may be generated by (malicious) applications at any point, they can only appear in local (per-application) ring buffers. When the network server copies packets to the global medium, it verifies MAC, IP, and port metadata which must already be valid for a given application. Races are impossible as prior users of the same MAC, IP, and port must have already transmitted all packets to the global medium and closed their connections.

We also considered an alternative solution based on the netfront/netback model [4] used by Xen. The netback driver typically runs in an OS instance with a physical device (Driver Domain). The netfront driver is used by guest OSes as a virtual NIC to establish network connectivity through the netback driver. Rumprun already implements a very simple netfront driver so that it can access a virtual NIC when running on top of the Xen hypervisor. However, it lacks the netback driver that would be required for the server. This model is also fundamentally different from our approach: transmitted data must traverse additional OS layers on the network server side and all traffic needs to go through network address translation (NAT) or bridging, as each application (VM instance) gets its own IP address. While the Xen approach is suitable when running guest OSes, it introduces unnecessary layers of indirection and overheads as we show in Section 5.3.

3.3 Dynamic Mode Switch

LibrettOS's use of the same code base for both direct and server modes is crucial for implementing the dynamic mode switch. We implement this feature for networking, which is more or less stateless. As NVMe's emerging support for SR-IOV becomes publicly available, our technique can also be considered for storage devices. However, storage devices are more stateful and additional challenges need to be overcome.

Figure 6 shows LibrettOS's dynamic switching mechanism for the network server. LibrettOS is the only OS that supports this mechanism. Moreover, applications do not need to be modified to benefit from this feature. As we further discuss in Section 5, dynamic switching does not incur large overhead, and switching latency is small (56-69*ms*).

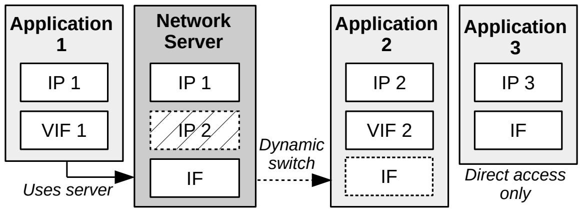


Figure 6. Dynamic switching mechanism.

The biggest challenge for dynamic switching is that application migration must be completely transparent. When using direct access (VF), we need a dedicated IP address for each NIC interface. On the other hand, our network server shares the same IP address across all applications. A complicating factor is that once connections are established, IP addresses used by applications cannot be changed easily.

To allow dynamic switching, we leverage NetBSD's support for multiple IPs per an interface (IF). Unlike hardware Rx/Tx queues, IPs are merely logical resources; they are practically unlimited when using NAT. A pool of available IPs is given to a server, and addresses are recycled as connections are closed. When launching an application that needs dynamic switching (Application 2), the network server allocates an IP address that will be used by this application. Applications use virtual interfaces (VIF) which are initially connected to the network server (i.e., similar to Application 1). The network server adds the IP to the list of configured addresses for its IF (PF or VF).

As I/O load increases, an application can be decoupled from the network server to use direct access for better throughput and CPU utilization. The NIC interface used by the network server deactivates application's IP (which remains *unchanged* throughout the lifetime of an application). The application then configures an available NIC interface (typically, VF) with its dedicated IP. Changing IP on an interface triggers ARP invalidation. A following ARP probe detects a new MAC address for the interface-assigned IP. These events are masked from applications, as they typically use higher-level protocols such as TCP/IP. No pause-and-drain is needed, and on-the-fly requests are simply ignored because TCP automatically resends packets without ACKs.

When running in this mode, all traffic from the NIC interface (direct access) is redirected through VIF, as all application socket bindings belong to this VIF. This special VIF is very thin: *mbufs* are redirected to/from IF without copying, i.e., avoiding extra overhead. When the load decreases, applications can return to the original state by performing the above steps in the reverse order. If applications always use direct access (Application 3), VIFs are not needed, i.e., all traffic goes directly through IF.

Dynamic switching helps conserve SR-IOV resources, i.e., it enables more effective use of available hardware resources across different applications. Policies for switching (e.g., for managing resources, security, etc.) are beyond the scope of this paper, as our focus is on the switching mechanism. In our evaluation, users manually trigger dynamic switching.

4 Implementation

While NetBSD code and its rump kernel layer are SMP-aware (except rump kernel bugs that we fixed), none of the existing rumprun versions (Xen PV, KVM, or seL4) support SMP.

4.1 Effort

We redesigned low-level parts of rumprun (SMP BIOS initialization, CPU initialization, interrupt handling) as well as higher-level components such as the CPU scheduler. Our implementation avoids coarse-grained locks on performance-critical paths. Additionally, the original rumprun was only intended for paravirtualized Xen. Thus, we added support for SMP-safe, Xen's HVM-mode *event channels* (VIRQs) and *grant tables* (shared memory pages) when implementing our network server. We implemented

Component	Description	LOC
SMP support	SMP BIOS init/trampoline, M:N scheduler, interrupts	2092
HVM support	Xen grant tables and VIRQ	1240
PV clock	SMP-safe Xen/KVM clock	148
Network server	mbuf routing, dynamic switching	1869
Xen	Network server registrar	516
NetBSD, network	Network server forwarder	51
NetBSD, kqueue	EVFILT_USER support	303
NetBSD, rump	Race conditions in SMP	45
NetBSD, glue	New drivers (NVMe, ixgbe)	234
NFS, rpcbind, and mountd	Porting from NetBSD, avoiding fork(2)	154
Total:		**6652**

Table 2. Added or modified code.

the Xen HVM interrupt callback mechanism by binding it to the per-CPU interrupt vectors as it is implemented in Linux and FreeBSD. The HVM mode uses hardware-assisted virtualization and is more preferable nowadays for security reasons [53]. Moreover, it has better IOMMU support when using device PCI passthrough. We also fixed race conditions in the rump layer, and added NVMe and 10GbE ixgbe driver glue code which was previously unavailable for rumprun. Additionally, we ported a patch that adds support for the EVFILT_USER extension in kqueue(2), which is required by some applications and supported by FreeBSD but is, for some reason, still unsupported by NetBSD.

Since rumprun does not currently support preemption, we designed a non-preemptive, lock-free M:N scheduler with global and per-VCPU queues. A lock-free scheduler has an advantage that, it is not adversely affected by preemption which still takes place in the hypervisor. (When the hypervisor de-schedules a VCPU holding a lock, other VCPUs cannot get the same lock and make further progress.) Lock-free data structures for OS schedulers were already advocated in [45].

In Table 2, we present the effort that was required to implement our current LibrettOS prototype. Most changes pertain to rumprun and the network server implementation, whereas drivers and applications did not require substantial changes.

4.2 Limitations

In our current prototype, applications cannot create processes through *fork(2)*, as its implementation would require complex inter-VM interactions which Xen is not designed for. We consider such an implementation to be realistic for microkernels such as seL4. Each application, however, can already support virtually unlimited number of threads. Moreover, typical applications that rely on fork [82] can be configured to use the multi-threading mode instead.

We did not implement POSIX's IPCs such as *pipe(2)*, but Xen-based mechanisms can be used with no restrictions.

As *fork(2)* is not currently supported, we did not consider sharing socket file descriptors or *listen(2)*/*accept(2)* handoff across processes. This may create additional challenges in the future. We believe that the problem is tractable, but requires modification of the network stack. Also, Rx/Tx ring buffers will have to be created per connection rather than per process to allow sharing per-socket/connection buffers across processes while not violating process isolation requirements.

As previously mentioned, we did not develop policies for dynamic switching as our focus was on mechanism. Section 5 uses a simple policy wherein a user triggers an event.

Processor	2 x Intel Xeon Silver 4114, 2.20GHz
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	Intel DC P3700 NVMe 400 GB

Table 3. Experimental setup.

5 Evaluation

We evaluate LibrettOS using a set of micro- and macrobenchmarks and compare it against NetBSD and Linux. Both of these baselines are relevant since LibrettOS is directly based on the NetBSD code base, and Linux is a popular server OS. LibrettOS typically has better performance, especially when using direct mode. Occasionally, Linux demonstrates better performance since it currently has more optimizations than NetBSD in its drivers and network stack, especially for 10GbE+, as we further discuss in Section 5.1.

In the evaluation, we only consider standard POSIX compatible applications, as compatibility is one of our important design goals. Since kernel-bypass libraries and typical library OSes [7, 77, 84] are not compatible with POSIX, they cannot be directly compared. Moreover, kernel-bypass libraries require special TCP stacks [40] and high engineering effort to adopt existing applications [89]. Likewise, comparison against existing multiserver designs [30, 56] is challenging due to very limited application and device driver support.

Table 3 shows our experimental setup. We run Xen 4.10.1, and Ubuntu 17.10 with Linux 4.13 as Xen's Dom0 (for system initialization only). We use the same version of Linux as our baseline. Finally, we use NetBSD 8.0 with the NET_MPSAFE feature[5] enabled for better network scalability [58]. LibrettOS uses the same NetBSD code base, also with NET_MPSAFE enabled. We set the maximum transmission unit (MTU) to 9000 in all experiments to follow general recommendations [51] for the optimal 10GbE performance. We measure each data point 10 times and present the average; the relative standard deviation is mostly less than 1%.

LibrettOS not only outperforms NetBSD, but also outperforms Linux in certain tests. This is despite the fact that NetBSD can be slower than Linux. LibrettOS's advantage comes from its superior multiserver-library OS design and better resource management (compared to NetBSD).

5.1 NetBSD and Linux performance

Since neither the original rumprun nor other unikernels support 10GbE+ or NVMe physical devices, one of our goals is to show how this cutting-edge hardware works in LibrettOS.

Although NetBSD, Linux, and rumprun's 1GbE are known to be on par with each other [14], 10GbE+ networking puts more pressure on the system, and we found occasional performance gaps between NetBSD and Linux. However, they are mostly due to specific technical limitations, most of which are already being addressed by the NetBSD community.

One contributing factor is the network buffer size. NetBSD uses *mbuf* chains; each *mbuf* is currently limited to 512 bytes in x86-64. In contrast, Linux has larger (frame-sized) *sk_buff* primitives. Since 512-byte units are too small for 10GbE+ jumbo frames, we increased the *mbuf* size to 4K; this improved performance by ≈10% on macrobenchmarks. *mbuf* cannot be increased beyond the 4K page size easily without bigger changes to the network stack, but this is likely to be addressed by the NetBSD community sooner or later.

Another contributing factor is the use of global SMP locks by NetBSD. Since NET_MPSAFE was introduced very recently, and the effort to remove the remaining coarse-grained locks is still ongoing, Linux's network stack and network drivers may still have superior performance in some tests which use high-end 10GbE+ networking.

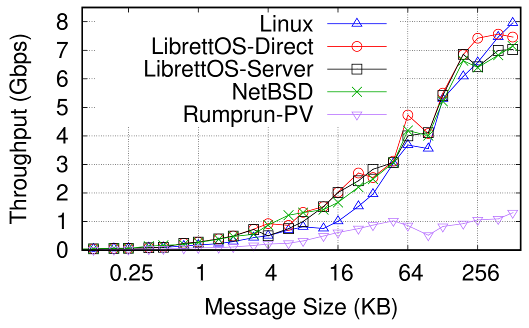
We also found that the adaptive interrupt moderation option in NetBSD's ixgbe driver adversely affected performance in some tests such as NetPIPE (e.g., NetBSD was only able to reach half of the throughput). Thus, we disabled it.

[5]NET_MPSAFE, recently introduced in NetBSD, reduces global SMP locks in the network stack. By and large, the network stack is the only major place where global locks still remain. The ongoing effort will eliminate them.

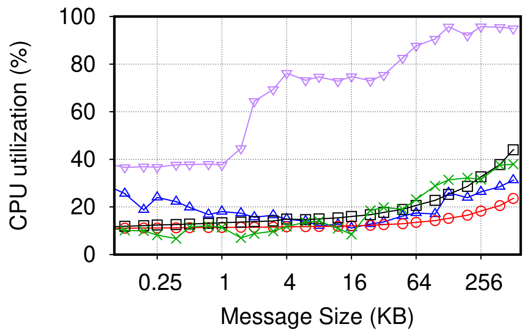
5.2 Sysbench/CPU

To verify our new scheduler and SMP support, we ran the Sysbench/CPU [80] multi-threaded test which measures the elapsed time for finding prime numbers. Overall, LibrettOS, Linux, and NetBSD show roughly similar performance.

5.3 NetPIPE



(a) Throughput



(b) CPU utilization (1 CPU is 100%)

Figure 7. NetPIPE (log2 scale).

To measure overall network performance, we use NetPIPE, a popular ping pong benchmark. It exchanges a fixed size message between servers to measure the latency and bandwidth of a single flow. We evaluate Linux, LibrettOS (direct access mode), LibrettOS (using our network server), NetBSD, and the original rumprun with the netfront driver. On the client machine, Linux is running for all the cases. We configure NetPIPE to send 10,000 messages from 64 bytes to 512KB. Figure 7(a) shows the throughput for different message sizes. LibrettOS (direct access) achieves 7.4Gbps with 256KB. It also has a latency of 282μs for 256KB messages. LibrettOS (network server) achieves 6.4Gbps with 256KB, and the latency of 256KB messages is 326μs. Linux achieves 6.5Gbps with 256KB messages and the latency of 318.9μs. NetBSD reaches 6.4Gbps in this test with the latency of 326μs. The original rumprun with the netfront driver reaches only 1.0Gbps with 256KB messages and has a large latency of 1,983μs.

Overall, all systems except the original rumprun have comparable performance. With smaller messages (< 256KB), LibrettOS-Direct can be a bit faster than Linux or NetBSD. We attribute this improvement to the fact that for smaller messages, the cost of system calls is non-negligible [76]. In LibrettOS, all system calls are replaced with regular function calls. Even LibrettOS-Server avoids IPC costs, as the server and application VMs typically run on different CPU cores.

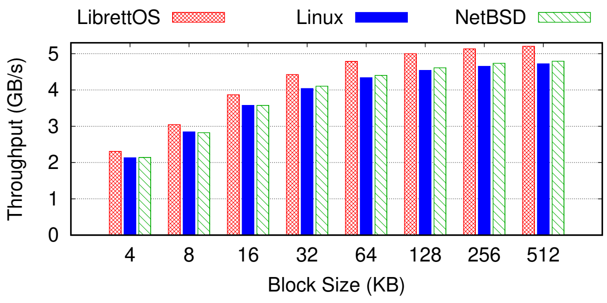


Figure 8. NFS server.

LibrettOS-Direct is more efficient not just on throughput but, more importantly, on CPU utilization (Figure 7(b)). The saved CPU resources can be used in myriad ways – e.g., running more applications, obtaining higher overall performance. For LibrettOS-Server, there is a 1.8x gap in overall CPU consumption due to an extra layer where packets are transferred from an application to the server. The same argument applies to other experiments in this section – e.g., dynamic switching.

5.4 NFS Server

To evaluate NVMe storage, we run an NFS server macrobenchmark by using Sysbench/FileIO [80] from the client. We measure mixed file I/O composed of page cache and storage I/O because users benefit from the page cache in general. We mount an NVMe partition initialized with the ext3 file system and export it through the NFS server. For LibrettOS, we use direct NIC and NVMe access. We mount the provided NFS share on the client side and run the Sysbench test with different block sizes (single-threaded). Figure 8 shows that for all block sizes, LibrettOS outperforms both NetBSD and Linux. LibrettOS is 9% consistently faster than both of them. This is because, LibrettOS avoids the system call layer which is known to cause a non-negligible performance overhead [76].

5.5 Nginx HTTP Server

To run a network-bound macrobenchmark, we choose Nginx 1.8.0 [60], a popular web server. We use the Apache Benchmark [81] to generate server requests from the client side. The benchmark is set to send 10,000 requests with various levels of concurrency on the client side. (In this benchmark, concurrency is the number of concurrent requests.) Figure 9 shows the throughput with different concurrency levels and file sizes. LibrettOS (direct access and server) is always faster than NetBSD. Except very large blocks (128KB), LibrettOS-Direct also outperforms Linux. Particularly, for 8KB/50, LibrettOS (direct access) is 66% faster than NetBSD and 27% faster than Linux. Even LibrettOS-Server outperforms Linux in many cases due to its optimized, fast IPC mechanism. We note that LibrettOS is able to outperform Linux even though the original NetBSD is slower than Linux in this test. We generally attribute this improvement to the fact that LibrettOS removes the system call layer. Furthermore, compared to NetBSD, LibrettOS improves scheduling and resource utilization. For very large blocks (128KB), NetBSD's network stack current limitations outweigh other gains.

We also evaluated dynamic switching when running Nginx. In Figure 10, LibrettOS-Hybrid represents a test with runtime switching from the server to direct mode and back to the server mode (the concurrency level is 20). Overall, Nginx spends 50% of time in the server mode and 50% in the direct mode. LibrettOS-Hybrid's average throughput shows benefits of using this mode compared to LibrettOS-Server, Linux, and NetBSD. The relative gain (Figure 10) over Linux, let alone NetBSD, is quite significant in this test. We also measured the cost of dynamic switching to be $56ms$ from the server to direct mode, and $69ms$ in the opposite direction.

5.6 Memcached

We also tested LibrettOS's network performance with memcached [57], a well-known distributed memory caching system, which caches objects in RAM to speed up database accesses. We use memcached 1.4.33 on the server side. On the client side, we run memtier_benchmark 1.2.15 [71] which acts as a load generator. The benchmark performs SET and GET operations using the *memcache_binary* protocol. We use the default ratio of operations 1:10 to ensure a typical, read-dominated workload. In Figure 11, we present the number of operations per second for a small and large block sizes and using a different number of threads (1-20). Each thread runs 10 clients, and each client performs 100,000 operations.

LibrettOS-Direct and Linux show similar overall performance. LibrettOS-Server reveals a runtime overhead since it has to run a server in a separate VM. Finally, NetBSD shows the worst performance as in the previous test.

5.7 Redis

Finally, we ran Redis, an in-memory key-value store used by many cloud applications [70]. We use Redis-server 4.0.9 and measure the throughput of 1,000,000 SET/GET operations. In the pipeline mode, the Redis benchmark sends requests without waiting for responses which improves performance. We set the pipeline size to 1000. We measure the performance of Redis for various number of concurrent connections (1-20). We show results for the SET/GET operation (128 bytes); trends for other sizes are similar. LibrettOS's GET throughput (Figure 12) is close to Linux, but the SET throughput is smaller. NetBSD reveals very poor performance in this test, much slower than that of LibrettOS and Linux. We suspect that the current Redis implementation is suboptimal for certain BSD systems, which results in worse performance that we did not observe in other tests.

5.8 Failure Recovery and Software Upgrades

LibrettOS can recover from any faults occurring in its servers, including memory access violations, deadlocks, and interrupt handling routine failures. Our model assumes that failures are localized to one server, i.e., do not indirectly propagate elsewhere. Applications that use our network server do not necessarily fail: the portmap state is kept in the hypervisor, and if recovery happens quickly, TCP can still simply resend packets without timing out (applications keep their TCP states). Additionally, when upgrading the server, an artificial fault can be injected so that an application can use a newer version of the server without restarting the application itself.

To demonstrate LibrettOS's failure recovery (or transparent server upgrades), we designed two experiments. In the first experiment, an application has two jobs: one uses networking (Nginx) and the other one accesses a file system. When LibrettOS's network server fails, the Nginx server is unable to talk to the NIC hardware, and we report outage that is observable on the client side. This failure, nonetheless, does not affect the file system job. In Figure 13(a), we show the corresponding network and file system transfer speeds. After rebooting the network server, Nginx can again reach the network, and clients continue their data transfers. In the other test, Figure 13(b), we run two applications. Nginx uses direct access while Redis uses our network server. We show that a failure of Nginx does not affect Redis. Then we also show that a network server failure does not impact Nginx.

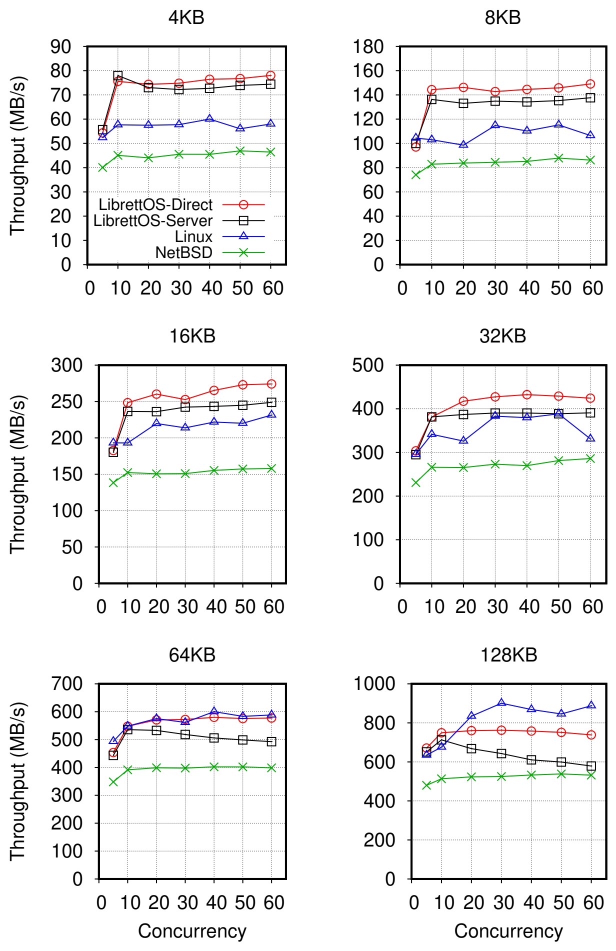


Figure 9. Nginx HTTP server.

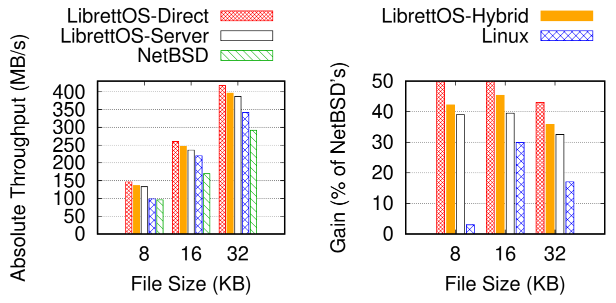


Figure 10. Dynamic switch (Nginx).

6 Related Work

LibrettOS's design is at the intersection of three OS research topics: system component isolation for security, fault-tolerance, and transparent software upgrades (multiserver OS); application-specialized OS components for performance, security, and software bloat reduction (library OS); and kernel bypassing with direct hardware access to applications for performance.

Isolation of system software components in separate address spaces is the key principle of the microkernel model [1, 13, 27, 30, 33, 36, 52]. Compared to the classical monolithic OS design, microkernels provide stronger security and reliability guarantees [22, 31, 46]. With microkernels, only essential functionalities (scheduling, memory management, IPC) are implemented within the kernel. L4 [52] is a family of microkernels, which is known to be used for various purposes. A notable member of this family is seL4 [46], a formally verified microkernel. Multiserver OSes such as MINIX 3 [30], GNU Hurd [8], Mach-US [79], and SawMill [20] are a specialization of microkernels where OS components (e.g., network and storage stacks, device drivers) run in separate user processes known as "servers."

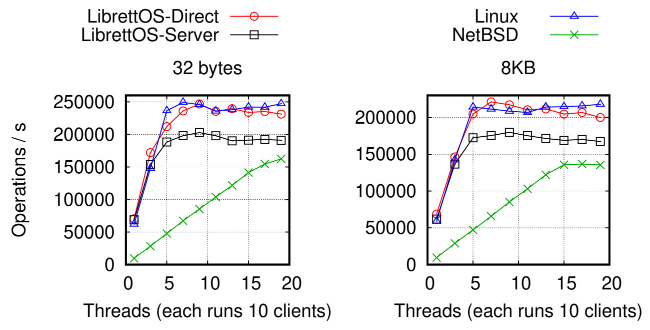


Figure 11. Memcached (distributed object system).

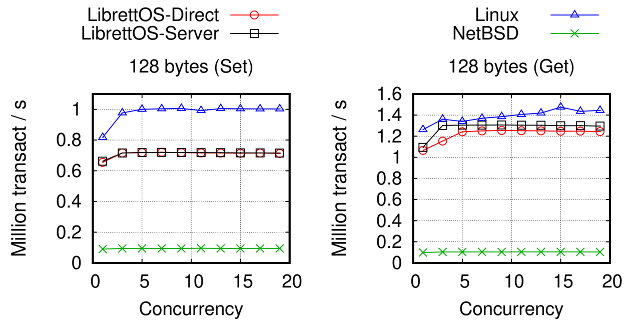
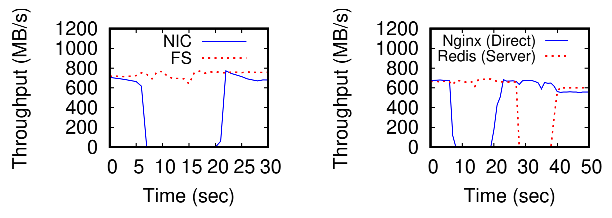


Figure 12. Redis key-value store.



(a) Single application (Nginx) (b) Multiple applications

Figure 13. Failure recovery and transparent upgrades.

Decomposition and isolation have proved to be beneficial for various layers of system software in virtualized environments: hypervisor [75], management toolstack [11], and guest kernel [62]. Security and reliability are particularly important in virtualized environments due to their multi-tenancy characteristics and due to the increased reliance on cloud computing for today's workloads. In the desktop domain, Qubes OS [73] leverages Xen to run applications in separate VMs and provides strong isolation for local desktop environments. LibrettOS also uses Xen to play the role of the microkernel. However, the LibrettOS design is independent of the microkernel used and can be applied as-is to other microkernels supporting virtualization, with potentially smaller code bases [46, 78, 87].

The network stack is a crucial OS service, and its reliability has been the subject of multiple studies based on microkernels/multiserver OSes using state replication [12], partitioning [35], and checkpointing [21]. Several user-space network stacks such as mTCP [40], MegaPipe [25], and OpenOnload [65] have been designed to bypass the OS and avoid various related overheads. While it is not their primary objective, such user-space network stacks do provide a moderate degree of reliability: since they do not reside in the kernel, a fault in the network stack will not affect the applications that are not using it.

In its multiserver mode, LibrettOS obtains the security and reliability benefits of microkernels by decoupling the application from system components such as drivers that are particularly prone to faults [23, 32] and vulnerabilities [9].

IX [7] and Arrakis [64] bypass traditional OS layers to improve network performance compared to commodity OSes. IX implements its own custom API while using the DPDK library [84] to access NICs. Arrakis supports POSIX but builds on top of the Barrelfish OS [5], for which device driver support is limited. LibrettOS gains similar performance benefits when using direct hardware access (library OS) mode, while reducing code development and maintenance costs as it reuses a very large base of drivers and software from a popular OS, NetBSD. Additionally, in contrast to LibrettOS, these works do not consider recoverability of OS servers.

Certain aspects of the multiserver design can be employed using existing monolithic OSes. VirtuOS [62] is a fault-tolerant multiserver design based on Linux that provides strong isolation of the kernel components by running them in virtualized service domains on top of Xen. Snap [56] implements a network server in Linux to improve performance and simplify system upgrades. However, Snap uses its private (non-TCP) protocol which cannot be easily integrated into existent applications. Moreover, Snap is limited to networking and requires re-implementing all network device drivers.

On-demand virtualization [47] implements full OS migration, but does not specifically target SR-IOV devices. In contrast, LibrettOS allows dynamic device migrations.

To optimize I/O in VMs, researchers have proposed the sidecore approach [19, 48] which avoids VM exits and offloads the I/O work to sidecores. Since this approach is wasteful when I/O activity reduces, Kuperman et al. [49] proposed to consolidate sidecores from different machines onto a single server.

Exokernel [15] was among the first to propose compiling OS components as libraries and link them to applications. Nemesis [72] implemented a library OS with an extremely lightweight kernel. Drawbridge [66], Graphene [85], and Graphene-SGX [86] are more recent works that leverage the security benefits of library OSes due to the increased isolation resulting from the lesser degree of interaction between applications and the privileged layer. Bascule [6] demonstrated OS-independent extensions for library OSes. EbbRT [74] proposed a framework for building per-application library OSes for performance.

Library OSes specialized for the cloud – unikernels [10, 43, 45, 50, 55, 90] – have also emerged in recent years. They are dedicated to running in the cloud and are a practical implementation of the library OS model where the hypervisor plays the role of the exokernel. In its library OS mode, LibrettOS builds upon and reaps the benefits of this model. Additionally, LibrettOS gains the same benefits of multiserver OSes such as failure recovery. LibrettOS is also the first SMP design that runs unikernels with unmodified drivers that belong to the hardened NetBSD driver code base.

Red Hat has recently initiated an effort, which is in its early stage, to create a Linux unikernel (UKL) [69]. UKL promises to resolve the problem of maintaining a large base of drivers and specialized I/O stacks (e.g., DPDK [84], SPDK [77], and mTCP [40]). UKL's long-term goal of replacing DPDK and SPDK by using unikernelized applications is already achieved in the LibrettOS design, which also encapsulates a more powerful multiserver/library OS design.

7 Conclusion

We presented LibrettOS, an OS design that unites two fundamental models, the multiserver OS and the library OS, to reap their combined benefits. In essence, LibrettOS uses a microkernel to run core OS components as servers for better isolation, reliability, and recoverability. For selected applications, LibrettOS acts as a library OS such that these applications are given direct access to I/O devices to improve performance. We also support dynamic runtime switching between the two modes. In LibrettOS's unique design, not only most OS code stays identical as an application switches between different modes of operation, more importantly, the application does not have to be modified either.

We built a prototype of the LibrettOS design based on rump kernels. LibrettOS is compatible both in terms of BSD and POSIX user-space applications, and also in terms of the large driver code base of the NetBSD kernel. Our prototype implementation necessitated a significant redesign of the rumprun environment which executes rump kernels, in particular, the implementation of SMP support. To demonstrate the multiserver paradigm, we implemented a network server based on direct L2 frame forwarding. We evaluated this prototype and demonstrated successful recovery from OS component failures (and transparent server upgrades) as well as increased I/O performance compared to NetBSD and Linux, especially when using direct access.

Finally, although kernel-bypass advantages are widely discussed in the literature, unlike prior works [7, 56, 64], LibrettOS is the first to achieve greater performance with ordinary POSIX applications and out-of-the-box device drivers.

Availability

LibrettOS is available at: https://ssrg-vt.github.io/librettos

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Dilma Da Silva for their insightful comments and suggestions, which helped greatly improve this paper. We also thank Pierre Olivier for his comments and suggestions.

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, and N00014-18-1-2022.

References

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

[2] AMD, Inc. AMD I/O Virtualization Technology (IOMMU) Specification, 2016. http://developer.amd.com/wordpress/media/2013/12/48882_IOMMU.pdf.

[3] T. E. Anderson. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*, pages 92–94, April 1992.

[4] Paul Barham, 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 19th ACM Symposium on Operating Systems Principles*, SOSP'03, pages 164–177, 2003.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[6] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th European Conference on Computer Systems*, EuroSys'13, pages 239–252, 2013.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation*, OSDI'14, 2014.

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

[9] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, APSys'11, 2011.

[10] Cloudozer LLP. LING/Erlang on Xen, 2018. http://erlangonxen.org/.

[11] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 189–202, 2011.

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

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

[14] Kevin Elphinstone, Amirreza Zarrabi, Kent Mcleod, and Gernot Heiser. A Performance Evaluation of Rump Kernels As a Multi-server OS Building Block on seL4. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys'17, pages 11:1–11:8, 2017.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP'95, pages 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[16] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design & Implementation*, OSDI'96, pages 137–151, 1996.

[17] Keir Fraser, H. Steven, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on-demand IT InfraStructure*, OASIS'04, 2004.

[18] Free Software Foundation, Inc. GNU Hurd: New Driver Framework, 2017. http://www.gnu.org/software/hurd/community/gsoc/project_ideas/driver_glue_code.html.

[19] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High-Performance Hypervisor Architectures: Virtualization in HPC Systems. HPCVirt'07, 2007.

[20] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 109–114, 2000.

[21] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. We Crashed, Now What? In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, pages 1–8, 2010.

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

[23] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP'09, pages 103–116, 2009.

[24] Google. Fuchsia Project, 2019. http://fuchsia.dev/.

[25] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, 2012.

[26] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, HotOS'05, 2005.

[27] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[28] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are virtual-machine monitors microkernels done right? *SIGOPS Operating Systems Review*, 40(1):95–99, January 2006.

[29] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. The architecture of a fault-resilient operating system. In *Proceedings of 12th ASCI Conference*, ASCI'06, pages 74–81, 2006.

[30] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Reorganizing UNIX for Reliability. In *Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'06, pages 81–94, 2006.

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

[32] Jorrit N Herder, David C Van Moolenbroek, Raja Appuswamy, Bingzheng Wu, Ben Gras, and Andrew S Tanenbaum. Dealing with driver failures in the storage stack. In *Dependable Computing, 2009. LADC'09. 4th Latin-American Symposium on*, pages 119–126, 2009.

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

[34] Jon Howell, Bryan Parno, and John R. Douceur. How to Run POSIX Apps in a Minimal Picoprocess. In *Proceedings of the 2013 USENIX Annual Technical Conference*, ATC'13, pages 321–332, 2013.

[35] Tomas Hruby, Cristiano Giuffrida, Lionel Sambuc, Herbert Bos, and Andrew S Tanenbaum. A NEaT Design for Reliable and Scalable Network Stacks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 359–373, 2016.

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

[37] Intel. Management Engine, 2019. http://www.intel.com/content/www/us/en/support/products/34227/software/chipset-software/intel-management-engine.html.

[38] Intel Corporation. Intel 82599 10 GbE Controller Datasheet, 2019. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[39] Intel Corporation. Intel's Virtualization for Directed I/O, 2019. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf.

[40] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[41] Antti Kantee. Rump File Systems: Kernel Code Reborn. In *Proceedings of the 2009 USENIX Annual Technical Conference*, ATC'09, 2009.

[42] Antti Kantee. Flexible operating system internals: The design and implementation of the anykernel and rump kernels. In *Ph.D. thesis, Department of Computer Science and Engineering, Aalto University*, 2012.

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

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

[45] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. OSv - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*, ATC'14, page 61, 2014.

[46] 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. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, 2009.

[47] Thawan Kooburat and Michael Swift. The Best of Both Worlds with On-demand Virtualization. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, HotOS'11, Berkeley, CA, USA, 2011. USENIX Association.

[48] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Rearchitecting VMMs for Multicore Systems: The Sidecore Approach.

WIOSCA'07, 2007.

[49] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual remote i/o. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 49–65, New York, NY, USA, 2016. ACM.

[50] Stefan Lankes, 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 2016, 2016.

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

[52] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[53] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[54] Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. File Systems as Processes. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage'19, 2019.

[55] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 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, pages 461–472, 2013.

[56] 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. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 399–413, 2019.

[57] Memcached Contributors. Memcached, 2019. http://memcached.org/.

[58] NetBSD Contributors. The NetBSD Project: Announcing NetBSD 8.0, 2018. http://www.netbsd.org/releases/formal-8/NetBSD-8.0.html.

[59] NetBSD Contributors. The NetBSD Project, 2019. http://netbsd.org/.

[60] NGINX Contributors. Nginx: High Performance Load Balancer, Web Server, Reverse Proxy, 2019. http://nginx.org/.

[61] Ruslan Nikolaev. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *Proceedings of the 33rd International Symposium on Distributed Computing, DISC 2019*, volume 146 of *LIPIcs*, pages 28:1–28:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[62] Ruslan Nikolaev and Godmar Back. VirtuOS: An Operating System with Kernel Virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 116–132, 2013.

[63] PCI-SIG. Single Root I/O Virtualization and Sharing Specification, 2019. http://pcisig.com/specifications/iov/.

[64] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, 2014.

[65] Steve Pope and David Riddoch. OpenOnload A User-level Network Stack, 2008. http://www.openonload.org/openonload-google-talk.pdf.

[66] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. *SIGARCH Comput. Archit. News*, 39(1):291–304, March 2011.

[67] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333, June 2010.

[68] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC'07, pages 179–188, 2007.

[69] Ali Raza, 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 17th Workshop on Hot Topics in Operating Systems*, HotOS'19, pages 7–13, New York, NY, USA, 2019. ACM.

[70] Redis Contributors. Redis, 2019. http://redis.io/.

[71] Redis Labs. Memtier Benchmark, 2019. http://github.com/RedisLabs/memtier_benchmark/.

[72] Dickon Reed and Robin Fairbairns. Nemesis, The Kernel-Overview. 1997.

[73] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 54, 2010.

[74] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A Framework for Building per-Application Library Operating Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 671–688, USA, 2016. USENIX Association.

[75] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *The Network and Distributed System Security Symposium*, NDSS'17, 2017.

[76] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'10, pages 1–8, 2010.

[77] SPDK Contributors. Storage Performance Development Kit (SPDK), 2019. http://spdk.io/.

[78] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys'10, pages 209–222, 2010.

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

[80] Sysbench Contributors. SysBench 1.0: A System Performance Benchmark, 2019. http://github.com/akopytov/sysbench/.

[81] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool, 2019. http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[82] The Apache Software Foundation. Apache HTTP server, 2019. http://httpd.apache.org/.

[83] The Free Software Foundation. GPL-Compatible Free Software Licenses, 2019. http://www.gnu.org/licenses/license-list.en.html#GPLCompatibleLicenses.

[84] The Linux Foundation. Data Plane Development Kit (DPDK), 2019. http://dpdk.org/.

[85] 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. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys'14, pages 9:1–9:14, 2014.

[86] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, ATC'17, pages 645–658, 2017.

[87] Alexander Warg and Adam Lackorzynski. L4Re Runtime Environment, 2018. http://l4re.org/.

[88] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design & Implementation*, OSDI'02, pages 195–209, 2002.

[89] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS'19, pages 73–80, New York, NY, USA, 2019. ACM.

[90] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*, ATC'18, 2018.