MROS: A secure and scalable microkernel

[Paper draft, do not distribute!]

Xi Wang
xkw5215@psu.edu
The Pennsylvania State University
University Park, PA, USA

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

ABSTRACT

Microkernels are experiencing a renaissance and found many applications from embedded systems to more classical OSs, e.g., Intel ME, seL4, Google's Zircon, among many other academic and industry prototypes and systems. Achieving both great security and high scalability still remains a challenge, nonetheless, e.g., seL4 neither formally verifies its multi-core variant nor supports address space layout randomization (ASLR) in its formally verified version.

We introduce an early prototype of MROS, a microkernel that we develop from scratch in Rust by carefully considering issues related to multi-core performance, security, reliability, and compatibility with applications. In MROS, we demonstrate that a secure user-space asynchronous and fast IPC is feasible, despite conventional wisdom and prior beliefs of state-of-the-art microkernel developers. For task scheduling, we propose a novel approach of inter-process thread migration which enables workload self-balancing between user processes and microkernel servers.

We evaluate our current single-core MROS implementation and compare its performance against seL4 and Linux. MROS shows competitive IPC performance with respect to Linux system calls and better performance than that of seL4, especially for cross-core IPCs.

1 INTRODUCTION

Microkernels [8, 10, 18, 22] as well as microkernel-style paradigms [15, 16] have been studied for decades, with the primary goal of increasing isolation and security. Microkernels have a tiny TCB (trusted computing base) due to the underlying kernel minimalism.

Microkernels typically handle virtual memory and perform task scheduling only. Most system components such as file systems, device drivers, and other modules are moved into the user space. Since many components run in user space, the communication between various modules and applications is done via IPC (inter-process communication) rather than system calls. A number of works focused on optimizing IPCs, some of them [17] even propose to use specialized hardware extensions.

seL4 [10], a formally-verified microkernel, uses a capability system to manage resource access permissions. seL4 inherits the IPC mechanism [12] from the L4 family, which separate IPCs into slowpath and fastpath variants. Slowpath IPC works in a traditional system-call way. Fastpath IPC optimizes communication performance by using registers for short messages. While seL4 is often considered a state-of-the-art microkernel, to maintain feasibility of formal verification, seL4 uses the big kernel lock (BKL) widely [20], which hinders its scalability on modern, many-core systems. Moreover, to keep the kernel verifiable, seL4 makes several design choices that can be suboptimal for high-contention circumstances. For example, seL4 pushes the allocation policy to user space and requires finding out all outstanding capabilities before returning a memory block. This reclamation process may harm performance as it cannot be implemented via a simple reference counter. seL4 also uses a polling-based kernel event model for yield, interrupts, and exceptions. Polling for all interrupts and exceptions could waste many cycles, which can be bad for both power efficiency and CPU utilization.

In this paper, we propose to fully rethink how a new secure and scalable microkernel should be designed. We build an early prototype for a single core[1] with the intention to extend its support for multiple cores in the future. First, our MROS microkernel is built from scratch in Rust. We take a practical approach to enhance security of the underlying microkernel by designing it in Rust and relying on its memory safety features. Rust receives increasing interest from the systems community as it provides good support for both memory security and concurrency programming. It enforces memory safety through the ownership system. Moreover, we believe that formal verification would be more sustainable in the future for the Rust-based microkernel, as it already enforces stricter rules than plain C.

Second, we propose to implement user-space IPC instead of more traditional in-kernel IPC. In seL4, IPCs are implemented in the kernel space through mode switches and/or context switches. It leads to the overhead of mode switching for every message passing. Like in a recent work [11], we propose to revisit user-space IPCs, which were used historically

[1]To be precise, our prototype supports two cores – one for a server and the other one for an application, but the prototype lacks support for multiple cores in a more general case.

in microkernels. However, unlike this prior work, we adopt a number of advanced lock-free synchronization techniques, making it feasible to achieve scalability for modern workloads. By fully avoiding the kernel, we can greatly cut costs related to mode and context switching, as user space process can easily and safely manipulate data structures directly from user space.

We also propose an innovative thread scheduling model, which will be integrated with our user-space IPC. On the surface, it is an M:N thread core-aware scheduler, which is similar to scheduler activations [3] and Arachne [21]. However, our approach introduces the concept of *inter-process thread migration*, which enables workload self-balancing. The key idea is to provision per-core user threads for better parallelism, but migrate idle user threads directly into a server process to provide additional CPU resources for IPC handling. The key innovation is that our thread migration will enable self-balancing in highly-dynamic workloads.

The main contributions of this paper are as follows:

- An early prototype of the microkernel that is written in safe Rust; it currently can run applications using one core.
- A new user-space IPC mechanism, which avoids address space and mode switch costs.
- We propose a thread migration mechanism for dynamic workloads to be implemented as part of the future work.

2 ARCHITECTURE

Figure 1 shows the architecture of MROS. The current design of MROS contains two main components. The IPC library provides system calls for processes to establish new IPC channels with other processes. The kernel only involves in allocating, granting, and freeing the IPC channel to the communication ends. All messages passing later are processed in the user space.

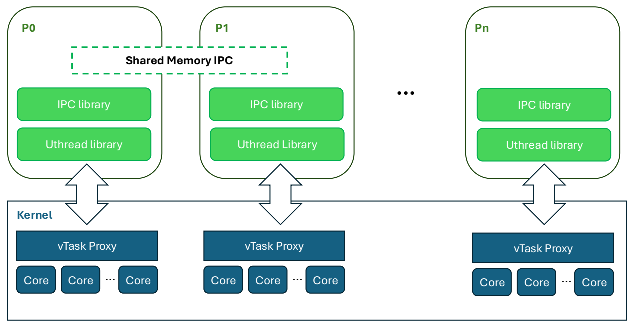

Figure 1: The architecture of MROS.

We propose a self-balanced M:N threading model, where M threads in a process run on top of N underlying kernel-managed threads, known as virtual Tasks (vTasks). We allow dynamic thread migration from a user process to a corresponding server process. Each process can allocate one vTask per core and pin it to the core. For example, A client process may initially use 4 vTasks attached to 4 physical cores, then connect to a server which uses 1 vTask only. With an increasing number of IPC requests, vTasks migrate to the server side. vTasks can migrate back when the server load decreases (i.e., vTask is becoming idle). Assuming that we avoid frequent and premature migration, we gradually reach an equilibrium where the number of vTasks on each side roughly correspond to the total workload.

The key feature in this proposed design is that *both* the IPC mechanism and the migratable threading model will benefit from each other. As MROS partitions the physical core into a disjoint set for different address spaces, the IPC avoids the address space switch between the client/server process pair since different cores are used for corresponding vTasks. On the other hand, the threading model will self-adjust to avoid core overprovisioning on the user side and underprovisioning on the server side, and vice versa. While we still need mode switches and context switches during thread migration, this a relatively more infrequent event compared to the regular IPC request which is fully handled in user space.

3 DESIGN

3.1 User Space IPC

IPC (Inter-Process Communication) is the core communication component in microkernels. Traditionally, IPC has several steps: (1) trapping into the kernel, (2) switching address space to the receiver process, (3) returning to user mode. It also has extra overheads, e.g., due to message copying. High overheads limit the overall performance of microkernels.

We implement a shared-memory user space IPC in MROS. Figure 3 depicts the IPC workflow. We adapt a high-performant lock-free MPMC (Multi-Producer Multi-Consumer) queue [19] as the communication channel, but make another step to better support our threading and migration model described above by using multiple *coalesced* queues in the *same* ring buffer. More specifically, we allow multiple heads and multiple tails (to avoid contention) while also increasing the total number of entries. This approach is different from just using multiple ring buffers because we have flexibility and can decide how many heads and how many tails we use (the number of heads is not necessarily the same as the number of tails, and we can adjust their number dynamically).

In Figure 2, we present the main idea of the coalesced queues and modification to the original algorithm. Multiple ring buffers are coalesced together into a single ring buffer. For both dequeuers and enqueuers, we provide virtual queue (vQueue) views, which are associated with different head

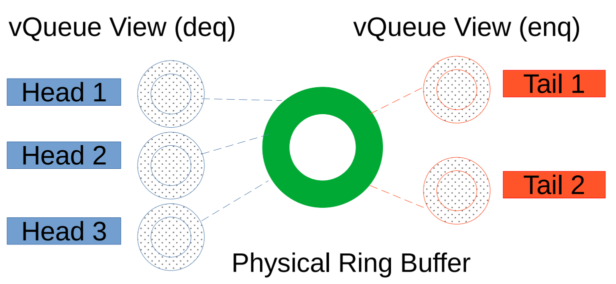

Figure 2: Queue Coalescing and vQueue (virtual queue).

and tail pointers. This structure is feasible due to the flexibility of the original queue [19] design. The original design already somewhat decoupled the ring buffer from head and tail pointers to simplify synchronization. However, we went a step further and made substantial modifications to intertwine multiple independent ring buffer together, which was not envisioned by the original algorithm.

We call the handles inserted into the user processes endpoints. When creating a new process, the kernel maps a read-only upcall trampoline into its address space. The upcall trampoline dispatches kernel signals to user-registered handlers. Each process holds an open IPC table to record available IPC endpoints.

As in Figure 3, an IPC register process works as follows: 1. a process starts IPC by calling the `createipc()` system call. 2. The kernel allocates a chunk of memory for the IPC channel, then maps the channel into the virtual address space of both the client and the server process, and writes the bind information into the channel table. 3. The kernel upcalls into the server through the upcall trampoline. The server's runtime registers the new endpoint provided by the upcall into its open IPC table. 4. The `createipc()` writes a new endpoint into the caller's open IPC table and returns it after all steps are finished.

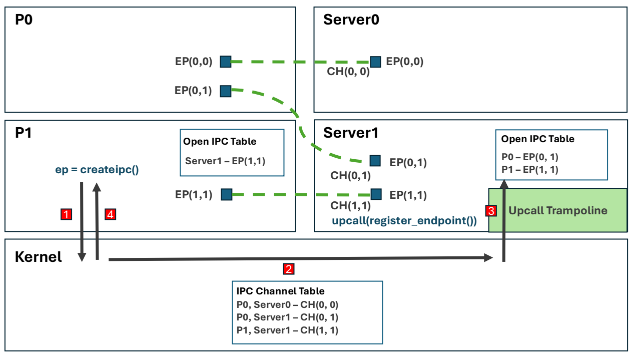

Figure 3: Userspace Inter-Process Communication.

An IPC entry contains a pointer to the thread's TCB (Task Control Block), a slot for the return value, and an array of arguments. IPC entries are aligned to cache line size to avoid false sharing. Both communication side polls on their endpoints to get the new messages.

The client calls `closeipc()` to end an IPC. The kernel upcalls the server to remove the endpoint from its open IPC table, then deallocate the IPC channel and unmap it from both side's address spaces. The client process frees the endpoint from its open IPC table when receiving the acknowledgment from the kernel.

We believe the IPC design brings several performance gains: (1) it does not require the involvement of the kernel for each communication. The kernel interferes only when building and destroying an IPC channel, (2) it avoids using IPI to communicate between two processes located on different cores. (3) the modified MPMC coalesced queue yields high throughput under high contention. Together with our thread model, the IPC also reduces address space switch overheads between client and server because they run on disjoint core sets.

3.2 Self-balanced M:N Threading

For our future implementation, we propose a threading model that shares a similar design to scheduler activations [3] and Arachne [21]. Processes use proxies to execute user threads on physical cores. We call this proxy vTask. vTask is a *kernel-visible* thread: it contains a vTask context, a page table directory pointer, a kernel stack, and information about the core it is currently assigned to. vTask is mapped to an image in user runtime. The user space image consists of a context and a user stack.

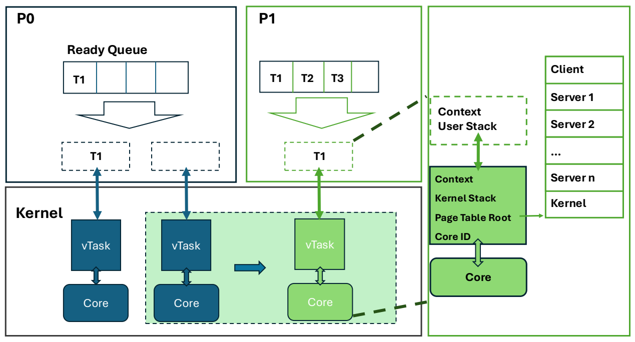

Figure 4: MROS Thread Model.

As in Figure 4, each process is assigned a set of cores, and each core is pinned to a VCPU. MROS provides an interface for the process to require extra physical cores. When a thread calls `createipc()`, the kernel maps the server's address space into the vTask's page directory. If a client process connects to multiple servers, all servers' address spaces are mapped into the same vTaks's page directory. Mapping address space of all communication ends is for fast thread migration among processes.

The user runtime checks the load factor of the ready queue and the number of outstanding IPC messages. We temporarily set a fixed threshold for these factors. When either factor exceeds this threshold, MROS tries to collect a core from other members of its communication group. MROS migrates a vTask with the core it works on. If the vTask doesn't belong to any core, then the kernel only migrates vTask itself. After migration, the kernel will try to allocate a new physical core to this vTask. If failed, the vTask shared a core with other vTasks in the same process. The self-balanced process will end when the load factor of both sides reaches an equilibrium.

4 PRELIMINARY EVALUATION

We evaluate MROS's IPC latency and compare it against seL4's IPC and Linux's system call latency. For seL4 [10], we use `seL4_Call()` and `seL4_ReplyRecv()` for evaluation. For Linux, we use `gettid()` as a zero-argument NULL-syscall, and an invalid `mmap()` call which returns EINVAL as a 6-argument NULL-syscall. We also considered SkyBridge [17], an IPC approach which relies on special Intel VM extensions, but could not evaluate it due to absence of its code and incompatibility with AMD CPUs. However, we indirectly compare with SkyBridge's previously reported numbers.

We measure latencies on AMD EPYC 9554 (3.75 GHz) server, which has 64 cores, 128 hardware threads (via hyperthreading), and 384 GiB of RAM. Linux executes natively, while both seL4 and MROS run through hardware-assisted virtualization in KVM. For Linux, kernel page-table isolation is disabled (i.e., there is no additional syscall overhead) as the CPU is not susceptible to the Meltdown [14] security attack.

It should be noted that while absolute results are heavily influenced by the the underlying system configuration, the overall trends remain the same. For example, the server exhibits a higher syscall overhead for Linux than a similar workstation with a substantially *lower* number of cores. However, the *same* is also true for the IPC synchronization costs in MROS. That is, the relative difference would stay fairly close no matter what system we use.

In Figure 5, MROS exhibits the smallest latency among the three systems as it fully eliminates the cost of address space and mode switches. Linux requires a mode switch, which incurs a direct syscall cost. seL4's IPC workflow contains both address space and kernel-user mode switches. Its capability system also participates in the IPC workflow, which increases costs. Although seL4 [10] also supports fast path

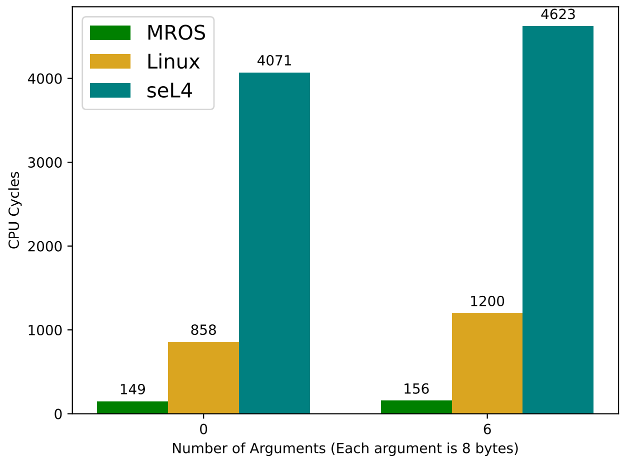

Figure 5: IPC performance.

IPC, it is still slower than Linux syscall and can only be triggered when satisfying specific conditions. SkyBridge's [17] reported overhead were around 400 cycles, which makes it worse than MROS despite SkyBridge's reliance on special hardware VM features.

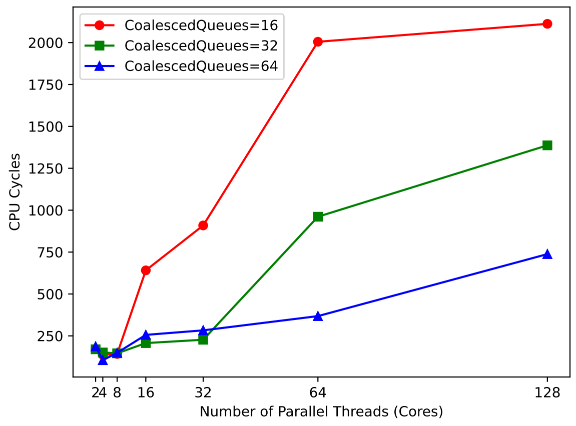

Figure 6: MROS' highly-contended parallel IPC requests.

The above results demonstrate an uncontended (single user thread) case. However, the IPC performance can degrade when using all 128 parallel threads, especially when using shared-memory mechanisms such as in MROS. To mitigate this issue, we developed coalesced IPC queues (with multiple heads and multiple tails). We evaluate the coalesced IPC queue in Figure 6 (we evaluate the IPC queue outside of MROS currently). In the test, we assume that the number of requester (user) threads equals to the number of receiver (server) threads, but it does not have to be the case in general for our coalesced IPC queue design. We observe that 64 queues largely mitigate all contention on 128 parallel threads, while 32 coalesced queues still provide fairly good performance, comparable to that of Linux. Our general observation is that the total number of cores should be 4x of the total number of coalesced queues, which is fairly reasonable in practice.

5 FUTURE WORK

We plan to implement a full-blown microkernel for multicore systems and support user space device drivers as the next step. Rumpkernel [4] is a NetBSD-based library OS which provides libc and libpthread and can run unmodified NetBSD drivers. A version of rumpkernel for seL4 [6] uses it to run unmodified POSIX applications, but has a number of limitations (e.g., no multicore support). We may adopt existing multicore support [24] for our microkernel. Alternatively, we are considering using FreeBSD-based code instead.

Yet another alternative to this would be to build user-space device driver based on DPDK [26] and SPDK [23]. Furthermore, both DPDK and SPDK can be improved with the coalesced queues as the ring buffers in these libraries do not scale on multicore systems that well.

6 RELATED WORK

L4 [13] microkernels have gained particular attention, especially seL4 [10], which is a formally verified microkernel. Sometimes, microkernels are used to build multiserver OSs are a type of microkernel where OS components run in distinct user processes, referred to as system servers, e.g., MINIX 3 [8], GNU Hurd [5], Mach-US [25], and SawMill [7]. Microkernels can also isolate entire OSs as in case of L4Linux.

Though seL4 [10] goes to great length to optimize IPCs, it still needs mode and context switches in general. seL4 [10] uses a so-called fastpath IPC, which optimizes IPC performance by using registers for short IPC messages. The main drawback of this approach is it puts constraints on the message size and does not work across multiple cores, as each core holds its own register sets.

SkyBridge [17] reduces address space switch by leveraging EPTP (extended page table pointer) switching via VMFUNC, a specialized Intel EPT feature. Since VMFUNC is a VM rather than process feature, SkyBridge [17] inserts a small hypervisor underneath the microkernel, mapping both client and server page directories into the same EPT. Although SkyBridge claims to reduce IPC overheads, it inevitably introduces the cost of virtualization, including indirect costs which are harder to measure. The approach makes it impossible to run the OS in a VM without resorting to nested virtualization and also lacks broader architecture support (e.g., AMD x86-64 or ARM64 processors).

RedLeaf [18] is a novel microkernel design. Instead of traditional hardware-based features such as page-table isolation, RedLeaf extends the Rust ownership system to implement isolation completely at the software level. Though RedLeaf avoids costly context switches, its isolation mechanism has a major drawback – it lacks the ability to support critical system calls, e.g., fork(2). This drawback makes it only support a subset of POSIX interfaces excluding many system calls that involve address-space manipulation. Unfortunately, popular applications are built on this standard process interface. It also severely limits how applications should be written, which might not be feasible in many cases.

Scheduler activations [3] provide each application with an in-kernel execution proxy. The scheduler activations are responsible for handling events such as thread execution and making requests to the kernel. Scheduler activations allow preempting cores during blocking kernel calls but lack any mechanism for thread migration. It was reported [2] that task migration is increasingly common in NUMA systems with the prevalence of virtual machines and containers.

Arachne [21] allows applications to use their own core policy. Arachne is especially useful for very short-lived threads. While we share a similar goal of low-latency non-traditional thread scheduling, we assume the connection between threads and servers is long-lived, e.g., when a database process connects to a file system server, and have a different set of trade-offs.

Hardware vendors such as Intel introduced user interrupts (UINTR) in Sapphire Rapids processors [1], offering more possibility to offload kernel-grade functionality to user space [9]. These approaches, while not universally available, can be complementary to the proposed MROS design.

7 CONCLUSIONS

In this paper, we propose MROS, a secure and scalable microkernel design. MROS guarantees memory security with Rust and proposes to co-design IPC mechanism with its thread model to improve scalability. The preliminary evaluation shows that MROS has the potential to outperform current state-of-the-art microkernels and provide comparable performance to monolithic kernels. When using special optimizations such as coalesced queues, the user-space IPC is capable of retaining good performance even in the presence of highly-contended IPC requests to the same server.

Our current prototype already implements basic memory management, system calls, and scheduling. We will be extending the system to support real-life applications and address a number of challenges, e.g., multicore support and device drivers, as part of our future work.

REFERENCES

[1] 2023. Intel 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/.

[2] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 283–300. https://doi.org/10.1145/3373376.3378468

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 95–109. https://doi.org/10.1145/121132.121151

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

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

[6] Kevin Elphinstone, Amirreza Zarrabi, Kent Mcleod, and Gernot Heiser. 2017. 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* (Mumbai, India) *(APSys'17)*. Article 11, 8 pages.

[7] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. 2000. The SawMill Multiserver Approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*. 109–114. http://l4ka.org/publications/

[8] 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.

[9] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. 2024. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 265–279. https://doi.org/10.1145/3694715.3695973

[10] 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.

[11] Yauhen Klimiankou. 2021. Micro-CLK: returning to the asynchronicity with communication-less microkernel. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 106–114. https://doi.org/10.1145/3476886.3477521

[12] Jochen Liedtke. 1993. Improving IPC by kernel design. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 175–188. https://doi.org/10.1145/173668.168633

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

[14] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[15] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 819–835. https://doi.org/10.1145/3477132.3483581

[16] 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.

[17] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 15 pages. https://doi.org/10.1145/3302424.3303946

[18] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[19] 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. https://doi.org/10.4230/LIPIcs.DISC.2019.28

[20] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a Microkernel, a Big Lock Is Fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (Tokyo, Japan) *(APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. https://doi.org/10.1145/2797022.2797042

[21] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[22] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 170–185. https://doi.org/10.1145/319151.319163

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

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

[25] 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.

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