Fast Core Scheduling with Userspace Process Abstraction

Jiazhen Lin*† Tsinghua University
Youmin Chen* Tsinghua University
Shiwei Gao Tsinghua University
Youyou Lu‡ Tsinghua University

Abstract

We introduce uProcess, a pure userspace process abstraction that enables CPU cores to be rescheduled among applications at sub-microsecond timescale without trapping into the kernel. We achieve this by constructing a special privileged mode in userspace to achieve safe and efficient separation among uProcesses. The core idea is a careful combination of two emerging hardware features – userspace interrupts (`Uintr`) and memory protection keys (MPK). We materialize the uProcess abstraction by implementing Vessel, a userspace core scheduler that colocates latency-critical and best-effort applications with minimal switching overhead when they time-share CPU cores. Our experiment result shows that Vessel exhibits better overall performance and low latency when multiple applications are colocated.

CCS Concepts: • **Software and its engineering** → **Scheduling**; • **Computer systems organization**;

Keywords: Core Scheduling, Userspace Interrupt, Process Management

ACM Reference Format:
Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. 2024. Fast Core Scheduling with Userspace Process Abstraction. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3694715.3695976

1 Introduction

Currently, 100-200 Gbps networks are widely adopted by mainstream cloud providers [6, 18, 30, 37]; NVDIA BlueField-3 Data Processing Unit (DPU) even enables connectivity at up to 400 Gbps with sub-microsecond latencies [3]. Storage vendors are also further exploring the performance limit, and new products such as Optane memory [1], Z-NAND [5] and memory-semantic SSDs [2] can easily deliver more than one million operations per second (IOPS). However, the CPU performance has remained comparatively stagnant as Moore's law slows and is becoming the major bottleneck.

To save CPU resources, recent systems (e.g., Pangu [37], Snap [34], FaRM [14, 15], MICA [31], RAMCloud [40], among others) employ kernel-bypassing to eliminate the kernel overhead and realize bare-metal hardware performance. They map hardware devices into userspace and extensively use the *run-to-completion* (`RTC`) model to run the application logic. In the meantime, cloud applications (e.g., microservices, serverless, etc.) are typically packed into *shared* clusters (i.e., workload consolidation), enabling CPU cycles left unused by latency-critical applications (`L-apps`) to be harvested by best-effort applications (`B-apps`). In large-scale clouds, increasing utilization by a few percentage points can save up to millions of dollars [50].

However, kernel-bypassing and workload consolidation are hard to reconcile with the existing operating system design. Kernel-bypassing systems' `RTC` model heavily relies on core binding instead of sharing, busy-polling instead of interrupt, so physical CPU cores are not shared among different applications and a single machine can only run very few applications. Forcibly colocating busy-spinning threads on the same core can lead to extremely high latencies due to the millisecond-scale timeslice used by existing kernel scheduler [25]. Even worse, the offered load of `L-apps` typically follows a bursty arrival pattern that jitters not only over diurnally or seasonally long timescales, but also over μs-scale short intervals [39, 52]. To keep latency low, `L-apps` must reserve enough idle CPU cores all the time to handle bursty requests, which, however, is exactly contradictory to our desire for saving CPU resources.

Several recent systems like Shenango [39], Caladan [17, 35] and Arachne [42] use userspace core schedulers to colocate applications on the same machine. Specifically, the userspace scheduler grants each application a dedicated set of cores, performs load balancing across cores within an application, and does *fine-grained* core reallocation between applications. These systems achieve impressive performance compared with the Linux scheduler. However, when serving short tasks with an average service time of 1 - 2 μs (which is typical for datacenter applications like Memcached [9] and Redis [4]), or densely colocating applications, they still sacrifice significant CPU efficiency. As we will show in §2.1, colocating an `L-app` with a `B-app` with Caladan degrades performance by up to 18% compared with running them alone, while colocating multiple `L-apps` on the same core amplifies the latency by multiple times [25, 27]. The root cause is that these systems rely on the standard Linux process to run applications. When switching a core between two applications, the CPU core

*Both authors contributed equally to this research.
†Tsinghua University and BNRist.
‡Corresponding Author: Youyou Lu (luyouyou@tsinghua.edu.cn)

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695976

needs to trap into the Linux kernel to switch the address space. In Caladan, for example, switching a core involves one inter-core communication and four user-kernel crossings. With bursty load or dense colocation, core reallocation happens frequently and wastes CPU cycles.

In this paper, we ask a fundamental question: *is it possible to let the CPU core switch between processes directly in userspace without violating the original semantics of the kernel process?* We show this is possible by introducing a new process abstraction named uProcess. uProcesses are rearchitected to run different applications within a single shared memory address space; in this way, a CPU core can switch to another uProcess by issuing jump instructions directly. While the idea sounds intuitive, aligning uProcess semantically with the standard Linux process abstraction is still challenging. First, Linux processes enforce resource isolation by strictly separating their address spaces, while uProcesses share the address space. This is problematic since malicious or buggy uProcesses can access and modify the whole address space arbitrarily. Hence, we require an efficient and safe way to allow uProcesses to both share and isolate their address spaces. Second, for fair scheduling, preemption is a must [17, 28] to avoid head of line blocking: threads running within a kernel process can be preempted flexibly through hardware timers or inter-processor interrupts (`IPIs`) to avoid their long-term occupation of cores. However, there are no such existing mechanisms for a thread to be preempted in userspace without trapping into the kernel.

We address the first challenge by introducing a user-level *privileged mode* to enable safe address space sharing. By default, different uProcesses' memory segments (i.e., text, data, stack, etc.) are protected by different memory protection keys (MPK). Switching between uProcesses requires a core to 'trap' into such a *privileged mode* through a carefully crafted call gate, which functions similarly as the boundary between the user and kernel space of the Linux kernel. To tackle the second challenge, we turn to an entirely new hardware mechanism named userspace interrupt (`Uintr`), which allows delivering interrupts directly to user-level processes. Therefore, a uProcess thread can either park itself actively or be preempted passively by `Uintr` signals and thus enables flexible scheduling policies. Based on the uProcess abstraction, we build Vessel, a pure userspace core scheduling system. Traditionally, load-balancing tasks among threads within an application (e.g., work stealing) incur much lower overhead than reallocating cores across applications, so existing scheduling policies tend to be *conservative* – a core is reallocated only after it fails to steal any work within the current application. Given the fast core reallocation performance of uProcesses, such a two-level scheduling policy is not necessary; instead, Vessel assigns (preempts) cores to (from) applications by maintaining a *global view* of CPU resources, and enforces a one-level scheduling policy, which is approaching better CPU efficiency.

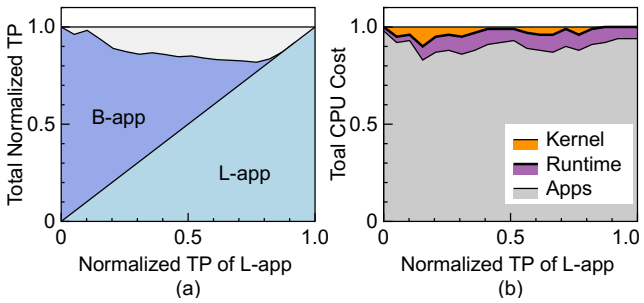

Figure 1. Cost of application colocation.

We compare Vessel with Caladan [17] and other systems under a variety of setups. Vessel can be used to colocate `L-apps` and `B-apps`, densely colocate applications, and regulate memory bandwidth usage accurately between memory-intensive workloads. Our contributions are summarized as follows.

1) We empirically study the CPU inefficiency of workload consolidation with existing core schedulers, with an in-depth analysis of the core reallocation overhead.
2) We introduce uProcess, a userspace process abstraction that enables fast core reallocation without trapping into the kernel via safe and efficient address space sharing.
3) With uProcess, we build Vessel, a userspace core scheduler to further explore innovations in scheduling policies.
4) Our experimental results reveal that Vessel enhances CPU efficiency when context switches happen frequently.

2 Motivation and Background

In this section, we empirically analyze the overhead of application colocation and introduce important hardware features we used when mitigating such overhead.

2.1 Cost of Application Colocation

We first investigate the cost of application colocation. We perform this experiment with Caladan [17, 35], the state-of-the-art CPU scheduler that allows applications to allocate their dedicated resources and do *fine-grained* resource reallocation between applications at μs timescale. We deploy Caladan with the optimized scheduling policies (i.e., Delay Range) enabled [35]. We demonstrate the overhead by colocating an `L-app`, memcached [9], with a `B-app` that does scientific computation [13] on one server. We measure their performance and consumed CPU cycles by varying the load of `L-apps`. For Memcached, the clients running on separate machines issue requests to the server following a Poisson arrival process (detailed experimental setup is given in §6.1).

Intuitively, an ideal CPU scheduler should ensure that `L-apps` always have sufficient CPU cycles, and any *unused* CPU cycles of `L-apps` should be reallocated to `B-apps` immediately, where the reallocation itself causes zero overhead. Hence, two applications running with an ideal scheduler

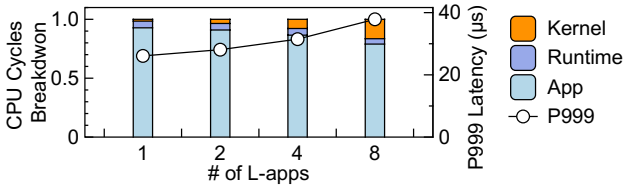

Figure 2. Cost of dense colocation. `L-app`s *run on a single core.*

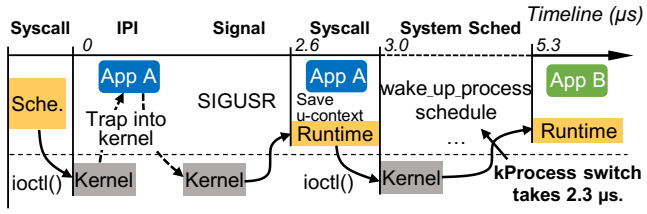

Figure 3. The timeline of core reallocation with Caladan.

should achieve a total normalized throughput[1] that remains to be 1 as the load of the `L-app` varies. Unfortunately, the performance of Caladan still has a nonnegligible gap from the ideal one. As sketched in Figure 1a, the performance of co-located applications running with Caladan exhibits a decline of 18% at most in terms of the total normalized throughput as the load of the `L-app` increases.

Further, we collect the number of CPU cores each application actually consumes as the load of the `L-app` varies (shown in Figure 1b). We find that up to 17% of CPU cycles are not spent on executing the application logic, which are instead spent in the kernel and runtime. Even at the peak throughput, there are still CPU cycles spent on the runtime. We also perform the same experiment by colocating multiple `L-apps` on a single CPU core (Figure 2), and observe that as the number of colocated applications increases, the CPU cycles spent in the kernel increase as well.

Caladan's scheduler features a two-level approach: an idle core first steals workload from other cores within the same application before being reallocated to other applications. The reason lies in that reallocating cores across applications is time-consuming, despite Caladan introduces mechanisms to accelerate such operations. Figure 3 depicts the whole timeline of Caladan to perform a core reallocation. To bind a new application task (`App-B`) on a CPU core, the scheduler first issues an `IPI` (inter-processor interrupt) to the victim core via an `ioctl` syscall to preempt its current task (e.g., `App-A`). On receiving the signal, the core traps into the kernel, and issues a `SIGUSR` to `App-A`, enabling the userspace runtime to save the current state; then the core in the kernel switches to `App-B` task by updating related kernel data structures, switching page tables, and finally restoring to `App-B`. Completing the entire operation takes an average of 5.3 μs, during which the target CPU core is unable to run the application. Hence, Caladan reallocates a core only after it fails to find any task to process for a certain period; such a conservative policy again wastes CPU cycles in the runtime.

2.2 User-space Interrupt

IPI is a widely used hardware feature for direct communications between CPU cores. Due to its preemption-based nature, IPIs must be invoked and received in kernel space, which inevitably incurs high overhead. Recently, `Uintr` is starting to be supported in the 4th generation Intel Xeon Scalable CPUs [11]. RISC-V privileged architecture also introduces the "N" extension for user-level interrupts [51]. `Uintr` enables two kernel threads (sender and receiver, respectively) to create a channel with each other and send and receive interrupts directly in userspace, achieving up to 15× lower latencies than IPI-based signals [36]. Specifically, a `Uintr` receiver CPU core holds a User Posted Interrupt Descriptor (UPID) that stores receiver interrupt vector information and notification state; each `Uintr` sender CPU core holds a User Interrupt Target Table (UITT) that stores UPID pointers and vector information for routing interrupts. When a sender issues a `senduipi<index>`, the hardware refers to its UITT table entry pointed by the `<index>` and posts the interrupt vector into the receiver's UPID. If the receiver is running, the sender CPU core sends a physical IPI to it. On the receiver side, this IPI is detected as a user interrupt, and the hardware pushes the vector number onto the stack and directly invokes the receiver's user interrupt handler in the userspace without kernel trap. If the receiver is in the kernel space or has been context-switched out, the interrupt delivery is deferred until the receiver is active again. To transfer the control back to the interrupted instruction, the `uiret` instruction should be executed by the receiver CPU, and the interrupted context will be restored by hardware with the information on the stack.

2.3 Memory Protection Key

A common way to achieve memory access control relies on syscalls like `mprotect()` in Linux, which modifies the permission bits in the corresponding page table entries but incurs high latencies. Intel memory protection key (MPK) is an extension to enforce memory access control in userspace. It leverages the 4-bit reserved space in each page table entry to separate the memory space of a process into at most 16 regions and adds a new register named `PKRU` to each CPU core to restrict memory access to these regions. `PKRU` contains 16 pairs of 2-bit permissions, each of which specifies whether the related region is read-only, read-write, or inaccessible. A core can manipulate its `PKRU` via non-privileged instructions (e.g., `WRPKRU` and `XRSTOR`) to change its access rights to these regions. For example, `WRPKRU` writes the value of the operand register into `PKRU` to control the permissions. This instruction endures low latency[49], requiring only 11 to 260

[1] Defined as: $T^{cur}_{B\text{-app}}/T^{max}_{B\text{-app}} + T^{cur}_{L\text{-app}}/T^{max}_{L\text{-app}}$, where T^{max}_{*} is the maximum throughput each application can achieve when running alone on all cores.

cycles to finish. Linux kernel also provides APIs to use MPK, among them, `pkey_alloc()` enables the usage of certain keys; `pkey_mprotect()` binds a memory area with the given key.

3 Challenges and Our Approach

Our overarching goal is to optimize CPU efficiency by minimizing the core reallocation overhead when colocating applications. Achieving low-latency core reallocation is challenging since all existing approaches, even the state-of-the-art (e.g., Caladan [17, 35], Shenango [39]), need to trap into the kernel to accomplish privileged context switches and other tasks, which incurs high latencies and wastes CPU cycles.

To this end, we explore a new core reallocation mechanism that can be done purely in *userspace*; we make this happen via a novel process abstraction named uProcess, which shares the same address space with other uProcesses directly; in this way, a core running within a uProcess can be reallocated to another uProcess by jumping to there directly. However, the abstraction of uProcess violates basic semantics (e.g., isolation of process-level resources and flexible scheduling) of original Linux processes and requires careful treatment.

Efficient protection of the shared address space. uProcesses require their address spaces to be shared to enable userspace direct core reallocation, eliminating the need to switch page tables in a privileged mode. However, using a shared address space is problematic since a uProcess can access the whole space arbitrarily, despite the memory region belonging to other uProcesses. To provide similar guarantees as that of Linux process, we require an efficient and safe way to allow uProcesses to both *share* and *isolate* their address spaces, which is a seemingly contradictory demand for us.

Light-weight preemption-based task scheduling. A standard operating system provides both passive (e.g., interrupts) and active (e.g., traps) mechanisms to enable flexible scheduling of processes. Userspace core schedulers can easily support active approaches for core scheduling; in Caladan, for instance, an application task can be scheduled whenever it parks itself by invoking `yield` functions. However, to achieve preemption-based passive scheduling, the scheduler needs to issue an `IPI` signal in a privileged mode to the victim core, and the victim core then accomplishes the scheduling in the kernel space, which incurs high overhead.

3.1 Our Approach

Figure 4 presents a sketch of uProcess and how it interacts with other components, including a scheduler that makes scheduling decisions, and a manager that receives users' commands and creates and initializes uProcesses. As shown in the figure, multiple uProcesses are grouped together into a scheduling domain, and each application runs in a uProcess; these uProcesses share a configured subset of CPU cores and interact with the same scheduler and manager.

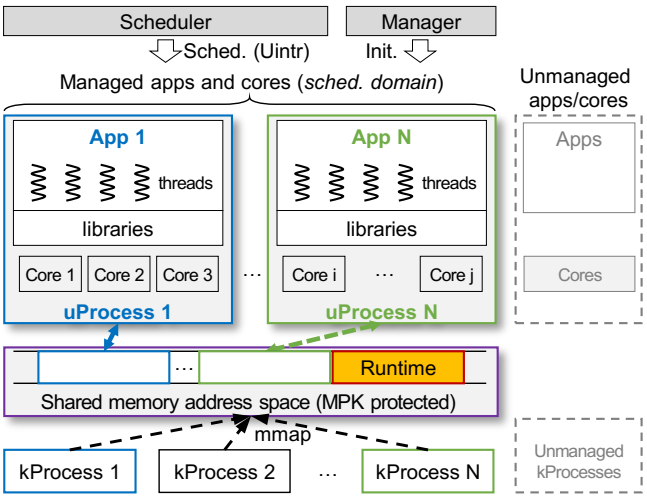

Figure 4. Overall architecture of uProcess.

uProcess is essentially similar to the Linux process (which we refer to as kProcess thereafter); specifically, a uProcess is launched by creating a dedicated kProcess beforehand, within which a number of threads can be created and attached to CPU cores to run application tasks. Despite these commonalities, uProcess is designed with a different address space organization than kProcess (§4.1). Specifically, uProcesses within the same scheduling domain use a *shared memory address space* (`SMAS`) to accommodate their text, data, stack, and other segments. The executable and the libraries for each uProcess are loaded to `SMAS`, and different applications can run inside different uProcesses.

To achieve efficient isolation and sharing of `SMAS`, uProcesses' memory segments are manipulated individually by MPK – by default, a uProcess only has access rights to its owned regions, while others are invisible. uProcess constructs a special privileged mode in userspace, which corresponds to the Linux kernel mode, to enable the safe core switching between uProcesses. Specifically, we place a *runtime region* at the end of `SMAS` to imitate the kernel space; the runtime region contains the runtime code that provides privileged function calls (similar to Linux's syscalls) to uProcesses, and a uProcess must use a carefully crafted *call gate* (§4.2) to 'trap' into such a privileged mode before invoking these privileged runtime operations.

uProcess enables light-weight task preemption with `Uintr`, where a CPU core running with a uProcess can be context switched to other uProcesses directly in userspace. To achieve this, the scheduler first issues a `Uintr` signal to the victim core. Then, the victim core executes a pre-registered userspace interrupt handler and passes through the *call gate* to upgrade its privilege level; here the core can invoke privileged operations and make scheduling decisions. Once decided, its address space is switched to the region accommodating the

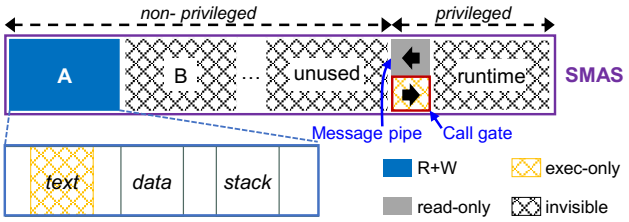

Figure 5. Layout of the shared memory address space (SMAS). *uProcess A's access permissions to* SMAS *are specified as well.*

target uProcess by setting MPKs to be that of the target application. In this way, core reallocation between uProcess eliminates userspace-kernel crossing completely and thus delivers extremely low core reallocation latency.

Moreover, with the uProcess abstraction, the overheads of reallocating cores across applications and load-balancing tasks across cores within an application (e.g., work stealing) become similar. With this insight, we implement Vessel, an efficient core scheduling system that enables flexible scheduling of CPU core from a global scope (§5).

Vessel is designed to coexist inside an unmodified Linux environment; the scheduler can be configured to manage a subset of cores running a subset of applications, while the Linux scheduler manages others. Moreover, Vessel also supports managing a subset of threads within an application, leaving others managed by the Linux scheduler.

4 uProcess Abstraction

In this section, we introduce uProcess by describing its core components – the address space organization (§4.1), the call gate design that enables privileged switching (§4.2), signal handling (§4.3), and core reallocation workflow (§4.4).

4.1 Address Space and Protection

Figure 5 exhibits the address space layout of SMAS, created by the manager and shared by all uProcesses within this scheduling domain. Similar to Linux kernel's user and kernel space isolation, SMAS is split into *non-privileged* regions that correspond to uProcesses (named uProcess regions) and *privileged* regions (e.g., runtime and message pipe) for providing system-level functionalities. These memory regions are managed with different protection keys.

uProcess region is either contiguous or fragmented areas to accommodate a uProcess's data, stack, and others except for the text segment (which will be described later). To safely isolate these regions between uProcesses, each uProcess region owns a separate memory protection key, and only the uProcess owning this region has read and write permissions to it. Moreover, we rely on our customized program loader and memory allocator to manage the address space of these regions and correctly install application programs into them (§5.2.1).

Runtime region is a special privileged memory area that stores the runtime's data and is invisible to all uProcesses. The runtime provides a group of privileged operations to uProcesses of the current scheduling domain, such as scheduling primitives, memory allocation, and other system-level services (e.g., storage and networking). When a uProcess needs to invoke these privileged operations, it must use the call gate (a piece of code framed by a red rectangle in Figure 5) to enter a privileged mode. In a privileged mode, the CPU core can access the whole address space of SMAS.

Text region. The text segments of all uProcesses, call gate, and the runtime are `executable-only` (neither readable nor writable) since we do not want applications to modify their binary code dynamically to issue illegal instructions. Specifically, the text segment of a uProcess shares the same protection key as that of the corresponding uProcess region; meanwhile, the permission bits in the page table entries of the text segment are set to `executable-only`. This is practicable since MPK is supplementary to the existing page permission bits and both permissions will be checked during memory access [12].

Note that `executable-only` text segments can be executed by arbitrary uProcesses; however, this is both necessary and safe. On one hand, sharing the text region allows non-privileged uProcesses to invoke the call gate (§4.2) directly. On the other hand, when an application improperly jumps to the text segments of other uProcesses or the runtime without using the call gate, it still does not have permission to access their data (e.g., stack, global variables, etc.), and any execution of illegal load/store instructions will be terminated by MPK.

Yet, sharing text segments raises the following risks. ① Data leakage. Although the data region is secured with the MPK-based mechanism, the constant data stored in the text region, like the immediate operand in an instruction (e.g., `MOV %EXA 0x1234`) inlined by the compiler, may contain confidential information and leak to the context during the execution. To mitigate the leakage, variables can be enforced to be stored in the secured static data segment. ② Unauthorized code execution. Although privileged functionality has been secured, the code execution for unprivileged functionality in a uProcess, like scientific computing, can not be completely prohibited. However, these attacks can be mitigated with address space layout randomization (ASLR).

Message pipe region is a special memory area that provides a unidirectional channel for the runtime to expose necessary information to uProcesses (e.g., the mapping of CPU cores and threads). All uProcesses only have read permissions to it while the runtime can both read and write it.

With the above design, one scheduling domain supports up to 13 uProcesses/applications, where the rest keys are used by the runtime region, and message pipe region, respectively[2]. Multiple scheduling domains can be used when the number of uProcesses exceeds this limit.

[2]We do not use `0` as the protection key to allow kProcess's unmanaged memory space other than SMAS to work as usual.

4.2 Call Gate

Threat Model. uProcess does not trust the application code and the shared and static libraries it depends on. The OS kernel, hardware, and the runtime, scheduler, and manager of uProcess are trusted. We focus on attacks where a malicious application tries to upgrade its privilege illegally (e.g., control-flow hijack attacks [49]). Attacks on the security vulnerabilities of hardware and OS are beyond the scope of this paper.

Goal. uProcesses must 'trap' into the privileged mode to invoke privileged runtime operations. Hence, the call gate should be designed to allow non-privileged uProcesses to switch into the runtime in a legal way. The connotation of legality here is: *as long as a uProcess is in privileged mode, it must be executing the **trusted** runtime code*.

Existing approaches. To achieve the above goal, a basic execution flow of a call gate works like this (see Listing 1): set the PKRU to attain access rights to the whole SMAS (Line 3), execute the privileged operation (Line 7), and set it back to an application's permission at return (Lines 10-13). However, a malicious application can easily bypass such an execution flow in a number of ways. One approach is using WRPKRU instructions directly in application codes to change PKRU. The other is control-flow hijack attack: WRPKRU uses the eax register as its input, so a malicious application can modify eax arbitrarily and jump directly to the WRPKRU instruction at Line 13 to upgrade its privilege.

```
1  // Enter call gate
2  // Stage 1: Set PKRU to RUNTIME_KEY
3  wrpkru(RUNTIME_KEY);
4  // Stage 2: Switch stack and call runtime function
5  MOV CPUID_TO_TASK_MAP[get_cpuid()].RSP $RSP;
6  MOV $RSP CPUID_TO_RUNTIME_MAP[get_cpuid()].RSP;
7  ((func_t) (FUNC_ADDRESS)) ();
8  MOV $RSP CPUID_TO_TASK_MAP[get_cpuid()].RSP;
9  // Stage 3: Set PKRU to user app's key
10 reset_pkru:
11 userapp_pkru =
12     CPUID_TO_TASK_MAP[get_cpuid()].PKRU;
13 wrpkru(userapp_pkru);
14 // Stage 4: Check PKRU
15 userapp_pkru_check =
16     CPUID_TO_TASK_MAP[get_cpuid()].PKRU;
17 cur_pkru = rdpkru();
18 // Reset, if inconsistent (control-flow hijack)
19 if (cur_pkru != userapp_pkru_check)
20     goto reset_pkru;
21 // Leave call gate
```

Listing 1. uProcess call gate in pseudo-code.

Both Hodor [22] and ERIM [49] are MPK-based software sandboxes to address the above problems. At a high level, both systems use code inspection techniques to neutralize WRPKRU instructions in application and library code except those used in the call gate. They also monitor mmap and mprotect syscalls issued by applications to prevent untrusted components from introducing new executable code that contains illegal WRPKRU instructions. Both systems also prevent control-flow hijack attacks. After restoring the PKRU register at Line 13, the call gate cannot simply return to the application code; instead, the call gate needs to recheck if the PKRU has been updated correctly (Lines 15 - 20). Any attack that jumps directly to line 13 with any other value in eax would fail the check and jump back (Lines 19 - 20).

Our approach. Despite the above efforts, however, both systems are still vulnerable to a variety of software attacks [10]. To complement this, we further introduce the following mechanisms to prevent potential vulnerabilities.

First, both ERIM and Hodor allow safe pages (i.e., no illegal WRPKRU instructions) to be dynamically mapped executable but unwritable. This, however, leaves an attack surface for malicious applications to subvert memory permissions or change code by relocation through a number of ways [10]. In our design, all mmap, mprotect, and other memory configuration syscalls that intend to make a memory page executable are intercepted and prohibited. On-demand program loading and other operations based on these syscalls can be achieved by invoking the call gates of uProcess runtime with the corresponding functionality.

Second, the WRPKRU instruction at Line 3 is immediately followed by a runtime function call on Line 7, which corresponds to a privileged operation invoked by a uProcess. However, ordinary function calls to privileged operations raise security risks here. The non-static functions of a user's shared library are commonly called via the PLT (Procedure Linkage Table). Thus, a malicious application can change the PLT entry to point to a self-defined function; along with the control-flow hijack attack, this application can execute any code in privileged mode. To address this issue, we use a static function pointer vector kept in the message pipe region (read-only to uProcesses) to point to runtime functions; in other words, any runtime function that can be invoked by uProcesses should own an entry in the function pointer vector. In this way, the call gate avoids the use of PLT by referring to function addresses directly (a direct control transfer).

Third, to invoke a runtime function call, the return address is pushed to the top of the stack. Hence, other threads within the same uProcess can easily modify the return address to another value. Once the caller thread executes a ret, it can execute arbitrary code in a privileged mode instead of jumping back to the next line within the call gate. We address this issue by switching the caller thread's stack to a stack segment within the runtime region before invoking the runtime function and switching it back after the return (Lines 5, 6, and 8). Here, CPUID_TO_RUNTIME_MAP maps CPU cores to their stack segments within the runtime region; CPUID_TO_TASK_MAP maps CPU cores to their running tasks, which corresponds to the contexts of their currently running threads. Both maps are stored in the message pipe region.

Recall that the text segment of the call gate is executable-only and can be invoked directly. Moreover, the call gate never accesses data that a uProcess does not have access rights to, which ensures that the calling uProcess can legally pass through the call gate without being terminated by MPK.

4.3 Signal Handling

uProcess handles both `Uintr`- and kernel-initiated signals.

The scheduler uses `Uintr` signals for uProcess scheduling (i.e., preempts a uProcess and reschedules it with a new task). To handle `Uintr` signals, the runtime first registers a handler via the `uintr_register_handler()` syscall and creates lock-free FIFO queues for all CPU cores with the scheduler within the current scheduling domain. When the scheduler performs scheduling on a victim core, it first pushes a command into the corresponding queue to describe the concrete scheduling action (i.e., which new thread to run with) and then issues a `Uintr` signal to this core. On receiving the signal, the corresponding handler is invoked on this core, which further passes through the call gate to upgrade its privilege level, and executes the actual handling function within the runtime. Here, the core pops the scheduling command, parses it, and performs scheduling.

uProcess handles kernel-initiatied signals as usual, but there are some special cases to consider. With our current design, uProcesses can interchangeably run within different kProcesses. Hence, some faulty events (e.g., memory corruption) of one uProcess may cause unexpected termination of other kProcesses (i.e., a bigger blast radius). It's essential to build a barrier between kProcesses to shield such faults. To achieve this, we let the runtime also register fault signal (e.g., segmentation fault) handlers in advance before loading uProcesses, and handle kernel-initiated signals similarly to that of `Uintr` signals. For example, when a memory fault occurs, the signal handler receives the signal, invokes the call gate, and 'traps' into the runtime for signal handling. The handler first identifies the current faulty uProcess via the `CPUID_TO_TASK_MAP`, and broadcasts this signal to all CPU cores running inside this uProcess. Here, the handling function only needs to push the signal into FIFO queues of all related cores, instead of sending `Uintrs` to them; when these cores enter a privileged mode due to a future scheduling event, such signals will finally be processed, and the uProcess is terminated.

4.4 Context Switch

The runtime provides two types of primitives to support efficient core scheduling. First, using the call gate, a thread can `park` itself actively whenever it fails to find new requests or needs to be blocked (e.g., acquiring a lock, waiting for a response, etc.). Second, we also leverage `Uintr` to achieve preemption-based core scheduling. Preemption happens when a high-priority task is blocked by a low-priority one. Both the two primitives incur context switches of a CPU core between different uProcesses.

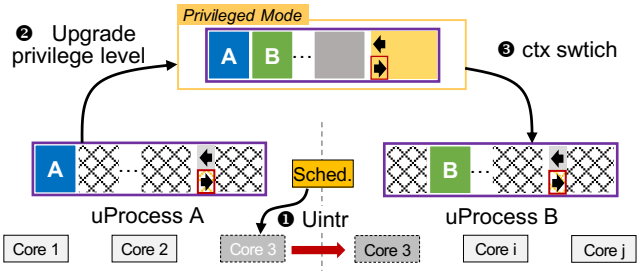

Figure 6. State transition of the address space when preempting a core. *The scheduler issues a* `Uintr` *to core 3 (❶); the core then enters into the privileged mode via the call gate (❷); and switches its address space to uProcess B's (❸).*

Figure 6 shows how a core is preempted and switched to run another uProcess's thread. Once the scheduler decides which victim core to preempt, it sends a `Uintr` signal to this core (❶). The victim core handles this `Uintr` signal within the runtime function in a privileged mode (❷), where the core first saves its context to the task queue, including necessary registers and the return address (the next to be executed instruction after returning from this function, i.e., Line 7 in Listing 1). Then, the runtime function pops a new thread from its task queue for execution. Here the new task corresponds to a pending thread belonging to uProcess B, so `CPUID_TO_TASK_MAP` should be updated as well to reflect the new mapping between this CPU core and and the new thread. At this stage, the core can restore the target thread's context and jump to the last saved return address (❸), which is exactly at Line 7 in Listing 1. From here on, the core's RSP register is restored to point to the new thread's stack; then, the core resets its `PKRU` to change its access permission to `SMAS` to be identical to uProcess B's.

4.5 Scheduler

With a fast path of userspace context switch, load balancing within an application and core reallocation among applications incur similar latencies. Benefiting from this, the uProcess runtime can adopt a *one-level* scheduling policy to schedule global CPU resources at a finer granularity.

As shown in Figure 7b, the runtime maintains per-core FIFO queues to track the threads running on each core. Since CPU cores can be switched to arbitrary applications with low overhead, they are not affiliated to a certain application; instead, the FIFO queue of a core contains threads of different applications. The scheduler is responsible for balancing the load of all cores within the current scheduling domain. We largely reuse Caladan's metrics (e.g., queuing delay, processing time, memory bandwidth, cache miss, etc.) for determining whether a core is overloaded.

The scheduler scans these FIFO queues repeatedly to find overloaded cores and dispatch their tasks to other cores. At each iteration, the scheduler performs the following steps. First, for each overloaded core, the scheduler preempts it if the

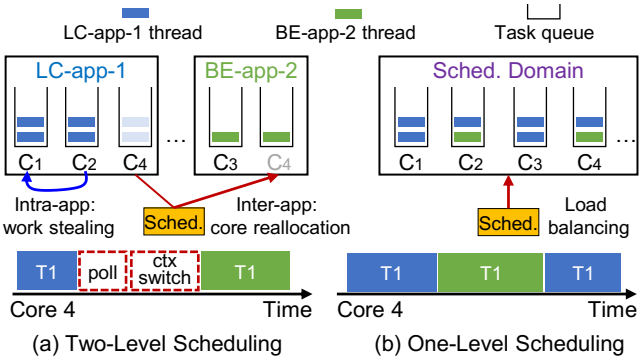

Figure 7. The uProcess scheduler.

core is executing a BE thread; otherwise, pops a thread from its FIFO queue. Second, the scheduler reassigns these threads to underloaded cores. Note that the preempted BE threads are put back into a global BE task queue. For each core, if the running thread parks itself or is preempted, it performs a context switch by suspending the current thread (i.e., saves its context and pushes it back to the FIFO queue) and popping a new thread from its FIFO queue for execution. When the core does not have active threads to execute (e.g., the FIFO queue is empty or no new request comes), it pops a BE thread from the global queue for execution. If the global BE task queue is empty, the core notifies the scheduler and enters an idle mode using UMWAIT[3].

Compared with the uProcess runtime, Caladan's scheduling policy is more conservative (see Figure 7a). In Caladan, an idle core needs to keep stealing tasks within an application for at least 2μs before parking itself, leaving fewer opportunities for it to be rescheduled to other applications, Moreover, Caladan reallocates cores among applications every 10μs since reallocation is expensive. Instead, the runtime does not have such restrictions and enables flexible scheduling of CPU core from a global scope. The lower side of Figure 7 also compares the execution timelines of a core under the two schedulers, we can observe that the uProcess's scheduler can fill the core with the applications' workloads.

5 Vessel Implementation

Vessel is a materialized implementation of the uProcess abstraction. We currently implement Vessel to support glibc 2.34/31/27 and Linux 5.16.15. The Uintr patch provided by Intel has not been merged into the mainstream kernel release, so we modify the Linux kernel to accommodate this patch. In addition to this, Vessel does not require any other changes to the Linux kernel, In this section, we describe Vessel's core components and how they work together to enable applications to run inside uProcesses and be flexibly scheduled (see Figure 8).

[3]A new feature supported by Intel's 4th Gen Xeon processors that can enter a light-weight power state while monitoring a range of addresses.

5.1 Vessel Manager

Vessel's manager is a standalone auxiliary program that processes users' commands for creating and destroying uProcesses. When the manager is launched, it first creates SMAS using the mmap() syscall and is responsible for managing the address space of SMAS. In the runtime region, the manager creates FIFO queues for every managed CPU core in the current scheduling domain.

Upon receiving a uProcess creation command, the manager first creates a kProcess with the fork() and binds it to a specified CPU core. This kProcess initially runs a special booting program, which maps SMAS to its own virtual address space and begins to busy-poll new commands from its own FIFO queue. Meanwhile, the manager allocates a uProcess region for this newly created kProcess, and uses pkey_mprotect() and mprotect() syscalls to associate this region with proper memory protection keys and permission bits. Then it sends an init command to the kProcess containing necessary information (e.g., the starting address of the uProcess region, the path name of the application's executable file, etc.). When the kProcess receives the init command from the queue, it invokes the program loader within the runtime to install and execute the real program (§5.2.1). When a user needs to destroy an uProcess, the manager simply sends a kill command to all CPU cores running inside this uProcess. These cores will finally terminate the uProcess when they see the command in a privileged mode due to a future scheduling event; meanwhile, system resources, such as the uProcess region and CPU cores, are returned back to the manager.

5.2 Vessel Runtime

The runtime in privileged mode contains a number of important functionalities (see Figure 8), including the program loader that installs a program's segments to proper locations (§5.2.1), a customized memory allocator that manages the uProcess's heap space (§5.2.3), and dataplane for accessing network and storage (§5.2.4). Signal handlers have already been illustrated in §4.3.

5.2.1 Program Loader. The program loader is responsible for replacing a kProcess's executable file from the original booting program to the real application program (i.e., similar to the exec() syscall). A standard UNIX-like loader works with the following steps, including validation of permissions and memory requirements, memory-mapping the executable files into main memory following the ELF format, copying the command-line arguments into memory, initializing registers (e.g., RSP), and finally jumping to the program entry point.

In Vessel, we make three modifications to the standard steps. First, in the validation phase, we perform static code inspection to exclude any illegal use of WRPKRU instructions. Second, before jumping to the program entry point, we should initialize the PKRU register via the call gate to enforce memory protection. Third, we make minor modifications to the dynamic

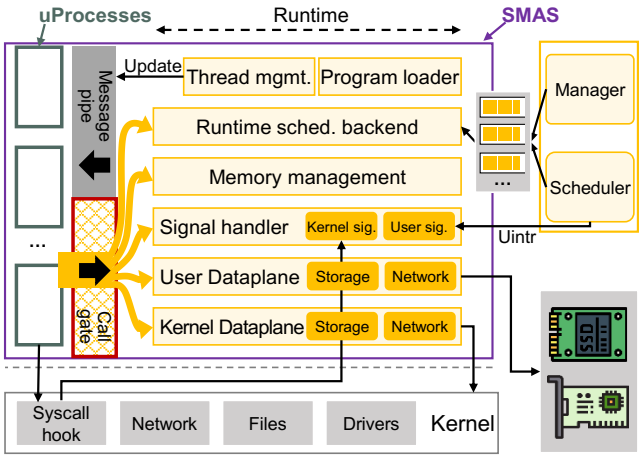

Figure 8. The architecture of VESSEL and basic functionalities that the runtime supports.

linker (i.e., `ld-linux.so*`), which originally uses `mmap()` syscall to load the shared libraries needed by a program. Since the `SMAS` space is created via `mmap()`, we cannot map shared libraries within the same address space; instead, we allocate space for them via our memory allocator (§5.2.3). Note that the text segments of the program and shared libraries reside in the text region and should be associated with the proper `executable-only` permission.

5.2.2 Thread Management. The thread implementation of VESSEL is essentially similar to Linux threads. Conceptually, a thread is just a collection of states (registers, stack, thread-local storage, etc.) and a CPU core operating on these states.

VESSEL provides an identical thread abstraction but manages these states completely in userspace. Specifically, when an application invokes a `pthread_create()`, VESSEL allocates a dedicated stack and thread-local storage for this thread; meanwhile, we also create a context structure to track the thread's states. In a scheduling domain, we use `CPUID_TO_TASK_MAP` to map CPU cores to their currently running threads and per-core task queues to keep threads assigned to them. Whenever a core is rescheduled to another thread, the suspended thread's context is saved, while the core is mapped to the new thread's context structure.

5.2.3 Memory Allocator. By sharing `SMAS`, different uProcesses' heaps reside in the same virtual address space, the layout of which is incompatible with `glibc`'s memory allocator. To this end, we use `jemalloc` to manage uProcess's heap space. By default, `jemalloc` uses `mmap()` and `munmap()` syscalls as its final source of memory. We replace these calls to use the uProcess region instead, which is already MPK-protected when the manager creates it. Applications running inside uProcesses are preloaded with `jemalloc`, and `malloc` functions are intercepted and replaced to use `jemalloc`.

5.2.4 Kernel Syscalls. When using uProcesses to run applications, we do not recommend applications using kernel-level file systems and other services. This is because when a CPU core traps into the kernel, VESSEL will lose control of it, which incurs high latency and degraded performance. Moreover, VESSEL allows uProcesses to be scheduled within arbitrary kProcesses, while the Linux kernel is completely unaware of the uProcess abstraction, which may even raise correctness and security issues. For example, both uProcess A and B are scheduled to run inside kProcess A, and uProcess A creates a file; uProcess B can open this file by performing a bruteforce test over different file descriptors since they are essentially part of the same kProcess (security issue). Furthermore, when uProcess A is rescheduled again to run inside kProcess B, it may not have permission to open the previously created file (correctness issue). For safety reasons, VESSEL only allows the trusted runtime to execute syscall on behalf of uProcesses. Specifically, all syscalls are intercepted and redirected to the runtime via the call gate. The runtime creates a map to track the created descriptors of different uProcesses, and uses this map to enforce access control of these syscalls. For the correctness issue, the manager creates kProcesses with the same ACL permission since the runtime has already performed access control.

5.2.5 Userspace Networking and Storage. In addition to supporting OS system calls, VESSEL also provides kernel-bypassing network and storage libraries for uProcesses to communicate with remote nodes and storage devices. We largely reuse Caladan's network dataplane [17] and SPDK. These dataplane libraries are placed in the runtime and we instrument them with `park()` calls to avoid threads running inside uProcesses from occupying CPU cores for too long when they busy-spinning on completion. Moreover, the software queues of these dataplane libraries are also exposed to the scheduler to assist in making scheduling decisions.

5.3 VESSEL and uProcess Semantics and Limitations

By comparing the life cycle of a uProcess and a Linux process, the semantics and limitations of uProcess implemented in VESSEL are sketched as follows.

Creation: New uProcesses are created by a manager while Linux processes are created by the kernel via standard syscalls. For uProcess, the executable should be loaded to the `SMAS`, so the position-dependent executables may cause memory address collision. Thus, uProcess only supports application executables and libraries compiled and linked with `PIE`.

Running: Applications running inside a uProcess execute code similarly to the Linux process, except for handling signals and syscalls. For signals, applications register signal handlers to the uProcess runtime, and the runtime works as a proxy that pre-registers all handlers and redirects signals to the corresponding uProcesses. In uProcess, all syscalls are intercepted and redirected by the runtime and processed

by the kernel. Due to these design choices, we recommend applications running inside uProcess use the system-level services of the runtime instead of kernel syscalls for efficiency.

Termination: A uProcess can be terminated by the manager and external signals (`SIGKILL`/`SIGTERM`). The termination of a specific thread of one uProcess is not directly supported: the threads are managed by the userspace scheduler, and the Linux kernel is unaware of them; one solution is to use the `sigqueue` syscall with extra parameters to specify thread IDs.

Besides, uProcess's semantics differ from Linux process in two aspects – the uProcess `fork` and on-demand loading. For the `fork`, the `fork`ed processes should have the same address space layout as their parent process; however, uProcesses share a single address space, and this causes a limitation that the child uProcess cannot coexist in the same scheduling domain because its address conflicts with the parent uProcess. Instead, uProcess clones child uProcesses by creating a new `SMAS` and synchronizing data; this allows the child uProcess to own an identical address space.

For on-demand loading, the uProcess runtime exposes the on-demand loading API like `dlopen()` to load new libraries to `SMAS`. This requires the runtime to render the page both non-writable and non-executable, perform code inspection, and then mark the page executable.

6 Evaluation

Our evaluation demonstrates the benefits and quantifies the overhead of Vessel. Specifically, it asks:

- How does Vessel compare with existing systems?
- Can Vessel maintain low latency and high CPU utilization when densely colocating applications?
- How do the internal techniques of Vessel perform?

6.1 Experimental Setup

Platform. Our server node is equipped with two 32-core Intel 4th Gen Xeon Gold 6430 processors and 256GB of RAM running Fedora 35 with kernel 5.16.15 (patched with Intel's `Uintr` support). We use four client nodes, each of which is equipped with two 12-physical-core Intel Xeon E5-2650 v4 processors and 128GB of RAM running Ubuntu 18.04 with kernel 5.4.0. These server and client nodes are connected with the 100Gbps ConnectX-5 Mellanox network. Similar to Caladan[17], we disable TurboBoost, CPU idle states, CPU frequency scaling, transparent hugepages, and numa-balancing for all the machines. By default, Vessel manages 32 hyperthreads at the server node. For a fair comparison, we also disable CPU mitigation for the Meltdown, MDS, and Spectre vulnerabilities, to reduce context switching costs.

Workloads. We evaluate two `L-apps`. First, *memcached* [9] is a popular in-memory key-value store. We choose Facebook's USR request distribution, which has both reads and writes with a service time of 1μs on average. Second, Silo is a scalable and efficient in-memory OLTP database [48], and we stress it with the TPC-C request pattern. TPC-C has high service time variability (20μs at median and 280μs at 99.9th percentile). All requests generated from the above workloads follow a Poisson arrival distribution. For `B-apps`, we choose both CPU- and memory-intensive workloads. A parallel version of *Linpack* [13] is a float-point benchmark that can approximate how fast a high-performance computer performs when solving scientific computation problems. We also implement *membench*, another `B-app` that continually repeats two phases, memory access and calculation, to simulate the behavior of the current data processing applications (e.g., AI recommendation [38]).

Compared systems. Caladan [17] is a CPU scheduler that reallocates cores between applications every 10μs to improve the CPU efficiency under bursty workloads and owns a more scalable control plane (e.g., IOKernel). We also compare with its optimized version (Caladan-DR-L/H) that uses *Delay Range* to make a tradeoff between CPU efficiency and tail latency. Arachne [42] is a user-level thread management system that can estimate how many cores an application needs and allocate cores between applications. Linux's completely fair scheduler (`CFS`) is a process scheduler that handles CPU resource allocation for executing processes and aims to maximize overall CPU utilization. We configure `L-apps` with a nice value of -19 (highest priority) and `B-apps` with a nice value of 20.

6.2 Resource Efficiency

Then, we demonstrate the resource efficiency of Vessel by first colocating one `L-app` and one `B-app` (§6.2.1). Then, we evaluate the scheduling capability of Vessel under densely colocated workloads (§6.2.2).

6.2.1 Core Reallocation. When colocating applications, we choose Memcached and Silo as the `L-app`, and Linpack as the `B-app`. We report the total normalized throughput, `B-app`'s throughput, and the 99.9th percentile latency of the `L-app` as we increase the load of the `L-app`. The experimental results are shown in Figure 9. We make the following observations.

First, when running Linpack with Memcached (i.e., short request service time), Vessel achieves the highest peak throughput among the evaluated systems. Specifically, at a target 99.9th percentile latency of 50μs, Memcached's throughput is 8.3% higher when running with Vessel compared to running with Caladan. With a target load at 16Mops/s, Vessel's P999 latency is 42.1%, 18.6%, 44.0% lower than that of Caladan, Caladan-DR-L, and Caladan-DR-H, respectively. When running with Linux `CFS` and Arachne, Memcached's P999 latencies spikes to over 10ms, so we only increase the load to 1Mops/s at most for Arachne and 0.3Mops/s for Linux `CFS`.

Second, the total normalized throughput of Memcached and Linpack is almost unchanged when running with Vessel (6.6% decline on average), while Caladan shows a performance

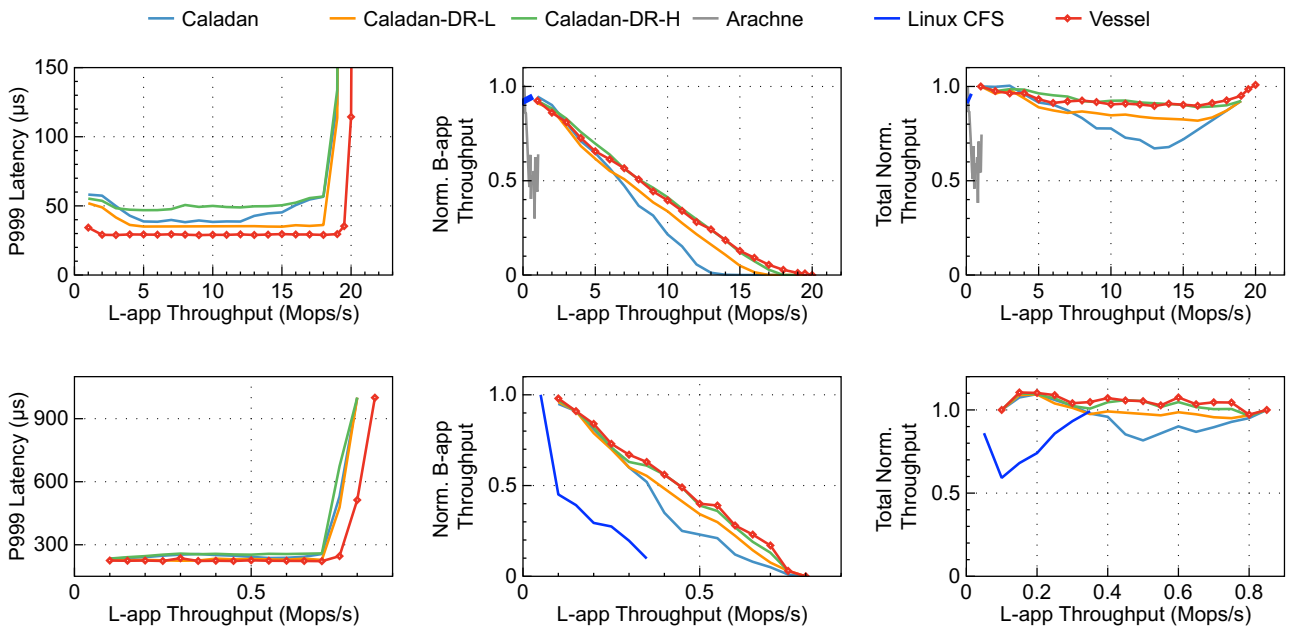

Figure 9. The performance of colocating an `L-app` and a `B-app` (Linpack). *Top: Memcached as the* `L-app`*; bottom: Silo as the* `L-app`*. We report the total normalized throughput,* `B-app`*'s throughput, and the P999 latency of the* `L-app` *as the load of the* `L-app` *increases. In Caladan, DR-L indicates Delay Range 0.5-1µs; DR-H indicates Delay Range 1-4µs.*

decline of 16.1% on average and 32.1% at most. *Delay Range* makes a tradeoff between CPU efficiency and tail latency: with a lower time threshold (i.e., DR-L), Caladan shows good P999 latencies, but it wastes up to 18% CPU cycles; with a higher time threshold (i.e., DR-H), Caladan achieves a similar CPU efficiency as that of Vessel, but exhibits 79% higher P999 latencies. We attribute the performance of Vessel to its lightweight scheduling. On the one hand, when `L-app` parks, it can switch to the next work immediately by scanning local task queues. On the other hand, when the `L-app` is busy, `B-app`'s core can be preempted just in time bypassing the kernel, too. Owing to these lightweight and low-latency scheduling strategies, CPU cycles are better utilized to run applications. The performance of applications running inside Arachne shows a sharp decline (40% on average) within our provided load. Linux `CFS` shows good total throughput given our provided load (from 0 to 0.3 Mops/s) but at the cost of extremely high latencies for the `L-app`. We can observe that Linux `CFS` always grants cores to execute `B-app` despite that `L-app` has a higher priority. This is because Memcached's worker threads suspend CPU cores frequently when they wait for network requests.

Third, when running Linpack with Silo (with service times ranging between 20 - 280μs). Both Caladan and Vessel show good total throughput, which is approaching the ideal case. Since the overhead of processing a Silo request is much higher, core reallocation overhead is amortized and becomes negligible. Linux `CFS` shows much lower total throughput at a low load since the low-priority `B-app` cannot attain enough cores for processing.

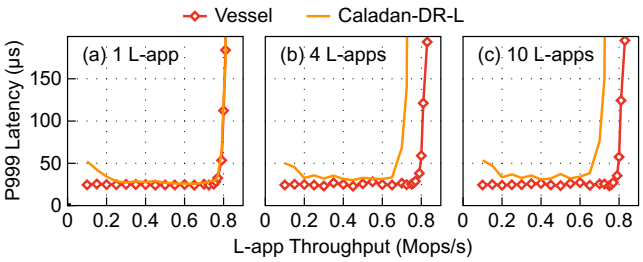

Figure 10. Performance with dense colocation. *Colocating a varying number of Memcached instances on a single core; we report the aggregated throughput and P999 latencies.*

6.2.2 Dense Colocation. We also evaluate Vessel's performance under a dense colocation scenario: we deploy an increasing number of `L-apps` on a single CPU core and measure their aggregated throughput as well as the tail latency as client nodes issue bursty requests. We create 10 client connections per application. In this experiment, we use Memcached as the `L-app` and we only compare with Caladan-DR-L since other systems show orders of magnitude higher latencies than ours. The experimental results are exhibited in Figure 10.

When there is only a single application (core reallocation between applications is not required), both Caladan-DR-L and Vessel show very close peak throughput and tail latency (see the left part of Figure 10). However, as we use 10 Memcached instances, Caladan's peak throughput declines by 25% on average and the P999 latency increases by 20% as the throughput peaks; instead, Vessel's aggregated throughput and tail latencies are almost unchanged. When multiple latency-critical applications are colocated, both Vessel and Caladan need to perform context switches frequently to avoid starved applications from causing long queueing delays. However, this inevitably incurs high overhead for Caladan since reallocation is expensive. In Vessel, core reallocation between applications and load balancing within an application incurs a similar overhead, so it can run 10 applications simultaneously without showing any performance degradation.

6.3 Microbenchmarks

6.3.1 Latency of Context Switch. To measure the latency of context switch, we bind two single-threaded applications on the same core and let them `park()` themselves repeatedly. In this way, the CPU core can switch between the two applications. We instrument two timestamps before and after the `park()` call, which are T_1 and T_2, respectively; then the context switch latency is $(T_2 - T_1)/2$. Table 1 shows the latency results. Caladan requires 2.103 μs on average and 6.461 μs at 99.9th percentile to perform a context switch, while Vessel can achieve sub-microsecond latency. CPU cores managed by Vessel can be efficiently switched between uProcesses since it is achieved via pure function calls, and the core does not need to trap into the kernel. Note that, in Caladan, the latency of core reallocation is lower than that of core preemption (Figure 3). In the latter case, one more time of user-kernel crossing is required to save the application's context.

6.3.2 Cache Friendliness. In this section, we explore the cache thrashing issue and show that Vessel enables two uProcesses multiplexing shared CPU cores to be more cache-friendly. We run two applications on the same core that randomly read and write objects with a uniform distribution. As shown in Figure 11, Vessel reduces the cache miss rate from Caladan's 4.6% to around 0.0415%, and the completion time of Vessel is 6% to 24% lower. The experimental results show that a cache-friendly memory layout can reduce cache conflicts when applications within a single address space run on a single CPU.

Table 1. The latency of core reallocation (μs).

	Avg.	P50	P90	P99	P999
Vessel	0.161	0.160	0.162	0.173	0.706
Caladan	2.103	2.063	2.091	2.420	5.461

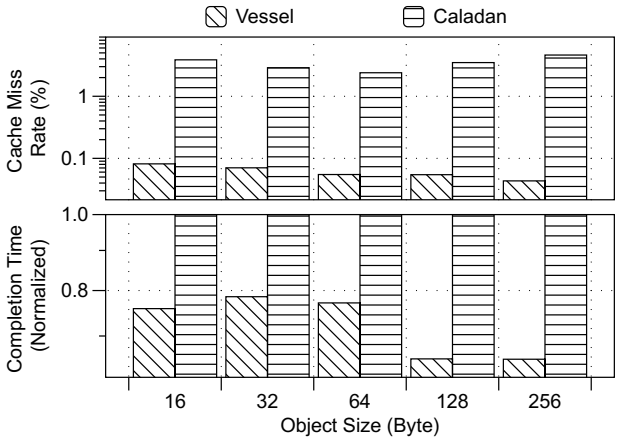

Figure 11. Cache friendliness evaluation. *Two single-threaded* `L-app`*s run on the same core, each of which runs an object copy.*

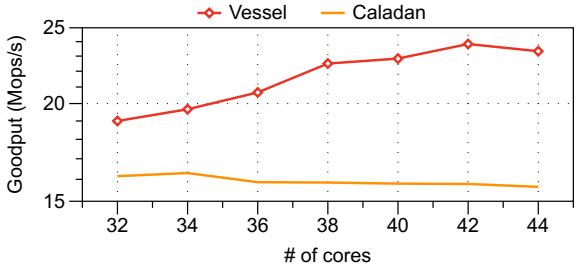

Figure 12. Performance with different numbers of CPU cores. *Goodput represents the maximum throughput each system can achieve within the P999 latency limit of 60 μs.*

6.3.3 CPU Core Scalability. One Vessel scheduling domain can schedule up to 42 cores in our experimental setup. We colocate one L-app with a `B-app` to explore the peak processing capacity of each system. We use the maximum throughput that one system can achieve at the P999 latency limit of 60 μs to indicate the peak processing capacity, which is referred to as goodput. Figure 12 demonstrates that as the number of CPU cores increases from 32 to 42, the goodput of Vessel rises around 25.44%. And, it reduces to 22.81% with 44 cores. In the case of Caladan, the goodput increases around 1.45% from 32 to 34 cores, and it begins to decrease, which indicates that Caladan can scale up to 34 cores.

6.3.4 Memory Bandwidth Regulation. Vessel can reallocate cores at a much finer timescale, which is also beneficial for regulating memory bandwidth consumption of an application. We conduct two experiments to verify this. First, both Caladan and Vessel support using memory bandwidth consumption as a metric for core scheduling (e.g., reduce CPU cores of a `B-app` when it consumes higher memory bandwidth than a threshold), so we colocate an `L-app` (Memcached) with a memory-intensive `B-app` (i.e., *membench*) and measure their

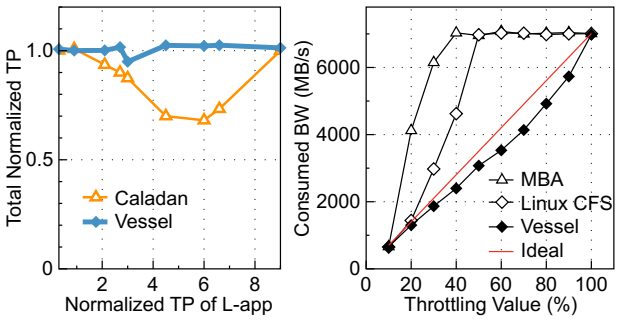

Figure 13. Memory bandwidth regulation. *(a) Colocating an* `L-app` *with a memory-intensive* `B-app` *membench. (b) The accurancy of memory bandwidth regulation.*

total normalized throughput under the tail latency constraints. As shown in Figure 13a, Vessel achieves up to 43% higher performance than Caladan.

Second, Vessel can also be used to assign an application fine-grained CPU quota for accurately regulating its memory bandwidth consumption. To demonstrate its efficacy, we compare Vessel with both hardware and software approaches. Among them, Intel Memory Bandwidth Allocation [21] (MBA) is a hardware approach that throttles memory bandwidth usage by inserting delays to memory requests. Linux cgroup works similarly to Vessel. We run the *membench* workload with a single thread and use the above approaches for bandwidth regulation. We measure the actually consumed bandwidth of them as we change the throttling value from 10% to 100% (see Figure 13b). Both MBA and Linux `CFS` use far higher memory bandwidth than desired (i.e., low accuracy). Instead, Vessel can accurately control memory bandwidth usage with sub-microsecond core scheduling.

7 Related Work

Userspace core scheduler. Existing systems leverage userspace core scheduling to handle microsecond-scale RPCs, including Arachne [42], Shinjuku [28], Shenango [39], and Caladan [17]. Syrup [29] and ghOSt [24] allow a userspace agent to write custom scheduling policies. These systems make timely core reallocation decisions in userspace. However, they must submit the decision to the kernel scheduler to finish the operation, which is less efficient than `Uintr`-based preemption in Vessel.

Hardware-offloaded core scheduling. Some recent studies try to offload the scheduler to SmartNICs. Shinjuku-offload [23], Turbo [47], and Ringleader [32] use the on-board FPGA or ARM of SmartNICs to implement load balancing and core allocation tasks. eRSS [44] makes core reallocation decisions on Taurus SmartNIC and delivers interrupts directly to host CPUs. Our work is orthogonal to them and Vessel's centralized scheduler can be offloaded to the NIC and work cooperatively with uProcesses.

Process abstraction. Recent work proposes new OS abstraction on top of the classical process. *lwCs* [33], *sthread* [7] and *Shreds* [8] divide a POSIX process into several isolated components to reduce the thread-scheduling overhead. In contrast, uProcess allows application tasks to be scheduled to run inside different processes. Junction [16] is a concurrent work optimized for densely colocating applications. It places multiple inter-trusted applications within a shared address space to enable fast scheduling. Vessel does not make such an assumption and uses MPK to allow untrusted applications to share the address space. Nu [43] proposes the *proclet* that includes heap data with runtime context and can be scheduled between POSIX processes and even across machines. However, applications require refactoring and recompilation to support the proclet. On the contrary, we do not intend to propose a new programming model; Vessel aims to accelerate existing applications used to run in kProcesses.

MPKs related memory protection. Previous work mainly utilize MPK to achieve intra-process isolation. Both ERIM [49] and Hodor [22] are software sandboxes that use MPK to isolate untrusted components from trusted ones within an application; however, they are still vulnerable to various software attacks [10]. UnderBridge [20] isolates the components in a microkernel to accelerate IPC data transferring. CubicleOS [45] isolates components in the system and allows the isolated components to share data dynamically with each other. Instead, Vessel utilizes MPK to ensure that the shared address space can be safely isolated and efficiently shared among uProcesses. Besides, many past works also enhanced the ease of use of MPK [19, 41], and proposed diverse techniques for safely using MPK in real cases [10, 26, 46, 49]. Vessel refers to parts of the design of these systems when designing the call gate, syscall hook, etc.

8 Conclusion

This paper attempts to break the boundaries between traditional processes and deploying applications within the same address space so that CPU cores can be freely switched between different applications with minimal switching overhead. To achieve this, two new hardware features – `Uintr` and MPK – are combined so that a single address space can be efficiently shared and safely isolated. With this new process abstraction, we implemented Vessel, a pure userspace core scheduler that is well-compatible with existing software and shows efficiency when time-sharing CPU cores among multiple applications.

Acknowledgement

We sincerely thank the anonymous reviewers and our shepherd Amy Ousterhout for their comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 62332011 and 62202255).

References

[1] [n. d.]. Intel Optane Memory - Responsive Memory, Accelerated Performance. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[2] [n. d.]. Memory-Semantic SSD. https://samsungmsl.com/ms-ssd/.

[3] [n. d.]. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[4] [n. d.]. Redis. http://redis.io.

[5] [n. d.]. Ultra-Low Latency with Samsung Z-NAND SSD. https://semiconductor.samsung.com/resources/brochure/Ultra-LowLatencywithSamsungZ-NANDSSD.pdf.

[6] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 49–67. https://www.usenix.org/conference/nsdi23/presentation/bai

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments

[8] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. 56–71. https://doi.org/10.1109/SP.2016.12

[9] Memcached community. [n. d.]. memcached – a distributed memory object caching system. https://lwn.net/ml/linux-kernel/20210913200132.3396598-1-sohil.mehta@intel.com/.

[10] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1409–1426. https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[11] Intel Corporation. [n. d.]. Intel® architecture instruction set extensions programming reference. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[12] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 478–493. https://doi.org/10.1145/3341301.3359637

[13] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 401–414.

[15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[16] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making Kernel Bypass Practical for the Cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 55–73. https://www.usenix.org/conference/nsdi24/presentation/fried

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 281–297.

[18] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 519–533. https://www.usenix.org/conference/nsdi21/presentation/gao

[19] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[20] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-kernel Isolation and Communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 401–417. https://www.usenix.org/conference/atc20/presentation/gu

[21] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011), 0–40.

[22] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 489–504. http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[23] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 60–68.

[24] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542

[25] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 113–128. https://www.usenix.org/conference/osdi21/presentation/hwang

[26] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-

documentation/analysis-speculative-execution-side-channels.html.

[27] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, Jing Li, and Xiaoning Ding. 2022. Achieving Low Latency in Public Edges by Hiding Workloads Mutual Interference. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 477–492. https://doi.org/10.1145/3542929.3563459

[28] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[29] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack. 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, 605–620. https://doi.org/10.1145/3477132.3483548

[30] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. 2023. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 331–346. https://www.usenix.org/conference/fast23/presentation/li-qiang-deployed

[31] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 429–444.

[32] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. 2023. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1293–1308. https://www.usenix.org/conference/nsdi23/presentation/lin

[33] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 49–64.

[34] 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)*. Association for Computing Machinery, New York, NY, USA, 399–413. https://doi.org/10.1145/3341301.3359657

[35] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient scheduling policies for {Microsecond-Scale} tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1–18.

[36] Sohil Mehta. [n. d.]. User Interrupts – A faster way to signal. https://lpc.events/event/11/contributions/985/attachments/756/1417/User_Interrupts_LPC_2021.pdf.

[37] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. 2022. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 753–766. https://doi.org/10.1145/3544216.3544238

[38] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[39] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.. In *NSDI*, Vol. 19. 361–378.

[40] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (aug 2015), 55 pages. https://doi.org/10.1145/2806887

[41] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK) *(USENIX ATC '19)*. USENIX Association, USA, 241–254.

[42] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 145–160.

[43] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan

[44] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019* (Beijing, China) *(APNet '19)*. Association for Computing Machinery, New York, NY, USA, 71–77. https://doi.org/10.1145/3343180.3343184

[45] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 546–558. https://doi.org/10.1145/3445814.3446731

[46] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. 936–952.

[47] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. 2023. Turbo: SmartNIC-enabled Dynamic Load Balancing of µs-scale RPCs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1045–1058. https://doi.org/10.1109/HPCA56546.2023.10071135

[48] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[49] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. 1221–1238.

[50] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[51] Andrew Waterman1 and Krste Asanović. 2023. The RISC-V Instruction Set Manua, Volume I: User-Level ISA, Document Version 2.2. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[52] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, Cheng Wang, Jian He, Guodong Yang, and Liping Zhang. 2022. Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 210–225. https://doi.org/10.1145/3542929.3563465