Skyloft: A General High-Efficient Scheduling Framework in User Space

Yuekai Jia[†*], Kaifu Tian[†*], Yuyang You[†], Yu Chen[‡†], Kang Chen[†]
[†]Tsinghua University, Beijing, China
[‡]Quan Cheng Laboratory, Jinan, China

Abstract

Skyloft is a general and highly efficient user-space scheduling framework. It leverages user-mode interrupt to deliver and process hardware timers directly in user space. This capability enables Skyloft to achieve μs-scale preemption. Skyloft offers a set of scheduling interfaces that supports different scheduling policies, including both preemptive and non-preemptive ones. Operating as a user-space scheduling framework, Skyloft is compatible with Linux and integrates seamlessly with high-performance I/O frameworks like DPDK.

Evaluation results show that optimizations in per-CPU scheduling with user-space timer interrupts allow Skyloft's Completely Fair Scheduler (CFS) and Round Robin (RR) to significantly reduce wake-up latency compared to their Linux counterparts (100μs vs. 10000μs). In comparison to the general scheduling framework ghOSt, Skyloft achieves a 1.2× increase in maximum throughput for Latency Critical (LC) applications. Additionally, unlike the specialized scheduling framework Shinjuku, Skyloft not only supports LC applications but also efficiently allocates CPU resources to Best Effort (BE) applications under low load conditions. By incorporating a 5μs preemption mechanism into the Work-Stealing strategy, a RocksDB Server running on Skyloft exhibits a performance improvement of 1.9× compared to Shenango.

CCS Concepts: • **Software and its engineering** → **Scheduling**.

Keywords: Operating systems, Scheduling, User interrupts

ACM Reference Format:
Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. 2024. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3694715.3695973

*Both authors contributed equally to this research.

1 Introduction

Cloud computing increasingly demands highly optimized and customized scheduling policies to enhance performance across key metrics such as tail latency, throughput, and resource utilization [21, 30, 45]. For distributed services within an application, improved scheduling is crucial for reducing tail latency, as demonstrated in large-scale RPC systems at Google [51]. Additionally, for varying workloads, application-specific scheduling is essential to meet service level objectives (SLOs) [16, 28, 30, 47].

Efforts have been made to customize Linux schedulers using user agents [26], live updates [37], and the Berkeley Packet Filter (BPF) [25], among others. However, scheduling threads through the kernel introduces performance overhead due to frequent mode switches and reduced locality [53]. Developing scheduling policies within the kernel is either constrained by limited interfaces [25] or requires significant modifications to the kernel's scheduling subsystem [26]. Additionally, kernel-level scheduling cannot take full advantage of widely deployed kernel-bypass drivers and frameworks, such as DPDK and SPDK [12, 18].

Recent works manage threads in user space and make scheduling decisions without kernel intervention for performance [32, 35, 52]. However, none of these approaches are sufficiently flexible to support both *μs-scale preemption* and *multiple applications* to offer the same level of versatility in scheduling policies as their kernel-based counterparts.

μs-scale preemption is crucial for meeting today's stringent μs-scale tail latency SLOs. In datacenter applications, the First Come, First Served (FCFS) policy is widely used and performs well for workloads with light-tailed distributions (e.g., Meta's USR workload [3] in Memcached). However, it suffers from head-of-line blocking for workloads with heavy-tailed distributions [16, 28, 30]. For instance, RocksDB can process GET requests within 10 microseconds, but range queries may take hundreds of microseconds or even milliseconds to complete. Consequently, schedulers must support preemptive scheduling to implement a Processor Sharing (PS) policy for heavy-tailed workloads [59]. User-space schedulers like ZygOS [48] and Shenango [45] lack support for preemption within a single application. Other user-space schedulers [8, 52] rely on Linux signals for preemption, which introduces extra overheads.

Supporting multiple applications is essential for improving CPU efficiency. Schedulers must dynamically allocate unused cores from **latency-critical (LC)** applications to **best-effort (BE)** applications and swiftly reallocate cores back to LC applications to handle burst loads. Shinjuku [30] and Concord [28] implement μs-scale preemptive scheduling through posted interrupts using virtual machines and compiler instrumentation, optimizing both tail latency and throughput for workloads with light- and heavy-tailed distributions. However, these systems do not support core sharing with other applications to enhance CPU efficiency at low loads, as they dedicate cores exclusively to a single application.

To enable universal user-space scheduling that supports both μs-scale preemption and multiple applications, we propose Skyloft, a general user-space scheduling framework designed to achieve the following goals.

High Flexibility. Skyloft provides flexible support for various scheduling policies across multiple applications. It accommodates both preemptive and non-preemptive scheduling policies, allowing efficient management of threads both among applications and within individual applications.

High Efficiency. Skyloft supports μs-scale preemption, enabling it to efficiently implement preemption-based scheduling policies. To achieve this, Skyloft needs to support lightweight context switching, efficient interrupt management, fast resource allocation and synchronization primitives for effective management of user-level threads.

High Compatibility. Skyloft is fully compatible with kernel-bypass I/O frameworks, such as DPDK, allowing high-performance I/O operations. Furthermore, Skyloft seamlessly coexists with Linux, offering flexibility in deployment. Applications have the freedom to choose between utilizing Linux's native schedulers or leveraging Skyloft's custom scheduling mechanisms, based on their specific performance or operational requirements.

Achieving μs-scale preemption and efficient multi-application switching is challenging. Both are closely tied to privileged operations typically restricted by the kernel, such as interrupt handling and page table management.

For μs-scale preemption, Skyloft leverages a new hardware feature, *User Interrupts* (UINTR), introduced in Intel Sapphire Rapids processors [13]. This feature enables user-space programs to register, deliver, and handle interrupts with minimal configuration overhead from initialization by the kernel. Both user-level threads and hardware components can act as interrupt senders. Skyloft supports both centralized preemptive scheduling and per-CPU preemptive scheduling, closely resembling the native Linux schedulers. By utilizing the per-CPU hardware timer design (x86 LAPIC), Skyloft handles timer interrupts in user space, offering improved scalability compared to dispatchers or timers simulated via Inter-Processor Interrupts (IPI).

To support multiple applications, Skyloft avoids reliance on a global controller for making decisions. Instead, it utilizes a main loop on each core, benefiting from a clean interface between the user and kernel for efficient application switching. Skyloft provides a set of scheduling operations to facilitate the implementation of various scheduling policies. A scheduler can leverage these operations, such as `task_enqueue` and `task_dequeue`, to define custom policies.

To evaluate Skyloft, we implement several per-CPU scheduling policies and achieve wakeup latencies that are orders of magnitude lower on schbench [39] compared to native Linux schedulers (§5.1). We also implement a centralized policy and evaluate Skyloft using synthetic workloads, demonstrating that Skyloft can achieve 1.2× higher maximum throughput than ghOSt [26], while maintaining a comparable core share for batch applications (§5.2). For real-world applications, we implement a work-stealing policy, where Skyloft delivers performance similar to Shenango on Memcached. After enabling preemption for the work-stealing policy, Skyloft achieves 1.2× higher performance than Shenango on a RocksDB server, while maintaining identical tail-latency SLOs (§5.3). Finally, we show that Skyloft's preemption overhead is 0.6μs from sending an interrupt on one core to handling the interrupt on another core, and 0.3μs to handle a timer interrupt, nearly 10x faster than a soft timer (§5.4).

The main contributions of this paper are:

- Skyloft supports kernel-bypass preemption by leveraging a new hardware feature (UINTR). To the best of our knowledge, Skyloft is the first general user-space framework to support μs-scale preemptive scheduling for multiple applications.
- Skyloft proposes a paradigm to develop various schedulers with a set of general operations. Based on Skyloft, scheduling policies – previously requiring significant modifications to specialized schedulers – can be implemented in just hundreds of lines of code.
- Skyloft achieves comparable or even better performance than existing works on both synthetic and real-world workloads for popular applications.

2 Motivation

2.1 User-Space Scheduling Frameworks

The user-space scheduling frameworks [5, 21, 26, 30, 45, 48, 49] are proposed to achieve both flexibility and performance, as with other modern kernel-bypass frameworks [12, 18, 29, 33, 47, 60]. Based on the generality, user-space scheduling frameworks can be classified based on two dimensions: whether or not a scheduling framework can support (1) *μs-scale preemption*; or (2) *multiple applications*.

Microsecond-scale preemptive scheduling. Non-preemptive scheduling policies are usually used in user space. For example, with the run-to-completion policy, a task never switches to another task throughout its lifetime. The cooperative policy allows switching to another task during execution, but the switch is invoked by the task itself at a deterministic switch point (e.g., yield or await). For short-lived RPC requests (such

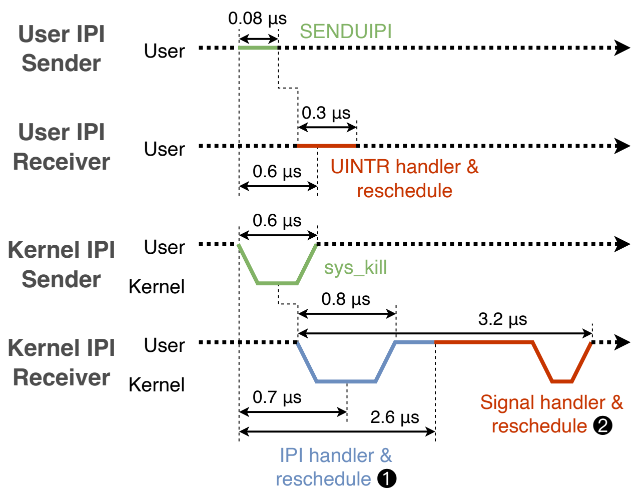


Figure 1. Comparsion between different preemptive scheduling frameworks. ❶ ghOSt [26] modifies kernel scheduler to reschedule threads on receiving kernel IPIs. ❷ Shenango [45] further delivers a signal to the thread to yield the core.

as GET/SET requests in key-value stores like in Memcached), a run-to-completion policy is often used to avoid the overhead of switching, while also contributing to improved data locality [5]. However, non-preemptive scheduling leads to higher tail latency under heavy-tailed workloads [30]. We need μs-scale preemption to approximate processor sharing (PS) policy, which performs best in terms of tail latency for such workloads [59].

Multiple applications. Today's large-scale datacenters need to support multiple applications with different characteristics, such as latency-sensitive applications and batch-processing applications requiring high throughput. To improve CPU efficiency, schedulers need to allocate unused cores from latency-sensitive applications to batch processing applications. Moreover, a scheduler is also required to reallocate cores back to latency-sensitive applications to handle peak load quickly. Linux can only achieve microsecond latency with low CPU utilization [34]. Alternatively, Shenango [45] and Caladan [21] support fast core allocations between multiple applications. Thus a flexible scheduling framework should also support multiple applications for efficient resource utilization.

2.2 Challenges in User-Space Scheduling

Scheduling is a core function of an operating system, relying on privileged operations such as handling interrupts and switching CPU states. It is challenging for user-space scheduling frameworks to support both *μs-scale preemption* and *multiple applications* with no serious negative impact on performance.

Microsecond-scale preemption support. Preemption requires the ability to interrupt and subsequently resume task execution at any point. As shown in Figure 1, preemption via kernel IPI (Inter-Processor Interrupt) has been widely used in existing work. In ghOSt, the agent sends messages to the kernel, and the kernel sends an IPI to preempt other cores, which introduces additional context-switching overhead. For user-space scheduling frameworks, such as Shenango, Linux signals are used to send preemption signals. This mechanism, in addition to kernel IPI handling, requires the signal to be handled in user space through a signal handler, introducing even more context-switching overhead. Additionally, signal handling faces scalability challenges when multiple threads on different cores receive signals simultaneously, leading to increased processing times due to data races [52].

Multi-application support. To enable scheduling threads from different applications, the first challenge is to address the context-switching issue for different applications. From the kernel's perspective, different applications are separate processes with their own address spaces. Switching to a thread in another application requires switching the address space. Kernel-space schedulers can easily handle address space switching, but these privileged operations are difficult to perform in user space. Additionally, the scheduler needs to have a global view of all threads (belonging to different applications), such as priorities, time slices, etc., in order to make the appropriate scheduling. Therefore, user-space schedulers need data sharing to get the global view of all threads.

	Scheduling unit	μs-scale preemption	Multiple applications
IX[5], ZygOS[48]	uthread	✘	✘
Shinjuku [30]	uthread	✔	✘
Shenango [45]	uthread	✘	✔
ghOSt [26]	kthread	✔	✔
Skyloft	uthread	✔	✔

Table 1. Comparison of different schedulers. Scheduling unit denotes the minimum scheduling unit (*uthread* for user-level thread, while *kthread* for Linux `task_struct`).

2.3 Limitations of Existing Work

Currently, a series of works have been dedicated to enabling applications to customize their scheduling strategies, including both user-space and kernel-space schedulers. Table 1 summarizes their comparisons with Skyloft.

User-level threading. The M:N threading model, supported by some runtime libraries [6, 32, 44, 58] and programming languages with asynchronous capabilities, like Go [22] and Rust [50], maps M user-level threads to N kernel-level threads. User-level threads are lighter and can handle many thread operations without needing the kernel, which reduces the overhead associated with thread creation, context switching, and synchronization. Despite these advantages, user-level threads struggle with low-overhead preemption and inter-application scheduling. Consequently, they are limited to functioning within a single application and cannot efficiently manage tasks across multiple applications. Moreover, because the kernel scheduler does not recognize user-space scheduling, it might inadvertently suspend a user-level thread that is holding a lock, potentially causing indefinite spinning [10, 55].

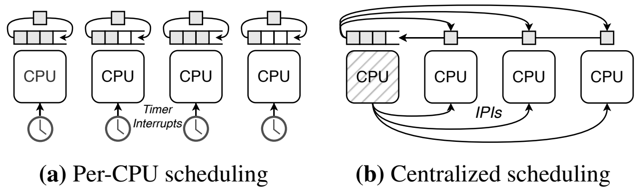
Dataplane OSes. Several works have integrated scheduling frameworks with kernel-bypass technologies like DPDK [18] to develop dataplane operating systems such as IX [5] and ZygOS [48]. These systems, however, primarily focus on executing tasks to completion and lack support for preemption. Shinjuku [30] and Concord [28] utilize posted interrupts and compiler instrumentation techniques, respectively, to facilitate μs-scale preemption. LibPreemptible [35] makes use of user interrupts. These strategies tightly integrate schedulers with application logic, which complicates adaptation to various scheduling policies and multiple applications. Moreover, all these methods restrict dedicated cores to a single application, leading to inefficient use of resources.

Core reallocation across applications. Several studies focus on achieving performance isolation between applications [21, 27, 40, 45, 49], aiming to meet the tail latency requirements of LC applications while optimizing the throughput of BE applications. These approaches typically monitor application congestion using various metrics periodically, making decisions to add or remove cores for each application and conducting intra-application load balance. For instance, Arachne [49] redistributes resources every 50ms, whereas Shenango [45] and Caladan [21] do so at frequencies as low as 5μs. However, the flexibility of these systems is limited by their tightly integrated schedulers. Additionally, these frameworks rely on either Linux signals or kernel IPIs for preemption, which are not optimal.

User-space delegation of kernel scheduling. Some studies utilize the Linux scheduling subsystem and delegate scheduling decisions to user space. ghOSt [26] facilitates this by informing user agents about changes in the state of kernel threads. However, this approach requires frequent interactions with the kernel, leading to considerable overhead. The recent `sched_ext` patch [25] in Linux enables control over kernel schedulers through BPF [20]. BPF programs are limited in their expressive capabilities due to security constraints, such as prohibitions on loops and floating-point arithmetic, which complicates the implementation of complex scheduling strategies. These methods depend on kernel-level threads, requiring user-to-kernel mode switches for thread creation and synchronization, which degrades performance.

2.4 Design Goals

Skyloft aims to propose a general scheduling framework to achieve both flexibility and performance. We consider the following requirements as goals for Skyloft:



(a) Per-CPU scheduling **(b)** Centralized scheduling

Figure 2. Two typical scheduling models.

High Flexibility. Skyloft should offer a range of general scheduling operation interfaces to help the implementation of various scheduling policies, such as per-CPU/centralized, or cooperative/preemptive. Additionally, Skyloft should not be limited to scheduling threads within a single application but also support thread scheduling across multiple applications. For instance, when all threads in one application are blocked, the corresponding cores should be allocated to other applications to enhance CPU utilization. This necessitates that the Skyloft scheduler maintains a global view of all applications to make informed scheduling decisions.

High Efficiency. Skyloft must support μs-scale preemption with minimal overhead for latency-critical applications. Preemption can be facilitated by CPU-local timer interrupts (Figure 2a) or through a dedicated dispatcher core that sends IPIs to other cores (Figure 2b). Skyloft should limit interactions with the kernel and implement most of its functionality in user space to decrease the overhead associated with context switching. For instance, employing user-level threads can minimize thread operation overhead, and handling interrupts in user space can reduce preemption overhead.

High Compatibility. Skyloft should require minimal modifications to the kernel to ensure compatibility with existing Linux systems. It should offer POSIX-compatible threading APIs, giving applications the flexibility to choose between using Skyloft or native Linux schedulers. Additionally, Skyloft should be compatible with user-space I/O frameworks, such as DPDK [18], for performance.

3 Design

3.1 Design Overview

Skyloft operates on top of Linux to ensure compatibility. To avoid interference from the kernel in Skyloft's scheduling, specific cores are isolated at boot time, exclusively dedicated to running applications within Skyloft. The remaining cores continue to function under the Linux scheduler for handling regular system applications.

Skyloft consists of two primary components (Figure 3): the *Skyloft Library OS*, which operates in user space, enabling kernel-bypass scheduling while offering POSIX-compatible APIs; and the *Skyloft Kernel Module*, which runs in kernel space, managing thread switching between applications and performing other privileged operations.

The core function of the Skyloft LibOS is to manage user threads, which are the fundamental units of scheduling. ❶

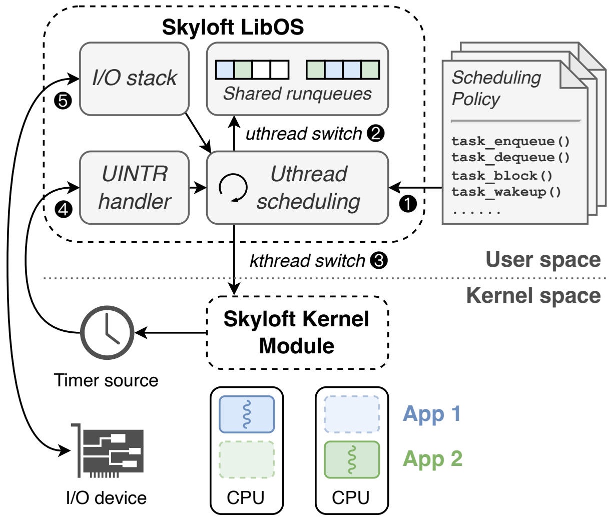


Figure 3. Skyloft architecture.

The user-space scheduler maintains a shared runqueue across all applications, follows various user-defined policies, and makes scheduling decisions accordingly. When a context switch is required, it either ❷ switches the user thread directly in user space or ❸ invokes the kernel module to switch the corresponding kernel thread , depending on the applications involved (§3.3). ❹ Additionally, the user-space scheduler handles user interrupts to enable μs-scale preemption (§3.2). ❺ Moreover, the Skyloft LibOS integrates kernel-bypass I/O stacks (§3.5).

3.2 User-Space Preemption

To enable μs-scale preemptive scheduling in user space, Skyloft utilizes a new hardware feature called *User Interrupts* (UINTR), available on Intel Sapphire Rapids servers. Intel has also submitted a patch for UINTR to the Linux kernel [41]. UINTR allows interrupts to be delivered directly to a user-space handler without switching address spaces or privilege levels, offering a low-overhead mechanism for event dispatch and inter-processor communication.

Resources related to UINTR. A dedicated *interrupt vector* is used to identify user interrupts [13], configured in the UINV (User-Interrupt Vector) register. Each receiving thread registers a user-interrupt handler through the UIHANDLER (User-Interrupt Handler) register and maintains a User Posted-Interrupt Descriptor (UPID) in memory, which contains essential information. For instance, the PIR (Posted-Interrupt Requests) field in the UPID holds interrupts posted by a sender; a non-empty PIR value indicates that a user interrupt has been raised. Before invoking the handler, the core sets the appropriate bit in the User Interrupt Request Register (UIRR), which is a bitmap representing up to 64 different interrupts, based on the PIR value.

Sending a user interrupt. User interrupts are typically generated using the `SENDUIPI` instruction. Each sender maintains a User-Interrupt Target Table (UITT) in memory, with each entry containing the UPID address of the receiver and the notification vector. The `SENDUIPI` instruction takes the index of the UITT entry as its operand, sets the corresponding bit in the PIR, and then sends a standard Inter-Processor Interrupt (IPI) to the target CPU via the local Advanced Programmable Interrupt Controller (APIC).

User-interrupt handling. The process of handling a user interrupt involves the following steps: (1) **Identification**: When the core receives an interrupt V, it proceeds to the next step if V matches the UINV value. Otherwise, V is treated as a legacy interrupt. (2) **Processing**: The core sets each bit in the UIRR based on the bits set in the PIR field of the UPID, detecting a pending user interrupt. (3) **Delivery**: Once the CPU enters user mode and the UIRR is set, the user interrupt is delivered. The current state, such as the stack pointer, instruction pointer, etc., is saved onto the stack, and control flow jumps to the user-interrupt handler. (4) **Handling**: The interrupt handler completes context saving and handles the interrupt. Once finished, it restores the context and executes the `UIRET` instruction to return to normal execution.

In Skyloft, preemption can be triggered through two mechanisms: by sending *IPIs* to other cores, or by utilizing hardware-generated *timer interrupts*.

Preemption with IPIs. This method is particularly well-suited for the centralized scheduling models illustrated in Figure 2b. A dispatcher periodically assesses the need to preempt tasks on other worker cores, acting as a clock source. When preemption is necessary, the dispatcher sends a user IPI via the `SENDUIPI` instruction to the targeted core, triggering the user-interrupt handler. The main advantage of this approach is its flexibility, as the frequency of preemption signals can be dynamically adjusted based on workload fluctuations [35]. However, this method requires a dedicated core for the dispatcher, which reduces overall throughput since it cannot be used for regular task execution. Moreover, the constant monitoring of all worker cores by the dispatcher can introduce bottlenecks, particularly in systems with many cores, thus limiting scalability [28, 30].

Preemption with timer interrupts. Timer interrupts are crucial for supporting per-CPU scheduling models, as illustrated in Figure 2a, and they eliminate the need for a dedicated dispatcher core. However, implementing user-space handling of hardware interrupts, including timer interrupts, presents significant challenges. To the best of our knowledge, no existing research or documentation has addressed the facilitation of user-space hardware interrupt handling, nor is it covered in the Intel manual [13]. Skyloft implements two critical steps to manage hardware interrupts, such as timer interrupts, in user space. The first step involves configuring the UINV register with the timer interrupt vector, enabling the core to recognize the timer interrupt as a user interrupt. However, this step alone is insufficient because the PIR is not automatically updated when a timer event occurs. Since the PIR remains empty, the core does not trigger the user-interrupt handling process. Therefore, the second step—updating the PIR—is more challenging. While the `SENDUIPI` instruction could be used to update the PIR, this would unnecessarily generate an IPI. Fortunately, we discovered that a bit in the UPID, known as SN (Suppress Notification), can prevent the generation of an actual IPI. As a result, Skyloft requires each core to *send itself an IPI with the SN bit set*, effectively updating the PIR without triggering an IPI.

To configure Skyloft for handling hardware interrupts, the following steps are performed: (1) Initialize each thread's UPID and set the SN bit. (2) Execute `SENDUIPI` to populate the PIR with a non-empty value (using any interrupt number), allowing the first hardware interrupt to be handled in user space. (3) After entering the interrupt handler, execute `SENDUIPI` again to ensure subsequent hardware interrupts are handled in user space. By following these steps, Skyloft successfully delegates local APIC timer interrupts to user space, enabling support for per-CPU preemptive scheduling policies.

3.3 Scheduling Threads Across Applications

Managing kernel threads. To simplify the overall design, Skyloft seeks to reuse the existing process and thread abstractions as much as possible. Process represents an application's global resources, such as address space and file table, while kernel threads represent the application's CPU resources.

Skyloft allocates a specific number of *isolated cores* for the applications it schedules. When an application is launched, Skyloft creates a number of kernel threads equal to the number of isolated cores and binds each thread to one of these cores. As a result, each core is associated with a set of kernel threads, corresponding to the number of initiated applications. Among these kernel threads, only those in the *runnable* state are selected by the kernel scheduler for execution. Threads that are exited, suspended, or blocked do not appear in the kernel scheduler's runqueue. These are referred to as *active* and *inactive* kernel threads, respectively. The primary challenge is to prevent scheduling interference from the kernel in Skyloft. To address this, Skyloft enforces the Single Binding Rule:

Single Binding Rule: ***No two or more active kernel threads may be bound to the same isolated core simultaneously***.

By following this rule, the kernel scheduler is entirely bypassed. Each core's single active kernel thread is associated with the application currently running on that core. Skyloft then manages the user threads within each application. As shown in Figure 3, every application has one active kernel thread occupying distinct isolated cores. To enforce the binding rule when launching a new application, Skyloft binds newly created kernel threads to isolated cores and immediately suspends them, making them inactive until they are later activated.

In principle, kernel threads not managed by Skyloft can still be bound to isolated cores if their CPU affinity is manually set and they are in a runnable state. As these threads do not share state with Skyloft, potential interference may occur, possibly leading to performance degradation. To avoid this issue, it is recommended that regular applications not managed by Skyloft refrain from manually setting CPU affinity.

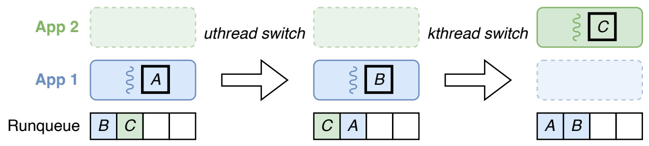


Figure 4. Skyloft switches user-level threads within the same application($A \rightarrow B$) and between different applications($B \rightarrow C$).

Scheduling user threads. Skyloft organizes all user threads from all applications into either a single global runqueue or per-CPU runqueues, depending on the chosen scheduling policy. These runqueues are shared across all applications.

A running user thread may yield or be preempted by another user thread. The next thread to execute is selected from the shared runqueue based on the scheduling policy. As illustrated in Figure 4, if the next thread belongs to the same application, the switch can occur directly in user space without kernel intervention. However, if the next thread is from a different application, an application switch is required. This involves setting the current application's active kernel thread to inactive (suspending it) and activating (waking up) the inactive kernel thread of the target application. Both steps must occur simultaneously in the kernel to uphold the binding rule. Since the overhead of switching between applications is considerably higher than switching within the same application (§5.4), minimizing inter-application thread switching is an important metric for scheduling policies to consider.

Application termination. When all user threads of an application have completed execution (or if the application manually invokes `exit()`), the application and all its kernel threads will be terminated. Before terminated, an active kernel thread of the application must wake up another thread on the same core first. Otherwise, once the sole active thread exits, there will be no active threads left on that core, leading to the potential issue of inactive kernel threads never being woke up.

Additionally, since a kernel thread must reclaim its resources before terminating, which must be performed by the thread itself, each kernel thread remains in a runnable state until it fully exits. To adhere to the binding rule, Skyloft rebinds active kernel threads to non-isolated cores before allowing them to exit independently, or it sends a termination signal to inactive threads.

As described, the kernel must perform several operations simultaneously, such as transitioning kernel thread states or adjusting CPU affinity. Skyloft implements these operations within a Linux kernel module, providing access to these functionalities for the Skyloft scheduler through the `ioctl()` interface (§4.2).

3.4 General Scheduling Operations

Inspired by both recent latency-optimized centralized schedulers [30] and traditional per-CPU schedulers in Linux [9, 56], Skyloft defines a set of operations to facilitate implementing schedulers in user space. As shown in Table 2, these operations include runqueue and task state manipulations for managing user threads, as well as event callbacks. For example, `sched_timer_tick()` is invoked in the timer interrupt handler to determine whether preemption is required. Based on these operations, high-level thread primitives (e.g., pthread functions) can be implemented.

Certain operations are tied to specific scheduling models. In per-CPU schedulers with multiple runqueues, task imbalance can significantly degrade performance [40, 45, 48]. To address this, Skyloft provides the `sched_balance()` operation, enabling policy-defined load balancing (e.g., work stealing [7]). On the other hand, centralized schedulers with a single queue are naturally balanced [30], so the `sched_balance()` operation is not necessary. Instead, the dispatcher must call `sched_poll()` to distribute tasks from the global runqueue to each worker core.

3.5 Integration with User-Space I/O Frameworks

Skyloft adopts a design similar to existing dataplane operating systems [5, 30, 45, 48], integrating high-performance user-space I/O frameworks for applications. For instance, Skyloft leverages DPDK [18] to poll network on a dedicated core. When a packet is received, it is distributed to the appropriate isolated core of Skyloft via a shared ring buffer based on the Receive Side Scaling (RSS) [15] hash value. A lightweight user-space TCP and UDP stack is integrated to parse network packets and provide POSIX-compliant socket interfaces to applications. When a thread is blocked by a request, the scheduler suspends it and switches to other threads. Idle cores also poll the ingress pool, creating new threads to process incoming packets.

4 Implementation

Skyloft operates on top of Linux, comprising 9K lines of C code for the LibOS and an additional 4K lines of C code to integrate DPDK and implement the TCP/UDP network stacks. Skyloft is linked with the application and runs as a process in user space. The Skyloft kernel module consists of 325 lines of C code. Additionally, we modified 163 lines of code in the UINTR kernel patch [41] to support user-space timer interrupts.

4.1 Skyloft LibOS

Scheduler Initialization. Skyloft uses the kernel command line parameter `isolcpus` to reserve isolated cores. Although Linux may still run some background threads on these cores [26], it does not interfere with Skyloft's scheduler and has minimal impact on performance.

A *daemon* starts before any applications and is responsible for initializing shared states. The daemon then creates a number of kernel threads equal to the number of isolated cores using the pthread library and binds these threads to the isolated cores with `sched_setaffinity()`. For subsequent applications, the initialization process is similar, but instead of directly binding the kernel threads, `skyloft_park_on_cpu()`, provided by the Skyloft kernel module (§4.2), is used to bind and suspend the kernel thread.

For each application, an *idle user thread* is created first on each kernel thread to run the main scheduling loop until it finds another runnable user thread to schedule. We implement a fast path for directly switching between two threads. If no runnable user thread is available, the scheduler saves the current thread's context and re-enters the main loop to perform further operations, such as load balancing.

Shared memory. Different applications use shared memory to exchange data, including application metadata, runqueues, memory pools for shared data structures. Each application must link an identical copy of the Skyloft library to ensure that the same version of the scheduler code and shared memory structures is used, regardless of which application is running. The application metadata stores the IDs of its kernel threads (obtained via `gettid()`), which are used by other applications to wake them up. The shared runqueue holds all runnable user threads, and the scheduler selects a user thread to run based on the defined scheduling policy. Each user thread includes fields that are visible to other applications, such as the thread state, the application it belongs to, and an extra field reserved for policy-defined data. This ensures that no matter which application is running, consistent scheduling information is visible. Skyloft maintains a memory pool in the shared memory dedicated to these structures. Additionally, each user thread has private fields, such as its context and stack, that are not visible to other applications.

User-interrupt configuration. The Linux UINTR patch [41] has introduced kernel APIs for user-space IPIs, enabling preemptive scheduling via user-space IPIs without requiring additional kernel modifications. A dedicated dispatcher thread is needed to send IPIs to other cores. Since the sender and receiver may belong to different applications, file descriptors opened by one are not visible to the other. To address this, we use `pidfd_get()` to share the opened user interrupt file descriptor (`uvec_fd`) between the sender and receiver, allowing the sender to establish a connection with the receiver for sending IPIs. Each application's kernel thread on each isolated

Scheduling Operations	Description
`void task_init(struct task_t* task)`	Initializes the policy-defined field of a task.
`void task_terminate(struct task_t* task)`	Finishes running a task and releases its policy-defined field.
`void task_enqueue(struct task_t* task, int flags)`	Puts a task to the runqueue.
`struct task_t* task_dequeue()`	Selects a task to run on and removes it from the runqueue.
`void task_block()`	Suspends the current task.
`void task_wakeup(struct task_t* task)`	Wakes up the task and puts it back to the runqueue.
`void sched_init(void* data)`	Initializes policy-defined states of the scheduler.
`bool sched_timer_tick()`	Updates scheduler states on each timer tick, returns `true` if rescheduling is required (current task is preempted).
`void sched_balance()`	Performs load balancing among runqueues (per-CPU policy only).
`void sched_poll()`	Polls the global runqueue and dispatches tasks to worker cores (centralized policy only).

Table 2. Scheduling interfaces defined by Skyloft to implement different scheduling policies.

	APIs	Description
App Lifecycle	`skyloft_park_on_cpu(cpu_id)`	Binds the current kernel thread to the specified core and suspends.
	`skyloft_switch_to(target_tid)`	Suspends the current kernel thread and wakes up the target kernel thread.
	`skyloft_wakeup(tid)`	Wakes up the specified kernel thread.
UINTR Configuration	`skyloft_timer_enable()`	Enables user-space timer interrupts on the current core.
	`skyloft_timer_set_hz(hz)`	Configures the timer frequency on the current core.

Table 3. APIs provided by the Skyloft kernel module.

```c
void  __attribute__((interrupt))
uintr_handler(struct __uintr_frame *ui_frame,
              unsigned long long vector) {
  if (is_timer_uintr(vector)) {
    _senduipi(uintr_idx); // reset UPID.PIR for next timer
  }
  // update policy-defined states, check preemption
  bool is_preempted = sched_timer_tick();
  // check if rescheduling is needed
  if (preempt_enabled() && is_preempted) {
    // put current task to end of runqueue
    task_enqueue(current);
    schedule(); // enter main scheduling loop
  }
}
```

Listing 1. Global user-interrupt handler for centralized and per-CPU schedulers.

core acts as a receiver, and the sender establishes connections with all of them at the start.

Skyloft also supports preemption via user-space timer interrupts without needing a dedicated dispatcher thread (§3.2), though this functionality is not provided by the UINTR patch. We address this with additional modifications to the UINTR patch, configuring specific UINTR-related registers and allowing the application to set the timer frequency.

User-interrupt handler. We implement a global user-interrupt handler, as shown in Listing 1, for both centralized and per-CPU schedulers using the operations described in §3.4. For timer interrupts, the handler first resets the UPID.PIR to prepare for the next timer interrupt (§3.2). The handler then updates the policy-defined states and makes a rescheduling decision if preemption is required.

4.2 Skyloft Kernel Module

The Skyloft kernel module is mounted as a miscellaneous device file at `/dev/skyloft`. It provides two key operations for the user-space scheduler: atomic invocation of kernel APIs for thread state transitions and execution of privileged operations for user-interrupt configuration. These operations are made available to user space through the `ioctl()` interface.

The upper half of Table 3 lists the operations for kernel thread state transitions. As detailed in §3.3, these operations are invoked when an application starts, switches, or terminates. They are implemented using Linux kernel APIs, such as setting CPU affinity, suspending, and waking up kernel threads, without additional kernel code modifications.

The lower half of Table 3 outlines the operations necessary for configuring user-space timer interrupts. As discussed in §3.2, user-space timer interrupts are enabled by setting UINV and UPID.SN, and the timer frequency can be configured through the local APIC registers.

5 Evaluation

We evaluate Skyloft schedulers to address the following questions:

Linux CFS (kernel/sched/fair.c)	6,592 LOC
Linux RT (kernel/sched/rt.c)	1,939 LOC
Linxu EEVDF (Linux v6.8 kernel/sched/fair.c)	7,102 LOC
ghOSt Shinjuku	710 LOC
ghOSt Shinjuku-Shenango	727 LOC
Skyloft Round-Robin (§5.1)	141 LOC
Skyloft CFS (§5.1)	430 LOC
Skyloft EEVDF (§5.1)	579 LOC
Skyloft Shinjuku (§5.2)	192 LOC
Skyloft Shinjuku-Shenango (§5.2)	444 LOC
Skyloft Work-Stealing (Preemptive) (§5.3)	150 LOC

Table 4. Lines of code for different schedulers.

- What is the impact of Skyloft's timer-interrupt delegation on per-CPU schedulers? (§5.1)
- How do Skyloft schedulers perform across multiple applications compared to ghOSt, a general-purpose scheduling framework? (§5.2)
- How does Skyloft handle real-world applications with light- and heavy-tailed workloads? (§5.3)
- What are the overheads associated with Skyloft operations? (§5.4)

Evaluated schedulers. To compare with existing systems, we have implemented several schedulers using Skyloft. Table 4 shows the lines of code (LOC) required for Skyloft and other systems, such as the Linux scheduling subsystem and ghOSt. Each Skyloft scheduler is implemented with only a few hundred lines of code, significantly fewer than the compared systems. This efficiency demonstrates the advantages of Skyloft's approach to developing preemptive schedulers entirely in user space, without requiring kernel intervention. In contrast, scheduling in previous user-space frameworks is often tightly integrated with other functionalities and lacks flexibility for different policies (e.g., Shinjuku [30] consists of 3,900 LOC and relies heavily on Dune [4]).

Experimental setup. We evaluated Skyloft on a Sapphire Rapids server that supports user interrupts. The server is equipped with two 24-core (48-hyperthread) Intel Xeon Gold 5418Y CPUs running at 2.0 GHz, along with 128 GB of RAM. To optimize for low latency, we configured the server according to best practices, disabling TurboBoost, CPU idle states, CPU frequency scaling, and transparent hugepages. For networking tests, we used a dual-socket client with 16-core Intel Xeon E5-2683 v4 CPUs at 2.1 GHz and 128 GB of RAM. Both the server and the client were equipped with a 10 Gb/s Intel 82599ES NIC. Unless otherwise noted, most evaluations on the server were conducted using DPDK 22.11 and a Linux kernel version 6.0.0, which was modified to support Intel user interrupts.

5.1 User-Space Timer Interrupts

To evaluate Skyloft's timer-interrupt delegation, we implemented per-CPU scheduling policies and compared them with Linux's schedulers, including `SCHED_RR` and `SCHED_BATCH`, which use Round-Robin (RR) with time slicing, and the Completely Fair Scheduler (CFS), respectively.

We use schbench [39] (v1.0), a widely used scheduler benchmarking tool [19], to evaluate various schedulers. Schbench simulates a typical network application by creating M message threads and T worker threads. The message threads continuously wake up the worker threads to perform simulated tasks (e.g., matrix multiplication). Once the tasks are completed, the worker threads enter a sleep state, waiting for the message threads to wake them up for the next request.

We configure Skyloft with 24 isolated cores. For Linux, schbench is also bound to 24 cores using the `taskset` command. The RR policy is applied using the `chrt` command. For both RR and CFS policies on Linux, schbench is set to the highest priority. Schbench is configured with default parameters (approximately 2,300 μs per request), using a single message thread, and the number of worker threads is gradually increased to saturate the cores. When the number of worker threads exceeds the number of cores, wakeup latency is affected by queuing time. Figure 5 compares the wakeup latencies of worker threads across the four schedulers. For both Skyloft and Linux, CFS slightly outperforms RR due to its compensation for blocked threads [56]. Since Skyloft can handle timer interrupts at μs scale with low overhead, it achieves significantly lower wakeup latencies compared to Linux schedulers , even when Linux CFS is tuned to reduce wakeup latency.

Similar to CFS, the Earliest Eligible Virtual Deadline First (EEVDF) scheduler aims to provide a fair share of CPU time to all runnable tasks, introduced in Linux v6.6 [9, 11, 54]. However, unlike CFS, which relies on various empirical heuristics, EEVDF uses a well-defined mechanism based on a lag value to decide which task to schedule. We implemented the EEVDF scheduling policy in Skyloft and compared it with CFS. As shown in Figure 5, Skyloft's EEVDF outperforms Skyloft's CFS. We also evaluated Linux EEVDF on Linux v6.8 (the latest Hardware Enablement (HWE) kernel on Ubuntu 22.04 LTS). Linux EEVDF performs similarly to CFS (with both default and tuned parameters), as the wakeup latency is constrained by the timer frequency.

Additionally, we evaluated the impact of different preemption frequencies on wakeup latency. As shown in Figure 6, the wakeup latency in schbench is roughly proportional to the time slice: smaller time slices result in lower wakeup latency.

5.2 Synthetic Workload Comparison

We evaluate a centralized scheduling policy from recent research, Shinjuku [30], designed to support μs-scale latency for network applications. In Shinjuku, a spinning dispatcher

	CONFIG_HZ (Linux) / TIMER_HZ (Skyloft)	min_granularity (CFS) / base_slice (EEVDF)	time_slice (RR) / sched_latency (CFS)
Linux RR (default)	250	-	100ms
Linux CFS (default)	250	3ms	24ms
Linux CFS (tuned)	1,000	12.5μs	50μs
Linux EEVDF (default)	1,000	3ms	-
Linux EEVDF (tuned)	1,000	12.5μs	-
Skyloft RR	100,000	-	50μs
Skyloft CFS	100,000	12.5μs	50μs
Skyloft EEVDF	100,000	12.5μs	-

Table 5. Parameters for scheduling policies. For Linux, the maximum configurable timer interrupt frequency is 1000 Hz.

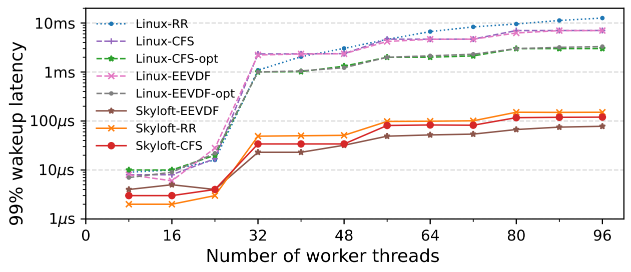


Figure 5. Schbench performance with different scheduling policies. The corresponding parameters of each test are shown in Table 5.

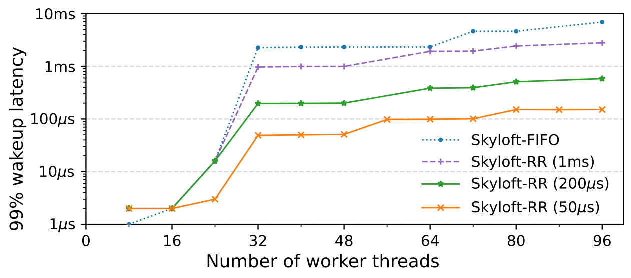


Figure 6. Schbench wakeup latency with different RR time slices. Skyloft-FIFO represents an infinite time slice and no preemption will occur.

runs on a dedicated core and maintains a global queue. The dispatcher manages incoming requests and dispatches them to idle workers running on other cores. Each request is preempted by the dispatcher and returned to the global queue if it exceeds a limited quantum. We compare four implementations of this policy:

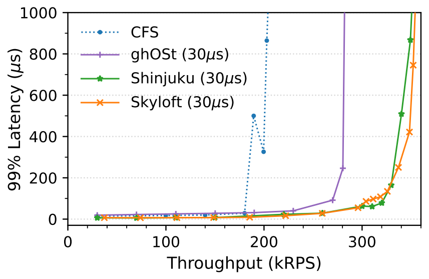
(1) The original Shinjuku system, which leverages posted interrupts to preempt workers. We run Shinjuku on Linux 4.4 due to compatibility issues with Dune on newer versions.
(2) The ghOSt-Shinjuku global agent, which makes scheduling decisions by committing transactions [26]. We run ghOSt on Linux 5.11 as porting ghOSt to newer versions requires significant effort.
(3) The Skyloft-Shinjuku dispatcher, which sends a user interrupt to preempt a worker.
(4) A non-preemptive version running on Linux CFS.

Single workload. As described in the ghOSt paper [26], we implement a load generator and request handlers for synthetic workloads. We generate a workload consisting of 99.5% short requests and 0.5% long requests, with execution times of 4μs and 10ms, respectively. One core is used for the load generator and dispatcher (except in the case of Linux CFS), and 20 cores are used as workers.

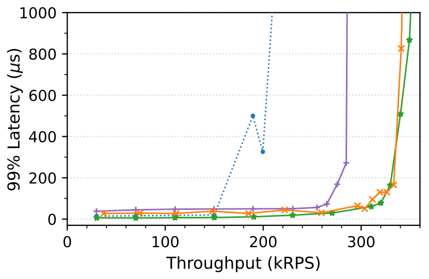
Figure 7a presents the performance of different implementations. The choice of preemption quantum has a significant impact on tail latency and maximum throughput. We find that a preemption quantum of 30μs yields the best results. While higher preemption frequencies can further reduce tail latency, they also increase the overhead from interrupt handling, which reduces maximum throughput. Linux CFS achieves 58.7% of the maximum throughput compared to Skyloft. This is because Linux CFS is designed for fairness rather than latency-sensitive applications. Skyloft and Shinjuku show similar performance, as both use low-overhead preemption mechanisms. In contrast, ghOSt performs worse, with maximum throughput reaching only 80.1% of Skyloft (at 30μs), and the 99% tail latency at low load is three times higher than that of Skyloft. This is due to ghOSt relying on Linux kernel threads, which incur significant overhead from context switches during preemption. Additionally, ghOSt's performance is affected by frequent communication between the user agent and the kernel.

Multiple workloads. A general-purpose scheduling framework should support scheduling multiple applications to improve CPU utilization. To demonstrate Skyloft's ability to handle multi-application scheduling, we evaluate a high-priority, latency-sensitive application co-located with a low-priority batch application on the same machine [26].

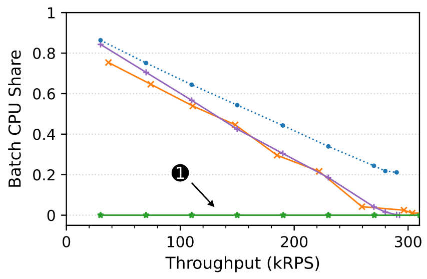
We implement Shenango's core allocation strategy [45] with the centralized policy mentioned earlier. Based on task queuing times, the dispatcher periodically checks if the global queue of requests is congested. If congestion is detected, the cores running the batch application are preempted and yield



(a) Tail latency for a dispersive load.



(b) Tail latency for a dispersive load co-located with a batch application.



(c) CPU share of the batch application.

Figure 7. Skyloft implements a centralized scheduling policy and ensures CPU sharing without compromising tail latency. Shinjuku ❶ is unable to allocate resources to run another application effectively, it cannot support the batch application (with zero CPU share).

to tasks from the latency-sensitive application. The strategy aims to allocate idle cores to the batch application when the latency-sensitive application is not busy, maximizing CPU utilization while ensuring low tail latency under peak loads.

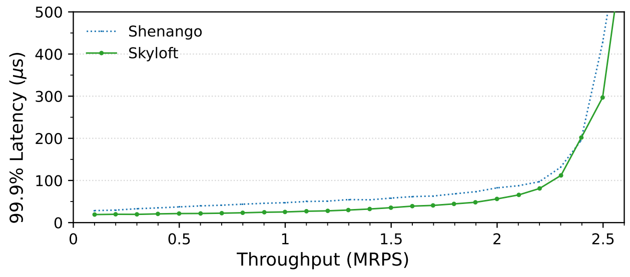
Figure 7b shows that we can achieve the same tail latency as the original centralized policy. Compared with ghOSt, Skyloft increases maximum throughput by 19% and reduces 99% tail latency by 33%. More importantly, Figure 7c demonstrates that we achieve a similar CPU share for the batch application at different load levels compared to both Linux and ghOSt.

5.3 Real-World Applications

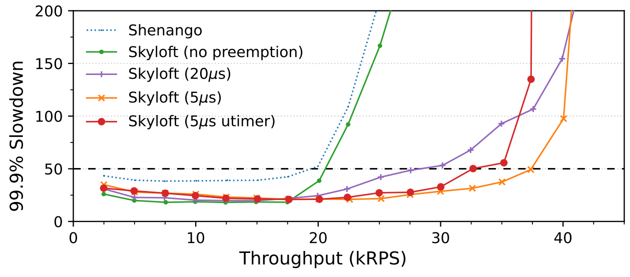
Skyloft can flexibly support scheduling policies for real-world applications. Skyloft leverages the approach described in Section 3.5 to provide kernel-bypass network functionality. We evaluate two popular applications: Memcached [17] (v1.5.6), and a UDP-based RocksDB [42] (v6.15.5) server, each handling different types of network requests. Memcached is an in-memory key-value store with light-tailed workloads, while RocksDB is a persistent key-value store with heavy-tailed workloads. A separate machine is used as a client, running an open-loop load generator to send requests to the Memcached or RocksDB server, following a Poisson arrival process.

Memcached. For Memcached, the client generates the USR workload [3], consisting of 99.8% GET requests and 0.2% SET requests. To saturate the server, we configure 4 worker cores for both Skyloft and Shenango. We implement a work-stealing policy similar to Shenango [45] to achieve load balance and low tail latency for light-tailed workloads. As shown in Figure 8a, Skyloft performs comparably to Shenango, within 2% of its maximum throughput. At low loads, Skyloft exhibits slightly lower tail latencies than Shenango, as Shenango incurs additional overhead due to frequent core adjustments, yielding, and wake-ups for a single application.

RocksDB server. We configure the client to generate a workload with a bimodal distribution, consisting of 50% GET and 50% SCAN requests, with processing times of 0.95μs and 591μs, respectively. We allocate 14 worker cores for both Skyloft and Shenango to ensure the server is fully saturated. Without any modifications to the scheduler, we enable Skyloft's timer-interrupt handler for the work-stealing policy. Skyloft improves the tail latency of short requests by preempting each core using timer interrupts. We compare the throughput at a target 99.9% *Slowdown*, which represents the ratio of total response time to the original service time. By using tail *Slowdown* instead of latency, we effectively evaluate the dispersive workload (where absolute latencies of requests vary significantly) against a common Service Level Objective (SLO). As shown in Figure 8b, Skyloft supports preemption quanta as low as 5μs. Due to its lack of μs-scale preemption, Shenango exceeds the slowdown SLO much earlier for the heavy-tailed workload. Skyloft sustains 1.9× more load than Shenango at a quantum of 5μs for a target slowdown of 50×.



(a) USR workload (99.8% GETs, 0.2% SETs) of Memcached.



(b) Bimodal workload (50% GETs, 50% SCANs) of RocksDB Server.

Figure 8. Skyloft flexibly supports both light- and heavy-tailed workloads for real-world applications.

	Send	Receive	Delivery
Signal	1,224	6,359	5,274
Kernel IPI	437	1,582	1,345
User IPI	**167**	**661**	**1,211**
User IPI (cross NUMA nodes)	178	883	1,782
`setitimer`	—	5,057	—
User timer interrupt	—	**642**	—

Table 6. Preemption mechanism comparison (cycles).

We also use a dedicated core to emulate a timer (called utimer) and send IPIs to 13 worker cores with a 5μs quantum. The results show that, compared to local APIC timer interrupts, software-emulated timer interrupts reduce performance by 13% due to the absence of one worker core.

5.4 Microbenchmarks

User-space preemption overhead. We measure the overhead of user interrupts and compare it with other notification mechanisms. The sender and receiver threads are bound to different cores on the same socket. The receiver's event handler is a no-op. In this setup, both Linux signals and kernel IPIs need to access the local APIC to send interrupts. We configure the local APIC in x2APIC mode, which is accessed through the MSR (Model-Specific Register) instead of MMIO (Memory-Mapped I/O). Table 6 shows the results, including the time spent by the sender on the send operation, the time spent by the receiver on event handling (including context saving and restoring), and the delivery latency. Linux signal has the highest overhead, which involves several switches between user and kernel. Neither sending nor receiving user IPIs involves a privilege level change, resulting in lower overhead than other mechanisms.

We also measure the overhead of user timer interrupts and compare it with the traditional signal-based user timer (`setitimer` in Linux). A timer interrupt can be generated either by the local APIC timer or from another core (utimer, as described in §5.3). The lower part of Table 6 shows the results. The signal-based timer requires several switches between user and kernel space, resulting in significantly higher overhead compared to the user timer interrupt. The overhead of simulating a timer on another core is similar to that of user IPIs but increases when signaling cores on different NUMA nodes. Therefore, using extra cores as timers limits scalability.

It is also worth noting that, as described in Section 3.2, receiving user-mode device interrupts requires an additional `senduipi` instruction (with UPID.SN = 1) in the interrupt handler, which takes approximately 123 cycles. Even with this additional operation, the overhead of receiving user timer interrupts remains lower than that of user IPIs, as timer interrupts do not incur the overhead of cross-core communication.

	pthread	Go	**Skyloft**
Yield	898	108	**37**
Spawn	15,418	503	**191**
Mutex	28	**25**	27
Condvar	2,532	262	**86**

Table 7. Threading operation comparison (ns).

User-level threading overhead. Skyloft implements low-overhead thread operations through its user-level thread library. We evaluate the time required for several common thread operations in Skyloft and compare them with native Linux kernel threads (pthread) and user-level threads in the Go programming language [22]. The results are shown in Table 7. Skyloft achieves the lowest overhead for almost all thread operations compared to other mechanisms.

We also measured the time for Skyloft's inter-application thread switching, which is 1,905ns. When switching between threads of different applications, Skyloft requires additional operations in the kernel, such as suspending and waking up kernel threads and manipulating the kernel's runqueue to comply with the binding rule described in §3.3.

For Linux, the thread-switching time is 1,124ns if both threads are *runnable*, and 2,471ns if one thread needs to wake up another (e.g., in inter-process communication). Further optimizations could provide a direct kernel thread context-switching operation in Skyloft without the need to manipulate the kernel's runqueue.

6 Discussion

Kernel-bypass timer reset. Currently, the timer is set through the kernel before running applications. We can enable applications to dynamically reset the timer by mapping the local APIC address range to the application's address space. However, a potential risk of this emulation is that the core may send an IPI to other cores, which should not be allowed in non-privileged mode. To address this, a new instruction could be introduced, allowing applications to modify the timer deadline directly. Recently, a similar hardware feature, called User-Timer Events, has been introduced in the programming reference for Intel's future processors [14].

Shared memory protection. Scheduling multiple applications in user space requires sharing data between different address spaces, which can raise safety concerns. For instance, malicious applications could tamper with the shared runqueue to manipulate scheduling decisions. However, since the user thread's context and stack are not shared between applications, this does not lead to memory-related attacks on other applications. Hardware mechanisms like Intel MPK (Memory Protection Keys) [46] can be employed for isolation. R/W permissions can be configured through a guardian code before entering the application. The risk arises from the fact that Skyloft's library and the application are linked together in the same address space, meaning the application could potentially modify permissions using the `WRPKRU` instruction. Prior works have addressed this issue to some extent [23, 24, 36]. Other mechanisms, such as VMFUNC, are also possible.

Peripheral interrupts. Skyloft provides a unified method for delegating peripheral interrupts to user space (§3.2). Both external interrupts via the I/O APIC and Message Signaled Interrupts (MSI) via the local APIC can be handled in user space. Additionally, kernel-bypass I/O drivers can be implemented with this mechanism, avoiding the need for polling or kernel signaling.

Blocking events. Skyloft enforces the Single Binding Rule, preventing multiple active kernel threads on the same core (§3.3). An active kernel thread can be blocked by events such as page faults or I/O operations. Skyloft can integrate various techniques to minimize performance impact. For active blocking (e.g., I/O syscalls), asynchronous syscalls like `aio` and `io_uring` can be used, or user-space file I/O can be implemented with SPDK. For passive blocking (e.g., page faults), `userfaultfd` can monitor the blockage on a non-isolated core and reschedule other kernel threads from different applications on the blocked core without violating the Single Binding Rule.

7 Related Works

User-level threading. To enable fast switching between application tasks, user-level threading libraries such as Capriccio [57], QThreads [58], Fred [32], and others [6, 8, 38, 44, 52] multiplex user threads atop one or more kernel (native) threads. With Skyloft providing POSIX-compatible threading APIs and lightweight user-space scheduling facilities, these libraries could be simplified by adopting a 1:1 thread mapping model, instead of the more complex M:N model.

Core allocation. Latency can generally be reduced by provisioning more cores for LC applications, but static partitioning may result in wasted resources under fluctuating workloads. Systems like Arachne [49], Shenango [45], Caladan [21], and others [40] dynamically reallocate cores between LC and BE applications, balancing latency and throughput. Skyloft adopts core allocation policies similar to Shenango [45] for multiple applications, while offering additional support for more scheduling policies to achieve better latency for both light- and heavy-tailed workloads.

Microsecond-scale preemption. Non-preemptive scheduling policies can suffer from head-of-line blocking in the presence of high-dispersion workloads, requiring frequent preemption to maintain lower tail latencies. Systems like Shinjuku [30], Concord [28], and LibPreemptible [35] achieve μs-scale preemption, with LibPreemptible being the most similar to Skyloft in terms of preemption mechanisms. However, LibPreemptible only supports preemptive scheduling within a single process, whereas Skyloft is capable of scheduling across multiple processes, thereby improving CPU efficiency. While Perséphone [16] implements a non-preemptive policy, it reserves a small number of cores specifically for short requests, sometimes outperforming preemptive systems under high-dispersion workloads. However, Perséphone relies on applications to group incoming requests and is not work-conserving[1].

Kernel-bypass and dataplane OSes. Emerging faster network and storage devices demand shorter software I/O paths, beyond what general-purpose kernels can provide. Libraries like DPDK [18], SPDK [12], mTCP [29], and Strata [33] move the driver and protocol stack to user space, tightly integrating them with the application. Dataplane operating systems separate the dataplane from the control plane to optimize throughput and latency. IX [5], Arrakis [47], and Demikernel [60] focus on optimizing shorter datapaths, while ZygOS [48] adopts work stealing for load balancing. Skyloft leverages DPDK and incorporates datapath optimizations from previous works to build a high-performance user-space networking system, in alignment with its μs-scale scheduling capabilities.

User-defined kernel scheduling. In addition to building scheduler systems from scratch, several approaches aim to improve existing kernel schedulers. Scheduler Activations [2] provides APIs that allow applications to coordinate with kernel thread scheduling. ghOSt [26] delegates kernel scheduling decisions to user-space agents, while Enoki [43] uploads user-provided schedulers to run inside the kernel. Syrup [31] coordinates thread and network scheduling with ghOSt and eBPF support. Plugsched [37] is an SDK that enables live updating of the Linux kernel scheduler. Like other user-space schedulers, Skyloft can achieve more efficient scheduling by bypassing kernel overhead.

8 Conclusion

Skyloft demonstrates significant advancements in user-space scheduling, offering a robust framework that leverages user-mode interrupt mechanisms to handle hardware timer interrupts directly in user space. This architecture enables μs-scale preemption, improving the performance of latency-sensitive applications. With these preemption capabilities, Skyloft provides a flexible scheduling framework that supports various scheduling policies.

Acknowledgments

We thank the anonymous reviewers and our shepherd Prof. KyoungSoo Park for their constructive suggestions. The authors from Tsinghua University are all in the Department of Computer Science and Technology. This work was supported by Quan Cheng Laboratory (QCLZD202305), Natural Science Foundation of China (62141216). Correspondence to: Yu Chen (yuchen@tsinghua.edu.cn), Kang Chen (chenkang@tsinghua.edu.cn).

References

[1] Work-conserving scheduler. https://en.wikipedia.org/wiki/Work-conserving_scheduler, 2024.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 95–109, New York, NY, USA, 1991. Association for Computing Machinery.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[8] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477. USENIX Association, July 2020.

[9] Jonathan Corbet. An EEVDF CPU scheduler for Linux. https://lwn.net/Articles/925371/, 2023.

[10] Jonathan Corbet. Deferred scheduling for user-space critical sections. https://lwn.net/Articles/948870/, 2023.

[11] Jonathan Corbet. Completing the EEVDF scheduler. https://lwn.net/Articles/969062/, 2024.

[12] Intel Corporation. Introduction to the Storage Performance Development Kit (SPDK). https://www.intel.com/content/www/us/en/developer/articles/tool/introduction-to-the-storage-performance-development-kit-spdk.html, 2016.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, chapter User Interrupts, pages 7.1–7.8. 2023.

[14] Intel Corporation. *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*, chapter User-Timer Events and Interrupts, pages 13.1–13.3. Number 319433-052. March 2024.

[15] Microsoft Corporation. Introduction to receive side scaling. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2023.

[16] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Dormando. memcached - a distributed memory object caching system. https://memcached.org, 2018.

[18] DPDK Project. DPDK. https://www.dpdk.org, 2023.

[19] Matt Fleming. A survey of scheduler benchmarks. https://lwn.net/Articles/725238/, 2017.

[20] Matt Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157/, 2017.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[22] Google. The Go Programming Language. https://go.dev, 2024.

[23] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020.

[24] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[25] Tejun Heo. sched: Implement BPF extensible scheduler class. https://lwn.net/Articles/951156/, 2023.

[26] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, Boston, MA, July 2018. USENIX Association.

[28] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 466–481, New York, NY, USA, 2023. Association for Computing Machinery.

[29] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

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

[31] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Martin Karsten and Saman Barghi. User-level threading: Have your cake and eat it too. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), may 2020.

[33] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G. Edward Suh, Kostis Kaffes, and Christina Delimitrou. LibPreemptible: Enabling fast, adaptive, and hardware-assisted userspace scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–936, 2024.

[36] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. Fast core scheduling with userspace process abstraction. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles*, SOSP '24, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Teng Ma, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. Efficient scheduler live update for linux kernel with modularization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 194–207, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 110–121, New York, NY, USA, 1991. Association for Computing Machinery.

[39] Chris Mason. schbench. https://kernel.googlesource.com/pub/scm/linux/kernel/git/mason/schbench/+/refs/tags/v1.0, 2023.

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

[41] Sohil Mehta. x86 user interrupts support. https://lwn.net/Articles/869140/, 2021.

[42] Meta Platforms, Inc. RocksDB - A persistent key-value store. https://rocksdb.org, 2022.

[43] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. Enoki: High velocity linux kernel scheduler development. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 962–980, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Jun Nakashima and Kenjiro Taura. MassiveThreads: A thread library for high productivity languages. In Gul Agha, Atsushi Igarashi, Naoki Kobayashi, Hidehiko Masuhara, Satoshi Matsuoka, Etsuya Shibayama, and Kenjiro Taura, editors, *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, pages 222–238, Berlin, Heidelberg, 2014. Springer.

[45] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[46] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[47] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[48] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, October 2018. USENIX Association.

[50] Rust Team. async/.await Primer. In *Asynchronous Programming in Rust*, 2023.

[51] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 498–514, New York, NY, USA, 2023. Association for Computing Machinery.

[52] Shumpei Shiina, Shintaro Iwasaki, Kenjiro Taura, and Pavan Balaji. Lightweight preemptive user-level threads. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 374–388, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 33–46, USA, 2010. USENIX Association.

[54] I. Stoica and H. Abdel-Wahab. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical report, USA, 1995.

[55] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. The lock holder and the lock waiter pre-emption problems: nip them in the bud using informed spinlocks (I-Spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 286–297, New York, NY, USA, 2017. Association for Computing Machinery.

[56] The kernel development community. CFS Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, 2024.

[57] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 268–281, New York, NY, USA, 2003. Association for Computing Machinery.

[58] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[59] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations Research*, 60(5):1249–1257, 2012.

[60] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.