LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling

Yueying Li
Cornell University
Ithaca, NY

Nikita Lazarev
Massachusetts Institute of Technology
Cambridge, MA

David Koufaty
Intel Labs
Hillsboro, OR

Tenny Yin
Cornell University
Ithaca, NY

Andy Anderson
Intel Labs
Hillsboro, OR

Zhiru Zhang
Cornell University
Ithaca, NY

G. Edward Suh
Cornell University
Ithaca, NY

Kostis Kaffes
Columbia University
New York, NY

Christina Delimitrou
Massachusetts Institute of Technology
Cambridge, MA

***Abstract*—Modern cloud applications are prone to high tail latencies since their requests typically follow highly-dispersive distributions. Prior work has proposed both OS- and system-level solutions to reduce tail latencies for microsecond-scale workloads through better scheduling. Unfortunately, existing approaches like customized dataplane OSes, require significant OS changes, experience scalability limitations, or do not reach the full performance capabilities hardware offers. We propose LibPreemptible, a preemptive user-level threading library that is flexible, lightweight, and scalable. LibPreemptible is based on three key techniques: 1) a fast and lightweight hardware mechanism for delivery of timed interrupts, 2) a general-purpose user-level scheduling interface, and 3) an API for users to express adaptive scheduling policies tailored to the needs of their applications. Compared to the prior state-of-the-art scheduling system Shinjuku, our system achieves significant tail latency and throughput improvements for various workloads without the need to modify the kernel. We also demonstrate the flexibility of LibPreemptible across scheduling policies for real applications experiencing varying load levels and characteristics.**

I. Introduction

A large portion of the world's computation is now hosted on either public or private cloud infrastructures. This has brought on several changes to the way cloud applications are designed, including moving away from monolithic services to inter-dependent microservices and event-driven serverless frameworks [21], [36]–[38], [61], [63]. These software design approaches enable high concurrency of fine-grained tasks, with thousands of user requests executing at any point in time. The fine-grained nature of these tasks also allows them to be coscheduled on the same physical host, which facilitates multi-tenancy and improves the cloud's resource efficiency. At the same time, applications must meet service-level objectives (SLOs), often defined in terms of tail latency.

Furthermore, datacenter suffers from *massive thread over-subscription*. We observe serious thread over-subscription consistently from recent Google traces [58] across widely-used applications, where more than 50 threads on average or sometimes around 500 threads can be scheduled to each core (Table I). To achieve higher CPU efficiency and low tail latency, one necessary precondition is fine-grained, scalable, adaptive, and low-overhead preemptive scheduling.

App (code name)	# of threads	# of cores	Threads/core
charlie	4842	10	484
delta	300	4	75
merced	5470	110	50
whiskey	1352	8	169

TABLE I: Datacenter thread oversubscription from four widely used applications in Google [58].

Preemption enables more fine-grained resource sharing among workloads and requests. Without preemptions, short requests can get stuck behind long requests, causing head-of-line (HoL) blocking. The lack of fine-grained preemptions results in prior work experiencing high latencies under long-tailed request service time [22], [27], [29], [49], [56].

Unfortunately, past proposals for preemptive scheduling are not suitable for microsecond-scale workloads running on exisiting cloud platform, for the following reasons. 1) Preemptive user level threads based on regular interrupts still incur high overheads during user- and kernel- level context switches [10], [24]; if the minimum time slice is 5ms and there are 200 threads on average per core, the scheduler cycle will be increased to 1 seconds, significantly increasing tail latency. 2) Optimizing preemption overhead, like Shinjuku [44]'s usage of Posted IPI can reduce the time slice; however, assigning the programmable interrupt controllers (APICs) to their runtime introduces security concerns for a shared cloud environment. Similarly, using APICs limits the approach's scalability, as it only supports a small number of logical cores.

An alternative scheduling approach to achieving low latency scheduling without preemption is by using request-specific knowledge. However, such information is difficult to obtain beforehand. Additionally, the duration of these requests' execution time has no upper bounds. This makes it challenging to apply rules, such as SRPT (Shortest Remaining Processing Time [40]) or any other rules that give priority to short requests, especially when operating at the microsecond scale [28].

We propose *LibPreemptible*, a hardware-assisted user-level threading library that enables fine-grained, configurable, and scalable preemptions in the cloud. LibPreemptible is built on top of user interrupt (UINTR), a new hardware capability in

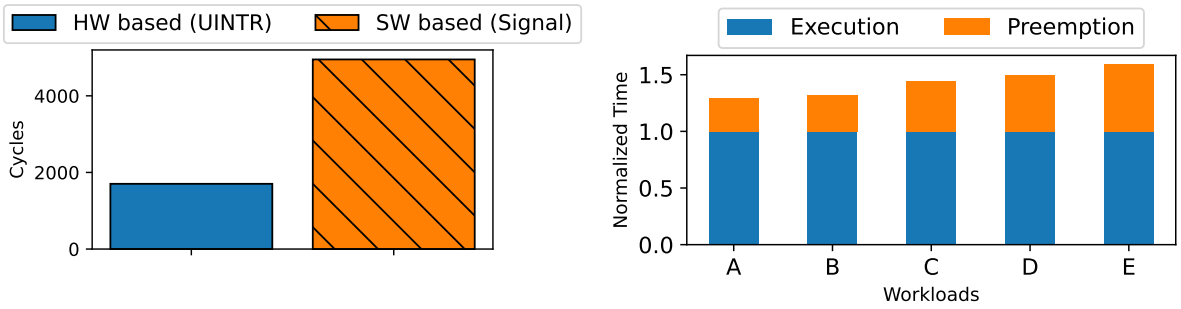


Fig. 1: Left: Performance gap between software- and hardware-based IPC delivery. Right: Normalized overhead of preemption w.r.t lean execution time for different microsecond-scale workloads running on Shinjuku (ranked by workload dispersion).

the Intel Xeon Scalable Processor codenamed Sapphire Rapids (SPR). UINTR is a low-overhead communication mechanism that allows application threads to directly send each other interrupts bypassing the kernel. However, native UINTR is not a good fit for dynamic workloads without knowledge of request service times. We leverage UINTR to design a user-level threading library that enables configurable and fine-grained periodic interrupts, and adjusts scheduling timeslices on the fly using statistical tests. LibPreemptible also introduces fast and accurate user timers, which can be used by a wide range of applications to easily build customizable scheduling policies.

We evaluate LibPreemptible via microbenchmarks, as well as synthetic and real applications widely used in private and public clouds today. We show that it achieves better performance, scalability, isolation, and flexibility compared to prior work. The tail latency with LibPreemptible is 10× better compared to Shinjuku [44], the previous state-of-the-art system, across various workloads. In an environment where applications time-share CPU cores, where a latency-critical application shares resources with a best-effort (BE) compression workload, LibPreemptible achieves on average 10% higher throughput for the BE job, while maintaining the same 99% tail latency SLO for the latency-critical application.

LibPreemptible's main contributions are:

- First paper to leverage a new hardware mechanism (UINTR) for fine-grained user-level interrupts, that scales much better than previous solutions and faster than traditional interrupts' millisecond timescales [24].
- Designing a new abstraction with user-level timers with fine-grained deadlines, called LibUtimer, accounting for the diverse needs of different microsecond-scale applications with a wide spectrum of scheduling policies.
- Running entirely outside the kernel and only requiring a kernel modification to regularly enable UINTR [14]. Applications using LibPreemptible can coexist with traditional applications and common cloud system software / hardware stacks with only a few lines of code change.

This paper focuses not only on just putting UINTR into use, but more on how we should build new abstractions and delegate scheduling decisions to user-level applications, offering adaptive policies that best meet their needs.

II. Motivation

Below we demonstrate the characteristics of cloud workloads that motivate the need for a preemptive user-level threading library that operates adaptively, at microsecond scale.

A. The Need for a µs-scale Preemption Mechanism

Previous work has demonstrated that preemption in user space is key to achieve low tail latency [44], [54]. Synchronous mechanism offers relatively low latency, but are not practical as they require the worker thread to be idle (for example, blocked in the kernel waiting for an eventfd, or polling or mwaiting on shared memory). Preemption requires the ability to asynchronously receive an event during execution of a user-level thread to trigger the context switch.

There are two approaches to deliver asynchronous events to drive preemptions; however, the overhead, lack of precision, modifications and security concerns often overshadow the benefit on performance.

One approach to delivering the asynchronous event is to use signals [10], [24], [62]. Typically, a thread creates a kernel timer at a specific future time and the kernel delivers a signal to the thread when the timer expires. In some cases, threads additionally communicate using signals to propagate the preemption event to other threads. However, a kernel timer is susceptible to kernel jittering and the signal overhead is significant and scales poorly as the number of threads increases due to signal contention (Section V-B). These reasons limit the applicability of this approach, and result in relatively large preemption timers, as in the case of the Go programming language [10], which recently introduced preemption to prevent starvation at a 10ms granularity.

Another approach is to use native interrupts, which provide a low latency mechanism for inter-processor communication (IPC). However, interrupts are restricted to kernel software due to its privileged nature, so it's challenging to use them directly within application threads. Furthermore, using interrupts would require system calls at sender threads and signals at the receiver threads. Prior works, such as Shinjuku [44],

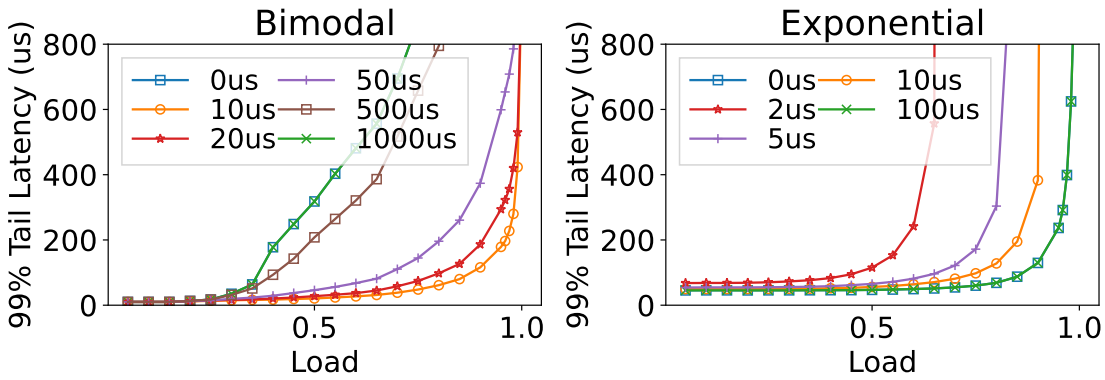


Fig. 2: Tail latency with different preemption quanta on bimodal (left) and exponential (right) workloads. Bimodal: 99.5% of requests are short (10us) and 0.5% are long (1000us). Exponential: mean as 10us. The preemption overhead, including sender and receiver cost, is about 1us across all settings.

have circumvented this overhead at the sender by mapping the physical APIC to the sender, which is untenable in the public cloud environment. Receiving events in user space still requires kernel-mediation (for example, to notify workers threads of the interrupt using signals) [1].

We profiled the overall CPU time spent in preemption vs. execution (normalized to execution time) with the time quantum chosen to offer the best tail latency for various latency-critical workloads running on baseline system [44]. It can be observed that preemption overhead is significant, especially for micro-second scale workloads with high dispersion (Figure 1 Right). Figure 1 Left shows the gap between software-based IPC using signals, and hardware-assisted IPC using UINTR. While previous software based optimizations achieve performance benefits compared to regular interrupts, there is still a gap to the latency of delivering and handling an interrupt with hardware.

B. The Need for Adaptive Preemption

Datacenter workloads have a wide variation in request processing times, especially at the microsecond level. Factors such as high load, imbalanced requests, low CPU processing speed due to interference, saturation, rate limiting, etc as well as high cache miss ratios due to skewed key distributions, and more, can cause high tail latencies in systems like Memcached [9].

Imbalanced request times can lead to head-of-line (HoL) blocking where short requests are blocked by longer ones, making adaptive scheduling policies necessary. The tail-optimal scheduling policy changes with the request service time distribution. To arrive at the best policy, we need to adapt to the workload. For example, in heavy-tailed workloads, c-FCFS (centralized first come first serve) scheduler with preemption is better than PS (processor sharing), while the throughput depends on the length of the time quantum. A time quantum that is too long causes HoL (Head of Line) issues, while one that is too short results in a decrease in CPU efficiency.

Figure 2 illustrates the importance of adaptive preemptive scheduling policies. With two typical service time distributions running on 16 cores and different preemption quanta[2], lower preemption quanta give better tail latency in heavy-tailed workloads (e.g. bimodal, until the time quantum becomes too small), wheras higher time quanta give better tail-latency for light tailed workloads such as exponential workloads. The tail-optimal scheduling policy varies with the request service time distribution [66].

While aggressive preemptive scheduling can help reduce tail latency in highly dispersive workloads, it's not always necessary under lower loads or lighter tailed workloads.

Current scheduling primitives (including OS or user-level green threads) do not expose such decisions to users for the following reasons. First, it was believed that delegating the scheduling decisions to kernel is sufficient, because of the privileged nature of kernel. However, with the tighter demand on workload latency, the traditional linux kernel is no longer a final rescue. Second, even if the application developers have the intention to do so, there is no lightweight mechanism to achieve this due to user-kernel context switching overhead.

III. User Interrupt, Limitation and Design

We first discuss the background of user interrupt, the challenges of using them, and design objectives of LibPreemptible, then present how we arrive at the right abstraction for fine-grained user-level interrupt delivery which the library is built upon, and finally show a framework for building application-specific adaptive request schedulers based on LibPreemptible.

A. User Interrupts

The kernel is used in almost all instances of communication across privilege boundaries, and it includes notification mechanisms based on hardware interrupts, signals, pipes, files,

[1]Other interrupt optimizations explored in Shinjuku (to avoid VM exits at sender) in a virtualized environment are becoming irrelevant as hardware vendors introduce support for inter-processor interrupts (IPI) virtualization [8].

[2]0us time quantum means no preemption.

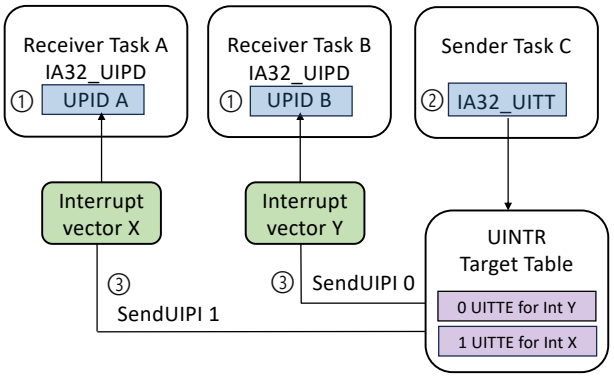


Fig. 3: User Interrupt overview. The mechanism can be decomposed into two phases, setup phase and delivery phase. In the setup phase, 1) receivers set up `UPID` and register the handler, and return `uintr_fd` to the sender; 2) senders use `uintr_fd` to allocate `UITTE` and get the UIPI index. In the delivery phase, 3) senders call `SENDUIPI`.

```
// Sender API
// Receive FD via inheritance or UNIX domain sockets
uipi_handle = uintr_register_sender(uintr_fd,flags);
int uintr_unregister_sender(uintr_fd, flags);
// x86 Instruction
void _senduipi(uipi_handle);
// Receiver API
// Register User Interrupt Handler
int uintr_register_handler(handler_func, flags);
int uintr_unregister_handler(flags);
// Create an fd representing the vector - priority
uintr_fd = uintr_create_fd(vector, flags);
void __attribute__ ((interrupt))
    u_handler(struct __uintr_frame
    *frame, unsigned long vector) {
    write(STDOUT_FILENO, "User Interrupt!\n", 16);
    uintr_received = 1;
}
```

Fig. 4: Native UINTR API.

etc. The only exception is when applications use memory for communication, which requires the receiver to poll, resulting in resource inefficiency.

User interrupt is a new hardware capability that delivers low overhead preemption without kernel mediation. User interrupts avoid transitions through the kernel, by enabling the possibility of sending and handling hardware interrupts directly in non-privileged applications in the user space [17]. While their setup requires kernel enabling, in runtime, delivery of a user interrupt to a running thread results in nearly the same latency as that of a regular interrupt. User interrupt also works with inter-processor interrupt virtualization, which allows guest software to send and receive UINTR without the cost of a VM Exit.

The interrupt delivery happens in three steps (Figure 3): 1) Each receiver stores interrupt information in a memory location defined by User Posted Interrupt Descriptor (UPID). 2) Each sender requires a table targeting receivers. The per-thread User Interrupt Target Table (UITT) contains the target UPID and interrupt vectors to deliver. User interrupts have 64 interrupt vectors per thread. 3) SENDUIPI takes an index to the UITT table to deliver an interrupt, and the UPID holds all the information necessary to post the request and send a notification to the CPU. The CPU control flow can be modified by a user interrupt, also called user interrupt delivery. It can be triggered by either the processor state or the state of memory data structures, managed by the processor and OS.

If the receiver is runnable, the UPID records the request and suppresses the notification. If blocked, the UPID uses an ordinary interrupt to unblock the receiver and inject the user interrupt.

B. Challenges and Abstractions for Preemption

To fully realize the benefit of hardware support for user-level interrupts, we design LibPreemptible, a scheduling library that achieves programming flexibility and dynamic time sharing built on top of UINTR. Native User interrupt (including unmodified UINTR instruction set architecture, and kernel APIs, shown in Figure 4) is not a good fit for micro-second scale scheduling for modern datacenter applications.

- First, the lack of exposure to application-level info makes it hard to dynamically change interrupt decisions based on runtime application characteristics. This requires defining an appropriate interface for users to express application requirements.
- Second, UINTR employs the security model of eventFD, which may cause DoS across untrusted processes. This requires defining an appropriate protocol to communicate scheduling decisions across untrusting processes without violating their respective constraints.

Based on these challenges, we discuss how to arrive at the right abstraction. Realizing fine-grained and adaptive preemption is challenging. OS threads are the natural choice due to the OS support for preemption and a rich set of functionality: per-thread signal masks, CPU affinity, etc. However, their benefits are limited when optimizing tail latency due to multiple reasons, most notably ❶ granularity of kernel context switching (in ms-scale), ❷ overhead of the context switch and ❸ limited information about an application's characteristics that the OS can use towards scheduling.

Ideally, the right abstraction should satisfy four requirements. First, for each request, it should allow the scheduler to express latency requirements as deadlines, which allows the preemption or cancellation of some long requests to release resources when otherwise SLO will be violated. Second, to allow the scheduler to make wise decisions, the abstraction should enable the user-level runtime to record past request information in a generic form. Third, it needs to express application-specific deadlines, to allow the scheduler to adjust to different loads and QoS constraints. Finally, the abstraction

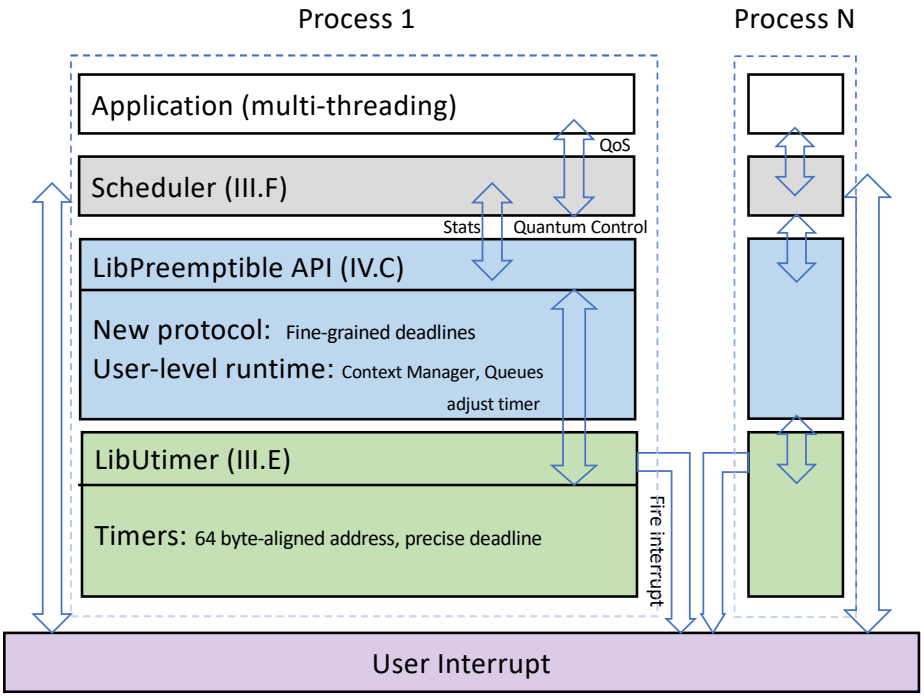


Fig. 5: LibPreemptible can be decomposed into three components: Scheduler, LibPreemptible API, and LibUtimer.

should defend against untrusted applications, e.g., they cannot preempt other applications based on how their timers are compiled.

To achieve the above requirements, we propose and build LibPreemptible, a user-level library with lightweight API and runtime support that can flexibly schedule microsecond-level requests. With 3us minimum time slice, it introduces accurate and fine-grained deadlines based on user-level threads. The key problem is to support fine-grained deadline abstraction for the dynamic and unknown workloads. To effectively tackle the challenge, we propose LibUtimer, a controllable user-level timer that implements fast and hardware-assisted preemptive timers in user space. Due to the applications' dynamic nature—both in terms of request service time and various loads, an online algorithm is proposed to achieve near-tail-optimal scheduling.

C. Design Goals

As a general purpose user-level threading library, the design of LibPreemptible should achieve four goals: enabling lightweight and fine-grained scheduling decisions, scalability, separation of mechanism and policy, and compatibility. We elaborate the goals as follows.

- ***Enabling lightweight and fine-grained preemption:*** the library should provide efficient task preemption mechanisms to allow the schedulers built on top to make scheduling decisions with time quanta in the order of *microseconds*;
- ***Scalability:*** the overhead of the preemption mechanisms should grow sub-linearly with the number of threads in the application;
- ***Separation of mechanism and policy:*** the design of LibPreemptible, which provides *mechanisms*, should be decoupled from the design of the scheduling *policies*. On the other side, the application developers should be able to express the policies on top with the *minimal API* provided by the library;
- ***Compatibility:*** as a standalone user-space library running on commodity servers, LibPreemptible should be compatible with various common cloud system software stacks.

D. Design Overview

LibPreemptible provides an API and runtime support for application developers to design efficient latency-critical request processing systems with custom scheduling policies. The library provides interfaces and APIs that are generalizable across application requirements. Figure 5 shows how different components interact with each other and with UINTR. LibPreemptible takes the scheduler's decisions on time quantum to adjust the deadlines (Quantum Control). The queues from LibPreemptible provide statistics of past requests (Stats). LibUtimer adjust timers and fire user interrupt to preempt worker threads.

In particular, Figure 6 represents how the data structures of LibPreemptible interact with each other. In the scheduler layer, for each request, LibPreemptible defines Function thread (Tn) consisting of the request Context (C) and the Deadline (D). The context encapsulates each request's state (e.g., stack, instruction pointer, etc.) at any point in time. In our current implementation, the context is based on the lightweight `fcontext`. The deadline represents the total time slice given for the requests by the scheduler according to the current

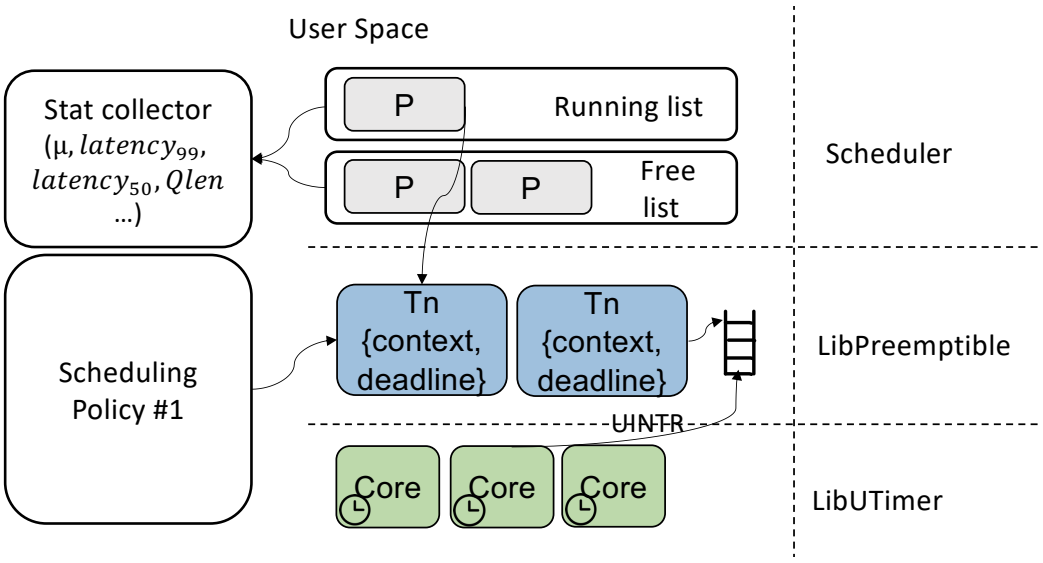


Fig. 6: A logical view of an adaptive request scheduler built upon LibPreemptible.

policy. When the deadline is reached, LibPreemptible preempts the request by storing the context in the preempted list (also defined as a part of the application) and returning control to the scheduler for the next scheduling decision.

LibPreemptible is based on another library, called LibUtimer. LibUtimer implements an accurate user-space timer that periodically fires preemption signals to all active requests when the corresponding deadlines are reached. The process of delivering these signals is crucial to the performance, scalability, and portability of the library.

E. Preemption Timer

We use UINTR to build an efficient, fast, and scalable preemption timer library LibUtimer.

A simple solution for preemption is to use kernel timers. Kernel timer comes with significant performance costs, mostly due to the kernel overheads of setting up the timer using system calls, and the signal delivery. System call overheads can be reduced by making a single system call with a periodic timer when the preemption interval is static. However, this approach does not work when the timer is dynamic or must be adjusted due to yields. Signals have a significant performance overhead that worsens as the contention in the kernel increases due to timer alignment; for example, when timers are created right after thread creation (creation-time). Prior work [62] tried to reduce this overhead by spreading the timers along the timer interval (staggered) or via a single kernel timer and signals to communicate the timer event to other threads (chaining).

However, these approaches are impractical for the fine-grained and dynamic timers needed to reduce tail latency, as we will show in Section V-B. They show poor scalability, as there is no sufficient spacing between timer events to reduce kernel contention, as the thread count increases [62]. The overhead of these timers would offset the benefit of the scheduler in terms of tail latency.

These shortcomings motivate a dedicated user-timer library based on UINTR that scales to a large number of applications issuing concurrent interrupts, reducing context switching overhead, and enabling latency-critical applications to meet their SLOs. LibUtimer is able to achieve 3us minimum time slice for scheduling, much lower than any other kernel timers. This fine-grained time unit enables a new design dimension for preemptive online scheduling. The API is described in detail in Section IV-A.

F. User-Level Scheduler

LibPreemptible exposes an API for users to easily integrate application-specific scheduling policies.

The scheduler is responsible for scheduling incoming requests across available worker threads. The scheduler is triggered either by the preemption signal from the UINTR, which is controlled by the deadline in the LibPreemptible library or when a new request arrives. It then makes the next scheduling decision based on the set of metrics (Stats) collected from the previous requests over a given time window, typically 10s (including the request load μ, median and tail latencies, the length of the local queues $Qlen$). The statistics are used to control QoS by changing the deadline or time quantum for each request individually.

In this example, we use a two-level scheduling mechanism to achieve adaptive QoS control with scheduling (Figure 6). Each worker thread is associated with a local queue of requests to be executed. The local scheduler acts in the following situations: 1) when it sees an incoming request on the queue, 2) or when a function got timed-out/preempted. Local queues for each worker manage execution of functions in a FIFO order. Preempted long-running functions go into the global "running list" together with their context. Finished functions go into the global "free list" and the contexts are reused. A

centralized running and free context list, which the scheduler maintains, help with better load balancing.

Furthermore, we can adaptively control the time quantum based on past latency distributions and real-time load. Algorithm 1 shows a time quantum controller that manages the change of user-level scheduler dynamically[3] (which is demonstrated in section V-C). During high load, the preemption interval becomes lower, ensuring timely and precise interrupt delivery to the right application thread. We set the period for the routine of changing time quanta as 10 sec, L_{high} as 90% of max load, and L_{low} as 10% of max load. The tail index ($0 \leq \alpha < 2$) is considered as a heavy tail distribution [26]. The UINTR and LibUtimer enable a minimum time quantum of 3 microseconds, a level that previous mechanisms couldn't attain due to issues with jitter. It ensures microsecond-scale tail latency. In practice, the hyperparameters can be adapted to different workloads from tracking past request data with our API.

Algorithm 1 Adaptive Time Quantum Controller

1: **Input:** past [$Qlen$, median and tail latencies], incoming load (μ)
2: **Hyperparameters:** $L_{high}, L_{low}, k_1, k_2, k_3, Q_{threshold}$
3: **Hyperparameters:** T_{min}, T_{max}
4: **Function:** f (fitting tail index with past statistics)
5: $TQ \leftarrow$ current time quantum
6: Tail index (α) $\leftarrow$ f(past [median and tail latencies])
7: **if** $\mu > L_{high}$ **then**
8: $\quad TQ \leftarrow \min\{TQ - k_1, T_{min}\}$
9: **end if**
10: **if** [$Qlen > Q_{threshold}$ or tail index (α) falls in heavy tail distribution] **then**
11: $\quad TQ \leftarrow \min\{TQ - k_2, T_{min}\}$
12: **end if**
13: **if** $\mu < L_{low}$ **then**
14: $\quad TQ \leftarrow \max\{TQ + k_3, T_{max}\}$
15: **end if**
16: **Output:** updated time quantum (TQ) =0

IV. Implementation

LibPreemptible is implemented in C, and can be deployed on existing applications with an additional dozen of lines of code (including setting deadlines when launching user level threads, arming / disarming timers, etc.), and compiled with a standard toolchain, such as `gcc`.

A. LibUtimer

LibUtimer implements fast, hardware-assisted preemptive timers in user space. The user interrupt delivers the preemption notification that will be used to trigger scheduling operations if needed. To deliver the user interrupt, LibUtimer requires that each thread registers a memory location where the time of its next preemption interrupt is located. We refer to this as the *deadline address*, and it is required to be allocated in a dedicated, naturally aligned 64 byte location to avoid false sharing. The deadline specifies the value of the timestamp counter (TSC) when the thread wants its next preemption interrupt. LibUtimer polls on the TSC, and sends a user interrupt to a thread when the TSC reaches its deadline.

The key interfaces exposed by LibUtimer are as follow:

1) `utimer_init`: This function creates a pool of timer threads, normally a single thread. This thread will use the RDTSC instruction to check the current value of the TSC and eventually send a user interrupt using the SENDUIPI instruction. It does so by periodically inspecting the value of the TSC and comparing it to the registered deadlines.
2) `utimer_register`: This function is called by application threads to specify the memory address of the deadline. It hides the low-level interactions with the kernel [14] to register the interrupt handler, create a file descriptor in the context of the application thread, etc.
3) `utimer_arm_deadline`: This function performs a memory write to set the deadline with the new time to fire the next preemption interrupt.

A timed interrupt fires periodically, causing our signal handler to be invoked, which in turn fires the user interrupt. Furthermore, for application with large thread counts and request for higher number of timers, we can opt in and use timing wheel techniques [64].

B. Context Management

We customize the fcontext library [1] for context management. The context structure consists of a machine-specific representation of the saved state, the signal mask, a pointer to the context stack, and a pointer to the context that will be resumed when this context finishes execution. The dispatcher allocates context objects and stack space for each request from a global memory pool; an application can define the size of this pool. When scheduler launches a function, a relevant context is attached to it. It is freed when the function related with the context completes execution and is returned to the pool of global contexts. When a function gets preempted, the context will be put into a global wait list.

Context can be reused by other requests once a function finished execution. The free contexts are maintained in a global free list. We further optimize the context switching overhead as done in Shinjuku [44].

C. Adaptive User-controlled API

LibPreemtible exposes a simple API that allows the creation and immediate execution of a function until the function completes, or its time slice has been reached. In either case, control is returned to the scheduler, which then can decide which function to resume.

The key interface exposed by LibPreemptible is as follow:

1) `fn_launch`: This function is used to create a preemtible function. Its execution begins immediately, and control is returned to the caller when the function completes or a timeout (the time slice) is reached. State for the preemptible function is allocated by the caller, and saved upon preemption.

[3]For heavy tail identification and definition of tail index on line 7 in the algorithm, please refer to the methodology [25], [26]. We use a similar statistical test to determine the direction of changing quanta, that can easily be implemented and enforced with our API under dynamic workloads. A feedback controller can also be integrated with our API.

```
for (i = 0; i < N; i++) {
  //  launch a preemptible function
  // and run it until the timeout
  fn_launch(my_function, &fn_args[i],
      &functions[i], timeout_us);
  // queue it for later execution if uncompleted
  if (!fn_completed(&functions[i]))
     enqueue(run_queue, &functions[i]);
}
  // round-robin scheduler
while (!empty(run_queue)) {
  // resume the function at the top of the queue
  f = dequeue(run_queue);
  fn_resume(f, timeout_us);
  // queue it for later execution if uncompleted
  if (!fn_completed(f))
     enqueue(run_queue, f);
}
```

Fig. 7: LibPreemptible example of a simple round-robin scheduler running N static user-level threads.

2) `fn_resume`: This function resumes execution of a preemptible function. As with `fn_launch`, control is returned to the caller when the function completes or a timeout is reached.
3) `fn_completed`: This function checks the status of a preemptible function and indicates its completion before the timeout expired, so that a reschedule is unnecessary.

When requests are scheduled, each runs on top of a lightweight preemptible function. When the request exceeds its timeout, its state is saved, including the registers and PC, and control returns to the worker thread. The worker thread can implement a custom scheduling policy, incorporating factors, such as the request's wait and execution time, and account for an application's SLO. Figure 7 shows a simple example of a round-robin scheduler using the LibPreemptible API.

V. Evaluation

LibPreemptible can be used by any datacenter applications written in C/C++. We deploy the system on an Intel Xeon Scalable Processor codenamed Sapphire Rapids with UINTR support (the network stack is DPDK or kernel TCP, and all machines run on Linux kernel version 5.15.0-rc1+ with turbo-boost disabled and fixed frequency scaling at 1.7GHz). The server has 56 CPUs, 112 hyperthreads and 2 sockets. Hyperthreading is enabled unless noted. For older CPUs, LibPreemptible will fall back to standard interrupts. Below we answer the following questions:

- Does LibPreemptible outperform prior preemptible scheduling techniques in terms of latency and throughput?
- Can LibPreemptible be deployed with third-party applications out-of-the-box?
- What is the overhead of LibPreemptible?
- Does adaptively setting the preemption quantum optimize performance and latency in a colocation environment?

A. Performance Comparison

We first compare LibPreemptible to Shinjuku [44] and Libinger [24], the most recent related work for preemption-based scheduling system. We first compare the three systems on synthetic loads, then on real workload. The synthetic workload is a server application where requests perform dummy work that we can control to emulate any target distribution of service times.

We experiment with several types of workloads with poisson arrival rates. These distributions are selected to match workloads found in object stores and databases that mix simple GET/PUT requests with complex range or relational queries [49], [54]. **A. Heavy tailed;** 1) a bimodal workload with 99.5% 0.5us and 0.5% 500us requests, and 2) a bimodal workload with 99.5% 5us and 0.5% 500us requests. **B. Lighter tailed;** an exponential workload with mean request time of 5us. **C. Dynamic;** a workload with first half as heavy tailed (A1) and second half as lighter tailed (B), representing a distribution shift in client request patterns.

We experimented with different worker numbers. To ensure a fair comparison and the benefit even at the cost of one timer core, the experiments present 1 network thread, 5 worker threads for Shinjuku and Libinger, and 1 network thread, 4 worker threads (+1 timer thread) for LibPreemptible, running for 2 minutes.

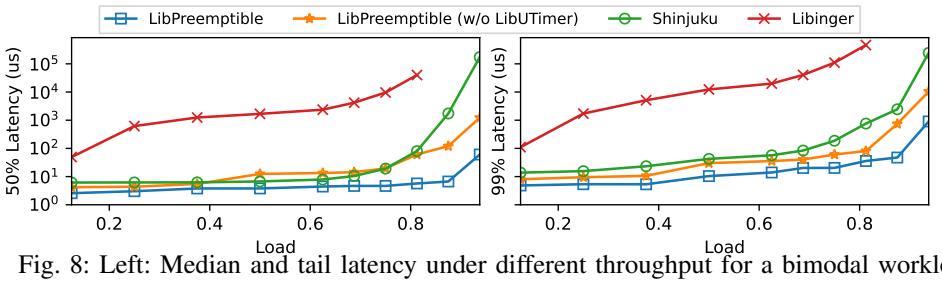
Figure 8 shows the comparison of median (left) and tail (right) latency and throughput for the three systems. Under high load, the median and tail latencies with LibPreemptible are $\sim 10\times$ better than Shinjuku. The maximum throughput is measured by bounding 99% tail latency by 200x the average latency in a stable system. Shinjuku needs to do careful profiling to select the right time quanta to achieve the desired throughput. LibPreemptible achieves better throughputs while dynamically adjusting the quanta under different workloads, as shown in Figure 8 (right), 22% higher than Shinjuku under workload A1, and 33% higher than Shinjuku under C.

To separate the benefit from the new hardware, we disabled UINTR in LibUtimer (orange line), which makes the library run on ordinary timed interrupts. As we will show later, because the interrupt-based timer's granularity is much worse, and the overhead of timer delivery is also larger, the tail latency under higher load becomes worse by more than 5x.

To evaluate the benefit of policy and deadline abstraction, Figure 9 shows how adaptive time quanta reduce SLO violations (as 50us) in workload C. The analysis to trigger the change in time quantum is only called every 10 sec and it is off the critical path, so it does not hurt the tail latency. Under lower load and low dispersion in service time, the time quantum is set to a higher value, consuming fewer CPU cycles for preemption.

B. LibPreemptible Analysis

LibPreemptible deployment overhead: We now measure the overhead of LibPreemptible for a simple gRPC server that uses no preemption by default. There are different threading models and configurations provided by the gRPC interface. We choose a thread pool threading model with blocking to ensure a low-latency baseline. LibPreemptible can also be incorporated on other threading models like Single Process Event-Driven



Workload	A(1/2)	B	C
Shinjuku	0.9 / 0.50	0.70	0.51
Libinger	0.35 / 0.23	0.12	NA
LibPreemptible	1.1 / 0.75	0.78	0.68

Fig. 8: Left: Median and tail latency under different throughput for a bimodal workload (0.5% 500 us, 99.5% 0.5 us). X axis is normalized with respect to the max load. Right: Tail latency bounded throughput (MRPS).

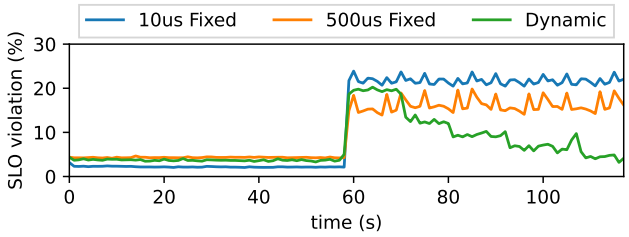


Fig. 9: Dynamic LibUtimer is better at adapting to a distribution shift (Workload C), resulting in much fewer SLO violations. The latter half of the workload exhibits a higher average service time so the SLO violation rate is higher under a static policy.

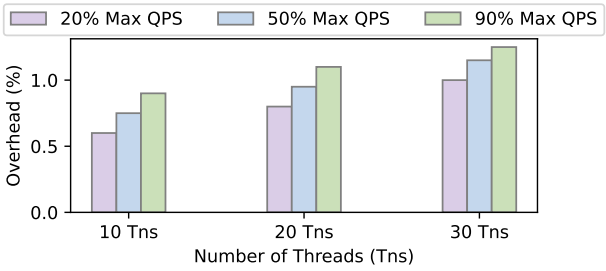


Fig. 10: End-to-end percentage overhead impact on 99% tail latency (Overhead %= $\frac{p99_{libfunc} - p99}{p99}$) across different loads with different numbers of user-level threads per kernel thread (T_n) compared to no preemption.

(SPED) [12] architecture which operates on asynchronous ready sockets.

We use a modified version of the open loop wrk2 [5] workload generator to generate requests with exponential service time. We measure the latency distributions at different QPS levels, with different numbers of user-level threads and kernel-level threads. We first fix the number of worker threads and measure the overhead of LibPreemptible with different numbers of user-level threads and user contexts at 20% - 90% of peak throughput, in terms of percentage degradation on p99 latency. As shown in the Figure 10, when the QPS is 20% - 80% of max load, the latency difference caused by LibPreemptible is negligible (below 1%) as the number of thread contexts increases, showing that hardware-assisted interrupt delivery scales well on larger systems.

We also study how the tail latency behaves under different loads. When the load is around 89% of max, we observe around 1.2% tail latency overhead, due to the higher number of requests per thread that are preempted per unit of time. Overhead increases sublinearly for higher loads and is minimal. The deployment overhead for LibPreemptible is also minimal: in our real workload setups, we deployed LibPreemptible in just 1 week (Sec V-C), which also included familiarizing ourselves with MICA and Zlib's original threading library. All the changes needed to integrate LibPreemptible are at user-level, while for Libinger and Shinjuku, we need to either customize the kernel or customize the glibc library. The code needed is only 3% of the original application code (excluding library code) to integrate LibPreemptible. The following tables show the additional time and code needed for integration (Table II and III).

Time needed	**MICA**	**Zlib**	**RPC**
LibPreemptible	4 hours	3 hours	7 hours
Shinjuku	NA	NA	11 hours
Libinger	9 hours	8 hours	12 hours

TABLE II: Time spent on integration by a researcher with no prior knowledge of either of the three frameworks.

Code needed	**MICA/Zlib**	**RPC**
LibPreemptible	3%	4%
Libinger	NA	7%

TABLE III: Additional code percentage. Since Shinjuku has a high percentage of kernel code we did not include it.

	avg (us)	**min (us)**	**std (us)**	**rate (msg/s)**
signal	15.325	3.584	3.478	63493
mq	10.468	8.960	2.017	95093
pipe	17.761	10.240	4.304	56151
eventFD	29.688	2.816	13.612	33629
uintrFd	0.734	0.512	0.698	857009
uintrFd *(blocked)*	2.393	2.048	0.212	409734

TABLE IV: Overhead of different IPC mechanisms.

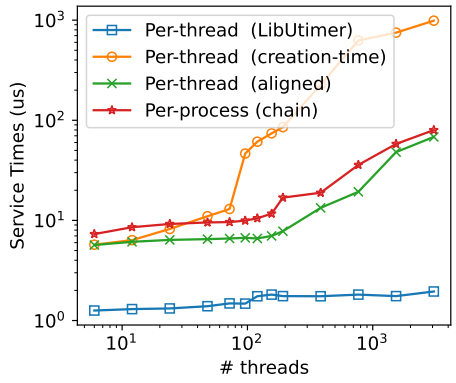


Fig. 11: Scalability of timer delivery overhead.

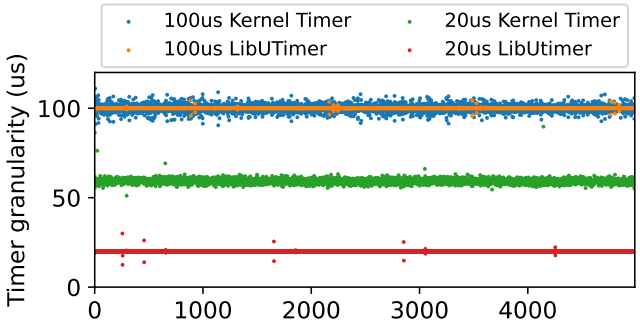


Fig. 12: Precision of LibUtimer. X axis shows number of samples.

Microbenchmarking: We now compare the interrupt delivery overhead for LibPreemptible to other IPC or event nofication mechanisms in Table IV. We adapt the microbenchmark suite [7], and use 1M ping-pong IPC notifications with message size=1B. The user interrupt (averaged between blocked and running) has $10\times$ better average latency compared to the fastest IPC mechanism (message queue). This enables us to deliver fine-grained interrupts to user code. Moreover, the delivery overhead has smaller variance, even when the receiver is blocked.

Scalability and overhead of LibUtimer: We consider two approaches to implementing preemption timers in LibUtimer: per-thread and per-process. With a per-thread timer, every thread has its own OS timer. With a per-process timer, all threads in the process share the same OS timer: one thread receives the signal and forwards it to other threads. Figure 11 shows the timer interrupt delivery overhead with 1000 interrupts on multiple threads, with interval between timer interrupt as 100us.

- **Per-thread (creation-time):** A naïve implementation of a per-thread timer in which the timer of each thread is set independently (e.g., on thread creation) does not scale well on systems with large core counts. In Linux, calling a signal handler involves taking a lock in the kernel, thus causing lock contention when multiple signals are issued at the same time. Figure 11 shows that the superlinearity is likely because of the lock contention, taking as much as 100 us for large core counts.
- **Per-thread (aligned):** The timer interrupts across the different threads are explicitly aligned to reduce contention in the kernel lock. This approach significantly reduces the timer interrupt overhead, especially for higher thread counts, by almost $10\times$ with 32 threads. However, it comes at a cost - the precision of the timer is impeded due to the delay of the issued interrupt.
- **Per-process (chain):** Shiina et al. proposed a new optimization to per-process timers, called "chained signals" [62]. The thread receiving interrupts handles the signal and then issues a signal to at most one other thread.
- **Per-thread (user-timer/LibUtimer):** LibUtimer achieves the best scalability across thread counts.

Moreover, compared with the scalability limitations of earlier approach used by Shinjuku (since the APIC supports only a limited number of logical processors), LibUtimer are able to scale to more tenants using more logical processors by design.

LibUtimer precision and power cost: Finally, we study and justify the timer thread overhead: As a justification of dedicating a core for timer threads, we measure the cost is about 1.2 Watts for the first core because UMWAIT can save energy for busy polling. With each additional core, the power cost is minimal. Besides, we also use `stress-ng` [13] to inject some contentions, and even with kernel activities (IRQ-affinity, TLB shootdowns, overcommitment), we observed that the timer preciseness will not be significantly impacted. In Figure 12, we measure the jittering of LibUtimer timer with background activities, when our target quanta is 100us and 20us. It shows the time between setting up a periodic kernel timer vs LibUtimer with 26 threads and visualizes the time (y-axis) between calls to the handler for 5000 consecutive samples (x-axis). Kernel timer's granularity cannot go down to $20us$ (which is why we see a line around $60us$) and shows high variations, and LibUtimer can consistently give precise timer (average relative error for timer delivery is around 1% for 5000 samples).

C. LibPreemptible in Real Workload Scenarios

To improve resource management and enable efficient CPU utilization, latency critical (LC) jobs can be time-sharing CPU resources with other applications, often best effort (BE), and use preemption to reclaim resources. This reclaimation must happen adaptively in micro-second scale to ensure low tail latency when needed. Achieving this in practice is challenging, especially for the jobs with μs- and sub-μs- SLO requirements, due to relatively large overhead of preemption in current systems. With LibPreemptible, this overhead is dramatically reduced; this can allow low-latency LC jobs to meet their SLO even when aggressively sharing resources with BE tasks.

1) Experiment Setup: We use the MICA [49] KVS system as the LC work in this experiment due to its sub-μs scale execution time for small requests. As the co-located BE processing, we use data compression based on zlib [16]. We run MICA under 5/95 SET/GET request distribution with the

App	Params	Latency 50%, 99%
MICA	EREW mode {k, v} = {16, 64}B SET/GET = 5/95 zipf w/ skewness 0.99	1us, 7 us
zlib	data size = 25 kB	100us, 250 us

TABLE V: Workloads for the co-location experiment.

skewness at 0.99. We use the default zipfian generator from the original MICA work. This yields a median request processing time of 1us. As the compression workload, we run zlib engines against 25 kB of raw data where median latency is 100 us. Table V summarizes the configuration of the datasets used for both workloads, and shows the median and tail request latencies when running them individually, without co-location, and on a single core.

The request generator issues uniformly distributed BE and LC requests with 2% and 98% rates respectively with both constant and bursty QPS. The dispatch thread is connected (over $dispatch\_queue$) to worker threads via our user-space request scheduler based on LibPreemptible. We showcase two different scheduling policies, to demonstrate how adaptability in the time quantum helps with achieving the best of both worlds for LC's SLO and BE's throughput.

2) Scheduling policy #1, ***FCFS with preemption, constant preemption interval****:* Here the scheduler picks up the first request from the $dispatch\_queue$ and runs it in the worker thread with the static and constant time quantum of 30 us. If the request runs for a longer time period, it gets preempted and pushed into the $long\_queue$. Then the scheduler picks up the next element from the $dispatch\_queue$ thereby giving preemptive priority to shorter jobs. If the $dispatch\_queue$ is empty, the scheduler resumes previously preempted requests from the $long\_queue$ via $fn\_resume()$ calls. The results of the experiment are shown in Figure 13 (left).

As Figure 13 (left) shows, our preemptive scheduler brings the 99th tail latency of the LE job (LC-Lib) down $3.2\times - 4.4\times$ times compared to non-preemptive execution (LC-Base). With the preemption interval of 30 us, the scheduler brings the tail latency of MICA KVS requests down to 33 us under 55 kRPS. When the preemption interval is set to 5 us (Figure 13, right), the scheduler brings the 99th tail latency of MICA requests down to 8 us, which is $18.5\times$ lower than non-preemptive execution under the same load and co-location environment. However, the preemption interval of 30 us results in 30% increase of the BE job latency, and the overhead reaches $2.2\times$ when using very low intervals of 5 us.

3) Scheduling policy #2, ***FCFS with preemption, dynamic preemption interval****:* One of the key benefits of the LibPreemptible API is the ability to dynamically set the preemption interval on a per-request basis. The ability to change the time quantum in runtime allows us to build an adaptive scheduling policy that would adjust preemption in interval depending on the current request rate. In this experiment, we modify the previous policy by adding two new components in the system: the QPS monitor, and the controller of the preemption interval.

According to this policy, the scheduler monitors the current QPS rate of incoming requests and sets the preemption interval in the worker threads according to the load. We test the policy with a spiky load generator that periodically issues bursty traffic (Figure 14). We allow the controller to set the preemption interval to values between 10 and 50 us, and our workload QPS changes from 40 to 110 kRPS.

Figure 14 (left) shows that a preemption interval of 50 us reduces the average latency of the LC job from 14 us to 10 us, under the QPS spikes. It does not affect the latency of the co-located BE job significantly when load is low, and only marginally during spikes. When using lower preemption intervals of 10 us (middle), we achieve a reduction of the LC average latency down to 3 us – $5\times$ lower than when running without LibPreemptible, however the latency penalty for the BE job is relatively higher. While complete avoidance of the latency penalty of the BE job is impossible due to preemptions, the overhead can be reduced when the load is low by setting the preemption interval to a higher value during these periods. Figure 14 (right) shows performance with our adaptive time quantum scheduler. The average latency of the LC job remain low during the time of the experiment, while the negative impact on the BE jobs during periods of low load is minimized. These results justify the need for adaptive, low-overhead preemption when the workload is spiky, and showcase the corresponding scheduler that can be build with LibPreemptible.

VI. Related Work

We now discuss prior proposals for scheduling for reducing tail latency and improving CPU efficiency, and quantify our differences.

Dataplane Operating Systems: A dataplane OS improves throughput and/or latency by separating its control and data planes. Initially, the systems that target microsecond-level workloads, such as Chronos [46], IX [22], and Arrakis [55], choose to statically pin threads to cores, and offload load balancing to the NIC hardware, thus eliminating the associated scheduling overheads. This approach works well for homogeneous workloads, however, it can introduce higher latencies for complex workloads with varying execution times, e.g., key-value stores [31], [49]. ZygOS [56] shows that load-balancing through work stealing among the pinned threads is necessary even at these timescales. Shinjuku [44] demonstrates that it is feasible to implement more complex scheduling policies without significant overheads by re-purposing low-overhead preemption mechanisms normally used by VMs. *These approaches require significant changes to the OS, and typically support specific types of applications, which are not scalable and do not coexist with common cloud system software stacks. LibPreemptible performs better than Shinjuku while operating safely on top of the Linux kernel. Our work will complement existing dataplane OSes and hardware offloading techniques.*

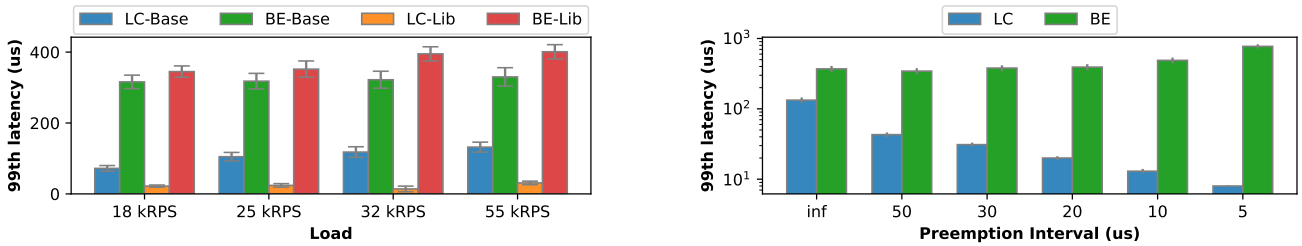


Fig. 13: Tail latency of co-located LC and BE jobs with LibPreemptible based preemptive scheduler with fixed time quantum of 30us (left) and with variable time quantum over fixed QPS of 55kRPS (right).

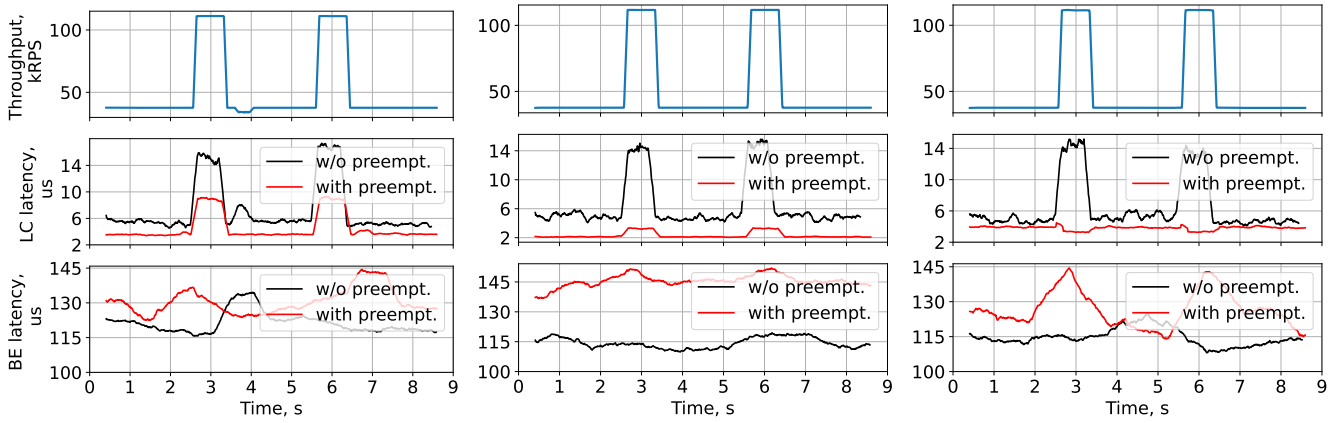


Fig. 14: Average latency of LC and BE jobs over time with a constant preemption interval of 50 us (left), 10us (middle), and with the dynamic policy (right); the top plot shows measured QPS of the bursty load in the dispatch thread, the middle and the bottom - average latency of LC and BE jobs respectively over the 10 s timeframe.

User-level Threading: There is a significant body of work on co-operative userspace thread libraries starting from Adya et al. [18] who proposes the concept of userspace threading with automatic stack management. Capriccio [65] introduces optimizations to improve scalability and scheduling, optimizing stack allocation while identifying some of the pitfalls of non-preemptive scheduling. Inspired by scheduler activations [19], Arachne [57] makes userspace threading core-aware and thread creation faster. Psyche [51] introduces first-class user-level threads that delivers software interrupts. Commercial languages and libraries, such as Go [6], folly::Fibers [4], Boost fibers, uThreads [11], and C++ coroutines [2]), have adopted many of these optimizations.

Libturquoise [24] is the first attempt to create a general-purpose user-level threading library using regular timer interrupts as the preemption mechanism, and mainly focusing on handling shared state and non-reentrant code. Shiina et al. [62] improves upon libturquoise via chained per-process timers and general-purpose, kernel-level thread switching. *LibPreemptible leverages hardware support to minimize the preemption overhead of user interrupts, and facilitates the development of custom user-level scheduling policies.*

Scheduling Policies: Based on the fact that First-Come-First-Serve (FCFS) scheduling has been shown [66] to be

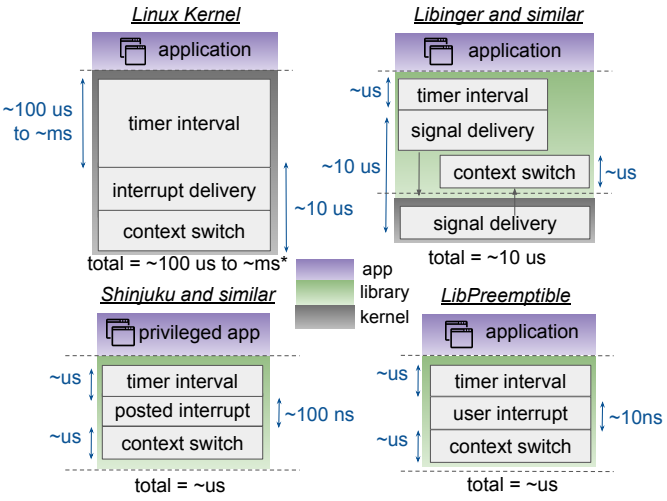


Fig. 15: Comparison of LibPreemptible to prior work.

tail-optimal for light-tailed homogeneous tasks, many older systems did hash-based load balancing on the network interface card (NIC) using receive side scaling (RSS) and running requests to completion [22], [49], [55]. To handle imbalance between workers, newer systems enhance RSS to take into account end-host load (RSS++ [20], eRSS [59]), employ

work-stealing (ZygOS [56], Shenango [54], Caladan [34], BWS [30], Elfen [67], Li at al. [48]), or use techniques, such as join-idle-Queue [50] or join-bounded-shortest-queue [47]. To accommodate highly-variable workloads, Shinjuku [44] takes a different approach by implementing centralized preemptive scheduling in a dedicated core, while Bertogna et al. [23] attempt to find the optimal preemption points. Persephone [29] leverages application-specific knowledge to reserve cores for short requests and avoid preemption altogether. Other proposals employ custom hardware with centralized scheduling (Mind the Gap [42], nanoPU [43], RPCValet [27]), priority queues (ExpressLane [53]), or fast context switching [41].

Recently, McClure et al. [52] show that the trade-offs between load balancing and core allocation are not easy to navigate even without considering request reordering and preemptive policies. It has generated a renewed interest in customizable and application-specific scheduling. Zhao et al. [68] focus on scalable, adaptive, SLO-aware scheduling systems by combining the effectiveness of SLO-aware migration policy and a hardware-based mechanism for short-lived RPCs. SKQ [69] makes event scheduling on top of the Linux kernel configurable. Ford et al. [32], Slite [35], and ghOSt [42] offload kernel thread scheduling decisions to userspace while Syrup [45] allows users to deploy custom scheduling functions throughout the stack safely. *LibPreemptible does not require additional hardware, simplifies the specification and deployment of custom thread scheduling policies, providing complete flexibility in how preemption is employed.*

Finally, LibPreemptible is orthogonal to and can benefit existing work which dynamically allocates CPU cores like [34], to schedulers that focus on microsecond-scale reallocation of resources, like Caladan [34] and Shenango [54], and to systems which use kernel bypass techniques, like DPDK [3]. Figure 15 shows how LibPreemptible differs from prior work.

VII. Discussion

A. Native User Interrupts Security Discussion

The security model for native User Interrupts is designed with a focus on trusted and cooperating processes. Note that the model allows any sender with access to `uintr_fd` to generate the associated interrupt vector for the receiver task that created the `fd`. This could potentially lead to issues with untrusted processes launching a Denial of Service attack.

The potential for a DoS attack by generating a storm of user interrupts is a valid concern. The fact that a user interrupt handler is invoked with interrupts disabled, but upon execution of `uiret`, interrupts get enabled by the hardware, could indeed lead to the handler being invoked before normal execution can resume.

LibPreemptible only allows timer threads to send preemption, which is under the same security domain. The quantum is also controlled by trusted application process. So LibPreemptible reduce the attack surface compared with native UINTR.

B. Attack Surface under LibPreemptible

Security of interrupt-handling is critical, especially in real cloud environments. Shinjuku uses inter-process interrupts (IPIs) directly, rather than Linux signals for preemption to reduce overhead, because it executes in privilege ring 0. To minimize the preemption overhead, it maps the local APIC for each application core to the same address space as the dispatcher. Because the APIC is directly mapped to ring 3 in Shinjuku, both the Shinjuku runtime and application must be trusted. Direct APIC access enables buggy or malicious code to flood the system with IPIs, creating a DoS attack vector against all cores. LibPreemptible avoids this exposure through the use of User Interrupts. While User Interrupts may be sent directly without kernel intervention, allowing low-latency signaling, they may only be sent to targets configured in the kernel-maintained User Interrupt Target Table (UITT). In LibPreemptible, the only configured vectors are between the timer cores and worker applications. Buggy runtime code can at most degrade application performance within a single runtime.

C. Other Use Cases and Future Work

Other use cases for Libpreemptible include traffic shaping [60], scheduling in 5G [33], and real-time DNN serving [39]. The accuracy of these timed actions is crucial to the performance of these real-time applications. Concurrent DNN serving with lightweight micro-second scale preemption on CPU can also improve the throughput and latency. LibPreemptible can provide a precise and lightweight solution to these use cases.

Additionally, while our evaluation uses a dedicated core to generate timer events using the UINTR capability, it is relatively easy to offload this capability directly to hardware. In fact, hardware vendors are exploring supporting this type of capability using a dedicated hardware timer that can deliver an interrupt directly to the application [15]. This hardware offloading approach will improve performance/watt at the cost of area of the chip.

VIII. Conclusions

We propose LibPreemptible, a hardware-assisted library for user-level scheduling. Our primitives achieve flexibility, good performance, and scalability at the same time. LibPreemptible leverages user-level interrupts, and its abstraction delegates scheduling decisions to applications and provides an interface that supports dynamic policies. Compared with prior systems it showed significantly better performance and scalability. LibPreemptible is compatible across application types and requires no changes to the existing OS kernels, making it easy to be deployed in cloud settings. This work can both highlight the benefits of hardware-assisted user-level scheduling and motivate follow-up work that improves the generality of dynamic application-aware scheduling policies.

IX. Acknowledgements

We sincerely thank Qizhe Cai, Chris De Sa, Sihang Liu, Kevin Tang, Yang Zhou, Sol Boucher, Tom Anderson, Ken Birman, Mengjia Yan, Xuehai Qian for their feedback. We thank the colleagues in SSR group in Intel Labs (especially Rajesh Sankaran and James Tsai) for the support. This work was supported in part by NSF CAREER Award CCF-1846046, SRC ACE center funding, and an Intel Research Award.

References

[1] Boost c++ libraries. https://www.boost.org/.

[2] Coroutines. https://en.cppreference.com/w/cpp/language/coroutines.

[3] Dpdk/dpdk: Data plane development kit. https://github.com/DPDK/dpdk.

[4] folly/readme.md at main · facebook/folly. https://github.com/facebook/folly/blob/main/folly/fibers/README.md.

[5] giltene/wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[6] The go programming language specification - the go programming language. https://go.dev/ref/spec.

[7] Ipc benchmark. https://github.com/goldsborough/ipc-bench.

[8] Ipi virtualization support for vm [lwn.net]. https://lwn.net/Articles/863190/.

[9] memcached - a distributed memory object caching system. https://memcached.org/.

[10] Proposal: Non-cooperative goroutine preemption. https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md.

[11] samanbarghi/uthreads: A concurrent user-level thread library implemented in c++. https://github.com/samanbarghi/uThreads.

[12] Single-process event-driven. https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/pai/pai_html/node7.html.

[13] stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng.

[14] Uintr kernel patch. https://lore.kernel.org/lkml/20210913200132.3396598-7-sohil.mehta@intel.com/.

[15] User timer directly programmed by application. https://patents.justia.com/patent/20220147393.

[16] Zlib compression. https://zlib.net/.

[17] Intel ® architecture instruction set extensions and future features programming reference. 2021.

[18] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATC '02, pages 289–302. USENIX Association, 2002.

[19] 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. *ACM Transactions on Computer Systems*, 10:53–79, 1992.

[20] Tom Barbette, Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. Rss++ load and state-aware receive side scaling. In *Proceedings of the 15th international conference on emerging networking experiments and technologies*, pages 318–333, 2019.

[21] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[22] 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 ix: A protected dataplane operating system for high throughput and low latency. 2014.

[23] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. *Proceedings - Euromicro Conference on Real-Time Systems*, pages 217–227, 2011.

[24] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Lightweight preemptible functions.

[25] Mark E. Crovella. Performance evaluation with heavy tailed distributions. https://www.cs.bu.edu/faculty/crovella/paper-archive/tools00-perfeval-ht.pdf, 2000. (Accessed on 04/09/2023).

[26] Mark E Crovella, Murad S Taqqu, et al. Estimating the heavy tail index from scaling properties. *Methodology and computing in applied probability*, 1(1):55–79, 1999.

[27] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[28] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, mar 2013.

[29] 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. *SOSP 2021 - Proceedings of the 28th ACM Symposium on Operating Systems Principles*, 17:621–637, 10 2021.

[30] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. Bws: Balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 365–378, New York, NY, USA, 2012. Association for Computing Machinery.

[31] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. The impact of thread-per-core architecture on application tail latency. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–8. IEEE, 2019.

[32] Bryan Ford and Sai Susarla. Cpu inheritance scheduling. *2nd USENIX Symposium on Operating Systems Design and Implementation, OSDI 1996*, 1996.

[33] Xenofon Foukas and Bozidar Radunovic. Concordia: Teaching the 5g vran to share compute. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 580–596, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. 2020.

[35] Phani Kishore Gadepalli, Runyu Pan, and Gabriel Parmer. Slite: Os support for near zero-cost, configurable scheduling. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 160–173. IEEE, 2020.

[36] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 135–151, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 19–33, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[40] Mor Harchol-Balter, Nikhil Bansal, and Bianca Schroeder. Implementation of srpt scheduling in web servers. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2000.

[41] Jack Tigar Humphries, Kostis Kaffes, David Mazires, and Christos Kozyrakis. A case against (most) context switches; a case against (most) context switches.

[42] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. Ghost: Fast and flexible user-space delegation of linux scheduling. *SOSP 2021 - Proceedings of the 28th ACM Symposium on Operating Systems Principles*, pages 588–604, 10 2021.

[43] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[44] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, David Mazires, Christos Kozyrakis, Adam Belay, and David Mazì Eres. Shinjuku: Preemptive scheduling for µsecond-scale tail latency shinjuku: Preemptive scheduling for µsecond-scale tail latency.

[45] Kostis Kaffes, Jack Tigar Humphries, David Mazires, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. *SOSP 2021 - Proceedings of the 28th ACM Symposium on Operating Systems Principles*, pages 605–620, 10 2021.

[46] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. *Proceedings of the 3rd ACM Symposium on Cloud Computing, SoCC 2012*, 2012.

[47] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association.

[48] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I. Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. Work stealing for interactive services to meet target latency. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, 12-16-March-2016, 2 2016.

[49] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage mica: A holistic approach to fast in-memory key-value storage. 2014.

[50] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68:1056–1071, 11 2011.

[51] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *SIGOPS Oper. Syst. Rev.*, 25(5):110–121, sep 1991.

[52] 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.

[53] Amirhossein Mirhosseini, Brendan L. West, Geoffrey W. Blake, and Thomas F. Wenisch. Express-lane scheduling and multithreading to minimize the tail latency of microservices. *Proceedings - 2019 IEEE International Conference on Autonomic Computing, ICAC 2019*, pages 194–199, 2019.

[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. 2019.

[55] 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. *ACM Transactions on Computer Systems*, 33, 11 2015.

[56] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 325–341, 10 2017.

[57] 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, 2018. USENIX Association.

[58] Parthasarathy Ranganathan and Victor Lee. Advancing systems research with open-source google workload traces. https://dynamorio.org/google_workload_traces.html, 2011. Accessed on April 15, 2023.

[59] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, page 71–77, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Ahmed Saeed, Nandita Dukkipati, Valas Valancius, Terry Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end-hosts. In *ACM SIGCOMM 2017*, 2017.

[61] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[62] Shumpei Shiina, Shintaro Iwasaki, Kenjiro Taura, and Pavan Balaji. Lightweight preemptive user-level threads. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, 1:374–388, 2 2021.

[63] Akshitha Sriraman and Thomas F. Wenisch. µTune: Auto-Tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[64] George Varghese and Tony Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 25–38, 1987.

[65] 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, pages 268–281, Bolton Landing, NY, USA, 2003. ACM.

[66] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations Research*, 60:1249–1257, 9 2012.

[67] Xi Yang, Stephen M Blackburn, and Kathryn S Mckinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. page 309.

[68] Jiechen Zhao, Iris Uwizeyimana, Karthik Ganesan, Mark C Jeffrey, and Natalie Enright Jerger. Altocumulus: Scalable scheduling for nanosecond-scale remote procedure calls. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–440. IEEE, 2022.

[69] Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh. SKQ: Event scheduling for optimizing tail latency in a traditional OS kernel. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 759–772. USENIX Association, July 2021.