# COPS: A coroutine-based priority scheduling framework perceived by the operating system

1st Fangliang Zhao
*Tsinghua University*
zfl22@mails.tsinghua.edu.cn

2nd Donghai Liao
*Beijing Institute of Technology*
ctrlz.donghai@gmail.com

3rd Jingbang Wu
*Beijing Technology and Business University*
wujingbang@btbu.edu.cn

4th Huimei Lu
*Beijing Institute of Technology*
luhuimei@bit.edu.cn

5th Yong Xiang*
*Quan Cheng Laboratory*
*Tsinghua University*
xyong@tsinghua.edu.cn

***Abstract*—The multi-threading model in the general operating systems is becoming insufficient in applications with increasing amounts of concurrency, due to the high context-switching costs associated with kernel multi-threading. In this paper, we propose a new concurrency model called COPS. COPS employs a priority-based coroutine model as the fundamental task unit, replacing the traditional multi-threading model in scenarios with high concurrency, and offers a unified priority-based scheduling framework for both kernel and user space coroutines. COPS introduces coroutines as first-class citizens within the OS to provide asynchronous I/O mechanism, using kernel coroutines as a bridge between I/O operations and devices, and user coroutines to bridge applications with the OS services.**

**We have designed a prototype web server based on COPS and conducted extensive experiments on an FPGA-based system to evaluate COPS. Results show that the proposed model achieves one to four times higher throughput while maintaining relatively lower overhead compared to the multi-threading model in large concurrency applications.**

***Index Terms*—Multi-threading, Concurrency, Coroutine.**

## I. Introduction

In today's era of data explosion, the ability of the general Operating Systems (OS) to process large amounts of data is receiving more attention. For example, Google's servers handle 883 billion requests per day in 2022 [1], with an average of 8 million requests per second. The increasing scale of the concurrency in the system poses severe challenges to the traditional multi-threading model, which has two main shortcomings in our point of view. First, the multi-threading model is nondeterministic [2]. The execution order of threads is uncertain, resulting in the access order of shared resources being uncertain. When used inappropriately in fixed work-flows, multi-threading can lead to greater overhead. In order to improve the performance of the multi-threading model in large concurrency scenarios such as the web server, some research has tried to optimize it, but has not solved the problem mechanically [3], [4].

Second, the multi-threading model is incompatible with the asynchronous I/O mechanism, which requires additional mechanisms (such as the mandatory use of the producer-consumer model, coupled with synchronous mutual exclusion mechanisms) to ensure the safety of asynchronous I/O. Moreover, additional asynchronous runtime is needed within the kernel to execute the kernel's asynchronous tasks and extra interfaces are required to provide user-mode programs with asynchronous I/O mechanisms.

As an alternative concurrency model, the concept of coroutines has not been widely adopted in modern programming languages in many decades. Currently, there is a renewal of interest in coroutines to explore their advantages as an alternative to multi-threading. In this paper, we rethink the concurrency model and asynchronous framework to find solutions that could meet the larger-scale and higher performance requirements. We propose COPS[1], a coroutine-based priority scheduling framework in OS. COPS improves high context-switching overhead and solve the problem of uncertain sequence of accessing shared resources by using coroutines as the fundamental task unit. In addition, COPS builds upon existing research on asynchronous I/O mechanisms, integrates coroutines within the kernel, treats them as first-class citizens, and combines them with the asynchronous I/O mechanism to provide a coordinated and unified scheduling framework for all tasks within the OS.

The rest of the paper is organized as follows: In Section II, we review related work and explain the fall and resurgence of coroutines and the coroutine facilities in Rust language. Section III details the design of COPS, while Section IV explores common patterns in asynchronous and concurrent programming. Performance results for COPS are presented in Section V. We review related work in Section VI and conclude in Section VII.

## II. Background

### A. Coroutine

Coroutines, introduced in the 1960s, offer a lightweight way to improve concurrency with lower resources and overheads

This paper is supported by Quan Cheng Laboratory (Grant No. QCLZD202305).

[1]The **CO** in COPS represents the coroutine, **P** represents the priority, and **S** represents scheduling. All tasks must be carried out under COPS's management.

```
pub trait Future {
  type Output;
  fn poll(self: Pin<&mut Self>, cx:
   ↪  &mut Context<'_>) ->
   ↪  Poll<Self::Output>;
}
enum Poll<T> {
  Ready(T),
  Pending,
}
```

Listing 1: Future trait.

compared to processes or threads. The core characteristics of coroutines, shown as follows, are summarized in [5]:

1) The values of data local to a coroutine persist between successive calls;
2) The execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

However, this general definition leaves open relevant issues regarding the coroutine construct, affecting how coroutine facilities are supported in programming languages. Misunderstandings about the expressive power of coroutines have been perpetuated by some implementations. Additionally, the introduction of first-class continuations and the adoption of multi-threading as a ”de facto standard” for concurrent programming have led to a decline in the use of coroutines.

The revival of coroutines stems from the fact that Moura et al. [6] have demonstrated that full coroutines have an expressive power equivalent to one-shot continuations and one-shot delimited continuations. Nowdays, modern programming languages, c++ 20 [7], Go [8], Rust [9], Python [10], Kotlin [11], etc., all provide varying degrees of support for coroutines.

### *B. The Coroutine facilities in Rust*

Rust, a modern language, excels in performance, security, and concurrency. It features a robust trait system that offers similar capabilities as object-oriented languages, including abstraction, inheritance, and polymorphism.

In Rust, the coroutine facilities is based on its trait system. Rust provides a special trait named **Future** in its core library. Any object that implements **Future** exhibits asynchronous, which is called as coroutines. The poll method defined by the Future trait advances the coroutine's execution. If poll returns Poll::Ready, it indicates the coroutine has completed. Returning Poll::Pending signifies the coroutine is awaiting an event before resuming. The Future trait is detailed in listing 1.

Besides manually implementing Future trait, Rust provides async/await keywords for creating and executing coroutines. The async keyword converts functions, blocks, or closures into futures without explicitly implementing the Future trait. The await keyword executes Future objects and, when an operation cannot continue, relinquishes the CPU.

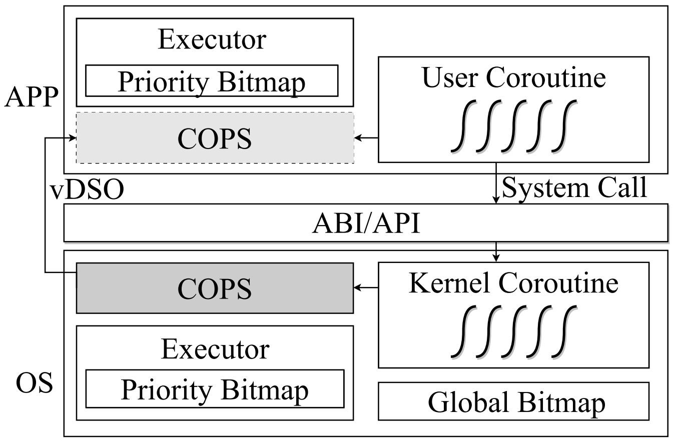


Fig. 1: The architecture of COPS.

Through the Future trait and async/await keywords in Rust, coroutines are treated as first-class citizens for developers at the language level. Moreover, Rust separates the coroutine runtime from the definition of coroutines, enabling customization of runtimes. This separation and the flexibility in definition could pave the way for introducing coroutines at the kernel level, beyond just user space, and even making coroutines as first-class citizens at the OS level.

## III. Design of COPS

COPS is designed to enhance system concurrency by treating coroutines as the fundamental task unit and integrating them as first-class citizens within the OS. COPS offers a unified priority-based scheduling system for coroutines in both kernel and user spaces, enabling both kernel and application developers to freely utilize coroutines. Fig.1 illustrates COPS's overall structure. Both applications and the OS maintain their own Executor data structures and share COPS's scheduling framework via vDSO [12]. They represent tasks as coroutines and manage tasks by using COPS's services through function calls. Applications access other system services via system calls. Additionally, the OS maintains a global bitmap for advanced priority-based cooperative scheduling.

In this section, we detail the design of COPS. We start by explaining the data structures behind COPS's coroutine runtime and how they enable coroutine scheduling (III-A). We then discuss the coroutine state transition model (III-B). Following that, we outline the COPS API (III-C). Lastly, we describe COPS's global cooperative scheduling mechanism (III-D).

### *A. Coroutine Runtime*

Rust offers the **Future** and **Wake** traits to facilitate coroutines without tying them to a specific runtime. This flexibility facilitates the creation of a coroutine runtime suitable for both kernel and user space. The runtime consists of two main components:

```
pub struct Coroutine{
  pub cid: CoroutineId,
  pub kind: CoroutineKind,
  pub priority: usize,
  pub future: Pin<Box<dyn
 ↪  Future<Output=()> + 'static +
 ↪  Send + Sync>>,
  pub waker: Arc<Waker>,
}
```

Listing 2: Coroutine control block.

*1) Coroutine Control Block:* As detailed in II-B, the poll method in Future trait is partially transparent, limiting the precise control of coroutines. To address this, we extends Rust's future and wake abstractions to create coroutine control blocks (CCBs) for more precise control. The CCB structure is displayed in listing 2. The Wake trait is responsible for saving and restoring coroutine contexts. Both the execution and the context-switching of the coroutine are done by the compiler and are transparent. Therefore, the future and wake must be described in the CCB. However, relying solely on the future and wake traits results in a basic polling mechanism that cannot allow for precise control or integration with asynchronous I/O. To enable precise control, we have enhanced the CCB with three additional fields: 1) The Cid is used to identify CCB, and plays a key role in asynchronous I/O mechanism; 2) The Kind denotes the type of the coroutine, guiding the subsequent processing after the coroutine reaches a certain stage; 3) The Priority reflects the coroutine's priority level, which is fundamental to COPS's scheduling framework. The state field is not included in the CCB because Rust coroutines only exist in pending or ready states, with their state implicitly indicated by the queue they reside in.

*2) Executor:* The core of the coroutine runtime is the Executor, which manages all coroutines in a process based on the coroutine control block. The Executor's primary components consist of the following:

1) Ready Queues and Priority Bitmaps: The Executor manages queues of coroutines sorted by priority, ensuring that the highest priority coroutine is always executed first. It also employs a priority bitmap to track the presence of coroutines at each priority level. This bitmap, while not needed for user-level Executors, aids in OS-level scheduling by providing the OS with visibility into user-level coroutines.
2) Blocking Set: This structure manages all coroutines that are blocked, holding them until the awaited event occurs and triggers their resumption.

These two data structures form the runtime foundation for coroutines and establish a basic priority scheduling mechanism for COPS. This ensures that within an address space, COPS can effectively prioritize and schedule the highest-priority coroutine each time.

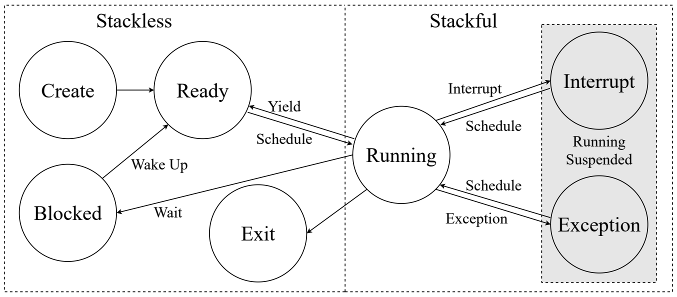


Fig. 2: Coroutine state transition model.

## B. *Coroutine State Transition Model*

Treating coroutines as first-class citizens within the OS has transformed traditional process and thread concepts. For instance, under Kernel Page Table Isolation(KPTI) [13], the kernel acts as a special process with address space switched upon entry and return. Threads now act as stack providers for coroutines and provide abstractions for multiprocessor systems, rather than being the primary unit of scheduling.

The traditional thread state model has been supplanted by the coroutine state model, which includes five basic states: create, ready, running, blocked, and exit, plus a special running-suspended state due to preemptive scheduling and other interruptions. When a coroutine's execution is interrupted by a timer interrupt, exception, or synchronous system call, it occupies the stack but does not actively execute on the CPU, hence the term "running-suspended." This can be further categorized into interrupt and exception sub-states. The coroutine state transition model is depicted in Fig.2.

1) When a coroutine is created, it begins in the ready state and moves to the running state once it is scheduled.
2) A running coroutine can either wait for an event and enter the blocked state, or yield to higher-priority coroutines and become ready again. This transition does not utilize the coroutine's execution stack. Alternatively, if an interrupt or exception occurs, the coroutine enters a suspended state. Upon task completion, it transitions to the exit state, where resource cleanup is performed.
3) In the blocked state, a coroutine awaits an event to transition back to the ready state. If it is in a suspended state, it bypasses the ready state and waits for the completion of the relevant handling routine before resuming execution.

## C. *COPS API*

Coroutines typically have three basic operations: *create*, *resume* and *yield*. In Rust, the *resume* and *yield* operators are replaced by the poll function and the await keyword respectively. Since the future must be driven by await keyword, it is not necessary to consider whether the *yield* is placed correctly within the asynchronous tasks. However, developers must be careful using loop to avoid occupying the CPU for a long time. It is worth mentioning that COPS addresses the issue of accidentally using tight loops by employing a global cooperative scheduling mechanism, detailed in section III-D.

COPS offers a *spawn* function to create new coroutines, along with additional APIs (*current, wake_up, set_priority, alloc_cpu*) to enhance the control of coroutines. The *spawn* initiates a coroutine and returns its identity, the *current* retrieves the identity of the current coroutine, the *wake_up* revives a specified coroutine, the *set_priority* adjusts a coroutine's priority, and *alloc_cpu* requests additional CPU resources. By default, applications are allocated on one CPU, but *alloc_cpu(1)* can be used to request more. Table I lists the COPS APIs.

With the COPS API, developers can fully leverage coroutines in both kernel and user space with minimal restrictions.

| Function | Return Value |
|---|---|
| spawn(future, priority) | cid |
| current() | cid |
| wake_up(cid) | |
| set_priority(cid, priority) | |
| alloc_cpu(cpu_num) | |

TABLE I: Interface of COPS.

## D. Global Cooperative Scheduling Mechanism

COPS offers an advanced global cooperative scheduling mechanism that includes both between kernel and user process and among user process. The priority bitmap, as described in III-A2, is crucial for this mechanism.

COPS facilitates coordination between kernel and user process coroutines through a global cooperative scheduling approach. Upon the timer interrupt, the kernel scans the priority bitmaps of user process Executors to construct a global priority bitmap, thereby giving the OS awareness of user-level coroutines. Additionally, the kernel possesses an Executor for managing its own coroutines.

COPS integrates coroutine and process scheduling within the kernel. One approach is to have distinct schedulers for processes and coroutines, prioritizing kernel coroutines over user processes. However, this approach can complicate the system and not fully address kernel asynchronous I/O issues discussed in I. A more elegant solution involves a special **switching coroutine** in the kernel that identifies the highest-priority user process, manages switching operations, and persists as long as user processes exist. It matches the highest process priority, ensuring cooperative scheduling between the kernel and user processes based on their priorities. The kernel statically assigns the **switching coroutine** the highest priority at initialization, with dynamic adjustments made during runtime based on priority bitmap scans. This allows the kernel to reuse COPS's priority scheduling mechanism for both process and coroutine scheduling. If the kernel has higher-priority coroutines, all user process coroutines are considered of lower priority and wait for the kernel coroutines to complete execution. The **switching coroutine** can then schedule the highest-priority process, coordinating between kernel coroutines and user processes.

Furthermore, COPS provides user processes with read-only access to the global priority bitmap, for coordinating coroutines across various processes. If a user process detects a higher-priority coroutine within the kernel or other processes, it will proactively yield to facilitate cooperative multitasking. However, excessive global coordination might lead to issues such as prolonged CPU usage by malicious processes or high switching overhead, which we aim to address in our future research. These concerns, while important, are beyond the scope of the present paper.

This mechanism allows COPS to offer an OS-aware coroutine scheduling framework that prevents any single coroutine from monopolizing CPU resources for extended periods. Fig.3 illustrates the code logic of COPS.

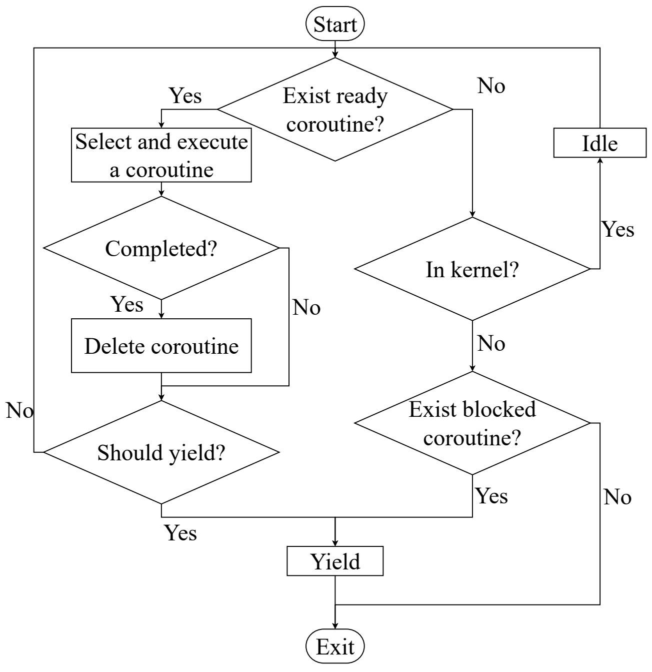


Fig. 3: The code logic of COPS.

## IV. Common usage patterns

In this section, we demonstrate concurrent and asynchronous programming patterns using COPS. COPS creates an environment where all tasks are executed as coroutines. The application's main task is to use the *spawn* function to initiate coroutines for specific tasks. The execution of these coroutines is automatically managed by COPS's coroutine runtime, operating transparently.

### A. Concurrency

A typical COPS use pattern is depicted in Algorithm 1, where the application initiates two coroutines: a producer and a consumer. The producer generates data and, when the message queue is full, signals the consumer and yields the CPU. The consumer then processes all data and waits. Subsequently, COPS reschedules the producer.

### B. Asynchronous Programming

Another common COPS pattern is for asynchronous programming. By leveraging COPS's coroutine runtime environ-

**Algorithm 1** Concurrent Programming

```
1:  $MSG\_QUEUE$;
2:  function MAIN
3:      $consumer\_cid \leftarrow spawn(||consumer, 1)$;
4:      $spawn(||producer(consumer\_cid), 0)$;
5:  end function
6:  function CONSUMER
7:      loop
8:          while MSG_QUEUE is empty do
9:              blocked;
10:         end while
11:         ...                    ▷ consume data from MSG_QUEUE
12:     end loop
13: end function
14: function PRODUCER(cid)
15:     loop
16:         while MSG_QUEUE is full do
17:             wake_up(cid);      ▷ wake up the consumer
18:             yield;
19:         end while
20:         ...                    ▷ produce data into MSG_QUEUE
21:     end loop
22: end function
```

ment, we can convert synchronous system calls into asynchronous ones using coroutines. However, simply changing the interface is not sufficient; the kernel must also implement support for these asynchronous system calls.

1) System Call Interfaces: We have introduced an **Asyncall** data structure that implements Rust's Future trait to handle system calls asynchronously. This structure checks the return values of system calls to determine whether asynchronous waits are required. Rust macros ensure a consistent format for both synchronous and asynchronous system calls, with the latter requiring an additional parameter. An example is shown in the read system call interface 3.
2) Kernel Asynchronous I/O Support: When a user coroutine initiates an asynchronous system call, the kernel creates a corresponding kernel coroutine that is not executed immediately. Control promptly returns to the user coroutine, allowing it to pause while enabling other user coroutines to run. Once the kernel coroutine completes the asynchronous operation, it awakens the related user coroutine using the cid provided by the system call.

Algorithm 2 illustrates asynchronous programming using the asynchronous system call interface. In the async_read function, it initiates an asynchronous *read!* system call. If the kernel buffer is empty, async_read yields the CPU. When data becomes available in the kernel buffer, async_read is awakened to continue execution.

To demonstrate how coroutines integrate with asynchronous I/O, consider the example of reading data from sockets asynchronously. When a read operation reaches the kernel, it is handled by a kernel coroutine, allowing the control flow to

**Algorithm 2** Asynchronous Programming

```
1: function MAIN
2:     $spawn(||async\_read, 0)$;
3: end function
4: function ASYNC_READ
5:     $buf \leftarrow [0; buf\_len]$;
6:     $current\_cid \leftarrow current()$;
7:     $read!(buf.ptr, buf.len, current\_cid)$;
8: end function
```

```
read!(fd, buffer, cid); // Async
read!(fd, buffer); // Sync
```

Listing 3: System call interface of read().

return to user space immediately. The current user coroutine is then blocked, waiting for the kernel to complete the read, while COPS switches to execute another user coroutine. The kernel coroutine may run on a different CPU. If the data is ready in the buffer, the kernel coroutine proceeds without waiting, leveraging multi-processor capabilities. If the data is not ready, the kernel coroutine waits for the network card to signal it. During this wait, other kernel coroutines can continue executing. Once the kernel detects the network card's interrupt and prepares the data, it awakens the corresponding kernel coroutine to finish the operation, such as copying data to a user-space buffer. Upon completion, the kernel sends a user-level interruption, which in turn wakes up the corresponding user coroutine.

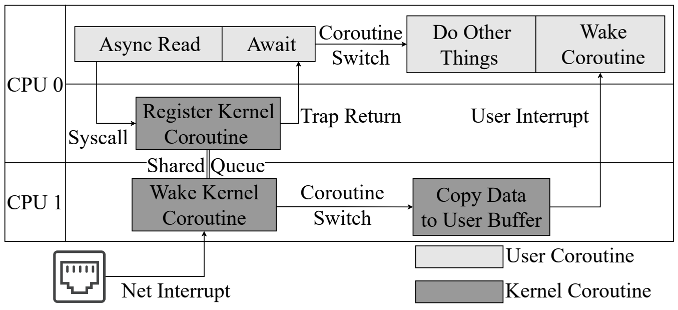


Fig. 4: Asynchronous system call.

## V. Performance Evaluation

To assess COPS's capability in creating highly concurrent asynchronous programs and its precise control over coroutines, we conducted an evaluation on an FPGA. We used the Zynq UltraScale+ XCZU15EG MPSoC model [14] to construct a five-stage RISC-V pipeline processor based on the rocket-chip [15], a RISC-V soft IP core. Implementing N extension [16] on rocket-chip facilitated user-level interrupt functions, essential for asynchronous system calls. We operated an OS designed with the COPS framework on the RISC-V subsystem and evaluated COPS by simulating real-world web server

| | | |
|---|---|---|
| FPGA | Zynq UltraScale+ XCZU15EG MPSoC [14] | |
| | RISC-V soft IP core | rocket-chip [15] with N extension, 4 Core, 100MHz |
| | Ethernet IP core | Xilinx AXI 1G/2.5G Ethernet Subsystem (1Gbps) [17] |
| Operating System | rCore-tutorial [18] | |
| Network Stack | smoltcp [19] | |

TABLE II: Configuration of evaluation.

application scenarios. The complete configuration parameters are detailed in Table II.

The simulated web server application scenario is divided into two parts: a client on the PC regularly sends matrix data of a certain length to the server and awaits the response; The server, hosted on the FPGA, integrates two usage patterns: it establishes a connection with the client, processes the received matrix data, and sends the results back. The server comprises three components:

1) A request receiver that takes client requests and queues them;
2) A request handler that dequeues requests, performs matrix operations, and queues the results;
3) A response sender that retrieves responses from the queue and communicates them to the client.

The PC client measures the latency of each request-response cycle and calculates the message throughput over a set time period. We assess COPS's performance by examining the web server's latency and throughput under various configurations.

## A. *Coroutine vs. Thread*

To demonstrate the coroutine model's suitability for large-scale concurrency, we implemented the kernel and the web server's three components respectively using coroutines and threads[2]. The coroutine model assigns three coroutines per client-server connection, while the thread model assigns three threads per connection, each handling a component. We defined four server models based on the use of coroutines(**C**) and threads(**T**) in the kernel(**K**) and userland(**U**):

**KCUC**: When a user coroutine calls read(), the kernel creates a coroutine to perform the operation, blocking the user coroutine. Once data is read and copied, the kernel coroutine triggers a user-level interrupt to awaken the user coroutine.

**KCUT**: Similar to KCUC, but with a user thread invoking read(). The kernel coroutine executes the operation and unblocks the user thread after the copy is complete.

**KTUT**: A user thread calls read(), and a corresponding kernel thread reads data from the socket, blocking until the copy operation is done, allowing other threads to run in the meantime.

**KTUC**: Similar to KCUC, but instead of the kernel coroutine handling the copy, it delegates the read operation to a separate kernel thread, which polls sockets for data and copies it. Once the copy is complete, it sends a user-level interrupt to awaken the user coroutine.

[2]The threads in the application are kernel-support thread.

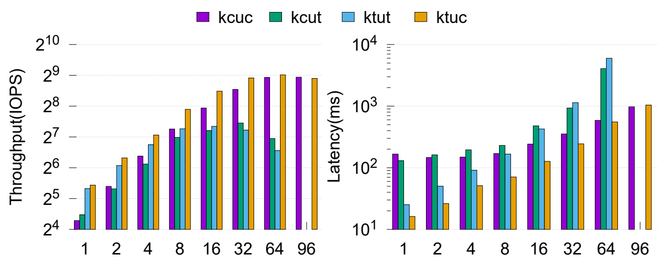


Fig. 5: Throughput and message latency. The horizontal axis represents the connections.

The testing begins once all client-server connections are set up to avoid the influence of creating coroutines/threads. The client sends requests at 100ms intervals for a total of 5 seconds, with each request containing a 15x15 matrix. The results of the experiment are depicted in Fig.5.

**Runtime overhead**: With few connections, coroutines incur more overhead than threads, as seen in comparisons like KCUC vs KCUT, KCUT vs KTUT, and KCUC vs KTUT in Fig.5. This overhead is due to COPS's executor being lock-protected. With a small number of connections, COPS can complete tasks with fewer cores but is given extra CPUs for a fair comparison (the coroutine model has the same core allocation as the thread model). The extra CPUs lead to increased scheduling overhead from unnecessary synchronization. Additionally, frequent yielding on the extra core due to a lack of ready coroutines causes overhead from privilege level switching.

**Lower coroutine context-switching overhead**: As the number of connections grows, the latency of the thread model (KCUT and KTUT) exceeds that of the coroutine model (KCUC). By the time connections hit 32, KCUT's latency is notably lower than KTUT's. Despite performing the same operations, coroutines have less context-switching overhead than threads, even when comparing to kernel threads with streamlined context switching. At two connections, KCUC's latency is lower than KCUT's due to most context switches happening in userland in the KCUC model, avoiding the privilege-switching overhead present in KCUT. However, as connections increase further, the throughput of KCUT and KTUT decreases, and their context-switching costs rise rapidly.

**Coroutines have obvious advantages with high concurrency**: With a small number of connections, the KTUC model exhibits the lowest latency due to its dedicated kernel thread that polls socket states efficiently, providing a quick response. However, this advantage diminishes as connection numbers grow, and the KTUC model's polling overhead increases. Comparing throughput, at 64 connections, the CPU is fully loaded. At 96 connections, the KCUC model shows lower latency than KTUC, as the latter, along with KCUT and KTUT models, cannot complete the test due to the heavy workload. The specific throughput and message latency are not displayed in Fig.5 for these cases. As connections continue to increase, KTUC's throughput drops significantly, whereas

KCUC's throughput remains relatively stable. Even with the use of epoll, which was not part of our experiment, the KTUC model's performance is unlikely to improve significantly due to additional synchronization overhead from its producer-consumer model and increased thread context-switching costs. In conclusion, after comparing KCUC with KTUC, COPS proves more suitable for large-scale concurrent scenarios.

**Less memory usage**: We also compared the memory usage between coroutines and threads. User-level threads, supported by the kernel, have two stacks: one for userland execution and another for kernel execution. In contrast, both kernel and userland coroutines run on a single stack, statically sized at 0x4000 bytes. The memory footprint of the three components using coroutines is as follows: 120 bytes for the request receiver, 80 bytes for the request handler, and 64 bytes for the response sender. The kernel coroutine itself is 176 bytes. Thus, establishing a connection in the KTUT model requires 0x4000 * 2 * 3 bytes, while the KCUC model requires only 0x4000 * 2 + 176 + 120 + 80 + 64 bytes. Table III illustrates the maximum number of connections possible for both the KTUT and KCUC models under specific configurations.

| Configuration | Size(bytes) | KCUC | KTUT |
|---|---|---|---|
| kernel heap | 0x80_0000 | | |
| kernel frame | 0x1A0_0000 | 385 | 186 |
| user heap | 0x20_0000 | | |

TABLE III: Maximum connections of KCUC and KTUT.The stack used by the thread is allocated from the kernel frame.

## *B. Priority orientation*

In a realistic scenario, a web server may manage tens of thousands of connections, many of which could be idle. System resources should favor active connections, prioritizing timely responses for them. We assign each connection a priority level in a tiered structure to ensure that higher-priority connections experience lower latency and less latency variability.

Similar to the previous experiment, but with both the kernel and application using coroutines. We establish 64 client-server connections with an equal distribution across 8 priority levels and measure throughput and message latency for each priority during the same time frame. The client sends requests to the server every 50ms over 5 seconds.

As shown in Fig.6, under limited resources, the throughput and latency for higher-priority connections are maintained. As resource availability increases(increasing the amount of CPU), lower-priority connections also see improved throughput and reduced latency, while the highest-priority connections consistently achieve the highest throughput and lowest latency.

Additionally, we created 64 connections between the client and server, evenly spread across 4 priority levels, to analyze the latency distribution for each priority. The results are depicted in Fig.7. Consistent with priority handling, higher-priority connections exhibit concentrated and low latency, whereas lower-priority connections show dispersed and higher latency. As resources increase, latency across all priorities decreases and becomes more concentrated.

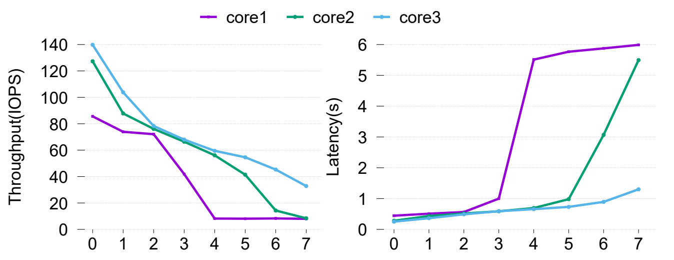
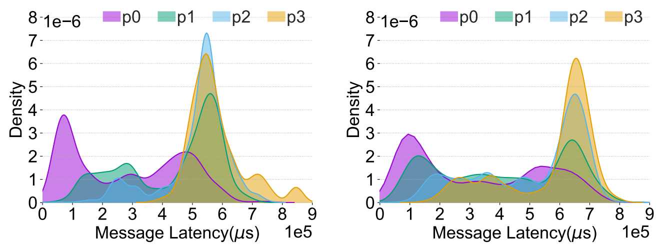


Fig. 6: Throughput and message latency of different priority connections. The horizontal axis represents the priorities.



(a) Core-2 (b) Core-4

Fig. 7: Latency distribution of different priority connections in different amount of cores.

## VI. Related Work

*Select* and *epoll* are widely used asynchronous I/O mechanisms in Linux that enable a single thread to handle multiple I/O events. However, they involve producer-consumer interaction and increase the cost of synchronous mutual exclusion, thereby adding to the kernel's complexity [20]. Instead, COPS follows the principle of treating coroutines as first-class citizens within the OS, assigns a coroutine per I/O event to maintain the event's state (as same as Demikernel [21], which assigns a coroutine per TCP connection for retransmissions, maintaining TCP state locally), thereby avoiding the overhead associated with maintaining global state. Besides, *Select* and *epoll* expose extra syscall interfaces to provide asynchronous I/O mechanism for user programs while COPS modifies the syscall implementations allowing user programs directly using the same syscall interfaces to obtaining asynchronous I/O services. Furthermore, because the OS lacks awareness of userland asynchronous tasks, asynchronous interfaces implemented in userland, such as POSIX AIO, result in increased overhead due to thread management, I/O buffer replication, and context switching across privilege levels [22]. COPS utilizes the user interrupt mechanism to awaken user coroutines and eliminates frequent context switching across privilege levels.

Similar to *Select* and *epoll*, Windows' I/O Completion Ports(IOCP) [23] also offers I/O multiplexing but can be challenging to program correctly due to callback functions [24]. In contrast, COPS utilizes the Rust coroutine facilities(async, await keyword), enables writing asynchronous functions in a manner similar to synchronous functions, thus aviods the callback hell.

LXDs [25] created an in-kernel asynchronous runtime for efficient cross-domain processing. Memif [26] offered a low-latency, low-overhead interface using asynchronous and hardware-accelerated techniques. Lee *et al.* [27] introduced the asynchronous I/O stack (AIOS) to the kernel to reduce the I/O latency. Despite these advancements, most methods function independently of the kernel's thread scheduler, which limits their adaptability and scalability and further complicates the kernel's design. Embassy [28], an asynchronous driver framework for embedded systems built on Rust coroutines, excels in managing device interrupts. It significantly outperforms FreeRTOS, a C-based system, in interrupt handling time, thread processing time, and interrupt latency. COPS, draws on the implementation of embassy but provides priority support for coroutines, employs a unified priority-based scheduling system for coroutines in both kernel and user spaces, which achieving better scalability.

## VII. Conclusion

This paper introduces COPS, a coroutine-based priority scheduling framework, which allows the operating system to perceive user-level coroutines through a priority bitmap mechanism. COPS integrates coroutines with asynchronous I/O mechanisms, thereby facilitating the development of highly concurrent applications, reducing the overhead of traditional multi-threading, and providing efficient asynchronous I/O and priority scheduling.

Our evaluation demonstrates COPS's ability to deliver high throughput and low latency for highly concurrent applications. Compared to threads (KCUC vs KTUT), COPS's coroutine abstraction can increase throughput by 1.05 to 3.93 times. Additionally, COPS's coroutine priority scheduling effectively meets various requirements and ensures the rational distribution of system resources.

## References

[1] M. Mohsin. (2023) 10 google search statistics you need to know in 2023 | oberlo. [Online]. Available: https://www.oberlo.com/blog/google-search-statistics

[2] E. A. Lee, "The problem with threads," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-1, Jan 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

[3] P. Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives," vol. 42, no. 6, pp. 189–199, 2007. [Online]. Available: https://dl.acm.org/doi/10.1145/1273442.1250756

[4] J. Howell, B. Bolosky, and J. J. Douceur, "Cooperative task management without manual stack management," in *Proceedings of USENIX 2002 Annual Technical Conference*. USENIX, 2002. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cooperative-task-management-without-manual-stack-management/

[5] C. D. Marlin, "Coroutines: A programming methodology, a language design and an implementation," *Lecture Notes in Computer Science*, vol. 95, 1980.

[6] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *Acm Transactions on Programming Languages & Systems*, vol. 31, no. 2, pp. 1–31, 2009.

[7] D. Mazières. (2021) My tutorial and take on c++20 coroutines. [Online]. Available: https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html

[8] A. Freeman, "Coordinating goroutines," *Pro Go*, pp. 811–835, 2022.

[9] K. Rosendahl, "Green threads in rust," Ph.D. dissertation, Master's thesis, Stanford University, Computer Science Department, 2017.

[10] E. V. Craeynest, "Asynchronous programming with coroutines in python," in *FOSDEM 2017*, 2017.

[11] R. E. B. A. Usmanov, "Kotlin coroutines: design and implementation," in *Onward! 2021: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2021.

[12] M. Kerrisk. vdso(7) - linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man7/vdso.7.html

[13] J. Corbet. (2017) KAISER: hiding the kernel from user space [LWN.net]. [Online]. Available: https://lwn.net/Articles/738975/

[14] (2022) Zynq UltraScale+ MPSoC data sheet: Overview (DS891). [Online]. Available: https://docs.xilinx.com/api/khub/documents/sbPbXcMUiRSJ2O5STvuGNQ/content

[15] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[16] A. Waterman, K. Asanovic, and C. Division, "Volume i: Unprivileged ISA," 2019.

[17] (2023) AXI 1g/2.5g ethernet subsystem v7.2 product guide. [Online]. Available: https://docs.xilinx.com/r/en-US/pg138-axi-ethernet

[18] rcore os, "rcore-tutorial-v3," https://github.com/rcore-os/rCore-Tutorial-v3, 2023.

[19] smoltcp rs. (2023) smoltcp. https://github.com/smoltcp-rs/smoltcp.

[20] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, "Comparing and evaluating epoll, select, and poll event mechanisms," 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:8488207

[21] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, "The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 195–211. [Online]. Available: https://dl.acm.org/doi/10.1145/3477132.3483569

[22] M. T. Jones, "Boost application performance using asynchronous i/o," *IBM Developer*, 2006. [Online]. Available: https://developer.ibm.com/articles/l-async/

[23] alvinashcraft. (2022) I/o completion ports - win32 apps. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports

[24] Z. Liew. (2017) Callbacks in JavaScript | zell liew. [Online]. Available: https://zellwk.com/blog/callbacks/

[25] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya *et al.*, "Lxds: Towards isolation of kernel subsystems." in *USENIX Annual Technical Conference*, 2019, pp. 269–284.

[26] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 369–383, 2016.

[27] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds." in *USENIX Annual Technical Conference*, 2019, pp. 603–616.

[28] "embassy-rs/embassy: Modern embedded framework, using Rust and async." 2023. [Online]. Available: https://github.com/embassy-rs/embassy