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

Anonymous Author(s)
Submission Id: 28

ABSTRACT

The multi-threading model in the general operating systems is becoming insufficient in applications with increasing amounts of concurrency, due to the context-switching costs in the kernel multi-threading~~, and the kernel's inability to accurately schedule the user-level multi-threads for higher resource utilization~~. In this paper, a new concurrency model named COPS is proposed. ~~We designed~~ COPS uses a priority-based coroutine model as the smallest task unit to replace the ~~multi-thread~~ multi-threading model in large concurrency scenarios, and ~~designed~~ provides a unified priority-based scheduling framework for kernel space and user space coroutines. COPS ~~utilize~~ introduces coroutines as first-class objects in the OS to provide asynchronous I/O mechanism, which takes kernel coroutines as a bridge between I/O operations and devices ~~to provide asynchronous I/O mechanisms~~and takes user coroutines as a bridge between applications and the OS's services respectively.

We design a prototype of web server based on COPS and conduct extensive experiments in an FPGA-based system to evaluate COPS. Results show that the proposed model achieves one to four times higher throughput while remains relatively lower overhead than that using the multi-threading model in the large concurrency applications.

CCS CONCEPTS

• **Software and its engineering** → **Scheduling**; **Multiprocessing / multiprogramming / multitasking**.

KEYWORDS

Multi-threading, Concurrency, Coroutine.

ACM Reference Format:
Anonymous Author(s). 2024. COPS: A coroutine-based priority scheduling framework perceived by the operating system. In *Proceedings of The 21st ACM International Conference on Computing Frontiers (CF 24).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 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 [23], 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 [14]. The execution order of threads is uncertain, resulting in the access order of shared resources being uncertain. When used inappropriately in fixed workflows, 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 [10, 17].

Second, the multi-threading model is incompatible with the asynchronous I/O mechanism in OS. The traditional OS such as Linux usually utilizes the event-driven model to achieve the asynchronous I/Os, resulting in the complexity of OS. For example, Linux provides system calls such as the select and epoll to support user-level asynchronous I/O tasks by reusing multiple I/O operations on a single thread [9]. The epoll process combines a single thread with an event-driven model, and requires interaction through the producer-consumer model, which increases the overhead of synchronous mutual exclusion. The I/O Completion Ports (IOCP) in the Windows OS provides a similar I/O multiplexing mechanism and uses the callback functions to achieve the asynchronous I/O operations [4]. However, because of the callback function procedures, it becomes very difficult to correctly write a program based on it [18]. The io_uring proposed in [11] utilizes the shared memory between user-space and kernel-space to avoid memory replication, thereby improving IO processing efficiency and throughput. However, overdesign leads to increased kernel complexity and greater difficulty in utilizing interfaces [16]. Additionally, since OS is unaware of asynchronous tasks in userland, those asynchronous interfaces implemented in userland further increase the overhead including thread process, I/O buffer replication, and cross-privilege context switching (e.g., POSIX AIO) [12].

The asynchronous I/O mechanism has proven to be efficient in large-scale concurrent web server scenarios. However, the design and implementation of the asynchronous I/O are difficult. OS not only needs to provide asynchronous I/O support for applications, but also needs to build a runtime for some of its own asynchronous tasks. There has been some work investigating asynchronous I/O mechanisms in OS. LXDs [25] developed a lightweight, asynchronous runtime environment in the kernel for cross-domain batch processing. Memif [19] proposed a low-latency and low-overhead interface based on asynchronous and hardware-accelerated implementations. Lee *et al.* [15] introduced the asynchronous I/O stack (AIOS) to the kernel to reduce the I/O latency. Results showed that the performance of test applications are significantly improved. The above work has made progress in the research of asynchronous I/O

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CF 24, May 2024, Ischia, Naples, Italy
© 2024 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

mechanisms, but these methods are often independent of the kernel thread scheduler, resulting in a lack of versatility and scalability, and increasing the complexity of the kernel.

As an alternative ~~to the multi-threading model, coroutines have attracted much attention in the design of system solving large-scale concurrent demands. DepFast~~ concurrency model, the concept of coroutines has not been widely adopted in modern programming languages in many decades. Currently, there is some renewal of interest in coroutines to explore the advantages of coroutines as an alternative to multi-threading. For example, DepFast [20] used coroutines in distributed arbitration systems; ~~Capriccio [30] used cooperative user-level threads to achieve a scalable, large-scale web server~~Another resurgence of coroutines is in the scripting languages, such as Python and JavaScript.

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 the high context-switching overhead and solve the problem of uncertain sequence of accessing shared resources by using coroutines as the basic task unit. In addition, COPS draws on the existing research on asynchronous I/O mechanism, introduces coroutines into the kernel, provides coroutines as first-class objects and combines it with the asynchronous I/O mechanism to provide a coordinated and unified scheduling framework for all tasks in OS.

The rest of the paper is organized as follows: we explain the fall and resurgence of coroutines, the coroutine facilities in Section 2. We describe the design of COPS in Section 3 and present the common asynchronous and concurrent programming coding patterns in Section 4. We evaluate the performance of COPS in Section 5. We discuss the related work in Section 6 and conclude the paper in Section 7.

2 BACKGROUND

2.1 Coroutine

~~Coroutine~~ The concept of coroutines was firstly introduced in the early 1960's and constitutes one of the oldest proposals of general control abstraction. It is a lightweight concurrency abstraction that enable for the cooperative scheduling of multiple execution flow on a single system thread. Compared to processes or system threads, coroutines have lower resources ~~requirement~~ requirements and context-switching overhead. ~~Modern programming languages , For example, c++ 20 [22], Go [8], Rust [27], Python [7], Kotlin [29], etc. , all provide varying degrees of support for coroutines . Coroutine can be divided into the following two categories by the implementation~~However, coroutines has not been widely adopted in modern programming languages in many decades. The absence of coroutines can be partly attributed to the lack of a uniform view of the concept, which was never precisely defined. The fundamental characteristics of coroutines, shown as follows, are summarized in some literatures [21]:

1) ~~Stackful coroutine : It can generally be considered user-level threads. Each Stackful coroutine saves the function call chain and local variables in its own running stack space, which is allocated by runtime. Compared with system threads, Stackful coroutines optimize the overhead of context switching, but the effect is limited.~~ The values of data local to a coroutine persist between successive calls;
2) ~~Stackless coroutine: It runs on a public stack. Normally, the local variables it uses are saved in the heap and used as needed. Therefore, stackless coroutines do not need to allocate a fixed size of stack space, and the overhead of context switching is minimal.~~ 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, such a general definition leaves open relevant issues with respect to a coroutine construct which makes the support of coroutine facilities in programming languages different. Some implementation even contributed to the misconception of the expressive power of coroutines. Besides the absence of a precise definition of the concept, the other factors that greatly contributed to the discard of this interesting concept are the introduction of first-class continuation, and the adoption of multi-threading as a "de facto standard" concurrent construct [24].

Moura et al. [24] demonstrate that full coroutines have an expressive power equivalent to one-shot continuations and one-shot delimited continuations, which promotes the revival of coroutiness as a convenient general control abstraction. Nowdays, modern programming languages, For example, c++ 20 [22], Go [8], Rust [27], Python [7], Kotlin [29] , etc., all provide varying degrees of support for coroutines.

2.2 ~~Asynchronous Programming~~ The Coroutine facilities in Rust

As a modern programming language designed by Mozilla, Rust is outstanding in terms of performance, security, and concurrent programming. ~~Unlike Python and Java, which use background processes to maintain the runtime (such as using a garbage collection process for memory management), Rust has a relatively lightweight runtime. It provides relatively complete asynchronous programming support, allowing programmers to easily write asynchronous operations without having to deal with complex IO callbacks, which is very suitable for building IO bound applications~~Rust provides a powerful trait system to achieve the same expressive power as other object-oriented programming languages, such as providing abstraction, inheritance, and polymorphism.

The ~~support of Rust asynchronous programming is achieved by the three key components: 1) Future trait , 2) async/await syntax, 3) runtime library. The first two components help facilitate the development of asynchronous programs, while the runtime library enables asynchronous programs to run smoothly.~~

~~The future traits: Traits in Rust are similar to interface functions in other programming languages, defining abstract common behaviors. Future~~ coroutine facilities in Rust is built on the trait system. Rust provides a special trait named **Future** in its core component. An object that implements the Future trait is an object with asynchronous behavior, ~~and the future trait is the core of Rust asynchronous programming~~which is called as a coroutine. The poll function ~~it specifies~~ defined by the Future trait is used to drive the execution

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

progress of ~~asynchronous objects~~the coroutine. When the poll function returns Poll::Ready, it means that the ~~asynchronous behavior~~ coroutine has ended. On the contrary, returning Poll::Pending means that the ~~asynchronous behavior~~ coroutine cannot continue to be executed and needs to wait for the corresponding event to occur before it can continue. The ~~future~~ definition of the Future trait is shown in listing 1~~, The~~.

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

Listing 1: Future trait.

Besides implementing Future trait manually, Rust also provides async/await ~~syntax:~~ keywords to create coroutines and drive the execution of coroutines. The async keyword is ~~syntactic sugar provided by Rust,~~ used to modify functions, code blocks, and closures, converting them into anonymous futures. The await keyword is another syntactic sugar~~. When asynchronous behavior cannot continue, await will cause the current future~~, not only it can be used to execute the Future object, but also can be used to give up CPU usage when the asynchronous behavior cannot continue, allowing some other futures to execute. ~~The Rust runtime: The asynchronous execution of futures is guaranteed by Rust's runtime. Multiple futures are combined into a task, and these tasks can be executed in a non-blocking sequence on a single thread. Typically, the scheduling of these tasks is determined by the Executor in the runtime, rather than the operating system scheduler. Therefore, this style of logic model is similar to green threads, and the operating system will not be aware of the Rust tasks in userland.~~

Listing 1: Future trait

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

Through the Future trait and async/await keywords in Rust, the coroutines are provided as first-class objects to developers in the level of language. Moreover, Rust decouples the coroutine runtime from the definition of the coroutine, which means that developers can customize the coroutine runtime according to their own needs. Both the definition of coroutines and the decoupling property of coroutine runtime provides the possibility of introducing coroutines into the kernel not only the user space. Going even further, providing coroutines as first-class objects in the OS level is foreseeable.

3 DESIGN OF COPS

~~We choose to use the stackless coroutine to replace the traditional multithreading model in large-scale concurrent web server scenario, and we utilize Rust coroutines due to the strict checking mechanism of the Rust language compiler and the significant advantages in terms of memory safety. However, the operating system cannot directly perceived of the coroutine mechanism provided by the programming language. As learning more about Rust coroutines , we gradually realize the relationship between coroutines and asynchronous I/O mechanism. So we try to introduce coroutines into kernel to handle a wide variety of asynchronous tasks. This provides an opportunity to define a unified asynchronous task scheduling mechanism in the operating system. Therefore, we propose a coroutine-based priority scheduling framework perceived by the operating system, which can provide a coordinated and unified~~ The design of COPS is tightly focused on improving the concurrency capability of the system. COPS take coroutines as the smallest task unit and provides coroutines as first-class objects in the OS. The kernel and application developers can directly use coroutines without any extra restrictions. It provides a unified priority-based scheduling framework for ~~all tasks in the operating system and provide a unified asynchronous I/O framework to meet the requirement of high concurrency~~kernel space and user space coroutines.

Fig. 1 shows the overall framework for COPS. The application and the operating system maintain their Executor data structure respectively and share the scheduling framework provided by COPS through vDSO [13]. Applications and tasks within the operating system are described in the form of coroutines. The COPS's services are provided for applications through function calls and other system services provided through system calls. A separate global bitmap is maintained within the operating system for more advanced priority cooperative scheduling.

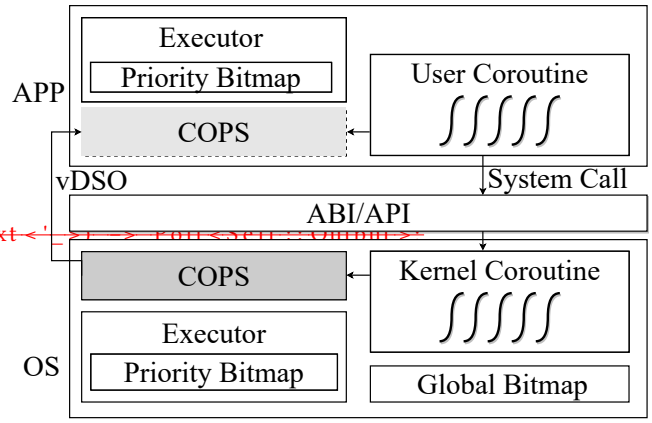


Figure 1: The architecture of COPS.

Next, we will cover the design details of COPS in the rest of this section. We will first introduce the data structure related to the coroutine runtime provided in COPS to describe how to implement the scheduling of coroutines(3.1). Next, we will introduce the

~~concurrency of COPS(??~~state transition model of the coroutines(3.2). Then, we will ~~introduce the state transition model of the coroutine (3.2~~describe the COPS API(3.3). Finally, we will describe the global cooperative scheduling mechanism provided by COPS(3.4).

~~The architecture of COPS.~~

3.1 Coroutine Runtime

The **Future** and **Wake** traits are provided by Rust to support the coroutine mechanism without limiting the specific runtime implementation. Therefore, we can use this decoupling property to customize a coroutine runtime that can be used in both kernel and userland. The Coroutine runtime is mainly composed of the following two parts: 1) Coroutine Control Block; 2) Executor.

3.1.1 Coroutine Control Block. As the description of 2.2, the polling process specified in future trait is partially transparent, which prevents us from accurately controlling the coroutine. Therefore, on the basis of future and waker abstractions provided by Rust language, we add additional fields to form coroutine control blocks, so as to achieve precise control of coroutines. The structure of the coroutine control block is shown in listing 2.

How to switch and save the context of the coroutine is the most important issue. The Wake trait is responsible for saving and switching context of coroutine. 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 coroutine control block. However, ~~Using~~ using these two fields alone means that the execution of coroutine can only use a rough polling way to promote, neither cannot achieve the purpose of accurately control, nor be combined with asynchronous I/O mechanisms to truly take the advantages of coroutines. For this purpose, we use three additional fields in the coroutine control block to achieve the accurately control of the coroutine. 1) The cid is used to identify coroutine control blocks, and plays a key role in asynchronous I/O mechanism; 2) The Kind field is used to indicate the type of the coroutine task. After promoting the execution of the coroutine to a certain stage, COPS will process the coroutine differently according to the task type; 3) The Priority field indicates the priority of coroutine and serves as the basis of the COPS's scheduling framework.

Listing 2: Coroutine control block

```
~~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>,~~
~~}~~
%DIF <
-
```

Note that we didn't label the coroutine with a state field because the Rust coroutines only have pending or ready states, so the state of the coroutine is implicitly described by the queue it is in.

```
\DIFadd{pub struct Coroutine}{
  \DIFadd{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.

3.1.2 Executor. The main part of the coroutine runtime is the Executor, which is based on the coroutine control block and is responsible for managing all coroutines within a process. Its main structure includes the following parts:

1) Ready queues and priority bitmaps: The Executor maintains ready queues of different priorities, and coroutines are stored in queues corresponding to their priorities. This guarantees that the coroutine with the highest priority can be executed firstly every time. In addition, the Executor maintains a priority bitmap structure corresponding to the ready queue to indicate the presence or absence of coroutines at the corresponding priority level. Although it is unnecessary to maintain this structure in user process Executor, this structure serves the purpose of scheduling within the operating system. Through this data structure, the operating system will gain a certain degree of awareness of user-level coroutines.
2) Blocking set: All coroutines that are blocked after execution will be managed by this structure until the event that the coroutine is waiting for occurs and then wakes up from this set.

These two data structures provide the runtime environment of coroutines, and provide a basic priority scheduling mechanism for COPS, which ensures that the coroutine priority scheduling in COPS can play a role in the address space of a single process and can be scheduled to the coroutine with the highest priority each time.

3.2 ~~Concurrency~~

~~Through the coroutine model, applications can support concurrent multitasking on a system thread. And the multiple tasks of the application can access shared resources through mutual cooperation, rather than disorderly competition in the form of multithreading, which can eliminate the uncertainty of multithreading model for the access to shared resources. However, in a web server with tens thousands of connections, the concurrency achieved by coroutines alone is not enough to cope with such a huge demand. To do this, COPS provides applications with a interface for creating system threads, but limits the way it can be only used to request more CPU resources to cope with large-scale concurrency.~~

3.2 Coroutine State Transition Model

The introduction of coroutines into the asynchronous I/O mechanism in the kernel to replace the original multithreading model has undoubtedly brought new changes to the basic concepts of process

and thread in the operating system. In the case of kernel page table isolation (KPTI) [6], the kernel can also be regarded as a special process, which means that the address space will be switched when entering the kernel and returning to user process. As for thread, its role has been greatly diminished, no longer as the basic unit of task scheduling, only to provide a running stack for coroutines, and as a parallel abstraction of multiprocessor systems. Therefore, the traditional thread state model is no longer needed in task scheduling, instead of the coroutine state model.

Similar to the thread state model, coroutines have five basic states: create, ready, running, blocked, and exit, but there is also a special state: the running-suspended state. This is due to the preemptive scheduling provided by the operating system and some other special cases. The coroutine only has the stack when it is in the running state, but the execution process of the coroutine in the running state may be interrupted by clock interruption, exception, or entering the kernel to perform synchronous system calls. In this case, the coroutine will occupy the running stack in a certain time scale, but it is no longer in the state of executing on the CPU, so we define it as the state of running-suspended. According to the cause, it can be further divided into operation interrupt state and operation ~~abnormal~~ exception state. The coroutine state transition model is shown in Fig. 2.

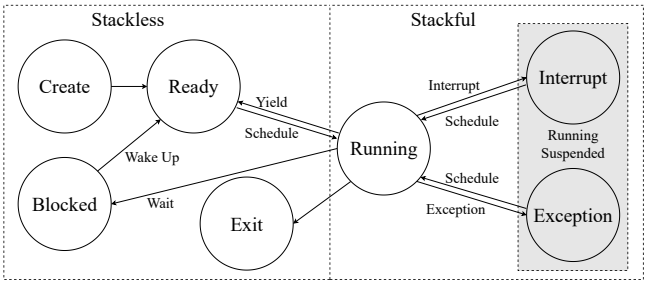


Figure 2: Coroutine state transition model.

(1) Once a coroutine is created, it goes into the ready state until it is scheduled and thus into the running state.
(2) For a coroutine in the running state, the possible state transitions can be divided into two categories. On the one hand, it may wait for an event to enter the blocked state, or it may yield actively and turn to ready state when detecting other coroutines with higher priority (including coroutines in other processes). This type of state transition does not occupy the running stack; On the other hand, if an interrupt or exception occurs during the running, the CPU will be preempted and the current coroutine will enter a running-suspended state. In addition, when the task is completed, the running coroutine will enter the exit state, waiting for the resource to be reclaimed.
(3) When the coroutine is in the blocked state, it must wait for an event to wake itself up and thus enter the ready state. However, when the coroutine is in the running-suspended state, it does not need to go through the ready state transition, and only needs to wait for the relevant handling to complete before entering the running state.

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

Table 1: Interface of COPS.

3.3 COPS API

According to the general definition of coroutines, the coroutines has three basic operators: *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 in the right places. Certainly, we cannot avoid that the developer may accidentally use a tight loop in the coroutine, which will cause the coroutine to occupy the CPU for a long time. Pure cooperative scheduling frameworks are helpless against this dilemma. However, COPS can deal this dilemma through a global cooperative scheduling mechanism. This will be introduced in the section 3.4.

COPS provides the *spawn* operator to create a new coroutine. Besides, COPS also provides additional interfaces(*current*, *wake_up*, *set_priority*, *alloc_cpu*) to enhance the control capability of coroutines. The *spawn* function create a new coroutine in the current process(kernel) and return the identity of the new coroutien. The first argument is the closure of the coroutine main body. The *current* function returns the identity of the current coroutine. The *wake_up* function wakes up the coroutine with the specified identity. And the *set_priority* function sets the priority of the coroutine with the specified identity. The *alloc_cpu* function is used to apply for more CPU resource. Default, an application is only allocated one CPU. Developers can call the *alloc_cpu(1)* to apply for an additional CPU for the current application. Table 1 shows the interfaces provided by COPS.

Using the COPS API, developers can take advantage of coroutine at mostly without any restrictions, no matter in the kernel or user space.

3.4 Global Cooperative Scheduling Mechanism

On the basis of the coroutine priority scheduling, COPS also provides a more advanced global cooperative scheduling mechanism: cooperative scheduling between the kernel and user processes and cooperative scheduling between user processes. The priority bitmap mentioned in 3.1.2 plays a key role in this process.

The first is the coordination between the coroutines in the kernel and the coroutines in the user process. When the kernel handles the time interrupts, it scans the priority bitmap in all user process Executor to generate a global priority bitmap, so that the operating system can perceive the user-level coroutine to a certain extent. There is also an Executor in the kernel to manage the kernel coroutines. Coordination between the operating system and user processes can be achieved by combining the global priority bitmap with the priority bitmap in the kernel Executor. In the process of coordination, kernel task scheduling is divided into coroutine scheduling and process scheduling. The most directly way to achieve this goal is to define a process scheduling and a coroutine scheduling separately, which can ensure that coroutines with high priority in

the kernel are executed first, and then to determine the execution of user processes. However, this extra mechanism can bloat the system and does not solve the problems with the kernel asynchronous I/O mechanism mentioned in 1. A more elegant design that can solve this problems is to introduce a special coroutine in the kernel: the switching coroutine. The switching coroutine is responsible for finding the user process with the highest priority and completing the switching operation, so it never ends as long as there is a user process. Its priority is consistent with the highest priority of all processes, so as to ensure that the scheduling of kernel coroutines and user processes can be cooperative according to the priority. When the kernel is initialized, it is staticly assigned the highest priority, then its priority changes dynamically once the system is running. The kernel determines the priority of the switching coroutine after scanning the priority bitmap. In this case, process and coroutine scheduling in the kernel can reuse the priority scheduling mechanism provided by COPS. When there are other coroutines with higher priority in the kernel, it means that all the coroutines within the user process are inferior to the coroutines in the kernel, need to wait for the kernel to finish executing the coroutine with higher priority. Then the process with the highest priority can be scheduled by switching coroutine. This ensures coordination between kernel coroutines and user processes.

In addition, we share the read-only permission of the above global priority bitmap with the user process, so as to achieve the coordination between coroutines in different processes. Once the user process detects the existence of a higher priority coroutine in the operating system or other processes while it is running, the user process will yield actively to achieve mutual coordination. However, blind global coordination may cause some malicious processes to occupy CPU for a long time or cause frequent switching overhead, which will be our future improvement direction, and this problem is not covered in this paper.

Through the above mechanism, COPS provides an operating system aware coroutine-based scheduling framework, which can avoid one coroutine occupying CPU for a long time. Fig. 3 shows the code logic of COPS.

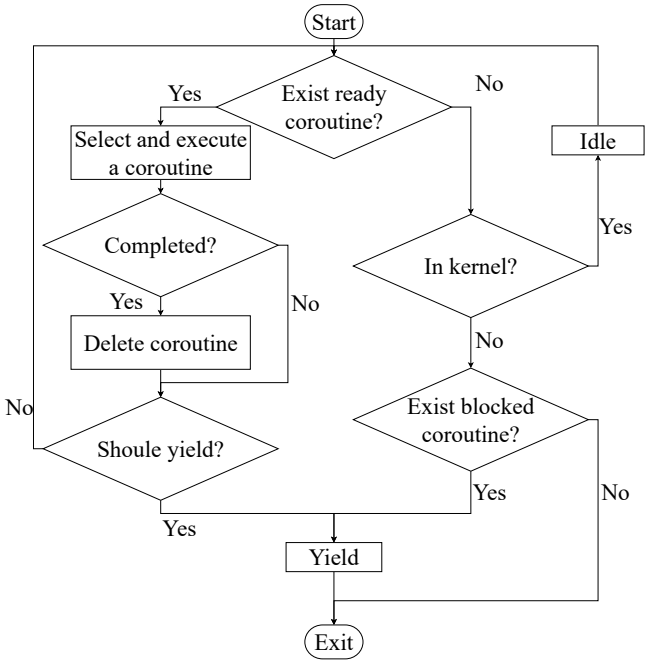


Figure 3: The code logic of COPS.

4 ~~BUILD PROGRAM WITH COPS~~COMMON USAGE PATTERNS

In ~~order to facilitate application developers to use COPS to build highly concurrent asynchronous programs, we provide a comprehensive set of programming interfaces and asynchronous system call interfaces.~~

4.1 ~~The Interface of COPS~~

~~In order to ensure that application development is compatible with traditional methods, we have adjusted the Unix-like runtime environment of the user process for compatibility, so that the main function is not executed immediately after the user process is initialized, but is wrapped into the main coroutine and added to the ready queue for unified scheduling. This means that after the user process is initialized,~~ this section, we illustrate COPS use patterns for concurrent and asynchronous programming. Both the patterns are broadly applicable. COPS provides an environment that all tasks exist in the form of coroutines~~, in a cooperative execution environment~~. What the main ~~coroutine~~ body of application needs to do is using the ~~interface of COPS to create different coroutines~~ *spawn* function to create coroutines with specific functionality. The running of coroutines is transparently driven by the coroutine runtime provided by COPS. ~~COPS provides the interfaces to application developers in Table 1.~~

~~Interface of COPS~~ ~~**Interface Description**~~ ~~spawn(future, prio) getcid(void) wake(cid) reprio(cid, prio) alloc_cpu(void)~~

4.1 ~~Asynchronous system call~~Concurrency

~~In addition to providing a programming interface to easily replace the original threading model, we also need to combine coroutines with asynchronous I/O mechanism to take full advantage of coroutines. If a synchronous I/O system call, such as "read", is invoked in a coroutine, this operation will block all ready coroutines that can run on the running stack, thus limiting concurrency. Therefore, it is necessary to transform the system call to an asynchronous form to ensure that only the current coroutine is blocked while other ready coroutinescontinue to be drived. After the analysis, we find that kernel coroutines can help transform the synchronous I/O operations into asynchronous ones. On the one hand, the problem of excessive granularity of the thread model resources is solved. On the other hand, the async/await synchronous style of code makes it easy to deduce the changes in the execution flow and avoid "callback hell" [18]. When the asynchronous task is blocked, the corresponding coroutine will enter the blocking set in the Executor and wait for the event to occur. After the event occurs, the callback function in the original event-driven model will be unified into a behavior, which is waking up the corresponding coroutine from the blocking set. The transformation of synchronous system calls to asynchronous~~

~~ones mainly involves two parts: the interface provided for the application and the support~~ A common COPS use pattern is to build a producer-consumer model, shown in Algorithm 1, where the application creates two coroutines: one producer and one consumer. The producer will firstly be driven by COPS to produce data. While the message queue is full, the producer will wake up the consumer and yield the CPU. Then, consumer will consume all data and block itself. Next, COPS will schedule the producer again.

Algorithm 1 Concurrent Programming

```
MSG_QUEUE;
function MAIN
    consumer_cid ← spawn(||consumer, 1);
    spawn(||producer(consumer_cid), 0);
end function
function CONSUMER
    loop
        while MSG_QUEUE is empty do
            blocked;
        end while
        ...                                  ▷ consume data from MSG_QUEUE
    end loop
end function
function PRODUCER(cid)
    loop
        while MSG_QUEUE is full do
            wake_up(cid);                    ▷ wake up the consumer
            yield;
        end while
        ...                                  ▷ produce data into MSG_QUEUE
    end loop
end function
```

4.2 Asynchronous Programming

Another common COPS usage pattern is for asynchronous programming. COPS provides a coroutine runtime environment, so asynchronous programming can be achieved by simply transforming the original synchronous system call into an asynchronous system call based on coroutines. However, it is not enough to modify the interface form alone, and support for asynchronous system calls must also be implemented in the kernel.

1) System call interfaces: In order to ensure that system calls can support asynchronous features, we add an **Asyncall** auxiliary data structure that implements the future trait specified in Rust, and this data structure will determine whether asynchronous waits are required according to the value returned by the system call. In addition, we use the macro mechanism in Rust to ensure that synchronous and asynchronous system calls are similar in form. The difference between the two is that an asynchronous system call requires an additional parameter. We show the read system call interface in listing ~~3~~3.
2) Kernel asynchronous I/O support: When a user-level coroutine invokes an asynchronous system call, the kernel will create a kernel coroutine corresponding to the user-level coroutine, which it is not be executed immediately. Then the control flow will immediately return to the user-level coroutine, blocking the current coroutine so that other user-level ready coroutines can continue to execute. Once the kernel has executed the coroutine and completed the corresponding asynchronous operation, it will wake up the corresponding user-level coroutine with the cid passed by the system call.

Listing 3: System call interface of read().

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

Algorithm 2 shows how to use the asynchronous system call interface for asynchronous programming. In the async_read function, it calls the *read!* system call interface for asynchronous reading. If there is no data in the kernel buffer, the async_read coroutine will give up the CPU. Once there is data in the kernel buffer, the async_read coroutine will be waked up.

Algorithm 2 Asynchronous Programming

```
function MAIN
    spawn(||async_read, 0);
end function
function ASYNC_READ
    buf ← [0; buf_len];
    current_cid ← current();
    read!(buf.ptr, buf.len, current_cid);
end function
```

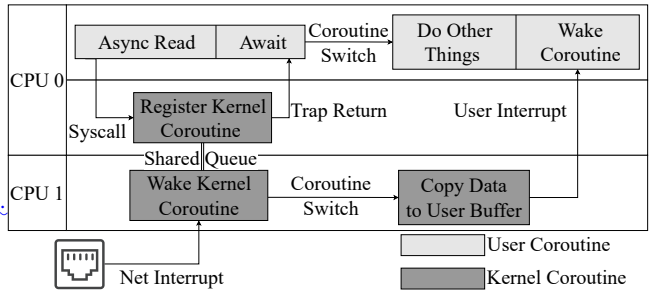


Figure 4: Asynchronous system call.

~~We will use the example of asynchronously reading data from sockets to explain~~ In order to illustrate that how coroutines can be combined with asynchronous I/O mechanism, we use the example of asynchron-ously reading data from sockets. Once inside the kernel, the operations that were previously done synchronously by the kernel are encapsulated in the kernel coroutine. The control flow then immediately returns to user space and blocks the current

```
\DIFadd{read!(fd, buffer, cid); // Async call
read!(fd, buffer); // Sync call
```

Listing 3: System call interface of read().

~~Configuration of evaluation~~

FPGA	Zynq UltraScale+ XCZU15EG-2FFVB1156 MPSoC [1]	
	RISC-V soft IP core	rocket-chip [5] with N extension, 4 Core, 100MHz
	Ethernet IP core	Xilinx AXI 1G/2.5G Ethernet Subsystem (1Gbps) [2]
Operating System	rCore-tutorial[26]	
Network Stack	smoltcp[28]	

Table 2: Configuration of evaluation.

coroutine, waiting for the kernel to finish reading operation. At this point, COPS switches and executes the next user coroutine. The kernel coroutine can be executed on another CPU. As if the data in this buffer is already ready, the kernel coroutine does not need to wait, which takes full advantage of the multi-processor; If the data in the buffer is not ready, the kernel coroutine will be blocked and wait for the network card to wake itself up. while the other kernel coroutines will be able to continue executing. Once the kernel has received the interrupt generated by the network card and has prepared the data in the buffer, the corresponding kernel coroutine is woken up to continue execution. Once the kernel coroutine finishes its work (in this case, copying data to the user-space buffer), it sends a user-level interruption, telling the user-level interruption handler to wake up the corresponding coroutine.

5 PERFORMANCE EVALUATION

In order to verify the effectiveness of COPS in building highly concurrent asynchronous programs and more accurate control of coroutines, we set an evaluation in FPGA. The FPGA model is Zynq UltraScale+ XCZU15EG-2FFVB1156 MPSoC [1]. We build a five-level RISC-V pipelining processor, which is based on the rocket-chip [5] (a RISC-V soft IP core), in FPGA. Since asynchronous system calls relys on the relevant functions of user-level interruption, we implement N extension [31] on rocket-chip. We run an operating system based on COPS framework on the RISC-V subsystem, and finally complete the evaluation of COPS by simulating the real web server application scenario. The total configuration parameters are shown in Table 2.

The simulated web server application scenario consists of two parts. One part is the client running on PC, sending a certain length of matrix data to the server regularly, and receiving the response from the server. The other part is the server in the FPGA, combined the two usage patterns, which establishes a connection with the client, performs matrix operations on the matrix data sent by the client and returns the results to the client. The server has the following three components:

(1) Receiving request component: It receives the request from the client and stores it in the request queue;
(2) Handling Request component: It removes the request from the request queue, performs matrix operations, and stores the result in the response queue;
(3) Sending response component: It retrieves the response message from the response queue and sends it to the client.

Finally, the client on the PC calculates the time latency between sending each request and receiving the response, and calculates the throughput of messages within a fixed period of time. We evaluate COPS by analyzing the time latency and throughput of the web server under different configurations.

5.1 Coroutine vs. Thread

To confirm that the coroutine model is more suitable for large-scale concurrency scenario, we use coroutines and threads in the kernel and application respectively to implement the three components of the web server mentioned above. Using the coroutine model creates three coroutines for each connection between the client and the server, while using the thread model creates three user threads for each connection, corresponding to the three components mentioned above. The web server can be divided into four models based on the combination of coroutines and threads used in the kernel and applications. **K** means in the kernel and **U** means in the userland. **C** means using coroutines and **T** means using threads. Note: The threads used in the application are kernel-supported threads.

~~KCUC~~**KCUC**: When the user-level coroutine invokes read() system call, the kernel will create a kernel coroutine to execute the operations and the current user-level coroutine will be blocked. Once the kernel coroutine reads data from the socket and completes the copy operation, the kernel coroutine will send a user-level interruption to wake up the corresponding user-level coroutine. ~~KCUT~~

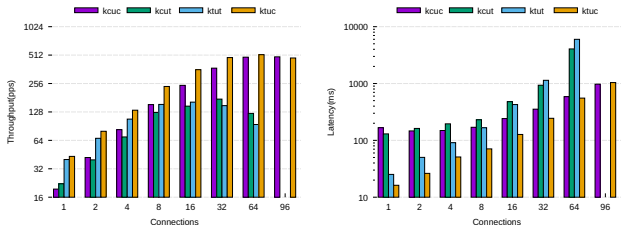
KCUT: When the user thread in userland invokes read() system call, it is similar to KCUC, the kernel will create a kernel coroutine to execute the operations and block the current user thread. The kernel coroutine will wake up the blocked thread after the copy operation is completed. ~~KTUT~~

KTUT: The user thread invokes the read() system call, and the corresponding kernel thread will continue to try to read data from the socket until the data copy operation is completed before returning to the user space to continue executing, during which other threads can be executed. ~~KTUC~~

KTUC: Similar to kcuc, but it no longer completes the copy operation through the kernel coroutine, instead of submitting the information of the reading operation to another separate kernel thread and directly returns to the userland. This kernel thread will constantly poll all the socket ports submitted by the user coroutines and copy data from the socket which has data in the buffer. After the data replication operation is completed, it will send a user-level interruption to wake up the corresponding user coroutine.

The test start after all connections between the client and the server are established, to eliminate the impact of coroutine/thread creation. The client sends requests to the server every 100ms for 5s, and the matrix size of each request is 15 x 15. The experimental results are as shown in Fig. ~~5:~~ 5.

~~COPS's synchronous mutual exclusion mechanism overhead: The concurrency provided in COPS results in that the synchronous mutual exclusion overhead is increased to ensure that data structures such as Executor are accessed correctly~~**Runtime overhead**: When the amount of connections is small, using coroutines will cause more overhead than using threads. This can be seen from the comparison of KCUC vs KCUT, KCUT vs KTUT, and KCUC vs KTUT



(a) Throughput **(b) Latency**

Figure 5: Throughput and message latency.

in ~~Fig. 5. Note that when~~ Figure 5. This is because the executor of COPS's runtime is protected by a lock. When the amount of connections is small, ~~due to~~ COPS requires only a small number of cores to complete the task, but it is allocated extra CPU for a fair comparison(The number of cores allocated to coroutine model is the same as the thread model). On the one hand, the allocation of excess CPU ~~to the application, COPS~~ will cause extra synchronous mutual exclusion overhead when scheduling the coroutines; On the other hand, COPS on the extra core will yield frequently because of no ready coroutine ~~, which results in COPS frequently acquires Executor locks when scheduling the kernel coroutines. The synchronous mutually exclusive overhead increases rapidly, which leads to low scheduling efficiency of kernel coroutines. So the KCUC and KCUT models are with high latency when the connection number is small~~which will cause the overhead of privilege level switching. Therefore, COPS is not applicable to applications with low concurrency. ~~Lower coroutine context-switching overhead~~

Lower coroutine context-switching overhead: According to ~~KC-UC~~ KCUC vs KCUT and KCUT vs KTUT, the latency of the threading model will gradually exceed that of the coroutine model as the number of connections increases. When the number of connections reaches 32, the latency of KCUT is lower than that of KTUT. The coroutines and threads in the kernel complete the same operation, so the overhead of coroutine context-switching is less than that of threads (even the kernel thread with simplified context switching). When the number of connections is 2, the latency of KCUC is lower than that of KCUT. This is because most of the coroutine context-switching in KCUC model are carried out in userland, while the privilege-switching exist in KCUT model. However, KCUT and KTUT models that use threads will decrease their throughput and increase their context-switching cost rapidly as the number of connections increases to a certain extent. ~~At large-scale concurrency, coroutines have obvious advantages~~

Coroutines have obvious advantages with high concurrency: when the number of connections is small, the KTUC model has the lowest latency, which is reasonable, KTUC uses a separate kernel thread to constantly poll the state of sockets, and can respond in a timely manner, so the latency is lowest. However, as the number of connections gradually increases, the advantage is no longer obvious, and the overhead of each poll of the KTUC kernel thread gradually increases. From the comparison of throughput, when the number of connections reaches 64, the CPU is running with the full workload. When the number of connections reaches 96, the latency of KCUC model is lower than that of KTUC model (KCUT and KTUT model cannot complete the test due to heavy workload. ~~Fig.~~ Figure 5 does not show the corresponding throughput and message latency). When the number of connections continues to increase, the throughput of KTUC model decreases significantly, while the throughput of KCUC model does not decrease significantly. Although we did not use a separate thread to complete the data replication operation in our experiment, the analysis shows that the effect of KTUC model will not be significantly improved even if epoll is adopted. On the one hand, epoll requires additional synchronous mutually exclusive overhead because of using producer-consumer model. On the other hand, the overhead of thread context-switching will increase. Therefore, after comparing KCUC with KTUC, we can conclude that COPS is suitable for large-scale concurrent scenario.

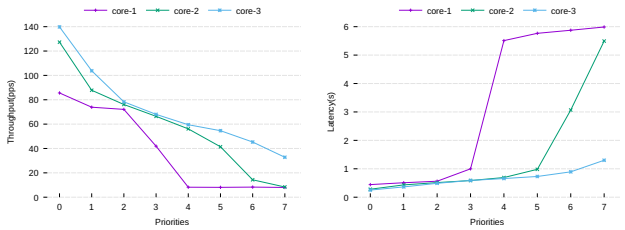
Configuration	Size(bytes)	KCUC	KTUT
kernel heap	0x80_0000		
kernel frame	0x1A0_0000	385	186
user heap	0x20_0000		

Table 3: Maximum connections of KCUC and KTUT. The stack used by the thread is allocated from the kernel frame.

Less memory usage: In addition to the comparison of throughput and latency, we also compare the memory usage of coroutines and threads. The user-level threads are kernel-supported, which have two stacks at the same time, one for execution in userland and the other for execution in kernel. Meanwhile, no matter the kernel coroutines or the userland coroutines, they must run on a stack. The stack size is staticly configured as 0x4000 bytes. The size of three components implemented by using coroutines are 120(Receiving request component), 80(Handling Request component) and 64(Sending response component) bytes. The size of kernel coroutine is 176 bytes. Therefore, establishing a connection in the KTUT model needs to occupy 0x4000 * 2 * 3 bytes, while the KCUC model only needs to occupy 0x4000 * 2 + 176 + 120 + 80 + 64 bytes. Table 3 shows the maximum connections that can be established for the KTUT model and the KCUC model under certain settings.

5.2 Priority orientation

In a real scenario, the web server needs to host tens thousands of connections, but a large part of the connections may be idle. The resources in the system should be biased to those active connections, and higher priority should be assigned to ensuring that these connections can receive timely responses. Therefore, we set the priority level of each connection in a hierarchical manner to ensure that connections with higher priority have lower latency and less latency jitter. Similar to the above experiment, but both the kernel and the application use coroutines. Priority scheduling is implemented by COPS. We set up 64 connections between the client and the server, divided into 8 priorities on average, and test the throughput and message latency of different priority connections in the same time period. The client sends a request to the server every 50ms for 5s. As shown in the ~~Fig.~~ Figure 6, the throughput and latency of connections with higher priority levels can be guaranteed under limited resource constraints. As the number of resources increases, the low priority connection can also achieve higher throughput and lower latency, while the connection with the highest priority still has the highest throughput and lowest latency.



(a) Throughput (b) Latency

Figure 6: Throughput and message latency of different priority connections.

In addition, we established 64 connections between the client and the server, evenly divided into 4 priorities, and analyzed the latency distribution for each priority connection. the final result is shown in Fig. 7. This is in line with the characteristics that the higher priority connections will be handled preferentially. High priority connections have concentrated latency distribution and low latency, while low priority connections have scattered latency and high latency. With the increase of resources, the latency of all priorities decreases and is concentrated.

6 RELATED WORK

Coroutines are lightweight, have outstanding performance in context switches and are well-suited for the state-machine-based asynchronous event handling that I/O stacks commonly require. In recent years, a lot of research work has taken advantage of these advantages of coroutines.

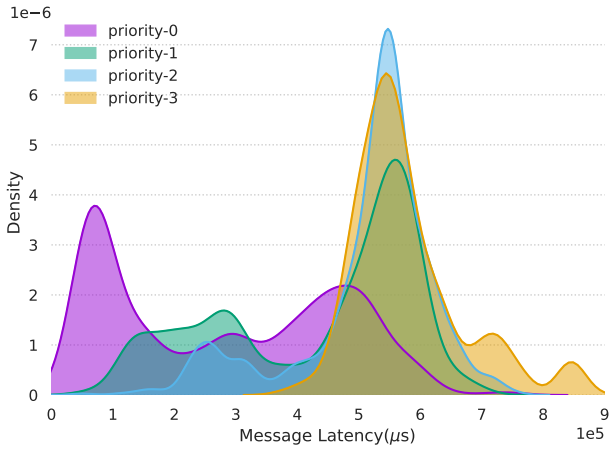
Demikernel[32] uses ~~rust~~ Rust coroutines to build their prototype system, avoiding the context switch overhead on the critical path of the IO stacks (approximately 12 cycles to complete the context switch). Besides, their TCP stack uses one coroutine per TCP connection for retransmissions, which keeps the relevant TCP state and removes the need for global TCP connection state management. They ultimately achieve microsecond latency.

Embassy[3] is a generation framework of asynchronous driver for embedded environments based on Rust coroutines, achieves remarkable results in handling device interruptions. Compared with FreeRTOS implemented in C, it has achieved overwhelming advantages in terms of interrupt time consuming, thread time consuming, interrupt latency, etc.

7 CONCLUSION

This paper proposes COPS, a coroutine-based priority scheduling framework that can be perceived by the operating system. COPS make the kernel perceive the user-level coroutines by the priority bitmap mechanism and combines kernel coroutines with asynchronous I/O mechanisms. It is proved that COPS can help to develop highly concurrent applications, reduce the overhead of traditional multithreading model, and provide convenient asynchronous I/O mechanism and priority scheduling mechanism. Through the evaluation, we proved that the COPS framework can have the characteristics of high throughput and low latency in the construction of highly concurrent applications. Using the coroutine abstraction provided by COPS can increase throughput to 1.05x-3.93x than threads

(a) Core-2



(b) Core-4

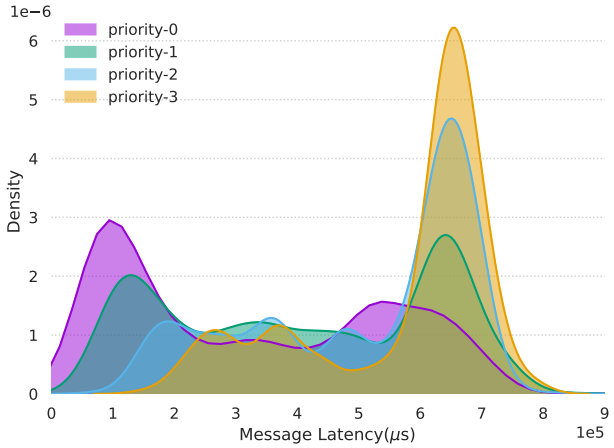


Figure 7: Latency distribution of different priority connections in different amount of cores.

(KCUC vs KTUT). At the same time, the coroutine priority scheduling provided by COPS framework can cope with different needs well and ensure the reasonable allocation of system resources.

REFERENCES

[1] 2022. *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891).* https://docs.xilinx.com/api/khub/documents/sbPbXcMUiRSJ2O5STvuGNQ/content
[2] 2023. *AXI 1G/2.5G Ethernet Subsystem v7.2 Product Guide.* https://docs.xilinx.com/r/en-US/pg138-axi-ethernet
[3] 2023. embassy-rs/embassy: Modern embedded framework, using Rust and async. https://github.com/embassy-rs/embassy
[4] alvinashcraft. 2022. *I/O Completion Ports - Win32 apps.* https://learn.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports
[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert

Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[6] Jonathan Corbet. 2017. *KAISER: hiding the kernel from user space [LWN.net]*. https://lwn.net/Articles/738975/

[7] Ewoud Van Craeynest. 2017. Asynchronous programming with Coroutines in Python. In *FOSDEM 2017*.

[8] Adam Freeman. 2022. Coordinating Goroutines. *Pro Go* (2022), 811–835.

[9] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. 2004. Comparing and Evaluating epoll, select, and poll Event Mechanisms. https://api.semanticscholar.org/CorpusID:8488207

[10] Jon Howell, Bill Bolosky, and John (JD) Douceur. 2002. Cooperative Task Management without Manual Stack Management. In *Proceedings of USENIX 2002 Annual Technical Conference* (2002-01). USENIX. https://www.microsoft.com/en-us/research/publication/cooperative-task-management-without-manual-stack-management/ Edition: Proceedings of USENIX 2002 Annual Technical Conference.

[11] Shuveb Hussain. 2020. *Welcome to Lord of the io_uring — Lord of the io_uring documentation*. https://unixism.net/loti/index.html

[12] M. T. Jones. 2006. Boost application performance using asynchronous I/O. *IBM Developer* (2006). https://developer.ibm.com/articles/l-async/#:~:text=Summary,CPU%20resources%20available%20to%20you

[13] Michael Kerrisk. [n. d.]. *vdso(7) - Linux manual page*. https://man7.org/linux/man-pages/man7/vdso.7.html

[14] Edward A. Lee. 2006. *The Problem with Threads*. Technical Report UCB/EECS-2006-1. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html The published version of this paper is in IEEE Computer 39(5):33-42, May 2006..

[15] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. 2019. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs.. In *USENIX Annual Technical Conference*. 603–616.

[16] Dingding Li, Niyang Zhang, Mianxiong Dong, Hao Chen, Kaoru Ota, and Yong Tang. 2021. Pm-aio: An effective asynchronous i/o system for persistent memory. *IEEE Transactions on Emerging Topics in Computing* 10, 3 (2021), 1558–1574.

[17] Peng Li and Steve Zdancewic. 2007. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. 42, 6 (2007), 189–199. https://doi.org/10.1145/1273442.1250756

[18] Zell Liew. 2017. *Callbacks in JavaScript | Zell Liew*. https://zellwk.com/blog/callbacks/

[19] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards programming heterogeneous memory asynchronously. *ACM SIGPLAN Notices* 51, 4 (2016), 369–383.

[20] Xuhao Luo, Weihai Shen, Shuai Mu, and Tianyin Xu. 2022. DepFast: Orchestrating Code of Quorum Systems. (2022).

[21] Chris D. Marlin. 1980. Coroutines: A Programming Methodology, a Language Design and an Implementation. *Lecture Notes in Computer Science* 95 (1980).

[22] David Mazières. 2021. *My tutorial and take on C++20 coroutines*. https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html

[23] Maryam Mohsin. 2023. *10 Google Search Statistics You Need to Know in 2023 | Oberlo*. https://www.oberlo.com/blog/google-search-statistics

[24] Ana Lucia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *Acm Transactions on Programming Languages & Systems* 31, 2 (2009), 1–31.

[25] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. 2019. LXDs: Towards Isolation of Kernel Subsystems.. In *USENIX Annual Technical Conference*. 269–284.

[26] rcore os. 2023. rCore-Tutorial-v3. https://github.com/rcore-os/rCore-Tutorial-v3.

[27] Kevin Rosendahl. 2017. *Green threads in rust*. Ph. D. Dissertation. Master's thesis, Stanford University, Computer Science Department.

[28] smoltcp rs. 2023. *smoltcp*. https://github.com/smoltcp-rs/smoltcp

[29] Roman Elizarov;Mikhail Belyaev;Marat Akhin;Ilmir Usmanov. 2021. 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*.

[30] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. 37, 5 (2003), 268–281. https://doi.org/10.1145/1165389.945471

[31] Andrew Waterman, Krste Asanovic, and CS Division. 2019. Volume I: Unprivileged ISA. (2019).

[32] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. https://doi.org/10.1145/3477132.3483569