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

Anonymous Author(s)
Submission Id: 59

ABSTRACT

Multithreading technology, a classic concurrency model, has been widely supported by many operating systems and is widely used in various applications. However, as concurrency increases, the high system overhead and context-switching costs of the multithreading model gradually become unacceptable. The user-level multithreading, although reduces the context-switching overhead, but is only supported in userland and not perceived by the kernel, which leads to the kernel cannot accurately estimate the workload of tasks in the system nor execute more accurate scheduling policies to achieve higher system resource utilization. Based on the above situation, we proposes SharedScheduler with the following characteristics: 1) It introduces coroutines into the kernel and applications as the smallest task unit, which is aimed to replace the traditional multithreading concurrency model. 2) It provides a unified priority-based scheduling framework for both user coroutines and kernel coroutines by attaching priorities to coroutines. 3) It uses kernel coroutines as the bridge between I/O operations and device interruptions to provide asynchronous I/O mechanism. Finally, we evaluates the characteristics of the SharedScheduler in a general web server scenario. In high-concurrency scenario, using coroutines can significantly reduce the context-switching overhead brought by the multithreading model, resulting in a throughput increase of 1.05-3.93x. Moreover, the priority-based scheduling ensures the effective resource utilization.

CCS CONCEPTS

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

KEYWORDS

Multithreading, Concurrency, Coroutine.

1 INTRODUCTION

As concurrency and scalability increases, a large-scale web server must accommodate tens thousands of simultaneous client connections without significant performance degradation. As of 2022, Google's servers are handling 883 billion requests per day [17], with an average of 8 million requests per second. For such a large amount of concurrent requests, multithreading model programs have achieved a certain degree of success, but as the demand gradually expands, the disadvantages of multithreading model are gradually exposed.

1) Threads, as a model of computation, are wildly nondeterministic [11]. Under the multithreading model, the uncertain execution sequence of programs leads to the uncertain accessing sequence of shared resources. Although reasonable synchronous exclusion mechanism can ensure the correct and safe accessing of shared resources in the concurrent environment, using multithreading is undoubtedly trying to obtain a deterministic goal in an uncertain way. In the concurrent scenario of web server, what needs to be done is receiving request package, processing it and sending response to the client. Using multithreading model blindly in such a deterministic resources accessing sequence scenario is very inappropriate.
2) The multithreading model is difficult to be combined with asynchronous I/O mechanism. Although it can be combined with the event-driven model underlying the operating system, it will increase the complexity of the operating system. For example, Linux provides system calls such as select and epoll [7] to support user-level asynchronous I/O tasks by reusing multiple I/O operations on a single thread. The epoll approach combines a single thread with an event-driven model, which leads to increased kernel complexity, requires interaction through the producer-consumer model and increases the overhead of synchronous mutual exclusion. Windows' I/O Completion Ports (IOCP) [2] provides a similar I/O multiplexing mechanism, which is a truly asynchronous I/O operation by using callback functions. However, due to the asynchronous callback functions, it is very difficult to develop a correct progame. The io_uring [9] approach utilizes shared memory between userland and kernel to avoid memory replication, thereby improving IO processing efficiency and throughput. However, excessive design leads to increased kernel complexity and greater difficulty in utilizing interfaces. In addition, some asynchronous interfaces implemented directly in userland are also not suitable because the operating system is unaware of asynchronous tasks, there is a lot of overhead, including thread creation, scheduling, destruction, I/O buffer replication, and cross-privilege context switching, such as POSIX AIO [10].
3) Threads is becoming expensive. The system threads need a complete set of hardware thread abstraction (stack, general register, program counter, etc.). Not only does it require more resources, but its context-switching is expensive which will bring serious performance degradation in large-scale concurrency scenario. Although user-level threads can reduce context-switching overhead and increase flexibility, it cannot be perceived by the operating system which prevent performing fine-grained resource scheduling.

ICPC 2024, April 2024, Lisbon, Portugal
© 2024 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

A lot of research has been done on the problem that the multithreading model is not suitable for large-scale concurrent web server scenario, but most still choose to continue using threads [8, 13] which cannot fundamentally solve the problem. In addition to the above problems, the asynchronous I/O mechanism is also needed in the large-scale concurrent web server scenario, which brings serious challenges to the operating system. The operating system 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. Although there has been some research, for example, LXDs [18] has developed a lightweight, asynchronous runtime environment in the kernel for cross-domain batch processing. Memif [14] provides a low latency and low overhead interface based on asynchronous and hardware-accelerated implementations; Lee et al. [12] significantly improved application performance by reducing I/O latency by introducing the Asynchronous I/O stack (AIOS) to the kernel; But these methods are often independent of the kernel thread scheduler, resulting in a lack of generality and scalability, and increasing the complexity of the kernel.

In this paper, we propose SharedScheduler, a coroutine-based scheduling framework perceived by the operating system. SharedScheduler 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, SharedScheduler draws on the existing research on asynchronous I/O mechanism, introduces coroutines into the kernel, and combines it with the asynchronous I/O mechanism to provide a coordinated and unified scheduling framework for all tasks in the operating system.

2 MOTIVATION

In recent years, coroutines have attracted much attention, DepFast [15] uses coroutines in distributed arbitration systems; Capriccio [21] uses cooperative user-level threads to achieve a scalable, large-scale web server. Based on previous studies, we believe that coroutines are very useful in constructing large-scale concurrent programs.

Coroutine 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, coroutine has lower resources requirement and context-switching overhead. Modern programming languages, For example, c++ 20 [16], Go [6], Rust [19], Python [5], Kotlin [20], etc., all provide varying degrees of support for coroutines. It can be divided into the following two categories by the implementation:

1) Stackful coroutines: It is also considered the user thread. The goroutine is a stackful coroutine implementation in the Go programming language that simplifies and enhances concurrent programming, allowing for a large number of concurrent executions within a single system thread. In the Go language, a goroutine is created using the keyword "go" followed by a function call. This function call runs concurrently with other goroutines in the same address space. Goroutine has its own running stack space, dynamically allocated and managed by the integrated runtime within the Go language, which can be used to hold local variables and function call relationships. Therefore, compared to the system thread, the stackful coroutine reduces the context-switching overhead but with limited effect.
2) Stackless coroutines: The other class is the stackless coroutines. Coroutines supported by the rust language are one of them. Rust Coroutines are implemented through the async/await syntax and the corresponding runtime library. In contrast to user-level threads, Rust coroutines are stackless and do not require a fixed size of stack space to be pre-allocated. Instead, the stack space for coroutines is dynamically allocated and released on demand. This makes the creation and destruction of coroutines very lightweight and efficient.

On the basis of the above analysis, as well as academic and industrial interest in coroutines, we rethink the concurrency model and asynchronous framework to find solutions that could meet the larger-scale and higher performance requirements. We choose to use the stackless coroutine to replace the traditional multithreading model in large-scale concurrent web server scenario, and we choose 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 preceived 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 scheduling framework perceived by the operating system, which can provide a coordinated and unified scheduling framework for all tasks in the operating system and provide a unified asynchronous I/O framework to meet the requirement of high concurrency.

3 DESIGN

This section introduces the design of SharedScheduler. Figure 1 shows the overall framework for SharedScheduler. The application and the operating system maintain their Executor data structure respectively and share the scheduling framework provided by SharedScheduler through vDSO. Applications and tasks within the operating system are described in the form of coroutines. The SharedScheduler'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.

Next, we will cover the design details of SharedScheduler in the rest of this section. We will first introduce the data structure related to the coroutine runtime provided in SharedScheduler to describe how to implement the scheduling of coroutines(3.1). Next, we will introduce the concurrency of SharedScheduler(3.2). Then, we will introduce the state transition model of the coroutine (3.3). Finally, we describe the global cooperative scheduling mechanism provided by SharedScheduler (3.4).

3.1 Coroutine Runtime

The Rust language provides two high-level abstractions, **Future** and **Wake**, to support the coroutine mechanism without limiting

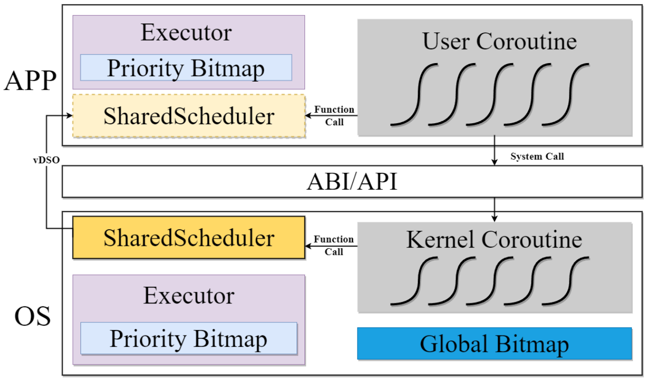


Figure 1: The architecture of SharedScheduler.

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.*

The Rust language provides the async/await keyword, which makes it very easy to create and execute coroutines. The compiler compilers the function with the async keyword into a finite state machine, and the coroutine execution is facilitated by polling. This polling process is partially transparent, so we cannot accurately complete the control of 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 1:

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

Listing 1: Coroutine control block

How to drive the execution of the coroutine and how to switch and save the context of the coroutine are the most important issues. Fortunately, Rust already offers two relatively well-developed abstractions, future and wake. The poll function required for the future abstraction is used to drive coroutine execution, while the Wake abstraction is closely related to save and switch 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 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, SharedScheduler 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 SharedScheduler's scheduling framework.

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.

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 SharedScheduler, which ensures that the coroutine priority scheduling in SharedScheduler 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, SharedScheduler 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.

When an application requests more CPU, because all the coroutines are managed by the Executor, the scheduling must be done through the Executor, so we use the mutex to ensure proper access to the Executor data structure. Despite the uncertainty introduced here, it is handled by the synchronous mutual exclusion mechanism inside SharedScheduler, which is completely transparent to the application. This approach has a certain impact on the application programming paradigm, application developers no longer need to consider the synchronous mutual exclusion problem of multi-threaded concurrency, only need to define the coroutine task and the order of access to shared resources between coroutines, reducing the development burden.

3.3 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) [4], 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 state. The coroutine state transition model is shown in Figure 2.

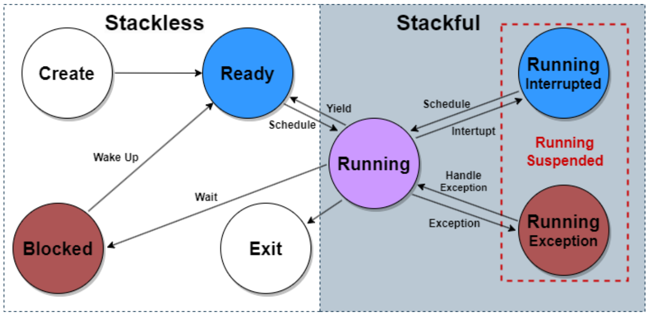


Figure 2: Coroutine state transition model.

(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.

3.4 Global Cooperative Scheduling Mechanism

On the basis of the coroutine priority scheduling, SharedScheduler 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 preceive 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 SharedScheduler. 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 article.

Through the above mechanism, SharedScheduler provides an operating system aware coroutine-based scheduling framework. Figure 3 shows the flowchart of SharedScheduler.

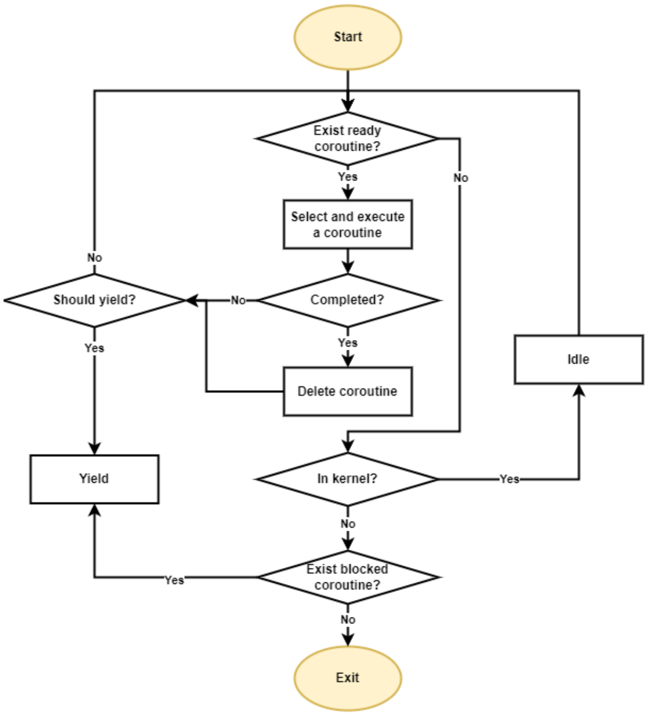


Figure 3: The control logic of SharedScheduler.

4 BUILD PROGRAM WITH SHAREDSCHEDULER

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

4.1 The Interface of SharedScheduler

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, all tasks exist in the form of coroutines, in a cooperative execution environment. What the main coroutine needs to do is using the interface of SharedScheduler to create different coroutines. The running of coroutines is transparently driven by the coroutine runtime provided by SharedScheduler. SharedScheduler provides the interfaces to application developers in Table 1:

Table 1: Interface of SharedScheduler

Interface	Description
spawn(future, prio)	Create a new coroutine with specific priority.
getcid()	Get the Id of current coroutine.
wake(cid)	Wake up the specific coroutine.
reprio(cid, prio)	Adjust the priority of the target coroutine.
alloc_cpu()	Allocate more cpu to support a higher degree of concurrency.

4.2 Asynchronous system call

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 coroutines continue to be drived. After analyzing, 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" [1]. 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 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 Rust language future abstraction, 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 2.

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.

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

Listing 2: System call interface of read().

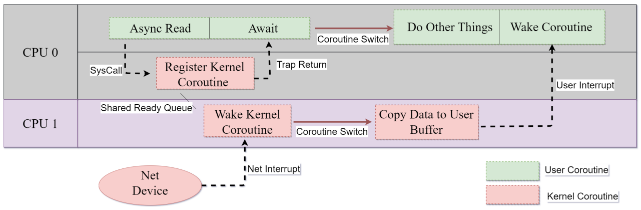


Figure 4: Asynchronous system call.

We will use the example of asynchronously reading data from sockets to explain how coroutines can be combined with asynchronous I/O mechanism. 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 coroutine, waiting for the kernel to finish reading operation. At this point, SharedScheduler switches and executes the next user coroutine. The kernel coroutines 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 SharedScheduler in building highly concurrent asynchronous programs and more accurate control of coroutines, we build a five-level RISC-V pipelining processor that supports N extension [22] in FPGA. We run an operating system based on SharedScheduler framework on the RISC-V processor, and finally complete the evaluation of SharedScheduler by simulating the real web server application scenario. The development board model is ALINX axu15eg which is equipped with Zynq UltraScale+ XCZU15EG-2FFVB1156 MPSoC. The RISC-V subsystem is based on rocket-chip project [3]. Since asynchronous system calls relys on the relevant functions of user-level interruption, we implement N extension for it. The total configuration parameters are shown in Table 2.

Table 2: Configuration of evaluation

RISC-V CPU	Cores	4
	Frequency	100MHz
	Memory Size	2GB
Device	Ethernet	Xilinx AXI 1G/2.5G Ethernet Subsystem
	DMA	Xilinx AXI DMA Controller
Operating System	rCore-tutorial	The teaching operating system used by Tsinghua University.
Network Stack	smoltcp	A standalone, event-driven TCP/IP stack for bare-metal, real-time systems.

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, 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.

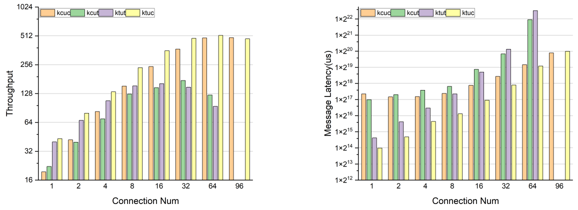
Finally, the client on the PC calculates the time latency between sending each request and receiving the response, and calculates the throughput of messages within a fixed period of time. We evaluate SharedScheduler by analyzing the time latency and throughput of the web server under different configurations.

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 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.

(1) 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.
(2) 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.
(3) 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.
(4) 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 coroutine 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 Figure 5:



(a) Throughput

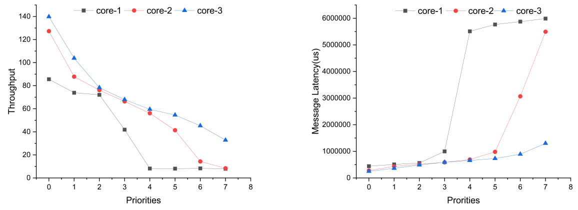
(b) Latency

Figure 5: Throughput and message latency.

- SharedScheduler's synchronous mutual exclusion mechanism overhead: The concurrency provided in SharedScheduler results in that the synchronous mutual exclusion overhead is increased to ensure that data structures such as Executor are accessed correctly. This can be seen from the comparison of KCUC vs KCUT, KCUT vs KCUT, and KCUC vs KTUT in Figure 5. Note that when the amount of connections is small, due to the allocation of excess CPU to the application, SharedScheduler yield frequently because of no ready coroutine, which results in SharedScheduler frequently acquires Executor locks when scheduling the kernel coroutine. 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. Therefore, SharedScheduler is not applicable to applications with low concurrency or low latency.
- Lower coroutine context-switching overhead: According to 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: when the number of connections is small, the KTUT 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. 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 SharedScheduler is suitable for large-scale concurrent scenario.

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 SharedScheduler. 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 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.



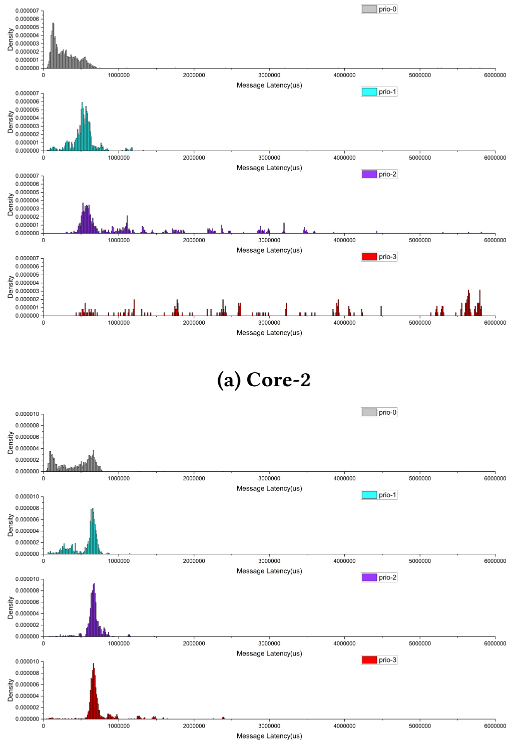
(a) Throughput **(b) Latency**

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 Figure 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 CONCLUSION

This paper proposes SharedScheduler, a coroutine scheduling framework that can be preceived by the operating system. SharedScheduler make the kernel preceive the user-level coroutines by the priority bitmap mechanism and combines kernel coroutines with asynchronous I/O mechanisms. It is proved that SharedScheduler 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 SharedScheduler framework can have the characteristics of high throughput and low latency in the construction of highly concurrent applications. Using the coroutine abstraction provided by SharedScheduler can increase throughput to 1.05x-3.93x than threads (KCUC vs KTUT). At the same time, the coroutine priority scheduling provided by SharedScheduler framework can cope with different needs well and ensure the reasonable allocation of system resources.



(a) Core-2

(b) Core-4

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

REFERENCES

[1] 2017. *Callbacks in JavaScript | Zell Liew.* https://zellwk.com/blog/callbacks/

[2] alvinashcraft. 2022. *I/O Completion Ports - Win32 apps.* https://learn.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports

[3] 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

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

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

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

[7] 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

[8] 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.

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

[10] 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

[11] 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..

[12] 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.

[13] 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

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

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

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

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

[18] 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.

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

[20] 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*.

[21] 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

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