A Design and Implementation of Rust Coroutine with priority in Operating System

1st Wenzhi Wang
College of Information Engineering
Capital Normal University
Beijing 100048, China
wwzcherry@gmail.com

2nd Yong Xiang
dept. of Computer Science and Technology
Tsinghua University
Beijing 100084, China
xyong@mail.tsinghua.edu.cn

3rd Weizhen Sun
College of Information Engineering
Capital Normal University
Beijing 100048, China
sunweizhen@mail.cnu.edu.cn

4th Jingbang Wu
School of Computer Science and Engineering
Beijing Technology and Business University
Beijing 100048, China
wujingbang@btbu.edu.cn

***Abstract*—Coroutines are usually used as an upper-level abstraction of the asynchronous mechanism, which is an effective tool for concurrent programming due to the low cost of cooperative switching and is widely used in applications and the kernel. However, flexible scheduling algorithms and convenient asynchronous support for the kernel are still lacking at present.**

This paper proposes a prioritized coroutine model that provides asynchronous support for both the applications and the kernel. In addition, we unified the scheduling of all coroutines in the system, which means that we can control all coroutines in the system from a global perspective.

Further, we present a reference implementation of a library based on the rust language, and compare the performance difference between threads and the coroutines implemented in this paper through experiments. The results show that with sufficient concurrency, the performance of the coroutine is significantly better than the thread.

***Index Terms*—Coroutines, Asynchronous Programming, Operating System, Scheduler, Rust**

I. Introduction

Coroutine is a lightweight scheduling unit, which can be regarded as the superior abstraction of the asynchronous function. The cooperative switching between coroutines will not cause the switching of the page table (process) nor the stack (thread) and does not require the participation of the kernel. The stack switching overhead of threads will become the performance bottleneck of multi-threading technology under high concurrency [1]. At this time, coroutines will be an effective solution [2] .

A. The Coroutine

The earliest description of the coroutine was proposed by Melvin Conway [3] in 1958, and it is a popular abstraction in computer science that has developed over the years. In the most general sense, coroutines [4], [5] are a generalization of subroutines, equipped with the ability to suspend their own execution, and be resumed by another part of the program later, at which point the coroutine continues execution at the point it previously suspended itself, up until the next suspension or the termination of the coroutine. It can be seen that it performs activities in an asynchronous manner.

The *stacked coroutine* [6] is similar to the thread which holds the call information on the stack. As a coroutine is created, stack memory is allocated to hold the context of the coroutine, and if a child coroutine is created, it continues to be pushed to the stack. The feature of the stacked coroutine is that users do not need to worry about the scheduling. The disadvantage is that it is difficult to allocate a reasonable stack size. The state-of-art languages that use stacked coroutines are Lua [7] and Golang [8].

Stackless coroutines are used like functions, which is usually stored in heap memory. It will not be allocated its own stack but use the thread's stack. The stackless coroutine is characterized by the low overhead of coroutine switching. The disadvantage is that coroutine scheduling is unbalanced and users need to intervene in the scheduling. Popular languages such as C++ [9], Rust [10], Kotlin [11] and Python [12] use stackless coroutines. The coroutine model in this paper is also stackless.

B. Rust and rCore OS

Performance, memory safety, asynchronous support, based on these three advantages, we recommend Rust as the implementation language. It is the same reason why Google's cross-platform operating system Fuchsia and meta's cryptocurrency Libra choose it.

Rust's asynchronous support refers to its *async/await* programming model. We can get asynchronous functions by modifying ordinary functions with the *async* keyword, and start asynchronous execution through *await* keyword.

rCore OS [13] is a Unix-like operating system developed by Tsinghua University. Many designs of its kernel make full use of the features of the Rust language to facilitate the extension of asynchronous mechanisms. This paper introduced the implemented coroutine library to rCore OS and conducted performance tests.

C. Related Work

Thanks to the fact that asynchronous operations allow for greater concurrency, coroutines have been used in many user-mode programs in recent years. Corobase [14] adopts the coroutine-to-transaction model to hide data stalls during data fetching. Zhu et al. [15] introduce the coroutine-to-SQL model to reduce the waiting cost of interactive transactions. FaSST [16] and Grappa [17] use the C++ coroutine package to hide the RDMA network latency. Nelson et al. [18] developed a runtime system for coroutines, which supports sufficient concurrency to tolerate the long latencies of their large high-bandwidth global memory system. Knoche [19] presented a coroutine-based approach to mitigate the potential performance penalty to batch jobs when migrating to microservices.

However, none of the above coroutines support priority scheduling. There will always be multiple ready coroutines in high concurrency, and their execution order cannot be distinguished by urgency. To make up for this shortcoming is one of the tasks of this paper.

The asynchronous mechanism also has powerful benefits for the work of the kernel. LXDs [20] develops an asynchronous runtime in the kernel to run lightweight cooperative threads, which can efficiently implement batching and pipelining of cross-domain calls. memif [21] is an OS service for memory movement, which implements a low-latency and low-overhead interface based on asynchrony and hardware acceleration. Lee et al. [22] reduced the I/O delay through the asynchronous I/O stack (AIOS) proposed for the kernel, so that the performance of the applications was significantly improved. There are many other works focusing on asynchronous I/O [23]–[26]. Sundaresan et al. [27] uses the asynchronous mechanism reduce synchronization bottleneck in the kernel to obtain a higher degree of parallelism.

It is easy to find that the asynchronous mechanism has a strong improvement effect in many scenarios of the applications and the kernel. However, current coroutines are only task units within a single application, and if the kernel expects to use the asynchronous mechanism, it usually needs to construct the asynchronous runtime and coroutine control block from scratch. So we started to conceive the work of this paper to make up for the above deficiencies.

D. Contributions and Paper Organization

This paper has the following two main contributions:

- This paper proposes a coroutine model with priority, which does not depend on the support of the operating system, and thus it can be used in both user-mode programs and the kernel. In addition, by introducing priority bitmaps, we unify the coroutine scheduling of user mode and kernel mode, which means that all coroutines in the system will be in the same scheduling framework.
- We present a reference implementation of a library [1] based on the Rust language, and verify through experiments that coroutines are significantly better than threads under high load conditions.

This paper will present the design and implementation details of the coroutine model in the next second section, give a performance comparison experiment in the third section and make a conclusion in the fourth section.

II. Design and Implementation

A. Overview

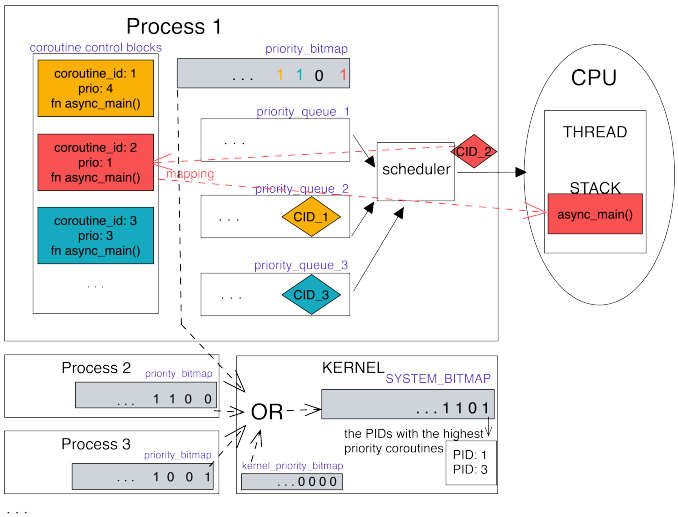

Fig. 1. The architecture of the coroutine model.

As shown in Fig. 1, the Rust language provides asynchronous functions, and the coroutine library provides priority and coroutine ID fields and encapsulates them into a data structure called a coroutine control block. Each coroutine control block represents a coroutine. And a coroutine can be regarded as a scheduling object corresponding to a global asynchronous function one-to-one. We use the coroutine ID as an index to store all the coroutines in a hash table on the process's heap memory. We can get or delete its corresponding coroutine control block through the coroutine ID.Considering that the coroutine will enter and exit the scheduler many times during its life cycle due to waiting for external events, compared to the scheduler directly managing the coroutine control block, it can reduce the memory overhead when the scheduler works.

The coroutine scheduler provides two necessary operations: push and pop, which are used to insert and delete coroutines, and implement the coroutine scheduling algorithm. For priority scheduling, we will set up multiple queues, each with a different priority, and store the coroutine ID at the corresponding priority in it. Query these queues from high to low priority to complete priority scheduling.

We use the priority bitmap to describe the priority layout of the coroutines within each process. We also use a bitmap in the kernel to describe the priority layout of the coroutines of all processes. The kernel can perceive the existence of coroutines and their priority through these bitmaps, and formulate the

[1] https://github.com/AmoyCherry/AsyncOS

scheduling strategy of the process, thereby indirectly intervening in the coroutine scheduling in the user mode. This makes coroutine scheduling no longer an independent behavior within each process but allows them to be considered uniformly from a global perspective. Its design and implementation will be introduced in Section 2.3.

The execution of functions must depend on the stack, the same is valid for asynchronous functions. For this, we have the following design: When a coroutine needs to be executed, we start the coroutine executor, which is essentially a function. The coroutine executor will take out all the coroutines in the ready state and run them in a loop. The coroutine will use this thread's stack to execute its asynchronous function. When the coroutine is completed or suspended due to waiting for external events, it will exit the occupied thread stack by returning the function. At this time, the thread running the coroutine scheduler can take out the next coroutine from the scheduler to run.

The coroutine needs to be woken up after it yields because it waits for external events. Wake-up is a key mechanism of the coroutine. This part is described in detail in Section 2.4.

B. Interface

The coroutine model will be presented as a function library, which only provides an interface-*coroutine_run*, it will be used to create coroutines and start the coroutine executor.

Algorithm 1 COROUTINE_RUN($async\_main, priority = DEFAULT$)

```
create_coroutine(async_main, priority)
if get_coroutine_size() <= 1 then
    run()
end if
```

We use the *async* keyword provided by the Rust language to create an asynchronous function, where the asynchronous code statement needs to be decorated with the *await* keyword, and this asynchronous function will run as the main function of the coroutine. When no coroutine priority is specified it will be given the default value. If we need to execute a batch of coroutines, this interface should be nested in an asynchronous function to create coroutines in batches. When the first coroutine is created, the interface needs to call the run function to start the coroutine executor; when the number of coroutines is greater than one, it means that the executor has been started, and this interface only needs to complete the creation of the coroutine.

C. Priority and Schedule

We handle the scheduling order of coroutines based on priority. Within a process, coroutines are executed from high to low priority; between processes, the process with the highest priority coroutine will be scheduled first.

1) Coroutine scheduling within a process: Each coroutine has a priority. After the coroutine is created, it will be inserted into the corresponding priority queue and wait to be scheduled for execution. The scheduler scans the queues according to the priority from high to low, and searches for the first non-empty queue to complete the priority scheduling. Only the coroutines in the ready state are saved in the queues, and the coroutines in the waiting state will be reinserted into the queue after being awakened, which ensures the query efficiency of the scheduler.

2) Priority Bitmap: Based on the previous design, the priority scheduling of coroutines can be implemented within the process. Now, we extend it to the entire system to perform priority scheduling on the coroutines of all processes.

We use bitmaps to describe the priority layout of coroutines. Each process will maintain a bitmap whose length is the number of priorities of the coroutines, and the value of each bit is zero or one, indicating whether there is a coroutine in the ready state of that priority.

We prefer to design the coroutine model in operating systems where the kernel has its own address space and page tables, so the kernel also has a priority bitmap to represent the priority information of the coroutines it manages, called the kernel bitmap.

In addition, we set an extra bitmap in the kernel to represent the priority information of the coroutines of the whole system, which is called the system bitmap. But the zero or one of each bit represents whether there are coroutines of this priority in all processes and the kernel of the entire system.

According to the definition of the system bitmap, we can know that its value is the result of the OR operations of all processes and kernel bitmaps-as long as at least one coroutine of at least one process belongs to this priority, the corresponding bit in the system bitmap is one; and only when all processes do not have coroutines of this priority, the corresponding bit in the system bitmap will be zero.

Now we explain how the kernel gets all the bitmaps. Our approach is to specify a virtual address as the address of the process accessing the bitmap in user mode. Then, when the process is initialized, apply to the kernel for a free physical page, map the specified virtual address to the starting address of the physical page, and grant the process permission to read and write this address. In this way, the process can read and write the bitmap in user mode, and the kernel can also directly access the bitmap of the process. We map each process ID with the physical address of its process bitmap, and we can get all process bitmaps in the kernel by enumerating the IDs of the processes.

3) Global Scheduling Based on Priority Bitmap: Based on the priority bitmap, we can guarantee the global priority scheduling when the clock interrupt arrives: the system bitmap will be updated when the clock interrupt arrives, and then the process or kernel with the highest priority coroutine can be scheduled to run according to the result, and the highest priority coroutine will be scheduled to run in this address space, We call this **relaxed global priority scheduling**.

Within one clock cycle, the changes in the priority layout of coroutines caused by the creation, wake-up and deletion of coroutines are synchronized to the system bitmap in time, so that the process with the highest priority coroutine is always prioritized scheduling, we call this **strict global priority scheduling**. Our specific measures are: the bitmap of the process can always be updated in real time by the execution of the coroutines, so before each execution of the coroutine, compare the highest priority recorded in the process bitmap and the system bitmap, If and only if there is a coroutine with a higher priority recorded in the system bitmap, it will enter the kernel through a system call to update the system bitmap, and then reschedule according to the result.

Obviously, adopting strict global priority scheduling will ensure that the highest priority coroutine in the system is always executed first, but it will also introduce a lot of overhead by frequently entering the kernel within a clock cycle, and relaxed scheduling is the opposite. Fortunately, which scheduling strategy to adopt is the behavior of the coroutine executor itself, and we can control which strategy it chooses through a parameter according to needs.

D. Coroutine Manager and Asynchronization

The Coroutine Manager is responsible for managing the coroutine control blocks, the scheduler, and the coroutine wakeup mechanism introduced in this section.

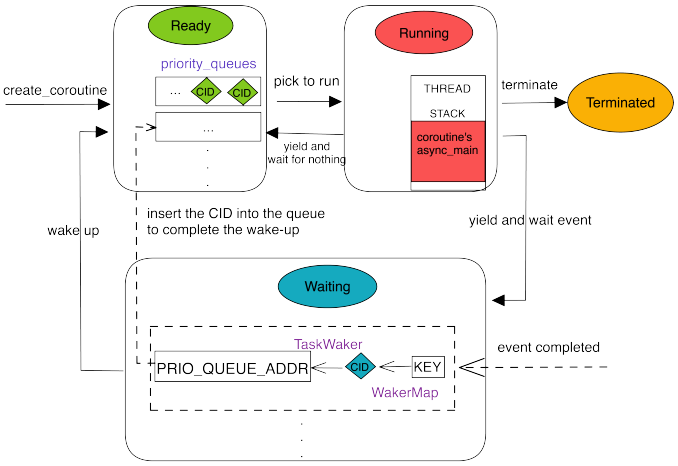

Fig. 2. The state transitions of a coroutine during its lifetime.

Only the IDs of the coroutines in the ready state are saved in the priority queue of the scheduler, so the wake-up operation can be completed by inserting the IDs of the coroutines awakened from the waiting state into the queue, and it only needs the ID and priority of the coroutine, and the address of the queue. We can encapsulate these three variables into a data structure called TaskWaker, and establish a mapping between the coroutine ID and the corresponding TaskWaker. So when a coroutine needs to be woken up, only its coroutine ID is needed. We will check whether its TaskWaker is created before each coroutine is executed, and delete it when the coroutine is deleted. Obviously, TaskWaker should be managed by the coroutine manager.

Now the coroutine executing the external event can use TaskWaker to wake up the suspended coroutine. But the key question is how it gets the ID of the coroutine it is going to wake up. We introduce a data structure called WakerMap here, which maintains the mapping of integer key values to coroutine IDs. For the waiting coroutine and the coroutine that performs the wake-up operation, they need to pass in the same key value when they are created (this requires the use of a global counter with mutually exclusive access within the process) to establish a connection. The waiting coroutine will insert the key and ID into WakerMap, and the wake-up coroutine can access TaskWaker and WakerMap through the same key to wake it up. If the wake-up coroutine is executed first, a null value will be read when accessing the WakerMap and the wake-up operation will be omitted, and then the waiting coroutine will not be suspended because the waiting event has already arrived (for example, the buffer has written enough data).

III. Performance Comparison

We introduce this coroutine library on rCore OS to compare the performance between coroutines and threads. Each experiment creates multiple processes to run, and each process runs coroutines or threads, which will perform some of the same work. Two times are collected at the beginning and end of the process, and the difference between them is used as the running time of each test. We will compare multiple runtimes under multiple concurrency levels-the number of coroutines or threads, and summarize the performance characteristics of coroutines and threads from them.

We conduct experiments on the RISC-V architecture simulated by QUME [28], which can emulate the machine's processor through dynamic binary translation and enable it to run various guest operating systems.

A. Experiment A: Test in User Mode without the Syscall

In this experiment, we test the performance of coroutines and threads in user mode, and will not actively call system calls to enter the kernel. In order to reflect the asynchronous characteristics of coroutines and the context switching of threads, coroutines or threads will mutually exclusive read and write a global variable. Specifically, multiple coroutines or threads will be created in the process, and they will be numbered sequentially starting from one, and the value of the global variable will be initialized to zero. When all coroutines or threads are created, the main thread of the process will modify the value of the global variable to one. Each coroutine or thread will read the global variable, and if the value is equal to their own number, they will add one to the value of the global variable. Otherwise, the coroutine will be suspended through the asynchronous mechanism, and the thread will be switched through the yield system call.

We tested the execution time under each concurrency from 200 to 4000 with an interval of 200, and the test results are

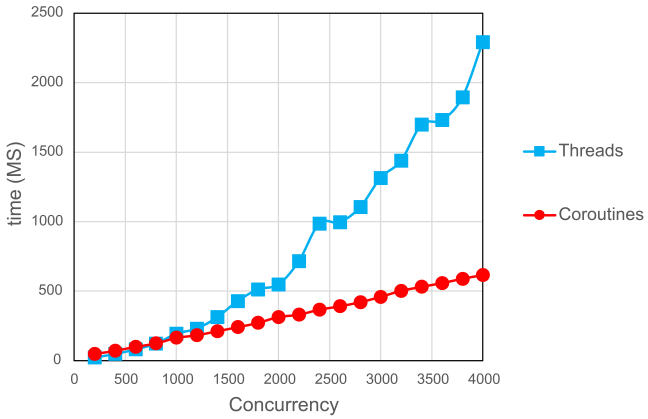

Fig. 3. Coroutines or threads mutually exclusive read and write a global variable in user mode.

TABLE I
THE START, INTERSECTION, AND END POINTS OF FIG. 3.

Concurrency	time (MS)	
	Threads	*Coroutines*
200	22	49
800	122	124
1000	193	164
4000	2090	616

shown in Fig. 3 and table I. We can see that when the amount of concurrency is small, the execution time of coroutines is slightly higher than that of threads. However, as the amount of concurrency increases, the execution time of coroutines will increase steadily along a slash, while the growth rate of threads will be much faster, and the difference between the two will gradually increase. When the concurrency is between 800 and 1000, their execution time will be equal, and then the performance of coroutines will exceed that of threads. When the concurrency is between 4000, the total execution time of coroutines is about 26% of threads.

B. Experiment B: Test in user mode with syscall

Here we will test the performance of coroutines and threads under high concurrency to read and write kernel buffers through system calls. We still need to ensure that the experiment can reflect the asynchronous characteristics of the coroutine and the context switching of the thread, so the design experiment is shown in the Fig. 4. For each test, we will create some coroutines or threads numbered from one in the process, and create pipes one more than them in number, and the number of pipes also starts from one. Each coroutine or thread will read fixed size data from the previous pipe and write it to the next pipe. Obviously, when the read is not completed, the coroutine will be suspended through the asynchronous mechanism, and the thread will be switched through the yield system call. After the creation work is done, the main thread of the process will write data to pipe one, and the data will flow between the coroutines or threads and the pipes.

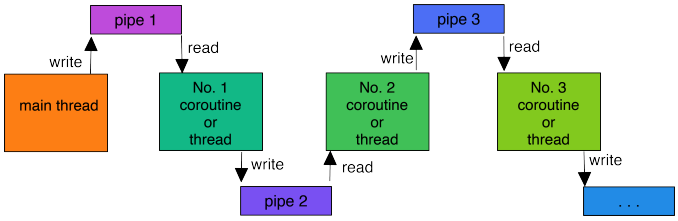

Fig. 4. Experiment B: Connect the front and back coroutines or threads through the read and write ends of the pipes.

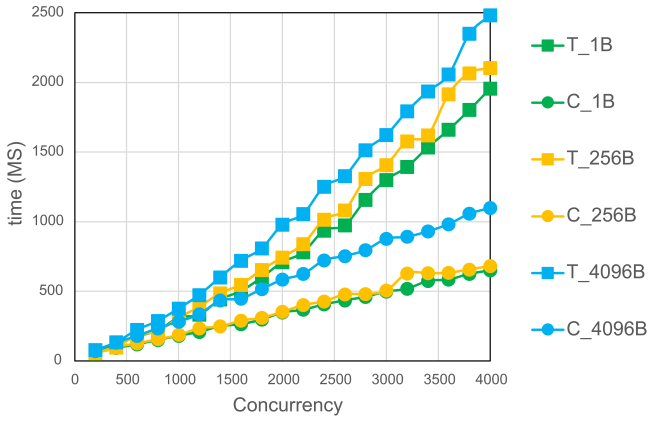

Fig. 5. The performance of coroutines and threads under different data sizes.

Since the following coroutines must wait for the previous coroutines to complete before they can run, we give the first coroutine the highest priority so that the flow of data can start as early as possible.

We tested three different data sizes, 1B, 256B, 4096B, and tested the execution time under each concurrency from 200 to 4000 with an interval of 200. The test results are shown in Fig. 5 and table II. We can see that the relative performance of coroutines and threads is similar to experiment A. When the concurrency reaches 4000, the total execution time of the coroutines is about 32%-44% of the threads.

C. Discussions

From the experiments, we can see that although coroutines and threads are also scheduling units, they have their own characteristics under different concurrency. When concurrency is small, the basic overhead of supporting the coroutine running, including the coroutine scheduler and the wake-up mechanism, has a significant impact, resulting in the performance of the coroutine at this time being close to the thread. However,

TABLE II
ALL ENDPOINTS IN FIG. 5.

Data Size	Concurrency	time (MS)	
		Threads	*Coroutines*
1B	4000	1956	651
256B	4000	2103	681
4096B	4000	2484	1099

as shown in the experimental data, the usage overhead of coroutines increases linearly with the increase of concurrency, which means that the increase of concurrency does not significantly impact the average execution time and average switching time of coroutines. This is a distinguishing feature of the coroutine from the thread and also the significance of the coroutine for concurrent programming. In contrast, the context switching cost of threads increases with the increase of concurrency, which causes the performance of coroutines to gradually exceed threads when the concurrency increases, and the gap between the two becomes larger and larger.

IV. Conclusions

This paper proposes a coroutine model that can provide asynchronous support for applications and the kernel. Additionally we present a reference implementation based on the Rust language and conduct performance tests. The results show that coroutines work well and are significantly better than threads in a high-concurrency environment. When the concurrency reaches 4000, the total execution time of the coroutines is about 26% - 44% of the threads.

References

[1] W.-m. Hwu, K. Keutzer, and T. G. Mattson, "The concurrency challenge," *IEEE Design & Test of Computers*, vol. 25, no. 4, pp. 312–320, 2008.

[2] B. Belson, J. Holdsworth, W. Xiang, and B. Philippa, "A survey of asynchronous programming using coroutines in the internet of things and embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 3, pp. 1–21, 2019.

[3] M. E. Conway, "Design of a separable transition-diagram compiler," *Communications of the ACM*, vol. 6, no. 7, pp. 396–408, 1963.

[4] L. Waern, "Coroutines for simics device modeling language," 2021. http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-453889.

[5] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 2, pp. 1–31, 2009.

[6] X. Wang and H. Huang, "Sgpm: A coroutine framework for transaction processing," *Parallel Computing*, vol. 114, p. 102980, 2022.

[7] A. L. De Moura, N. Rodriguez, and R. Ierusalimschy, "Coroutines in lua," *Journal of Universal Computer Science*, vol. 10, no. 7, pp. 910–925, 2004.

[8] R. Prabhakar and R. Kumar, "Concurrent programming with go," tech. rep., Citeseer, 2011.

[9] B. Belson, W. Xiang, J. Holdsworth, and B. Philippa, "C++20 coroutines on microcontrollers—what we learned," *IEEE Embedded Systems Letters*, vol. 13, no. 1, pp. 9–12, 2021.

[10] D. Weber and J. Fischer, "Process-based simulation with stackless coroutines," in *Proceedings of the 12th System Analysis and Modelling Conference*, pp. 84–93, 2020.

[11] R. Elizarov, M. Belyaev, M. Akhin, and I. Usmanov, "Kotlin coroutines: design and implementation," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 68–84, 2021.

[12] C. Tismer, "Continuations and stackless python," in *Proceedings of the 8th international python conference*, vol. 1, 2000.

[13] W. Yifan and Z. Yiren, "rcore-tutorial." https://github.com/rcore-os/rCore-Tutorial-v3, 2023.

[14] Y. He, J. Lu, and T. Wang, "Corobase: Coroutine-oriented main-memory database engine," *arXiv preprint arXiv:2010.15981*, 2020.

[15] T. Zhu, D. Wang, H. Hu, W. Qian, X. Wang, and A. Zhou, "Interactive transaction processing for in-memory database system," in *International Conference on Database Systems for Advanced Applications*, pp. 228–246, Springer, 2018.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 185–201, 2016.

[17] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 291–305, 2015.

[18] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, M. Oskin, *et al.*, "Crunching large graphs with commodity processors," in *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*, 2011.

[19] H. Knoche, "Improving batch performance when migrating to microservices with chunking and coroutines," *Softwaretechnik-Trends*, vol. 39, no. 4, pp. 20–22, 2019.

[20] V. Narayanan, A. Balasubramanian, C. Jacobsen, S. Spall, S. Bauer, M. Quigley, A. Hussain, A. Younis, J. Shen, M. Bhattacharyya, *et al.*, "Lxds: Towards isolation of kernel subsystems," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 269–284, 2019.

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

[22] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 603–616, 2019.

[23] X. Liao, Y. Lu, E. Xu, and J. Shu, "Write dependency disentanglement with horae," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 549–565, 2020.

[24] D. Li, N. Zhang, M. Dong, H. Chen, K. Ota, and Y. Tang, "Pm-aio: An effective asynchronous i/o system for persistent memory," *IEEE Transactions on Emerging Topics in Computing*, 2021.

[25] K. Magoutis, "Design and implementation of a direct access file system (dafs) kernel server for freebsd," in *BSDCon 2002 (BSDCon 2002)*, 2002.

[26] Z. Xu and J. Huang, "Detecting 10,000 cells in one second," in *International conference on medical image computing and computer-assisted intervention*, pp. 676–684, Springer, 2016.

[27] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 244–255, 2009.

[28] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, vol. 41, pp. 10–5555, Califor-nia, USA, 2005.