Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Exploring Asynchronous Operating System Design Using Rust

Author:
Yifan Wu

Supervisor:
Lluis Vilanova

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

August 2022

Abstract

System call is an important interface between the operating system and applications, there has been many works dedicated to increase the performance of system calls. A recent trend is turning system calls from synchronous to asynchronous which means that the application does not need to immediately trap into kernel mode on a system call. The most notable example is FlexSC[11], it makes the system call interface asynchronous based on a threading library. In this way, the frequency of CPU mode switches can be reduced to mitigate the processor state pollution problem.

Whether the system call interface is synchronous or asynchronous, we focus on the execution of the system calls inside the kernel. Based on zCore[13], an asynchronous operating system developed in Rust[10] which runs on a single core, we added another core to handle all system calls issued by the original core remotely and asynchronously. The results showed that the data locality was better in our implementation, but we have to execute more instructions, leading to the result that the overall performance were similar to the original zCore system on some benchmarks.

Contents

1 Introduction **1**

2 Background **5**
2.1 Technical Background 5
2.1.1 Coroutines 5
2.1.2 Asynchronous Programming in Rust 5
2.2 Academic Background 6
2.2.1 Cohort Scheduling 6
2.2.2 System Call Aggregation 6
2.2.3 Minimizing Granularity of Locks in Kernel 7
2.3 zCore 7
2.3.1 zCore Architecture 7
2.3.2 zCore Execution Model 9

3 Architecture **12**

4 Implementation **14**
4.1 Preparation 14
4.1.1 Multiboot With KVM Enabled 14
4.1.2 KVM Optimization 14
4.1.3 Emulated PCI Block Device Driver 15
4.2 Asynchronous System Call Module 15

5 Evaluation **16**
5.1 Testing Environment 16
5.2 Benchmarks 16
5.2.1 Task0 16
5.2.2 Task1 17
5.2.3 Task2 17
5.2.4 Task3 - Task5 17
5.3 Experiment Result Analysis 18
5.3.1 Task0 18
5.3.2 Task1 19
5.3.3 Task2 20
5.3.4 Task3 - Task5 21

6 Conclusion **24**

7 Future Work **25**

A Benchmark Source Code in C **26**

Chapter 1

Introduction

System call is one of the most important components in the Operating System (OS) since it serves as the interface between the OS and applications. Applications can request services provided by the OS via system calls, and the OS is responsible for allocating resources to the applications according to their requests while the security requirements including the isolation between applications must be guaranteed.

In the classical OS design, system calls are synchronous. To be specific, the application should execute a dedicated instruction provided by the Instruction Set Architecture (ISA) such as *syscall* on x86_64 platform or *ecall* on RISC-V platform. After that, the execution of the application is immediately stopped and an interrupt is triggered and captured by the processor. Then, the processor mode or the privilege level is switched from the user mode to the kernel mode, which means that the control is transferred to the code of the OS. The OS saves the execution context of the application, handles the system call requested by the application and then restores the execution context of the application. Finally, another dedicated instruction (e.g., *sysret* on x86_64 platform or *sret* on RISC-V platform) is executed, leading to the result that the processor mode is switched back to the user mode and the execution of the application is resumed.

During a system call, we have to take the overhead of saving and restoring the application context which requires storing several general-purpose registers to the memory or vice versa. However, there is also a much higher indirect overhead on a switch of the processor mode which we tend to ignore. This overhead is called **processor state pollution**, which is mentioned in [11]. We know that there are many mechanisms in modern processor architecture which accelerate the instruction execution assuming the temporal and spatial locality of accessed data. For example, multiple levels of data and instruction caches and prefetching are used to reduce the memory access latency on average; Translate Look-aside Buffer (TLB) can eliminate unnecessary page table traversals when translating a virtual address to a physical address in a virtual address space; branch prediction can help us quickly determine whether we should take the branch and what the branch target is. These components are called processor states and they can be trained as a flow of execution is consecutively executed on the processor to learn and record some information about the locality of the execution. Thus, the performance of this execution in terms of Instructions Per Cycle (IPC) can be maximized.

Why are the processor states polluted on a system call? Imagine that the processor has run an application for a period of time, now the processor states contain some information about the locality of the application's execution so that the application's code can be executed efficiently. If the processor traps into kernel mode to handle a system call, then the processor starts to work on a kernel stack and accesses kernel data structures which are completely isolated from the data accessed by the application, which means that the processor states have to be trained from scratch to fit into kernel's execution. Since the capacity of the processor states are limited, these kernel's locality information will substitute application's locality information, which we call the pollution of the application's processor state. After the system call returns and the application's execution is resumed, the polluted processor state will degrade the performance of the application's execution until the impact of the kernel's execution diminishes. We can see that the performance of both the application's and kernel's execution suffers from the processor state pollution, which is proven by the results of experiments [11]: the kernel-mode IPC is significantly lower than the user-mode IPC(up to 8x slower), and the degradation of user-mode IPC is up to 60% where the processor state pollution dominates the degradation if the system calls are invoked at a medium frequency.

How can we mitigate this problem? In general, we need to reduce the frequency of trapping into the kernel mode. There have been a number of works[11, 5, 3] which tried to achieve this goal. Among them, FlexSC [11] was a very successful attempt which made the system calls asynchronous. We are more familiar with synchronous system calls. This is because they are simpler and they can provide maximized transparency for applications: The application believe that the control is immediately transferred to the kernel after executing the dedicated instruction, and it is immediately returned back after the system call returns, which is as easy and natural as a subroutine call. However, given that the overhead of synchronous system calls is substantial, it is reasonable to sacrifice the transparency in pursuit of performance.

Apparently, asynchronous system calls are more complicated since applications should also be aware of these system calls. In FlexSC, there is a threading library called FlexSC-threads which is compatible with multithreading applications and responsible for transferring synchronous system calls issued by applications to asynchronous system calls. On a system call, FlexSC-threads only add a system call entry in system call pages shared between the application and the kernel rather than trap into the kernel mode. At the same time, FlexSC-threads blocks the current user-level thread and switch to another thread which is ready. If all user-level threads are waiting for the result of system calls, FlexSC-threads issues a synchronous system call to trap into the kernel mode. The kernel looks at the system call entries submitted by the application, handles these system calls accordingly and writes the results back to these system call entries. After at least one system call has completed, the control is given back to the application. Then, it is FlexSC-thread's turn to wake up threads according to the system call entries. We can see that FlexSC-threads also serves as a user-level scheduler. Additionally, FlexSC requires some modifications in the Linux kernel.

Although FlexSC significantly reduces the number of switches of processor mode, it still has some limitations. Firstly, it did not fully take advantage of the asynchronous system call interface, which allows the OS to reorder incoming system call requests according to requirements. FlexSC only handles system calls as the order they are issued, but obviously it could have done better if it can schedule system calls intentionally, especially in a multicore environment. Secondly, FlexSC uses threads as scheduling units in the OS, which leads to a higher memory consumption and context switch overhead. We want to use stackless coroutine instead since we believe that it is a more promising programming model and it is relatively easy to use in some programming languages like Rust.

We aims to further improve the OS implementation based on the asynchronous system call interface introduced by FlexSC in this project. In other words, the target of this project is designing a new asynchronous OS architecture which can handle asynchronous system call requests efficiently. The efficiency can be evaluated using the following kernel-side metrics:

- better core specialization and data locality
- lower synchronization overhead
- lower memory consumption and context switch overhead based on stackless coroutines
- be flexible enough to work on different hardware platforms including symmetric and asymmetric multicore environments

Although we prioritize the performance, we will also consider the compatibility with existing applications if possible.

The main challenge of this project is how we can reorder and distribute incoming system call requests to multicores to satisfy these performance requirements. A simple solution is just dedicating every core to a set of system calls. For example, core 0 is responsible for memory management while core 1 takes the responsibility for I/O operations. In this way, every core will execute and access a fixed set of code and data structures for a long time, which increases core specialization and data locality. For the reason that a type of system calls which access specific data structures protected by a group of locks are only handled on a single core, the data contention is also alleviated.

Inspired by the SEDA architecture[12], we can come up with a fine-grained architecture. We can divide the OS into several asynchronous modules, each of which represents a specific kind of kernel functionality. These modules can communicate with each other via asynchronous function calls. Every asynchronous module is an event-driven system which manages multiple coroutines which only executes this module's code. Every asynchronous module has its own scheduler and there is also a global scheduler which allocates CPU resources to these modules. Such architecture can allow us to utilize resource more flexibly. For example, we can let modules related to I/O run on cores with a low frequency since these modules are not CPU-bound. Mention that this architecture can provide more flexibility even if the OS

only supports synchronous system calls, so it is not limited by asynchronous system calls to some extent.

However, for the sake of time, we only separated the execution of all the system calls from zCore to get the only one asynchronous module which run on another core, i.e, there are only 2 cores in the system.

Chapter 2

Background

2.1 Technical Background

2.1.1 Coroutines

Coroutine is a programming model of cooperative multitasking. The execution of one coroutine can never be interrupted until itself decides to yield its control over the CPU. There are 2 kinds of coroutines: stackful and stackless coroutines, with the former being similar to threads since we need to allocate a stack for each stackful coroutine. Oppositely, a group of stackless coroutines only require a single stack since they use the stack alternatively. A typical implementation of a stackless coroutine is a finite state machine which only records the execution context at yield points. Other temporary data is stored on the shared stack. We can see that the stackless coroutine reduces the total memory consumption. We also expect that the context switch overhead between stackless coroutines to be lower under efficient implementation and compiler optimizations. This is the reason why we use stackless coroutines as the task unit instead of stackful coroutines.

2.1.2 Asynchronous Programming in Rust

In Rust[10], the **Future** trait describes the behaviour of a stackless coroutine (we also call it **task**), which is virtually a finite state machine from the perspective of Rust. There is only one method called **poll**, which means that the task tries to make some progress. If it successfully completes, the poll method returns **Poll::Ready** with the results. Otherwise, the task has to wait for an external event to go ahead. In this case, a **Poll::Pending** is returned.

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output>;
}
```

Programmers can provide their own implementation of the runtime to manage these tasks which implement the Future trait. Typically, every task managed by the

runtime is the root node of a Future tree, with each node on the tree implementing the Future trait. For each kind of external events, programmers design a dedicated structure and implement Future trait for it by themselves. They are the leaves in the Future tree. After that, the Rust compiler can combine these sub state machines into a large one level by level using the **async fn** and **.await** keywords until a Root-level Future is generated.

The runtime is an event-driven system which can be divided into 2 parts: **Executor** and **Reactor**. The runtime runs an event loop. In each iteration, the Reactor analyzes external events it receives and wakeup tasks accordingly while the Executor selects a task which is ready to be executed, i.e, calling its **poll** method provided by the Future trait and blocking the current task if **Poll::Pending** is returned. The executor servers as the task scheduler of the whole asynchronous runtime since it is responsible for the scheduling algorithm.

2.2 Academic Background

Here are some previous academic works related to this project:

2.2.1 Cohort Scheduling

Cohort scheduling[6] indicated that for a server application, handling a cohort of similar requests together by collecting and scheduling the requests can significantly reduce the cost per request in terms of cycles as the cohort size increases. This is because similar request tend to access the same group of data structures so that it can contribute to the data locality. We also take advantage of this observation to design our own architecture, but the difference is that we try to apply this idea in OS design based on asynchronous system calls and the huge impact of processor state pollution is considered.

The staged computation model[6] is similar to our asynchronous module architecture. However, the synchronization control per module is more difficult in kernel since it involves multitasking. Derived from the staged computation model, the Staged Event-Driven Architecture (SEDA) [12] use controllers and filters to handle excessive workloads gracefully, which we can also utilize to provide more flexibility and allow the OS to work on more platforms.

2.2.2 System Call Aggregation

In order to mitigate the impact of processor state pollution, another idea in addition to asynchronous system calls is to submit as much requests as possible on a single trap into the kernel, which is called system call aggregation. For example, the Effective System Call Aggregation (ESCA) [3] allows applications to submit a batch of system calls before trapping into the kernel. Anycall [5] allows applications to submit a piece of bytecode to the kernel so that the in-kernel compiler can translate the bytecode into native instructions or kernel operations. Although these methods

can reduce the times of processor mode switches, they are difficult to use and not transparent to applications.

2.2.3 Minimizing Granularity of Locks in Kernel

To eliminate the unnecessary synchronization overhead in the kernel, Corey OS[1] allows threads from the same process to explicitly indicate which resources it actually needs to access via modified system calls. This can reduce the in-kernel locks' granularity to a appropriate level so that the synchronization overhead can be mitigated. This idea is orthogonal to our project.

2.3 zCore

Since our work is based on zCore[13], here we have a brief introduction to zCore. zCore is an operating system kernel which reimplements the system calls of Zircon[16], the kernel of Google's next generation OS called Fuchsia[4] following Android, in Rust[10] with its asynchronous programming features. Because of its great modular design, it is also compatible with Linux system call interface and it can run on x86_64 and RISC-V bare-metal platforms and also Linux and MacOS host OS.

Currently it only supports synchronous system calls, but its internal implementation is asynchronous based on coroutines. Kernel-visible threads in classical OS design are substituted with coroutines and these coroutines are managed by a runtime inside zCore.

2.3.1 zCore Architecture

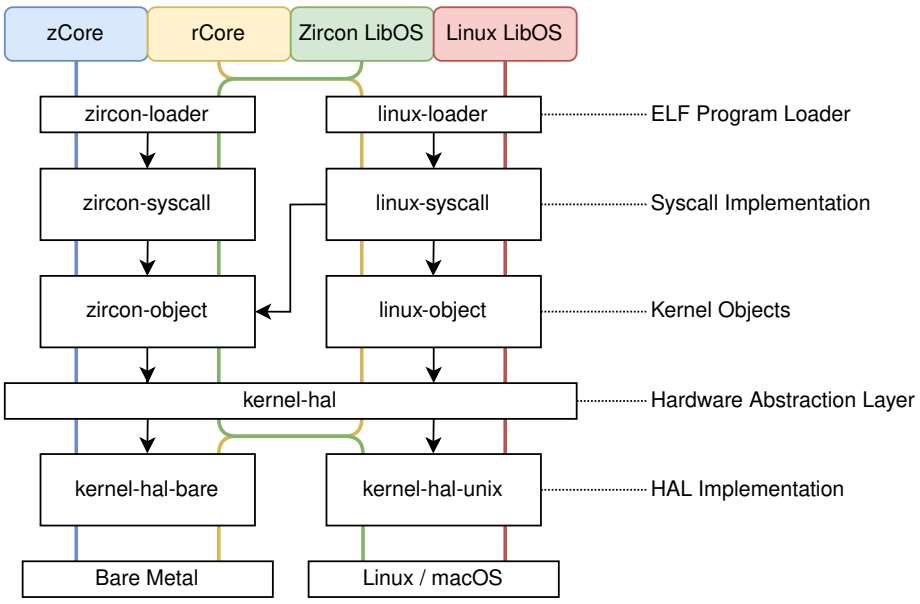

Figure 2.1: zCore Architecture[14]

From Figure 2.1 we can see that zCore is a huge project composed of a number of components which can be further divided into 5 layers. We describe these layers from bottom to top:

- Hardware Abstraction Layer (HAL): The **kernel-hal** encapsulates all the hardware primitives required during the execution of zCore. These hardware primitives includes physical memory management, virtual memory management, hardware thread management and trap handlers which are called when an interrupt or exception occurs.

  kernel-hal provides 2 different implementations which upper components can choose from depending on the platform the final zCore executable runs on. For example, the implementation **kernel-hal-bare** is selected if zCore wants to be directly run on a bare-metal platform. Therefore, **kernel-hal-bare** is implemented taking advantage of the instructions provided by the instruction set. By contrast, zCore uses **kernel-hal-unix** if it wants to be run as an normal application on a host OS such as Linux or MacOS. **kernel-hal-unix** implements the hardware primitives based on the services from the host OS including system calls and Rust standard library APIs.

- Kernel Objects Layer: This layer includes the implementation of important kernel data structures which are used for system calls. There are kernel objects which are responsible for task management(e.g., Job, Process and Thread), file system stack, network protocol stack, inter-process communication, synchronization, signal handling and so on.

  There are also 2 versions of the implementation. To be specific, **zircon-object** and **linux-object** are compatible with Zircon system call interface and Linux system call interface, respectively.

- Syscall Implementation Layer: **zircon-syscall** implements Zircon system call interface based on **zircon-object** while **linux-syscall** implements Linux system call interface based on **linux-object**.

- ELF Program Loader Layer: This layer reads an executable from the file system on the disk and turns it into an asynchronous task which is managed by the zCore runtime. Each asynchronous task corresponds to a thread in the process of an user-level application, and it is the smallest unit of task management in zCore. Similar to Kernel Objects Layer and Syscall Implementation Layer, it has 2 versions of implemented depending on the system call interface.

- Kernel Executable Layer: This layer contains behaviours during the initialization stage of zCore. In addition, it combines several components below to generate the final zCore executable. We can see that there are 4 types of executable available in different colours. Each type corresponds to a specific combination of the platform zCore runs on(bare-metal or a host OS) and the system call interface(Zircon or Linux).

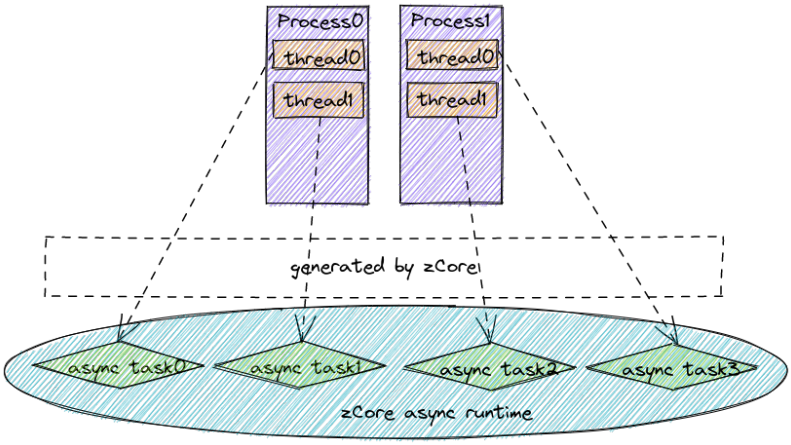

Figure 2.2: zCore Task Management Model

For example, the executable **rCore** in yellow color comprises components on the yellow path including **linux-loader**, **linux-syscall**, **linux-object** and **kernel-hal-bare**, which means that **rCore** is compatible with Linux system call interface and it supports a bare-metal platform. Mention that our modification is only for **rCore** rather than other types.

2.3.2 zCore Execution Model

The legacy version of zCore that we modify can be found in [15]. Here we only consider the **rCore** branch which implements Linux system call interface and runs on a bare-metal platform. First of all, we discuss its task management model. As Figure2.2 shows, it supports 2 abstractions in terms of task management: Process and Thread. Each Process contains one or more Threads. For each Thread, zCore generates an asynchronous task for it. All generated asynchronous tasks are managed by the zCore asynchronous runtime. Both the zCore asynchronous runtime and these asynchronous tasks run in kernel mode.

Listing 2.1: Pseudo Code of Asynchronous Task Generated for a Thread

```
async fn run_user(thread: &Thread) {
    loop {
        let ctx = thread.wait_for_run().await;
        thread.handle_signal();
        enter_uspace();
        handle_user_trap(thread, ctx).await;
    }
}
```

How is an asynchronous task generated? Listing 2.1 shows the pseudo code of each asynchronous task. For each Thread, its corresponding asynchronous task is an asynchronous function called **run_user** which takes the reference of the Thread

as input. Inside the **run_user** function, there is a loop, each iteration of which is comprised of 4 operations:

- Calling **thread.wait_for_run()** to get the application context of the thread. The application context contains the values of general registers and it is saved when the application traps into kernel mode on an interrupt or exception. The application context is stored in the Thread structure. However, **thread.wait_for_run()** does not always immediately returns. When the thread is suspended after a related system call, **thread.wait_for_run()** should wait for the thread to be resumed to return. Therefore, here we use **.await** keyword in Rust to wait for the asynchronous function **thread.wait_for_run()** to return.

- Calling **thread.handle_signal()** to handle signals on this thread. Mention that this is a synchronous operation which means that during this operation the current asynchronous task will not be scheduled out by the zCore asynchronous runtime.

- Using **enter_uspace()** to switch from kernel mode to user mode to execute the application code. Although it requires a change on the CPU privilege level, from the perspective of the asynchronous task it can be seen as a synchronous operation on the application context.

- After an interrupt or exception occurs during the application execution, the execution of **run_user** is resumed with the updated application context. Next, **handle_user_trap** is called to handle the incoming interrupt or exception including system calls. The application context contains the reason of the trap. **handle_user_trap** is also an asynchronous function since some system calls have to wait for specific events so that they cannot return at once. Similar to **thread.wait_for_run()**, we use **.await** here.

In each iteration of **run_user**, the state of the thread is checked several times. If it is detected that the thread has been exited(e.g., after a **sys_exit** system call), we break the loop which means that **run_user** returns. We did not show this mechanism in the pseudo code for simplicity.

zCore only uses a single core although it partially supports booting multicores and this core is managed by the zCore asynchronous runtime showed in Figure 2.3. zCore asynchronous runtime is divided into 2 parts: Executor and Reactor, which is the same as our discussion in Section 2.1.2.

After initialization, CPU starts to run Executor's loop. For each iteration, the Executor selects a task which is ready and then poll the task. If the result is **Poll::Ready** which means that the task has been completed, we release the resources corresponding to this task. Otherwise, the result is **Poll::Pending**. In this case, the task has to wait for an event to make further progress. Therefore, the Executor marks this task as blocked and the pair of the task and the event it waits is recorded. As for the Reactor, when an event comes in, it is used to wakeup the blocked tasks whick are waiting for this event.

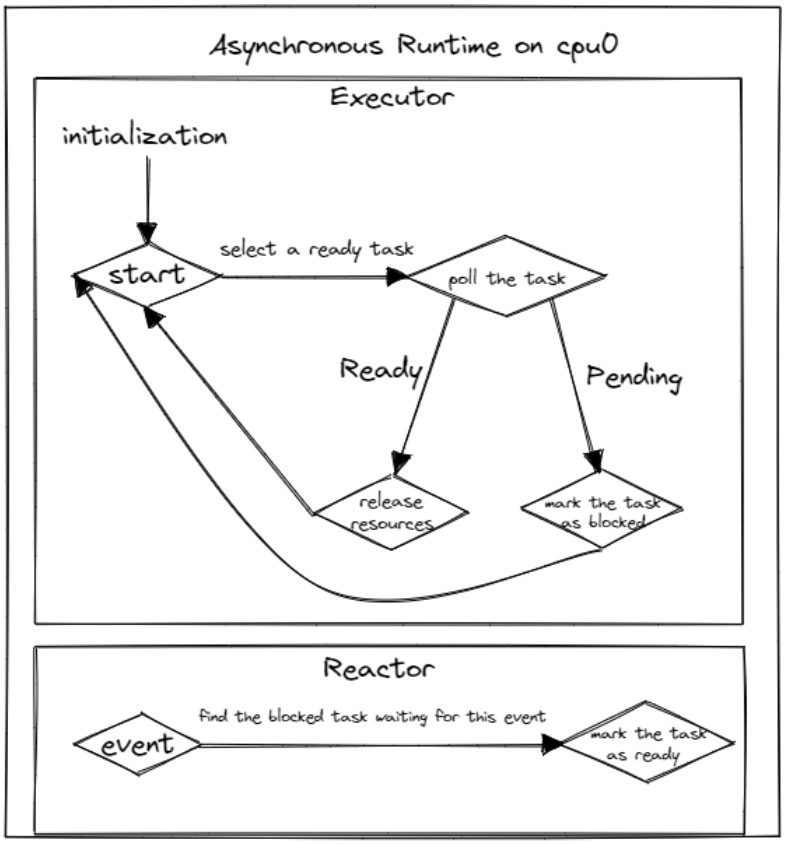

Figure 2.3: zCore Asynchronous Runtime

Chapter 3

Architecture

Based on zCore, our modification is that adding a new CPU which we refer to as CPU1 in addition to CPU0 and moving the execution of system calls from CPU0 to CPU1 asynchronously.

In the original zCore system, system call handling is a part of **handle_user_trap** we mentioned in listing 2.1. Thus, system calls are completely handled on CPU0. By contrast, Figure 3.1 shows the system call handling mechanism in our implementation. zCore asynchronous runtime still runs on CPU0 and it selects and polls tasks. The difference is that on a system call requested by the application code, CPU0 submits the system call request with system call ID and arguments to CPU1 instead of directly handling the system call itself. After that, zCore asynchronous runtime on CPU0 suspends current task to wait for CPU1 to complete the system call request remotely so that the runtime possibly switches to another ready task.

Similar to CPU0, there is another asynchronous runtime running on CPU1. However, this runtime is different from that on CPU0. To be specific, CPU1 is dedicated to handle system calls remotely submitted by CPU0. CPU1 periodically accept system call requests and handle them. After a system call request is completed, CPU1 calls the interface provided by CPU0 when CPU0 submits the request to wakeup the corresponding task which was blocked on CPU0, which means that the task is ready to be resumed. After a while, CPU0's asynchronous runtime selects the task and resumes its execution.

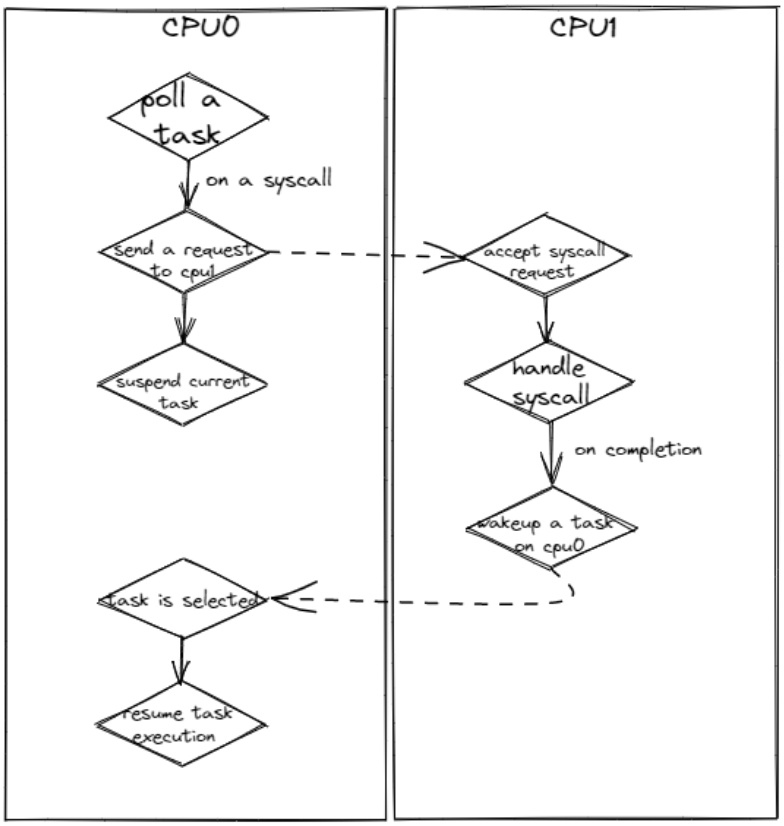

Figure 3.1: Architecture of Our Implementation

Chapter 4

Implementation

4.1 Preparation

In order to run our system on QEMU emulator with KVM acceleration enabled, we need to do some preparation works.

4.1.1 Multiboot With KVM Enabled

Since our implementation requires 2 vCPUs at the same time, we need to enable them after initialization. On x86_64 platform, zCore enables CPUs other than the bootstrap processor by sending inter-processor interrupts to these CPUs via the xapic system bus. However, when KVM is enabled, the interrupts should be sent via x2apic instead of xapic by default. Therefore, we disabled the x2apic feature by adding **-x2apic** flag to the QEMU emulator to solve this problem.

4.1.2 KVM Optimization

The reason why we use KVM acceleration is that we want to minimize the gap between the results when our system is run based on the emulator and on a real bare-metal machine. In fact, the QEMU emulator with KVM acceleration enable can be seen as a virtual machine monitor while our system runs as a virtual machine(or guest OS) inside the emulator. Some behaviours of our system will trigger the vmexit operation which exits the execution of our system and transfer the control to the emulator to handle some event. If the vmexit is frequently triggered, it will lead to a unacceptable overhead to the overall performance since each vmexit operation takes several microseconds. Therefore, we need to reduce to number of vmexits as much as possible.

Initially, 90% vmexits were triggered because of CPUID instructions. This is because the implementation of the spinlock used in zCore was not efficient. To be specific, each time the spinlock is acquired or released, a CPUID instruction is executed to get the ID of current CPU. However, we can store the CPU ID in the thread-local storage after the initialization of zCore so that we can directly get the

ID of current CPU without the need of the CPUID instruction. After that, the vmexits from the CPUID instruction were fully eliminated.

Other vmexits are triggered inevitably at a reasonable frequency:

- For each access to the emulated block device, an **EPT_MISCONFIG** vmexit is triggered;
- For each character printed to the screen, an **IO_INSTRUCTION** vmexit is triggered;
- For each timer interrupt, a **PREEMPTION_TIMER** vmexit is triggered;
- **EXTERNAL_INTERRUPT** is triggered at a fixed frequency at 270Hz, which is possibly related to some buses in the system.

Mention that we ignored some types of vmexits since the number they were triggered were too small.

4.1.3 Emulated PCI Block Device Driver

On x86_64 platform, the original zCore directly embeddeded the file system image into the zCore executable so that the whole file system is loaded into RAM when loading the zCore executable into memory, which we refer to as *ramfs* mode. However, this will bring a huge number of memory accesses during a file system operation which affects the cache performance. Therefore, we decided to use an emulated block device provided by the QEMU emulator called **virtio-blk-pci** as the storage device of the file system. We call this *disk* mode. To support the **virtio-blk-pci** device, we wrote some driver code and added it to zCore.

4.2 Asynchronous System Call Module

The asynchronous system call module run on CPU1 to receive system call requests, handle them and notify CPU0 on completion. It is also an Rust asynchronous runtime with an Executor and a Reactor. In each iteration of its main event loop, it first receive requests from CPU0 from a global atomic data structure shared between 2 CPUs and add requests to its task queue. After that, it selects a request from its task queue and handle the system call. The implementation can be found in **zCore/linux-syscall/src/async_syscall.rs**.

Chapter 5

Evaluation

5.1 Testing Environment

We tested our implementation on Ubuntu, which is the host OS. The original zCore system and our implementation are guest OSs which run based on the system-level emulator called QEMU[9] with KVM acceleration enabled. More detailed configuration can be found in Table 5.1.

5.2 Benchmarks

zCore is compatible with musl-libc[7], which is a lightweight C standard library. Therefore, we decide to come up with some simple benchmarks based on it. The benchmarks were compiled with the cross platform compiler used with musl-libc, which is **x86_64-linux-musl-gcc** 11.2.1 20211120. There are 6 different types of benchmarks: Task0 - Task5.

5.2.1 Task0

In task0, a huge number of simple system calls are executed in a process. From Listing A.1 **many_syscalls.c** we can see that 2 simple system calls, **getpid** and **getppid** are both called for 10,000,000 times.

By the way, to measure the time cost of a task, we added a tool **time.c**(Listing A.2). It takes a command as input from command line, forks a child process to execute the command and waits for the child process to be exited. After that, the time

CPU	Intel(R) Core(TM) i5-8279U CPU @ 2.40GHz
Operating System	Ubuntu 20.04.1
Linux Kernel	5.15.0-46-generic
Rust Compiler	nightly-2022-01-20
Emulator	qemu-system-x86_64 7.0.0

Table 5.1: Testing Environment

cost of the command is calculated and printed on the screen. For example, if we want to measure the time cost of the **many_syscalls** benchmark, we can change current working directory to the directory containing the benchmark executable(**/libc-test/src/added**) and type the command **./time.exe ./many_syscalls.exe**.

5.2.2 Task1

In zCore's default file system, there is a directory(**/libc-test/src/math**) which contains around 1,200 files of different size. Both Task1 and Task2 are operations on this directory and they are designed to evaluate the file system performance of our system. In Task1, we get the metadata for all the files in the target directory using **stat** system call. Task1 is implemented in **list_huge_dir.c**(Listing A.3).

Depending on the file system is stored directly on RAM or an emulated block device, which we refer to as *ramfs* mode and *disk* mode, respectively, we possibly repeat the operations for several times to make the overall time cost reasonable. To be specific, in *ramfs* mode, we repeat the operations for 100 times. By contrast, in *disk* mode, we only do the operations once. Therefore, the commands used for testing are:

- *ramfs* mode: **./time.exe ./list_huge_dir.exe 100**
- *disk* mode: **./time.exe ./list_huge_dir.exe**

5.2.3 Task2

We use Task2 to evaluate the performance of reading files of our system. Irrespective of the system runs in *ramfs* mode or *disk* mode, the command is **./time.exe /bin/busybox wc /libc-test/src/math/***. Here, we take advantage of the **wc** tool embedded in the **busybox**[2], a collection of Unix utilities, to read all the files under the target directory and find how many words are included in these files. This command will trigger approximately 44,000 times of **read** system calls, with each of them reading 1KiB of a file into a user-mode buffer area.

5.2.4 Task3 - Task5

In Task0 - Task2, there is at most one active task in the system at a time. Therefore, after CPU0 submits a system call request to CPU1, it cannot do nothing useful except for waiting the completion of the request on CPU1. This is why we introduce Task3 - Task5 to increase the concurrency of the workload. We use the tool **time_multiproc**(Listing A.4) to fork multiple processes to execute the given command multiple times concurrently. Our expection is that after CPU0 submits a request, it still has something meaningful to do, e.g, working on another task which is ready.

Here are the testing commands for Task3 - Task5:

- Task3: **./time_multiproc.exe 2 ./many_syscalls.exe**(2 concurrent Task0)

ramfs mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
4943.05ms	3.73×10^7	21421.63ms	2.23×10^8	$\frac{\text{CPU0}}{\text{CPU1}} = 1.78$
disk mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
5019.23ms	9.20×10^7	21742.38ms	2.65×10^8	$\frac{\text{CPU0}}{\text{CPU1}} = 2.00$

Table 5.2: Task 0 Results

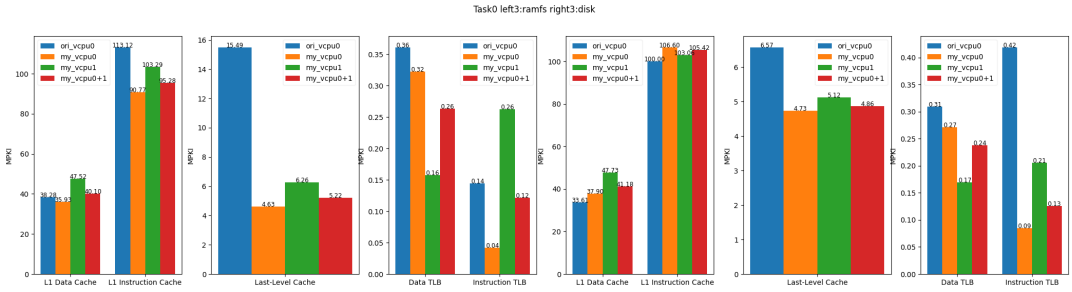

Figure 5.1: Task 0 Cache Performance Counters

- Task4(*ramfs* mode): **./time_multiproc.exe 2 ./list_huge_dir.exe 100**(2 concurrent Task1)
- Task4(*disk* mode): **./time_multiproc.exe 2 ./list_huge_dir.exe**(2 concurrent Task1)
- Task5: **./time_multiproc.exe 2 /bin/busybox wc /libc-test/src/math/***(2 concurrent Task2)

5.3 Experiment Result Analysis

For each Task, we execute the command in the original zCore system and also our implementation based on the QEMU emulator to compare their performance. The time cost of the command is recorded using either **time** or **time_multiproc** tool. In addition, we use Linux perf[8] tool to monitor the emulator process to collect some performance counters related to multiple levels of caches and so on.

5.3.1 Task0

Table 5.2 lists the time cost and overall number of instructions in the original zCore system and our implementation under *ramfs* and *disk* mode. Mention that there is an extra column named instruction ratio showing the instruction ratio on 2 CPUs in our implementation. For the original zCore system, there is not such a column since the original zCore only uses a single core.

Figure 5.1 shows the MPKI(Misses Per Kilo Instructions) on L1 Data Cache(L1D), L1 Instruction Cache(L1I), Last-Level Cache(LLC), Data TLB and Instruction TLB throughout the execution. The left half is for *ramfs* mode while the right half is for *disk* mode. For each type of cache, there are 4 bars. From left to right:

- The blue bar shows the MPKI of the cache on the unique CPU in the original zCore system;
- The orange bar show the MPKI of the cache on CPU0 in our implementation;
- The green bar show the MPKI of the cache on CPU1 in our implementation;
- The red bar show that aggregated MPKI of the cache on both CPU0 and CPU1 in our implementation.

From Figure 5.1 we can find that MPKI of LLC is significantly lower in our implementation than the original zCore system. This is because both CPU0 and CPU1 are more specialized on their own tasks so that the locality is better. It is only about $\frac{1}{3}$ of the original zCore's data under *ramfs* mode and $\frac{3}{4}$ under *disk* mode. LLC's MPKI is larger under *ramfs* mode since the file system is stored in RAM so that there will be more memory accesses. When it comes to Data and Instruction TLB, their MPKI are also smaller in our implementation for the same reason.

As for the L1D and L1I cache, there is little difference between their MPKI in the original zCore and our implementation. This is because their MPKI only corresponds to local memory access patterns. Therefore, our modification at the global level can hardly have an influence on them.

However, the time cost in our implementation is around 4 times of that in the original zCore. This is because far more instructions are executed in our implementation. For example, under *ramfs* mode about 6 times more instructions are executed in our implementation. In fact, for each system call we need to execute fixed extra 10 instructions in our implementation compared to the original zCore because of the remote system call handling mechanism, which is a huge overhead which cannot be ignored when the system call is simple. Obviously, in Task 0, the system calls we use for evaluation are very simple and the number of system calls is very large, leading to the bad result of the overall time cost. Nevertheless, the instruction execution is more efficient in our implementation since caches are more sufficiently used.

5.3.2 Task1

Result for Task1 can be found in Table 5.3 and Figure 5.2. Although the MPKI of almost every cache is lower in our implementation, it cannot effectively decrease the time cost given that we still need to execute more instructions in our implementation. what is the source of the extra instructions? In this case, the number of system call is small, which is only 120,000 even under *ramfs* mode. At the same time, the system call **stat** is more complicated than **getpid** in Task0. As a consequence, the fixed overhead of 10 instructions per system call can be ignored here.

Here, the extra instructions mainly comes from the synchronization between CPU0 and CPU1. For example, if CPU0 submits a request to CPU1 and there are no other

ramfs mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
7522.78ms	5.38×10^7	7526.08ms	8.15×10^7	$\frac{\text{CPU0}}{\text{CPU1}} = 1.70$
disk mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
14353.72ms	5.84×10^9	14257.45ms	7.41×10^9	$\frac{\text{CPU1}}{\text{CPU0}} = 4.48$

Table 5.3: Task 1 Results

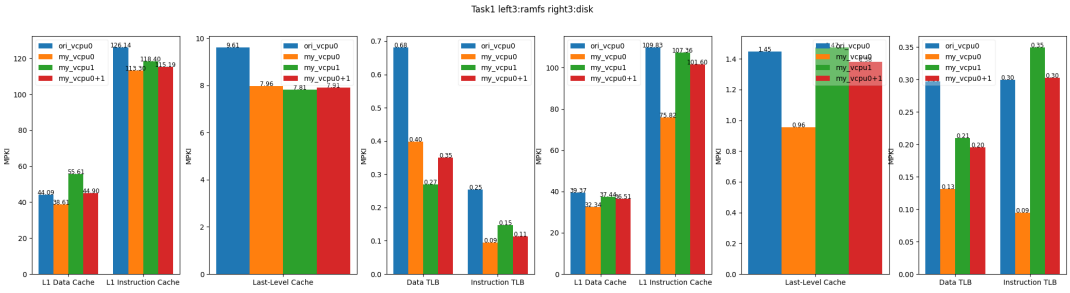

Figure 5.2: Task 1 Cache Performance Counters

ready threads on CPU0, CPU0 has to execute some extra instructions to wait for the completion of the system call request, which also applies to CPU1. If workloads on them are perfectly matched, this overhead can be eliminated. However, the tasks we use for evaluation are not balanced to some extent.

5.3.3 Task2

Results for Task2 can be found in Table 5.4 and Figure 5.3. Task2 is dominated by the system call execution since most instructions are executed on CPU1. Under *ramfs* mode, CPU0 only executes less than 1% instructions. If we focus on CPU1 in our implementation, we can find that MPKI of all caches on CPU1 are similar to or lower than the unique CPU in the original zCore.

ramfs mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
931.12ms	1.68×10^9	941.81ms	1.70×10^9	$\frac{\text{CPU1}}{\text{CPU0}} = 129.87$
disk mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
19501.75ms	8.63×10^9	19150.05ms	9.77×10^9	$\frac{\text{CPU1}}{\text{CPU0}} = 7.06$

Table 5.4: Task 2 Results

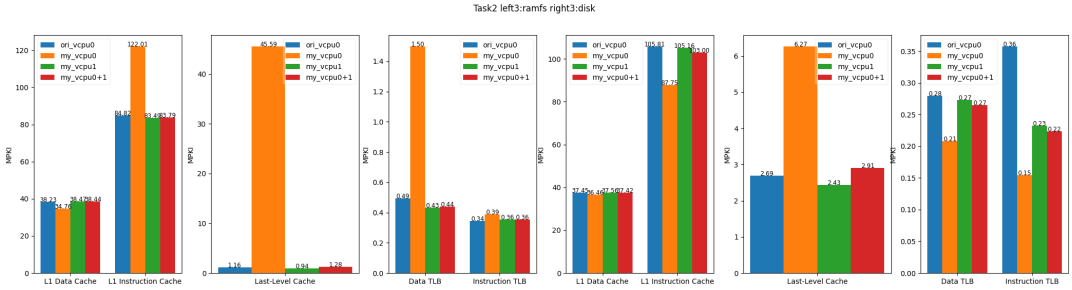

Figure 5.3: Task 2 Cache Performance Counters

ramfs mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
9871.63ms	6.65×10^7	38031.88ms	3.85×10^8	$\frac{\text{CPU0}}{\text{CPU1}} = 1.75$
disk mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
10012.26ms	1.31×10^8	38379.68ms	4.40×10^8	$\frac{\text{CPU0}}{\text{CPU1}} = 2.05$

Table 5.5: Task 3 Results

CPU0 in our implementation only executes a small number of instructions. Therefore, we think most misses on CPU0 are compulsory misses irrelevant to the cache capacity. This can explain why MPKI on CPU0 in our implementation are larger than the original zCore, especially the LLC's MPKI under *ramfs* mode. Consequently, the aggregated MPKI of LLC in our implementation is larger than the original zCore. But this will not pose a threat to the overall performance since we should concentrate on only CPU1 for this workload.

5.3.4 Task3 - Task5

Results for Task3 - Task5 can be found in Table 5.5 and Figure 5.4, Table 5.6 and Figure 5.5, Table 5.7 and Figure 5.6, respectively. Compared to Task0 - Task2, the

ramfs mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
15041.74ms	1.05×10^8	15071.40ms	1.62×10^8	$\frac{\text{CPU0}}{\text{CPU1}} = 1.87$
disk mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
28800.77ms	1.17×10^{10}	29262.01ms	1.41×10^{10}	$\frac{\text{CPU1}}{\text{CPU0}} = 4.56$

Table 5.6: Task 4 Results

ramfs mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
1655.45ms	3.36×10^9	1695.72ms	3.38×10^9	$\frac{\text{CPU1}}{\text{CPU0}} = 192.17$
disk mode				
the original zCore system		our implementation		
time	overall instructions	time	overall instructions	instruction ratio
38921.19ms	1.71×10^{10}	39215.17ms	1.93×10^{10}	$\frac{\text{CPU1}}{\text{CPU0}} = 7.26$

Table 5.7: Task 5 Results

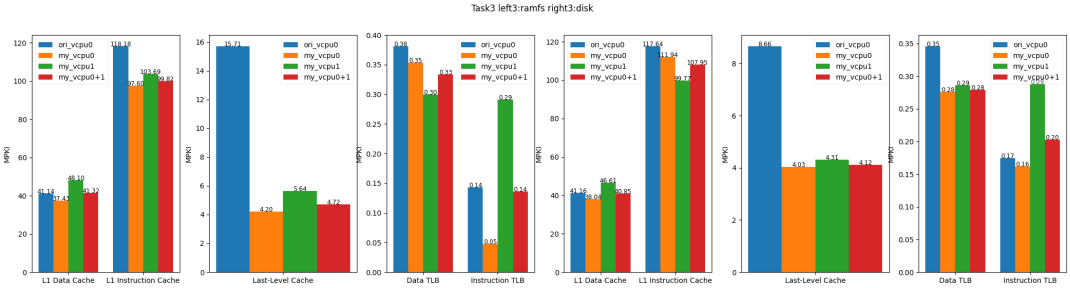

Figure 5.4: Task 3 Cache Performance Counters

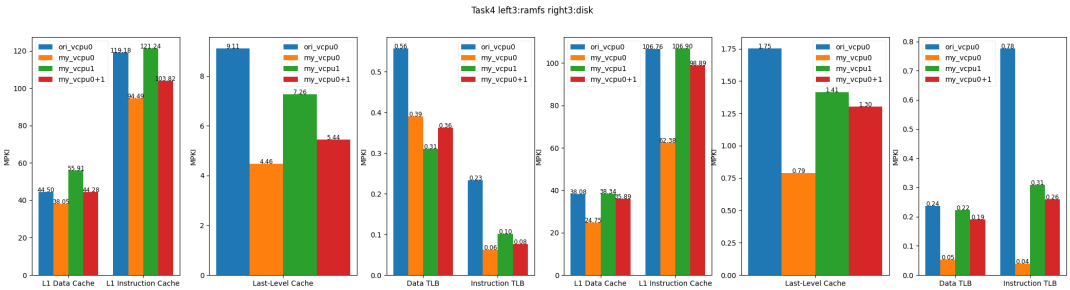

Figure 5.5: Task 4 Cache Performance Counters

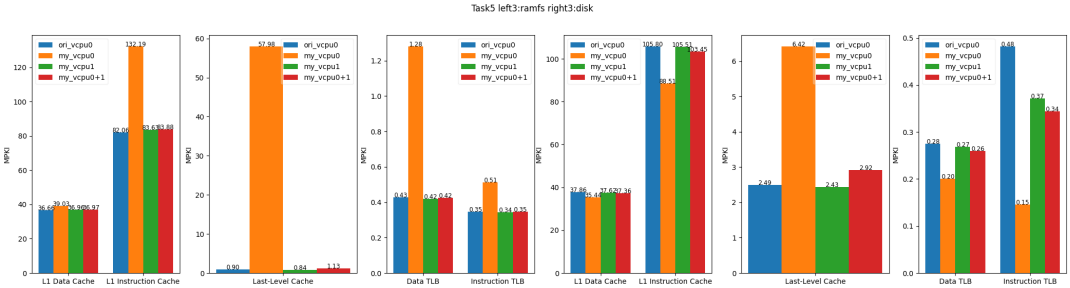

Figure 5.6: Task 5 Cache Performance Counters

aggregated MPKI of LLC in our implementation is further decreased since more useful works are filled into CPU's instruction slots. Time cost of Task3 is lower than twice of Task0's time cost, which means that less meaningless instructions for synchronization between CPU0 and CPU1 are executed. We believe that this should also apply to Task1 and Task2, but it cannot be seen from the data. Maybe this is because Task1 and Task2 are not as balanced as Task0 in terms of the workloads on CPU0 and CPU1.

Chapter 6

Conclusion

Based on zCore, an asynchronous operating system developed in Rust which only use a single CPU(CPU0), we tried to add another CPU(CPU1) and moved execution of all the system calls asynchronously onto CPU1. We expect that both CPU0 and CPU1 in our implementation are more specialized on their tasks so that their data locality will be better than the original zCore system. We evaluated the MPKI of a variety of caches including L1 Data Cache, L1 Instruction Cache, Last-Level Cache, Data TLB and Instruction TLB using a group of benchmarks and found that most of the time MPKI of our implementation was significantly lower than the original zCore.

However, we have to take the overhead of executing more instructions. There are 2 sources of these extra instructions: Firstly, a fixed number of instructions are executed per system call due to the remote system call handling mechanism. Secondly, both CPU0 and CPU1 have to execute some meaningless instructions for synchronization between them when workloads on them are not balanced. Therefore, although instructions are executed more efficiently because of a better data locality, there is little difference on the time cost of Task1/2/4/5. Our implementation even performed worse than the original zCore on Task0/3 since the number of system calls were huge and the system calls were simple.

Another reason why our implementation was not so good in terms of time cost of benchmarks is that the concurrency is limited in the system, leading to the result that the potential of this architecture cannot be fully revealed. After we increased the benchmark concurrency from Task0/1/2 to Task3/4/5, LLC's MPKI was further decreased and Task3 was executed more efficiently than Task0.

In conclusion, we think the asynchronous system call execution mechanism is interesting and it is advantageous for core specialization and data locality. However, it also brings some overhead and it works better in a highly concurrent environment.

Chapter 7

Future Work

Currently, we only have one asynchronous kernel module fixed on CPU1 to handle all system calls. If we have more time, we can further divide this module into multiple modules, each of which only executes a fixed set of system calls of similar functionality. Each module is managed by its own asynchronous runtime, which means that every 2 modules can have different scheduling algorithms. At the same time, we can move modules across CPUs. In this way, we can further increase the data locality and flexibility of the system.

Additionally, a notable fluctuation in the data was observed when running the experiment. Due to the time limit, we only manually run the same experiment for 4 times and calculated the average to eliminate the randomness, which was not enough. Therefore, we plan to write scripts to run the same experiment for at least 20 times to ensure the reliability of the data.

Appendix A

Benchmark Source Code in C

Listing A.1: Task0: many_syscalls.c

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;
    int N = 10;
    int M = 1000000;
    for (int i = 0; i < N; ++i) {
        printf("%d/%d\n", i, N);
        for (int j = 0; j < M; ++j) {
            pid = getpid();
            pid = getppid();
        }
    }
    return 0;
}
```

Listing A.2: Time Measurement Tool: time.c

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>
#include <assert.h>

int main(int argc, char* argv[], char* envp[]) {
    if (argc < 2) {
        printf("ERROR: argc < 2\n");
        return 0;
    }
    struct timespec begin, end;
    clock_gettime(CLOCK_REALTIME, &begin);
```

```
    pid_t pid = fork();
    if (pid == 0) {
        // child process
        execve(argv[1], &argv[1], envp);
    } else {
        int wstatus;
        pid_t p = waitpid(pid, &wstatus, 0);
        assert(p == pid);
    }
    clock_gettime(CLOCK_REALTIME, &end);
    double begin_ms = begin.tv_sec *
        (double)1000.0 + begin.tv_nsec / 1.0e6;
    double end_ms = end.tv_sec *
        (double)1000.0 + end.tv_nsec / 1.0e6;
    printf("elapsed time=%03lfms\n", end_ms - begin_ms);
    return 0;
}
```

Listing A.3: Task1: list_huge_dir.c

```
#include <dirent.h>
#include <sys/stat.h>

char* HUGE_DIR = "/libc-test/src/math/";
char full_path[128];

int main(int argc, char* argv[]) {
    DIR *d;
    struct dirent *dir;
    struct stat statbuf;
    int dir_len = strlen(HUGE_DIR);
    int filename_len;
    int count = 1;
    if (argc == 2)
        count = atoi(argv[1]);
    while (count--) {
        d = opendir(HUGE_DIR);
        if (d != NULL) {
            while((dir = readdir(d)) != NULL) {
                filename_len = strlen(dir->d_name);
                memset(full_path, 0, sizeof full_path);
                memcpy(full_path, HUGE_DIR, dir_len);
                memcpy(full_path + dir_len, dir->d_name, filename_len);
                full_path[dir_len + filename_len] = '\0';
                stat(full_path, &statbuf);
            }
        }
```

```
        closedir(d);
    }
    return 0;
}
```

Listing A.4: Concurrency Tool: time_multiproc.c

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>
#include <assert.h>
#include <stdlib.h>

int main(int argc, char* argv[], char* envp[]) {
    if (argc < 3) {
        printf("ERROR: argc < 2\n");
        return 0;
    }
    struct timespec begin, end;
    clock_gettime(CLOCK_REALTIME, &begin);
    int proc_number = atoi(argv[1]);
    for (int i = 0; i < proc_number; ++i) {
        pid_t pid = fork();
        if (pid == 0) {
            // child process
            execve(argv[2], &argv[2], envp);
        }
    }
    int wstatus;
    for (int i = 0; i < proc_number; ++i) {
        assert(waitpid(-1, &wstatus, 0) > 0);
    }
    assert(waitpid(-1, &wstatus, 0) == -1);
    clock_gettime(CLOCK_REALTIME, &end);
    double begin_ms = begin.tv_sec *
        (double)1000.0 + begin.tv_nsec / 1.0e6;
    double end_ms = end.tv_sec *
        (double)1000.0 + end.tv_nsec / 1.0e6;
    printf("elapsed time=%03lfms\n", end_ms - begin_ms);
    return 0;
}
```

References

[1] Silas Boyd-Wickizer et al. “Corey: An Operating System for Many Cores.” In: *OSDI*. Vol. 8. 2008, pp. 43–57.

[2] *BusyBox*. `https://busybox.net/`. Accessed: 2022-08-30.

[3] Yu-Cheng Cheng, Ching-Chun Jim Huang, and Chia-Heng Tu. “ESCA: Effective System Call Aggregation for Event-Driven Servers”. In: *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2022, pp. 18–25.

[4] *fuchsia*. `https://fuchsia.dev/`. Accessed: 2022-06-01.

[5] Luis Gerhorst et al. “AnyCall: Fast and Flexible System-Call Aggregation”. In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*. 2021, pp. 1–8.

[6] James R Larus and Michael Parkes. “Using cohort scheduling to enhance server performance”. In: *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*. 2001, pp. 182–187.

[7] *musl-libc*. `https://musl.libc.org/`. Accessed: 2022-08-30.

[8] *perf*. `https://perf.wiki.kernel.org/index.php/Main_Page`. Accessed: 2022-08-30.

[9] *QEMU*. `https://www.qemu.org/`. Accessed: 2022-08-30.

[10] *Rust*. `https://www.rust-lang.org/`. Accessed: 2022-06-01.

[11] Livio Soares and Michael Stumm. “{FlexSC}: Flexible System Call Scheduling with {Exception-Less} System Calls”. In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010.

[12] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An architecture for well-conditioned, scalable internet services”. In: *ACM SIGOPS operating systems review* 35.5 (2001), pp. 230–243.

[13] *zCore*. `https://github.com/rcore-os/zCore`. Accessed: 2022-06-01.

[14] *zCore-arch*. `https://github.com/rcore-os/zCore/blob/master/docs/structure.svg`. Accessed: 2022-06-01.

[15] *zCore-base*. `https://github.com/rcore-os/zCore/tree/a069efe05b8ade83e18dca2c79e80815f17ef0f1`. Accessed: 2022-08-29.

[16] *zircon*. `https://fuchsia.dev/fuchsia-src/concepts/kernel`. Accessed: 2022-06-01.