

國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

章瑋麟

Wei-Lin Chang

指導教授：黎士瑋 博士

Advisor: Shih-Wei Li Ph.D.

中華民國 112 年 7 月

July, 2023

# 國立臺灣大學碩士學位論文

## 口試委員會審定書



本論文係章瑋麟君（R09922117）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 7 月 1 日承下列考試委員審查通過及口試及格，特此證明

口試委員：\_\_\_\_\_

（指導教授）

_____	_____
_____	_____
_____	_____
_____	_____

所 長：\_\_\_\_\_





## Acknowledgements

常到外國朋友家吃飯。當蠟燭燃起，菜肴布好，客主就位，總是主人家的小男孩或小女孩舉起小手，低頭感謝上天的賜予，並歡迎客人的到來。

我剛到美國時，常鬧得尷尬。因為在國內養成的習慣，還沒有坐好，就開動了。

以後凡到朋友家吃飯時，總是先囑咐自己；今天不要忘了，可別太快開動啊！幾年來，我已變得很習慣了。但我一直認為只是一種不同的風俗儀式，在我這方面看來，忘或不忘，也沒有太大的關係。

前年有一次，我又是到一家去吃飯。而這次卻是由主人家的祖母謝飯。她雪白的頭髮，顫抖的聲音，在搖曳的燭光下，使我想起兒時的祖母。那天晚上，我忽然覺得我平靜如水的情感翻起滔天巨浪來。

在小時候，每當冬夜，我們一大家人圍著個大圓桌吃飯。我總是坐在祖母身旁。祖母總是摸著我的頭說：「老天爺賞我們家飽飯吃，記住，飯碗裡一粒米都不許剩，要是糟蹋糧食，老天爺就不給咱們飯了。」

剛上小學的我，正在念打倒偶像及破除迷信等為內容的課文，我的學校就是從前的關帝廟，我的書桌就是供桌，我曾給周倉畫上眼鏡，給關平戴上鬍子，祖母的話，老天爺也者，我覺得是既多餘，又落伍的。



不過，我卻很尊敬我的祖父母，因為這飯確實是他們掙的，這家確實是他們立的。我感謝面前的祖父母，不必感謝渺茫的老天爺。

這種想法並未因為年紀長大而有任何改變。多少年，就在這種哲學中過去了。

我在這個外國家庭晚飯後，由於這位外國老太太，我想起我的兒時，由於我的兒時，我想起一串很奇怪的現象。

祖父每年在「風裡雨裡的咬牙」，祖母每年在「茶裡飯裡的自苦」，他們明明知道要滴下眉毛上的汗珠，才能撿起田中的麥穗，而為什麼要謝天？我明明是個小孩子，混吃混玩，而我為什麼卻不感謝老天爺？

這種奇怪的心理狀態，一直是我心中的一個謎。

一直到前年，我在普林斯頓，瀏覽愛因斯坦的我所看見的世界得到了新的領悟。

這是一本非科學性的文集，專載些愛因斯坦在紀念會上啦，在歡迎會上啦，在朋友的喪禮中，他所發表的談話。

我在讀這本書時忽然發現愛因斯坦想盡量給聽眾一個印象：即他的貢獻不是源於甲，就是由於乙，而與愛因斯坦本人不太相干似的。

就連那篇亙古以來嶄新獨創的狹義相對論，並無參考可引，卻在最後天外飛來一筆，「感謝同事朋友貝索的時相討論。」

其他的文章，比如奮鬥苦思了十幾年的廣義相對論，數學部份推給了昔年好友的合作：這種謙抑，這種不居功，科學史中是少見的。

我就想，如此大功而竟不居，為什麼？像愛因斯坦之於相對論，像我祖母之

於我家。

幾年來自己的奔波，做了一些研究，寫了幾篇學術文章，真正做了一些小貢獻以後，才有了一種新的覺悟：即是無論什麼事，得之於人者太多，出之於己者太少。因為需要感謝的人太多了，就感謝天罷。無論什麼事，不是需要先人的遺愛與遺產，即是需要眾人的支持與合作，還要等候機會的到來。越是真正做過一點事，越是感覺自己的貢獻之渺小。

於是，創業的人，都會自然而然的想到上天，而敗家的人卻無時不想到自己。







# 摘要

中文摘要

關鍵字：LaTeX、中文、論文、模板







# Abstract

Abstract

**Keywords:** LaTeX, CJK, Thesis, Template

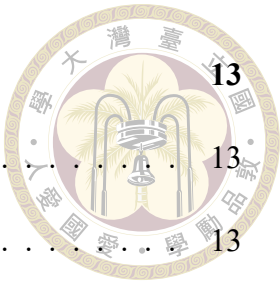




# Contents

	Page
Verification Letter from the Oral Examination Committee	i
Acknowledgements	iii
摘要	vii
Abstract	ix
Contents	xi
List of Figures	xiii
List of Tables	xv
Denotation	xvii
Chapter 1 Introduction	1
Chapter 2 Background and Related Work	3
2.1 The Rust Programming Language . . . . .	3
2.2 VM Protection . . . . .	9
Chapter 3 KrustVM Migration	11
3.1 Integration with Linux . . . . .	11
3.2 Hardware Enablement . . . . .	11
3.3 Code Migration . . . . .	11

<b>Chapter 4</b>	<b>KrustVM Design and Implementation</b>	<b>13</b>
4.1	Rcore Memory Regions . . . . .	13
4.2	Memory Region Isolation . . . . .	13
<b>Chapter 5</b>	<b>Evalutation</b>	<b>15</b>
<b>Chapter 6</b>	<b>Conclusions</b>	<b>19</b>
<b>References</b>		<b>21</b>
<b>Appendix A — Introduction</b>		<b>23</b>
A.1	Introduction . . . . .	23
A.2	Further Introduction . . . . .	23
<b>Appendix B — Introduction</b>		<b>25</b>
B.1	Introduction . . . . .	25
B.2	Further Introduction . . . . .	25





# List of Figures

Figure 5.1	Application Benchmark Performance . . . . .	17
------------	---	----





# List of Tables

Table 5.1	Application Benchmarks . . . . .	16
-----------	----------------------------------	----







# Denotation

HPC	高性能計算 (High Performance Computing)
cluster	集群
Itanium	安騰
SMP	對稱多處理
API	應用程序編程接口





# Chapter 1 Introduction





# Chapter 2 Background and Related Work

## 2.1 The Rust Programming Language

Rust, compared to C, is a relatively young programming language aiming to be safe and fast. It enables programs to be memory-safe without requiring programmers to painstakingly manage memory, as in traditional languages (e.g., C/C++). Unlike other memory-safe languages (e.g., Python, Go, etc.), Rust does not leverage garbage collection mechanisms to ensure memory safety. Instead, it introduces the concepts of lifetimes and ownership to mandate the programmer to follow specific rules. This paradigm of statically enforcing programming rules empowers Rust to perform comparably to C since Rust's compiler has complete control over the code that runs during runtime and can optimize it accordingly. Additionally, Rust's safety rules ensure that no memory safety bugs will be present when satisfied, and the compiler automatically checks and prevents any violation of these rules.

**Ownership and Lifetimes.** In Rust, each piece of data is said to be *owned* by a single variable, and it is automatically *dropped* (freed) when the variable's *lifetime* ends. A variable's lifetime ends as the program control flow exits the block in which the variable

is declared. In 1, *y*'s lifetime starts at line 5 and ends at line 7 as the block closes. Hence, the `println!` macro is unable to find the value *y*, whose lifetime has already ended. Ownership can be transferred or *moved*. For example, assigning the owning variable to a new variable moves the ownership of the data to the new variable. And passing the variable into a function also moves the data ownership into the function. In both situations, the original variable returns to the uninitialized state, and using it would result in a compilation error.

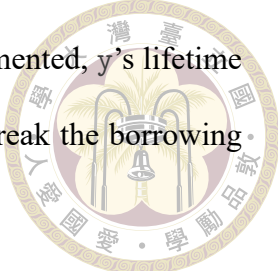
```
1 // this code sample does *not* compile
2 {
3   let x = 1;
4   {           // create new scope
5     let y;
6     y = x;
7   }           // y is dropped
8
9   // compilation error, y's lifetime has ended
10  println!("The value of 'y' is {}.", y);
11 }
```

Listing 1: Rust lifetime example

**Borrowing.** Ownership lacks the flexibility of argument passing. Rust addresses this by *borrowing*, a mechanism that allows accessing data without gaining ownership. A variable can borrow ownership from another variable to acquire a *reference* to the data. References can be divided into two categories, *shared* references and *exclusive* references. The reference can only be read and not modified with a shared reference. Nevertheless, multiple shared references for a specific value can be held simultaneously. On the other hand, exclusive references allow reading from and modifying the value. However, having any other kind of reference active simultaneously for that value is not permitted.

In summary, Rust's borrowing rule enforces *aliasing xor mutability* meaning there can be multiple shared references or a single exclusive reference. In 2, line 6 would not compile because it tries to create a mutable reference (*z*) to *x*, while *y* already borrowed *x* immutably. *y*'s lifetime ends on line 8 as it gets used for the last time; therefore *z* can be

created on line 10 and used on line 11. However, if line 13 is uncommented, *y*'s lifetime would be extended to line 13, making the creation of *z* on line 10 break the borrowing rules.



```
1 {
2   let mut x = vec![1, 2, 3];
3   let y = &x; // immutable borrow of x
4
5   // this line would fail to compile because x is already borrowed immutably by y
6   /* let z = &mut x; */
7
8   println!("x = {:?}", x); // This line works
9   println!("y = {:?}", y); // This line works
10
11  let z = &mut x; // mutable borrow of x
12  z.push(4);
13
14  // this line would fail to compile because x is borrowed mutably by z
15  /* println!("y = {:?}", y); */
16 }
```

Listing 2: Rust enforces *aliasing xor mutability*

**unsafe Rust.** Rust's safety checks are sometimes too restrictive regarding tasks like low-level hardware access or special optimizations. These operations are inherently unsafe and hence impossible to follow the rules mandated by Rust. However, they are still necessary for low-level software such as hypervisors. To provide flexibility for these operations, Rust allows parts of the program to opt out of its safety checks via the *unsafe* keyword. Traits, functions, and code blocks can be marked as unsafe to disable the checks that the compiler would normally enforce. However, using unsafe code also means that the responsibility for ensuring memory safety is shifted from the compiler to the programmer. Therefore, it is crucial to exercise caution when using unsafe code to avoid introducing bugs or security vulnerabilities.

**Interior unsafe.** While most low-level code is written in unsafe code, Rust introduces the concept of *interior unsafe* [5]. A function is considered interior unsafe if it exposes a safe interface but contains unsafe blocks in implementation. This allows unsafe operations to be encapsulated into safe abstractions. For instance, in 3, Rust's `replace`



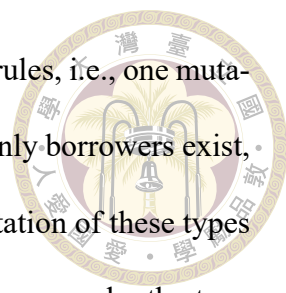
function can be called by safe Rust, but it is implemented using unsafe raw pointer operations. At line 6, `ptr::read` is used to copy a bit-wise value from `dest` into `result` without moving it, and at line 7, `ptr::write` overwrites the memory location pointed to by `dest` with the given value `src` without reading or dropping the old value. Lastly, at line 8, `result` is returned to the function's caller.

```
1 pub const fn replace<T>(dest: &mut T, src: T) -> T {
2     // SAFETY: We read from `dest` but directly write `src` into it afterward,
3     // such that the old value is not duplicated. Nothing is dropped and
4     // nothing here can panic.
5     unsafe {
6         let result = ptr::read(dest);
7         ptr::write(dest, src);
8         result
9     }
10 }
```

Listing 3: interior unsafe in Rust's `replace` function

This leads to a design practice that interior unsafe functions should provide the necessary checks that prevent the unsafe code from producing any undefined behavior or memory safety bugs. The callee in the safe world hence bears no responsibility to ensure safety.

**Interior Mutability.** Mutating the underlying data via an immutable reference is forbidden in Rust. However, this might be too restrictive for implementing efficient algorithms or data structures. For example, programmers might want to add a cache in a read-only search data structure to optimize the search time. Nevertheless, updating the state of the cache implies the need for mutability, which violates the read-only constraint. Hence, we need the ability to mutate states even under a read-only scene. To address this issue, the Rust standard library provides some special types that can mutate the underlying data even if we only have read-only access to the data holder. This design pattern is known as Interior Mutability. Implementing these types requires `unsafe` operations to bend Rust's usual rules that govern mutation and borrowing. To avoid violating the virtue

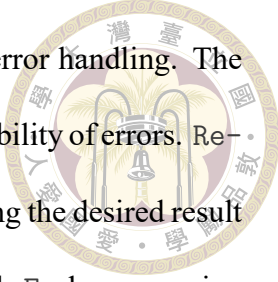


of the Rust safety assumptions, these types ensure that the borrowing rules, i.e., one mutable borrower at the same time and no mutable borrowers when read-only borrowers exist, will be followed at runtime. If these rules get violated, the implementation of these types has the responsibility to stop the behavior to avoid safety issues. For example, the type `Mutex` in Rust is a type that provides interior mutability. It uses a lock to ensure that only one borrower of the inner data can appear simultaneously to enforce safe mutation of the inner data, even without explicit mutability to `Mutex`. More precisely, when attempting to borrow data that has already been borrowed, the `Mutex` enforces a busy wait until the data is returned, thereby allowing only one borrower at a time. However, if a thread borrows the inner data of `Mutex` while it is already borrowing it, `Mutex` will wait forever, i.e., result in a self-deadlock.<sup>1</sup>

**Generics and Traits.** In addition to the safety mechanisms, Rust, as a modern programming language, provides handy features to make programming easier. `Generic` allows code to work with type parameters, reducing the effort of writing similar code for multiple types. For example, using generics, the `Mutex` type can hold and lock any arbitrary type. Rust traits are properties or interfaces that can be implemented on types; traits typically require the implementing type to supply function implementations for its trait methods. Additionally, combined with `Generic`, a trait can be treated as a restriction on type specifications such as function arguments or struct fields. The restriction is called a *trait bound*. For example, the `Clone` trait requires the implementing type to provide implementations for its `clone` and `clone_from` functions to make copies of themselves. A `Generic` function or type can use a trait bound to require its type argument to implement `Clone`, so that it can invoke the `clone` function that the argument implements.

---

<sup>1</sup>This is the behavior when using `Mutex` on Linux. On Windows, `Mutex` might panic.



**Graceful Error Handling.** Rust offers a graceful approach to error handling. The `Result<T, E>` and `Option<T>` types in Rust explicitly admit the possibility of errors. `Result` represents the outcome as either `Ok(T)` or `Err(E)`, with `T` denoting the desired result and `E` representing the error reason. Programmers are obligated to handle `E` when accessing `T`, and not doing so would result in a compilation error. To simplify error handling, Rust provides a convenient syntactic sugar, the `?` operator. It permits the retrieval of `T` from `Result` if it is `Ok(T)`, or early return of `E` if it is `Err(E)`. Similarly, `Option` simplifies error handling with two possibilities: `Some(T)` or `None`, where `None` can signify a trivial error. These types prevent unexpected errors when accessing a potentially non-existing value in the program.

**Copy and Drop Traits.** Some traits in Rust have intrinsic meaning to the compiler. For example, the `Drop` trait tells the compiler that a type has special freeing code, and the `Drop` trait's `drop` function should be invoked when an instance of the type goes out of scope. And the `Copy` trait, when implemented for a type indicates that the type should be byte-by-byte copied when the assignment (`=`) operator is used instead of Rust's typical semantic of moving the ownership to the new variable. Interestingly, Rust forbids a type from being `Drop` and `Copy` simultaneously, the designers of the language observed that if a type requires special deallocating code (the `drop` function), then it should also require a special copying function, rather than just copying it byte-by-byte. For instance, a type that holds a reference to the heap requires a `drop` function that frees the data pointed to by the reference, copying the object of the type in a byte-by-byte manner introduces risks of double-free, use-after-free, etc.

## 2.2 VM Protection







## Chapter 3 KrustVM Migration

### 3.1 Integration with Linux

challenge: Linux 5.15 does not include Rust support

solution: our way of source code organization, build system integration, how we link Rust and C, data layout issues, etc.

### 3.2 Hardware Enablement

we chose RPI4B to verify our implementation on real hardware

challenge: linker script allocation does not work for RPI4B

solution: memblock

### 3.3 Code Migration

challenge: difficult to do a top-down design due to the difficulty to debug and complexity of a hypervisor (is this challenge kinda weak?)

solution: two pass method, first function-by-function rewrite, then remove unsafe

and redesign after we have a working Rust hypervisor, and leverage Rust to achieve a stronger memory region isolation guarantee.





# Chapter 4 KrustVM Design and Implementation

Continues from Code Migration (the redesign after the function-by-function rewrite),  
talk about Rcore here.

## 4.1 Rcore Memory Regions

Rust has safe and unsafe code -> segregate unsafe code so that most of the hypervisor  
is written in safe Rust

## 4.2 Memory Region Isolation







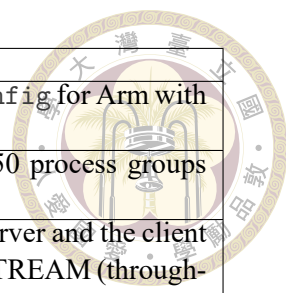
## Chapter 5 Evaluation

We evaluated the performance of various application benchmarks on a VM running on KrustVM and mainline KVM. We also tested the same benchmarks on bare metal environment performances to establish a baseline reference of the benchmark results. We ran the workloads on the Raspberry Pi 4 model B development board, with a Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC at 1.5GHz, 4GB of RAM, and a 1 GbE NIC device.

KrustVM and the mainline KVM are based on Linux 5.15. QEMU v4.0.0 was used to start the virtual machines on Ubuntu 20.04. The guest kernels also used Linux 5.15, and all kernels tested employed the same configuration. We extended QEMU v4.0.0 based on the artifact from [2] to support secure VM boot on KrustVM. We requested the authors of [4] and got a patch for the Linux guest kernel to enable virtio. `rustc` version 1.68.0-nightly was used to compile Rcore, while `clang` 15.0.0 was used to compile the remaining components of KrustVM and the mainline KVM.

We configured the hardware with 2 physical CPUs and 1 GB of RAM for the bare metal setup. Each VM that equips with 2 virtual CPUs for the VM setup, and 1 GB of RAM runs on the full hardware available.

We ran the benchmarks listed in Table 5.1 in the VMs on both KrustVM and the



Name	Description
Kernbench	Compilation of the Linux 6.0 kernel using <code>tinyconfig</code> for Arm with GCC 9.4.0.
Hackbench	<code>hackbench</code> [7] using Unix domain sockets and 50 process groups running in 50 loops.
Netperf	<code>netperf</code> [3] v2.6.0 running the netserver on the server and the client with its default parameters in three modes: TCP_STREAM (throughput), TCP_MAERTS (throughput), and TCP_RR (latency).
Apache	Apache v2.4.41 Web server running ApacheBench [8] v2.3 on the remote client, which measures the number of handled requests per second when serving the 41 KB <code>index.html</code> file of the GCC 4.4 manual using 100 concurrent requests.
Memcached	<code>memcached</code> v1.5.22 using the <code>memtier</code> [6] benchmark v1.2.3 with its default parameters.
YCSB-Redis	<code>redis</code> v7.0.11 using the YCSB [1] benchmark v0.17.0 with its default parameters.

Table 5.1: Application Benchmarks

mainline KVM. Figure 5.1 shows the normalized results. We normalized the results to bare-metal performance. 1.00 refers to no virtualization overhead. A higher value means higher overhead. The performance on real application workloads show modest overhead overall for KrustVM compared to mainline KVM.

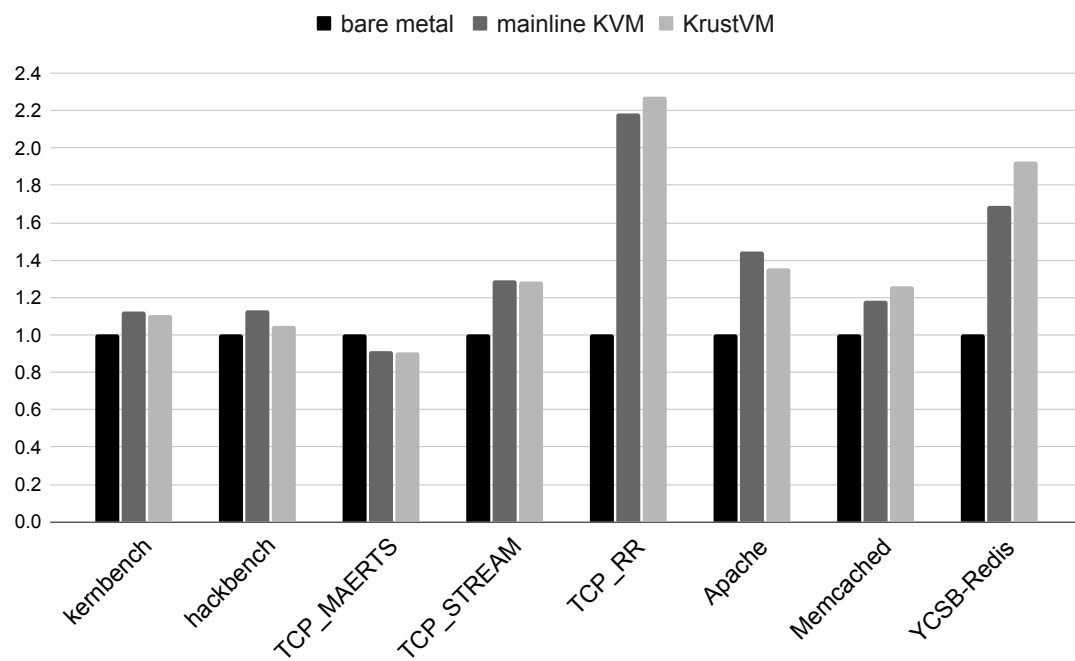


Figure 5.1: Application Benchmark Performance





## Chapter 6 Conclusions





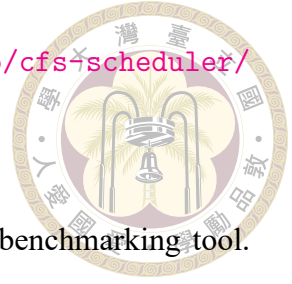
## References

- [1] Brian Cooper. Yahoo! Cloud Serving Benchmark. <https://github.com/brianfrankcooper/YCSB>, Feb. 2021.
- [2] Columbia University. SOSP 21: Artifact Evaluation: Verifying a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. <https://github.com/VeriGu/sosp-paper211-ae>, Sept. 2021.
- [3] R. Jones. Netperf. <https://github.com/HewlettPackard/netperf>, June 2018.
- [4] S.-W. Li, J. S. Koh, and J. Nieh. Protecting cloud virtual machines from commodity hypervisor and host operating system exploits. In Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19, page 1357–1374, USA, 2019. USENIX Association.
- [5] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, page 763–779, New York, NY, USA, 2020. Association for Computing Machinery.
- [6] Redis Labs. memtier\_benchmark. [https://github.com/RedisLabs/memtier\\_benchmark](https://github.com/RedisLabs/memtier_benchmark), Apr. 2015.



[7] R. Russell. Hackbench. [http://people.redhat.com/mingo/cfs-scheduler/  
tools/hackbench.c](http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c), Jan. 2008.

[8] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool.  
<http://httpd.apache.org/docs/2.4/programs/ab.html>, Apr. 2015.





# **Appendix A — Introduction**

## **A.1 Introduction**

## **A.2 Further Introduction**





## **Appendix B — Introduction**

### **B.1 Introduction**

### **B.2 Further Introduction**