Revolutionizing System Support: Supporting Firecracker Virtualization for Jinux Platform

*Group project I - Presentation III

1st Ruixiang JIANG
Southern University of Science and Technology
Shenzhen, China
12111611@mail.sustech.edu.cn

2nd Wenqian YAN
Southern University of Science and Technology
Shenzhen, China
12113020@mail.sustech.edu.cn

***Abstract*—Firecracker is an open-source virtualization technology developed by Amazon Web Services (AWS), tailored for the modern cloud computing landscape. At the same time, Jinux is a secure, fast, and general-purpose OS kernel, written in Rust and providing Linux-compatible ABI. Our propose is to support Firecracker virtualization for Jinux platform.**

***Index Terms*—Firecracker, Jinux, Rust**

I. Introduction

Firecracker, an open-source virtualization technology, has been meticulously engineered by Amazon Web Services (AWS) to cater to the evolving needs of the contemporary cloud computing ecosystem. Its design is characterized by a set of intricate features and optimizations that enable efficient virtualization and rapid deployment of virtual machines (VMs).

In parallel, Jinux, an exceptionally secure and high-performance operating system kernel written in the Rust programming language, boasts full compatibility with the Linux-compatible Application Binary Interface (ABI). Jinux's reputation for security and speed makes it an ideal candidate for enhancing the virtualization landscape.

Our primary research objective is to advance the compatibility and seamless integration of Firecracker virtualization within the Jinux platform. This collaboration will involve detailed analysis, testing, and the development of necessary interfaces and components. By bringing together the strengths of Firecracker and Jinux, we aim to deliver a robust and secure virtualization environment that not only meets the stringent demands of cloud computing but also fosters innovation in the field of virtualization technology.

II. Distinctive Characteristics of Firecracker

In the era of cloud computing, the demand for fast and efficient virtualization solutions has grown exponentially. Firecracker, introduced by AWS, offers a unique approach to meet these demands.

Firecracker provides a RESTful API through which a user can change some settings of the virtual machine. Because of its direct interaction with a user, the API server component is a good choice for our set of targets.

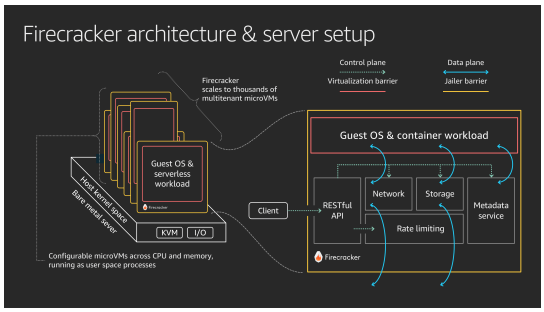

Fig. 1. Firecracker architecture & server setup [1]

The guest OS must interact with the hypervisor virtual devices for access to network and storage. A custom operating system crafted by a malicious user can use this communication link as an attack vector. Currently, in Firecracker only five emulated devices are available: (i) a network device, (ii) a block device, (iii) a vsock implementation, (iv) a serial console, and (v) a minimal keyboard controller. This seems to be a substantial attack surface, and we chose to focus our attention on the first three devices as they provide a complex implementation that is prone to hidden vulnerabilities. [2]

A. Rapid Boot Time

One of the standout features of Firecracker is its remarkable speed in launching virtual machines. With a startup time of less than one second, Firecracker is ideally suited for applications requiring rapid scaling, such as serverless functions.

There is a comparison of the boot times of different VMMs. The boot time is measured as the time between when VMM process is forked and the guest kernel forks its init process. For this experiment we use a minimal init implementation, which just writes to a pre-configured IO port. We modified all VMMs to call exit() when the write to this IO port triggers a VM exit. [3]

The figure below shows the cumulative distribution of (wall-clock) kernel boot times for 500 samples executed serially,

so only one boot was taking place on the system at a time. Firecracker results are presented in two ways: end-to-end, including fork- ing the Firecracker process and configuration through the API; and pre-configured where Firecracker has already been set up through the API and time represent the wall clock time from the final API call to start the VM until the init process gets executed. [3]

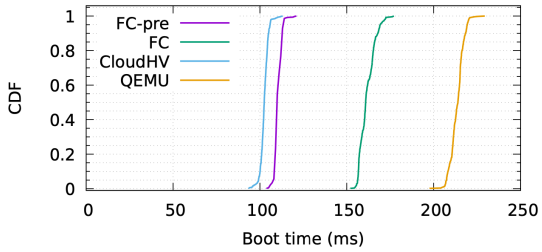

Fig. 2. Cumulative distribution of wall-clock times for start- ing MicroVMs in serial, for pre-configured Firecracker (FC- pre), end-to-end Firecracker, Cloud Hypervisor, and QEMU. [3]

This feature enables cloud providers to allocate resources dynamically, improving the overall user experience and resource utilization. Firecracker can achieve rapid boot time primarily due to its lightweight and minimalistic design, which eliminates many of the traditional virtualization overheads.

Here are the key reasons why Firecracker can provide such fast boot time below.

- MicroVM Architecture
  Firecracker uses a MicroVM (micro virtual machine) architecture. Unlike traditional VMs, MicroVMs are stripped down to include only essential components required for execution. This minimalistic approach reduces the time needed for the VM to start, as there are fewer services and processes to initialize.
- Firecracker Kernel
  Firecracker uses the Firecracker kernel, which is a custom-built, minimalistic and highly optimized Linux kernel tailored specifically for lightweight VMs. This specialized kernel is designed to boot quickly and manage VMs efficiently.
- Just-In-Time Initialization
  Firecracker employs a just-in-time initialization process, meaning that many VM components and services are initialized only when they are first used. This defers the startup of non-essential components until they are required, speeding up the initial VM launch.
- Pre-Allocated Resources
  Firecracker pre-allocates resources for VM instances, which eliminates the need for resource-intensive operations like dynamic memory allocation during boot-up. This approach reduces boot time by ensuring that the VM has immediate access to the resources it needs.
- Reduced Emulation Overhead
  Firecracker uses a KVM (Kernel-based Virtual Machine) backend, which allows it to leverage hardware virtualization support. This results in significantly reduced CPU and memory overhead compared to full virtualization, further contributing to faster boot time.
- Single-Purpose VMs
  Firecracker is designed for single-purpose VMs, such as serverless functions or container instances, which means it doesn't need to load a full-fledged operating system and associated services, further reducing startup time.

These design principles, along with a focus on simplicity and efficiency, enable Firecracker to achieve rapid boot time, making it well-suited for use cases where fast scalability and low latency are essential, such as serverless computing and containerization.

B. Resource Efficiency

Firecracker is designed with resource efficiency in mind. It utilizes a minimalistic approach to virtual machine management, resulting in lower resource overhead. This efficient resource usage allows for the deployment of numerous VM instances on the same physical host, increasing the density of workloads without sacrificing performance. It achieves resource efficiency through several design choices and optimizations that minimize resource overhead.

- Low Memory Footprint
  Firecracker's custom-built kernel and minimalistic design contribute to a small memory footprint for each VM. This reduction in memory usage allows for increased VM density on the same host, optimizing resource utilization.
- Multi-Tenant Isolation
  Firecracker's strong isolation mechanisms ensure that each VM is securely isolated from others, preventing resource contention and interference in multi-tenant environments.
- Resource C-group Management
  Firecracker uses resource control mechanisms like C-groups to manage resource allocation to VMs. Administrators can set resource limits and priorities, enabling fine-grained resource management and optimization.
- Efficient Component Initialization
  Firecracker employs a just-in-time initialization process to activate VM components and services only when they are needed. This design minimizes resource usage during VM boot-up, ensuring efficient resource allocation.

These resource-efficient design principles make Firecracker an excellent choice for cloud environments where efficient resource usage and scalability are crucial, as it allows for the efficient execution of workloads without compromising on performance or security.

C. High security

Firecracker emphasizes security in its design to provide a secure virtualization environment. The following features and design principles contribute to Firecracker's security.

- Isolation
  Firecracker ensures strong isolation between VM instances. Each VM runs in its own isolated environment,

and there is limited interaction between VMs. This isolation helps prevent security breaches and resource interference. What's more, it isolates VMs from the underlying host system, limiting the impact of potential attacks on the host. The attack surface presented to the host is minimized, enhancing overall system security.

- Minimal Attack Surface
  Firecracker's minimalist design reduces the attack surface. By stripping away unnecessary components and services, it decreases the potential entry points for attackers.
- Customizable Security Policies
  Firecracker allows administrators to customize security policies to meet their specific requirements. This flexibility enables the implementation of security controls tailored to the needs of a given environment.
- Monitoring and Auditing
  Firecracker can be integrated with monitoring and auditing tools to track VM activities and detect any suspicious behavior or security breaches. This proactive approach enhances overall security.

By incorporating these security features and design principles, Firecracker offers a secure virtualization platform, making it suitable for multi-tenant environments, critical workloads, and scenarios where strong security is a top priority.

D. Container Support

Firecracker complements containerization technologies like Docker and Kubernetes. It enables the quick encapsulation of containers into virtual machines, delivering improved isolation and security. This integration allows organizations to leverage the benefits of both containerization and virtualization for their workloads.

In conclusion, Firecracker represents a significant advancement in the field of virtualization, offering a host of compelling advantages for the modern cloud computing landscape. Its ability to deliver rapid boot time, resource efficiency, security, and container support positions it as a versatile and impactful solution for a wide range of cloud-based workloads and applications.

The rapid boot time of Firecracker, measured in milliseconds, is a game-changer in the dynamic world of cloud computing. This feature allows cloud providers to allocate resources dynamically and respond to varying levels of demand with unmatched agility. Whether in the context of serverless computing, microservices, or container orchestration, this quick provisioning capability ensures a responsive user experience and maximizes resource utilization.

Resource efficiency is another pillar of Firecracker's strength. Its MicroVM architecture and pre-allocation of resources not only reduce overhead but also enable multiple virtual machine instances to coexist harmoniously on the same hardware, thereby optimizing infrastructure utilization. This trait is especially relevant in cloud environments where resource efficiency can translate into substantial cost savings.

Security is a paramount concern in the cloud, and Firecracker addresses it admirably. By isolating each virtual machine, limiting the attack surface, and reducing the risk of resource contention, Firecracker fortifies the security posture of cloud-based workloads. In multi-tenant environments, these security features enhance overall system integrity and data protection.

As an open-source project, Firecracker fosters collaboration, innovation, and transparency. It empowers developers to customize and extend the technology to meet specific use cases, thus contributing to the ongoing evolution of the virtualization landscape.

Furthermore, Firecracker's seamless integration with container technologies such as Docker and Kubernetes provides an extra layer of isolation and security for containerized workloads. This compatibility allows organizations to harness the advantages of both containerization and virtualization, striking a harmonious balance between agility and security.

In a rapidly evolving cloud computing landscape, Firecracker's blend of rapid provisioning, resource efficiency, security, and container support makes it an invaluable tool. It accommodates the requirements of a diverse array of applications, from serverless functions to microservices, and ensures that cloud environments remain responsive, secure, and cost-effective. The dynamic interplay between these advantages positions Firecracker as a key enabler of innovation and efficiency in the digital era.

III. Distinctive Characteristics of Jinux

Briefly, Jinux is a secure, fast, and general-purpose OS kernel, written in Rust and providing Linux-compatible ABI. [4]

Jinux stands out in its commitment to the principle of least privilege while concurrently maintaining high performance standards. This distinctive approach is underpinned by the comprehensive utilization of the Rust programming language, which, in contrast to conventional system programming languages such as C/C++, boasts an array of exceptional attributes. These attributes encompass an expressive type system, memory safety with the option of unsafe extensions, versatile macros, and a customizable toolchain. By harnessing the intrinsic capabilities of Rust, Jinux successfully architects zero-cost abstractions that facilitate the enforcement of the principle of least privilege across three distinct levels, thereby enhancing both security and performance.

- The architectural level.
  Jinux is architected as a framekernel, where the entire OS resides in a single address space and unsafe Rust code is restricted to a tiny portion of the OS called Jinux Framework. The Framework exposes safe APIs to the rest of Jinux, which implements the most of OS functionalities in safe Rust code completely. Thanks to the framekernel architecture, Jinux's TCB for memory safety is minimized.
- The component level.
  Upon Jinux Framework is a set of OS components, each of which is responsible for a particular OS functionality, feature, or device. These OS components are Rust

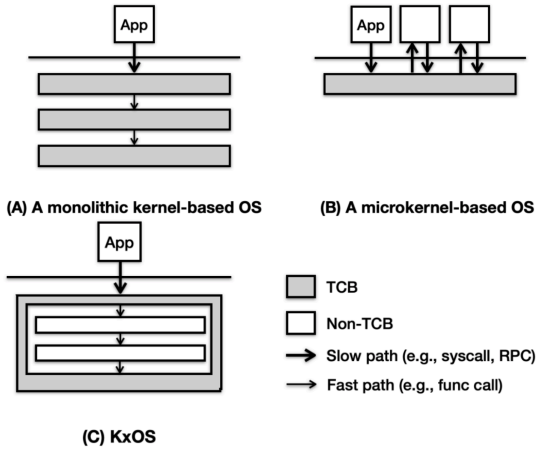

Fig. 3. A comparison between the architectures of traditional OSes and KxOS

crates with two traits: (1) containing safe Rust code, as demanded by the framekernel architecture, and (2) being governed by Jinux Component System, which can enforce a fine-grained access control to their public APIs. The access control policy is specified in a configuration file and enforced at compile time, using a static analysis tool.

- The object level.
  Jinux promotes the philosophy of everything-is-a-capability, which means all kernel resources, from files to threads, from virtual memory to physical pages, should be accessed through capabilities. In Jinux, capabilities are implemented as Rust objects that are constrained in their creation, acquisition, and usage. One common form of capabilities is those with access rights. Wherever possible, access rights are encoded in types (rather than values) so that they can be checked at compile time, eliminating any runtime costs. [4]

Here is an overview of the architecture of Jinux.

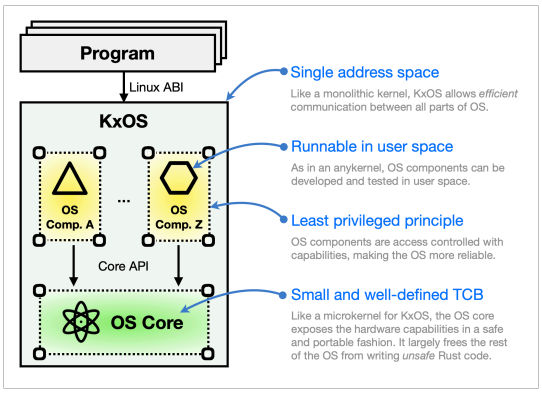

Fig. 4. An overview of architecture of Jinux [5]

Security is at the core of Jinux's design philosophy. The operating system adopts the "least privilege principle" as its guiding security best practice. This principle ensures that each component of the system has the minimum access and permissions required to perform its function, reducing the potential attack surface. Jinux enforces this principle by dividing itself into two distinct halves: a privileged OS core and unprivileged OS components. What sets Jinux apart is its use of Rust, a programming language known for its focus on memory safety and strong guarantees. While all OS components are written entirely in safe Rust, only the privileged OS core is allowed to incorporate unsafe Rust code. This careful separation of responsibilities enhances the system's security, reducing the risks associated with common programming errors and vulnerabilities. Additionally, Jinux introduces the concept of "everything-is-a-capability." Capabilities are elevated to the status of a ubiquitous security primitive used throughout the OS. Advanced features of Rust, such as type-level programming, are harnessed to make capabilities more accessible and efficient. The result is a robust security infrastructure that not only improves the system's resilience but also maintains its performance.

OS-level virtualization is a powerful tool for creating lightweight and efficient container environments. However, concerns have arisen regarding the security of containers, particularly when compared to traditional virtual machines. Malicious containers may exploit privilege escalation vulnerabilities in the underlying OS kernel, posing a threat to the host system. To address this issue, Jinux seeks to establish itself as a trustworthy OS-level virtualization platform. It aims to make OS-level virtualization as secure as VM-based solutions by ensuring the security of the OS kernel itself. By bolstering the underlying OS, Jinux aims to mitigate the risks associated with container security, allowing for a safer and more efficient approach to containerization.

Traditional OS kernel development can be a cumbersome and time-consuming process, involving numerous cycles of programming, testing, and debugging, often on bare-metal or virtual machines. Jinux acknowledges this pain point and takes steps to alleviate it. The Jinux design includes an OS core that provides high-level APIs, largely independent of the underlying hardware. These APIs are implemented with two targets in mind: one for the regular OS kernel space and the other for a library OS in user space. This design approach allows all OS components to be developed, tested, and debugged in user space. Developers can work more comfortably and efficiently, resulting in a more productive and enjoyable experience.

In conclusion, Jinux is an exciting addition to the world of operating systems. With its security-focused design, commitment to trustworthy OS-level virtualization, and emphasis on fast user-mode development, Jinux promises to be a game-changer in the field of operating systems. Its innovative use of Rust and the "least privilege principle" position it as a secure and efficient solution for modern computing needs.

IV. How to prepare the environment

A. Jinux

In terminal, follow this steps to run Jinux:

```
git clone [Jinux url]
docker pull jinuxdev/jinux:0.2.1
```

```
docker run -it --network=host -v `pwd`:/
    root/jinux jinuxdev/jinux:0.2.1
make build
make run ENABLE_KVM=0 BOOT_METHOD=microvm
```

The result is shown in Figure 1:

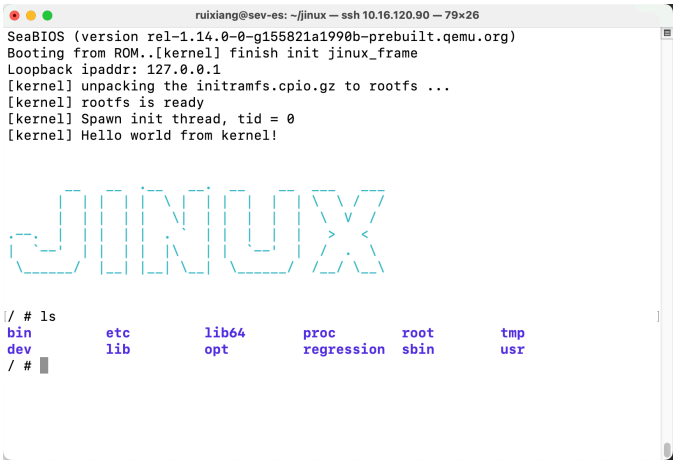

Fig. 5. Build Jinux

B. Firecracker

Firstly, download an official firecracker release from its github release page.

Then follow this steps to run Firecracker:

```
ARCH="$(uname -m)"
git clone https://github.com/firecracker-
    microvm/firecracker firecracker_src
systemctl start docker
./firecracker_src/tools/devtool build
cp ./firecracker_src/build/cargo_target/$
    {ARCH}-unknown-linux-musl/debug/
    firecracker firecracker

API_SOCKET="/tmp/firecracker.socket"
rm -f $API_SOCKET
./firecracker --api-sock "${API_SOCKET}"
```

One more terminal is need to run firecracker. The first one is running the firecracker binary, while the second one communicating with the firecracker process via HTTP requests.

Use how to run Ubuntu 22.04 in firecracker as an example. In another terminal, the instructions are shown in Code/Firecracker.txt. The output is shown in Figure 2.

It indicates that Ubuntu 22.04 is running on firecracker successfully.

C. Run Jinux on Firecracker

The bash instructions are shown in Codes/run.txt. Running the instructions we get a fault message:

```
{"fault_message":"Cannot load kernel due
    to invalid memory configuration or
    invalid kernel image: Kernel Loader:
    failed to load ELF kernel image"}
```

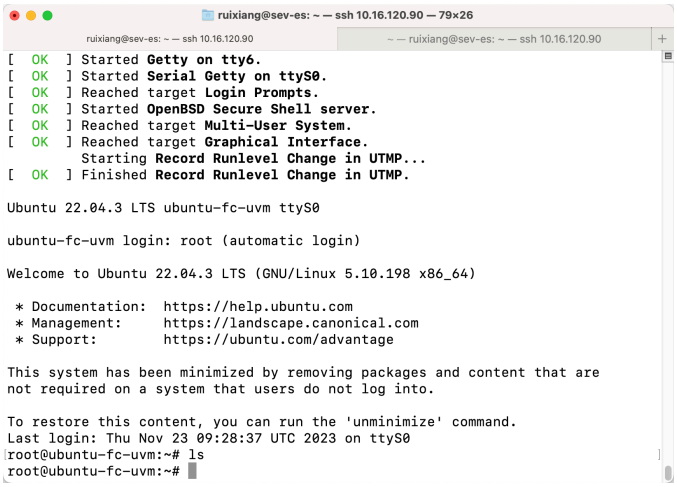

Fig. 6. Build Firecracker

Our goal is to modify the source code of Jinux, in order to run Jinux on firecracker.

V. Potential Reasons Analyses

The kernel is a core component of an operating system (OS). It serves as the central module that provides essential services for the functioning of the computer system. The kernel acts as an intermediary between the hardware and the user-level applications, managing hardware resources and providing a set of services for software.

ELF stands for Executable and Linkable Format. It is a common file format for executables, object code, shared libraries, and even core dumps in Unix and Unix-like systems. An ELF kernel image, in the context of operating systems, refers to the kernel binary file that adheres to the ELF format.

In the context of Linux and many other Unix-like operating systems, the kernel is often stored in an ELF format. When encounter an error message like ”Cannot load kernel due to invalid memory configuration or invalid kernel image: Kernel Loader: failed to load ELF kernel image,” it indicates an issue with loading the ELF-formatted kernel image into memory during the boot process. Possible reasons for such an error include a corrupted kernel image, incorrect configuration, or compatibility issues between the kernel and the boot loader or virtualization platform. Typically, Troubleshooting involves checking the integrity of the kernel image, ensuring proper configuration, and addressing any compatibility issues.

Noticing that Ubuntu has ran successfully on firecracker, we indicate that error occurs in ELF file generated after compiling the Jinux system. The most potential problem is that the ELF file fail to follow the manual of firecracker. Actually, according to multi-boot protocol, some bits in the ELF file should be polished.

Enabling the tracing of firecracker, we can position the fault, which is shown in Figure 3. Here fault raises at ‘/src/api_server/src/parsed_request.rs:198’, and the relevant code is shown in Figure 4.

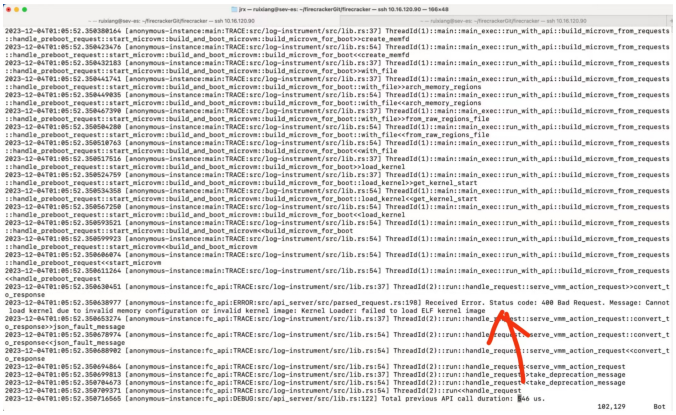

Fig. 7. Firecracker Tracing

```
                }
            );
            response.set_body(Body::new(ApiServer::json_fault_message(
                vmm_action_error.to_string(),
            )));
            response
        }
    }
```

Fig. 8. Code at parsed_request.rs:198

This error may occur due to the difference in the Linux systems supported by Jinux and Firecracker. Jinux supports Linux32, while Firecracker only supports Linux64.

Linux32 and Linux64 denote specific boot protocols or specifications used by boot loaders. These protocols could have distinct specifications and implementations, leading to variations in how applications are launched on different systems. Certain applications may encounter compatibility issues or fail to load correctly due to differences in the boot protocol, such as the startup procedure, loaded libraries, or other related factors, when transitioning from Linux32 to Linux64 boot protocols.

The error message indicates a failure to load the kernel due to an invalid memory configuration or an invalid kernel image. Specifically, the issue might stem from incompatibility in the boot protocols, such as Jinux utilizing the Linux32 boot protocol and Firecracker requiring applications using the Linux64 boot protocol. Another potential cause could be an incorrect or non-compliant format of the kernel image that fails to meet the requirements of Firecracker, leading to the loading failure. Considering both possibilities, it is crucial to ensure the use of the Linux64 boot protocol compliant with Firecracker specifications and verify that the kernel image adheres to the correct format and configuration.

A. *Successful Loading ELF*

Firstly, the issue concerning the erroneous loading of ELF files was rectified, enabling the successful execution of Firecracker by loading the ELF files on the KVM. Subsequently, a peculiar phenomenon was observed: immediately upon initiation of the VMM, it received a `KVM_SHUTDOWN_SIGNAL` and terminated without yielding any error messages.

In the absence of error messages, guidance on how to rectify the issue at hand remains elusive. It is, therefore,

```
$ ./firecracker --no-api --show-level --config-file config.json
2023-12-05T00:05:09.819659413 [anonymous-instance:main:INFO] Running Firecracker v1.6.0-dev
2023-12-05T00:05:09.820380338 [anonymous-instance:main:DEBUG:src/vmm/src/builder.rs:385] event_start: build microvm for boot
2023-12-05T00:05:09.887246112 [anonymous-instance:main:DEBUG:src/vmm/src/builder.rs:387] event_end: build microvm for boot
2023-12-05T00:05:09.887248094 [anonymous-instance:main:DEBUG:src/vmm/src/builder.rs:389] event_start: boot microvm
2023-12-05T00:05:09.887256776 [anonymous-instance:main:INFO:src/vmm/src/device_manager/mmio.rs:402] Artificially kick devices.
2023-12-05T00:05:09.887328415 [anonymous-instance:main:DEBUG:src/vmm/src/builder.rs:394] event_end: boot microvm
2023-12-05T00:05:09.887337962 [anonymous-instance:main:INFO:src/firecracker/src/main.rs:582] Successfully started microvm that was
configured from one single json
2023-12-05T00:05:09.887457569 [anonymous-instance:fc_vcpu 0:INFO:src/vmm/src/vstate/vcpu/mod.rs:484] Received KVM_EXIT_SHUTDOWN sig
nal
2023-12-05T00:05:09.887482989 [anonymous-instance:main:INFO:src/vmm/src/lib.rs:843] Vmm is stopping.
2023-12-05T00:05:09.887685110 [anonymous-instance:main:INFO:src/vmm/src/lib.rs:843] Vmm is stopping.
2023-12-05T00:05:09.910168258 [anonymous-instance:main:INFO:src/firecracker/src/main.rs:102] Firecracker exiting successfully. exit
_code=0
```

Fig. 9. Log of Launch

imperative to devise methods that enable the extraction of even a modicum of informative data. A review of the Linux kernel code reveals that the `KVM_SHUTDOWN_SIGNAL` signal appears trifariously, characterized by a lone instance associated with routine termination, and two occurrences induced by faults or similar contingencies.

Despite this knowledge, resolution remains impeded; the locus of error eludes precise identification, akin to the proverbial blind men attempting to discern an elephant by touch. In response, modifications were introduced into the Firecracker source code to facilitate the output of contextual information upon signal reception, encompassing, but not restricted to, register and control register values. This intervention is anticipated to significantly enhance error localization efforts.

Subsequent analysis pinpointed the terminal value of the `rip` register at `0xffffffff88b8da3f`. Disassembling the kernel yielded the insight that this corresponds to a `hlt` instruction. Further scrutiny of the boot assembly code disclosed the default utilization of the Multiboot booting protocol by Jinux. However, this protocols magic data failed to manifest during the startup, a plausible occurrence given Firecrackers incompatibility with Multiboot, culminating in a system halt.

jinux / framework / jinux-frame / src / arch / x86 / boot / **boot.S**

Code | Blame | 376 lines (320 loc) · 9.68 KB

```
47  .code32
48  .global __multiboot_boot
49  __multiboot_boot:
50      cli
51      cld
52
53      // Set the kernel call stack.
54      mov esp, offset boot_stack_top
55
56      push 0      // Upper 32-bits.
57      push eax    // multiboot magic ptr
58      push 0      // Upper 32-bits.
59      push ebx    // multiboot info ptr
60      // Tell the entry type from eax
61      cmp eax, MULTIBOOT_ENTRY_MAGIC
62      je magic_is_mb
63      cmp eax, MULTIBOOT2_ENTRY_MAGIC
64      je magic_is_mb2
65      jmp halt     // Should not be reachable!
66  magic_is_mb:
67      push 0       // Upper 32-bits
```

Fig. 10. Not-Multiboot Leads to Halt

B. *Boot into Rust*

The resolution appears straightforward: switching the boot protocol to Linux32 might suffice. However, this approach engendered an ancillary complication; the Linux32 boot process

is predicated on 32-bit assembly language. Executing this code in a 64-bit modeas is the practice with the Firecracker, which employs a Linux64 boot protocolresults in aberrant behavior and potentially leads to a different kind of segment fault.

To address this, the boot sequence was re-implemented using 64-bit assembly language. Thereafter, Jinux was capable of initiating normally, progressing into the Rust code segment, and accomplishing serial output. Thus, the initial phase of the startup process was concluded.

```
$ ../firecracker/build/cargo_target/x86_64-unknown-linux-musl/debug/firecracker --no-api --show-level --config-file config.json
2023-12-06T22:50:02.095633926 [anonymous-instance:main:INFO] Running Firecracker v1.6.0-dev
2023-12-06T22:50:02.157250648 [anonymous-instance:main:INFO:src/vmm/src/device_manager/mmio.rs:386] Artificially kick devices.
2023-12-06T22:50:02.157300648 [anonymous-instance:main:INFO:src/firecracker/src/main.rs:574] Successfully started microvm that
was configured from one single json
[INFO]: Found usable region, start:0, end:9fc00
[INFO]: Found usable region, start:100000, end:d0000000
[INFO]: Found usable region, start:100000000, end:130000000
[INFO]: Initializing trapframe...
[DEBUG]: new gdt:[0, 0, 0, 0, 0, 0, 0, 0, ff, ff, 0, 0, 0, 9a, af, 0, ff, ff, 0, 0, 0, 93, cf, 0, ff, ff, 0, 0, 0, 9a, cf, 0, 6
7, 0, 0, e0, d3, 89, 0, 88, ff, ff, ff, ff, 0, 0, 0, 0, 0, 0, 0, 0, 98, 20, 0, 0, 0, 0, 0, 0, 0, 92, 0, 0, ff, ff, 0, 0, 0, fa,
 cf, 0, ff, ff, 0, 0, 0, f2, cf, 0, 0, 0, 0, 0, 0, 0, f8, 20, 0], entry_count:11
[INFO]: GDT initialization completed
[INFO]: IDT initialization completed
[INFO]: Syscall related register initialization completed
[WARN]: ACPI info not found!
[INFO]: x2APIC ID:0, Version:14, Max LVT:5
[INFO]: [IOAPIC]: Not found ACPI talbes, using default address:fec00000
[INFO]: [IOAPIC]: IOAPIC id: 0, version:17, max_redirection_entry:24, interrupt base:0
```

Fig. 11. A Success Boot

Subsequently, issues related to device incompatibility emerged. Jinux was unable to initialize the APIC timer, and preliminary testing indicated a requirement for a network device to function properly. These compatibility issues with devices, among potentially others, necessitate further evaluation and remediation.

VI. Solution

A. Implement Linux64 Boot Protocol

This part involved the implementation of the Linux64 boot protocol for Jinux. The original Linux32 boot protocol of Jinux was evaluated and a comprehensive comparison between the Linux32 and Linux64 boot procedures was conducted. Due to the inherent scaffoldings laid out by the Linux64 boot protocol, where the bootloader is already set up in a long mode with Page Table enabled, our modifications were thus streamlined. We primarily focused on updating the page table, preparing the `boot_params` on the stack, and facilitating entry into Rust code.

B. Timer Initialization

In the course of our undertaking to migrate Jinux to Firecracker, we encountered a marked challenge regarding the initialization of the Advanced Programmable Interrupt Controller (APIC) timer. The common practice in Computing Architecture involves leveraging the Programmable Interval Timer (PIT) - recognized for its ready availability albeit lower precision - to assist in calculating the APIC frequency. However, upon initiating this process on Jinux, our team observed an unexpected halt during the APIC's initialization phase, an anomaly traced back to the system's failure to utilize the PIT timer. A deep-dive investigation revealed a discrepancy in the PIT's Interrupt Request (IRQ) number; with qemu operating on IRQ 2 whereas Firecracker employed IRQ 0. Promptly amending the IRQ number on Jinux, the operating system proved its ability to not only effectively setup the APIC timer but also successfully carry out subsequent initializations.

C. Correct VirtIO Device Discovery

A significant component of the migration was the rectification of the search range for the VirtIO devices, an issue we encountered during the initialization process. Traditionally, in environments like Qemu, VirtIO devices are allocated within a definitive memory space spanning from `0xFEB00000` to `0xFEB04000`. However, in Firecracker, we noted an alteration in this norm, with the devices being strategically placed in an area from `0xD0000000` to `0xFFFFF000`. Such disjointedness prompted an inability in Jinux to discover the devices in question. To address this, we incorporated Firecrackers range into Jinuxs search field, which resulted in successful device detection. Regrettably, a subsequent issue presented itself, involving a failure to initialize the detected virt-net device. This will be deliberated extensively in the following subsection.

D. Amend VirtIO Device Initialization

In the previous subsection, it's mentioned that we encountered and subsequently rectified an issue pertaining to the initialization procedures of VirtIO devices - a crucial element in enhancing the performance of the virtual machines. The problem originated from the distinctive state transition paradigms employed by Qemu and Firecracker. In the original Qemu model, the state machine managing each VirtIO device, which begins at the INIT state, is permitted to transition directly to the ACKNOWLEDGE+DRIVER state. However, the Firecracker model necessitates an intermediate transition to ACKNOWLEDGE state prior to achieving the ACKNOWLEDGE+DRIVER state. To facilitate a successful transition in accordance with Firecracker requirements, we modified the initialization procedure of the Jinux operating system so that it conforms specifically to this two-step state transition sequence. As a result, the Jinux system on Firecracker is now able to proceed from the INIT state, through the ACKNOWLEDGE state, and finally to the ACKNOWLEDGE+DRIVER state. This adjustment has effectively enabled successful and compliant initialization.

After applying these amending, we had made Jinux boot successfully.

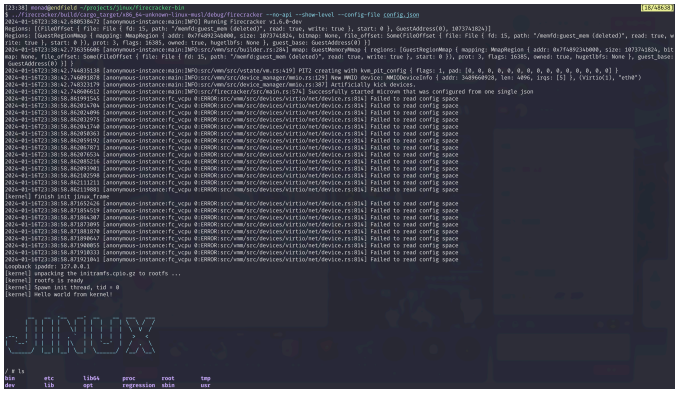

Fig. 12. Success Migration

VII. Conclusion

Adapting Jinux to run within the Firecracker virtualization environment represents a significant undertaking that involves integrating Firecracker support, reconfiguring MicroVM parameters, adding device emulation, and enhancing security features. Successfully achieving this adaptation will enable users to harness the lightweight, secure, and efficient capabilities of Firecracker while utilizing Jinux for managing MicroVM workloads. This integration could offer an exciting new dimension to the world of MicroVMs, expanding their utility in cloud computing and container orchestration platforms.

References

[1] "Firecracker open-source innovation", DevelopersIO. [Online]. Available: https://dev.classmethod.jp/articles/reinvent2019-opn402

[2] G. Mocanu, C. Caraba and N. pu, "Fuzz testing in AWS Firecracker hypervisor," 2021 20th International Symposium on Parallel and Distributed Computing (ISPDC), Cluj-Napoca, Romania, 2021, pp. 130-137, doi: 10.1109/ISPDC52870.2021.9521598.

[3] A. Agache et al., Firecracker: Lightweight virtualization for serverless applications, in 17th USENIX symposium on networked systems design and implementation (NSDI 20), 2020, pp. 419434.

[4] "Jinux", Github. [Online]. Private: https://github.com/jinzhao-dev/jinux

[5] "Introduction", Github. [Online]. Private: https://github.com/jinzhao-dev/jinux/tree/main/docs/src