Ataru: A Lightweight VMM+Runtime for Low Latency Serverless Functions

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Faculty of Engineering

BY

Prakhar Gupta

Computational Data Science
Indian Institute of Science
Bangalore – 560 012 (INDIA)

October, 2024

Declaration of Originality

I, **Prakhar Gupta**, with SR No. **06-18-01-37-51-21-1-20068** hereby declare that the material presented in the thesis titled

Ataru: A Lightweight VMM+Runtime for Low Latency Serverless Functions

represents original work carried out by me in the **Department of Computational Data Science** at **Indian Institute of Science** during the years **2021-23**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: 10-05-2023 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Dr. Lakshmi Jagarlamudi Advisor Signature

Acknowledgements

I am grateful to Dr Lakshmi Jagarlamudi, my esteemed supervisor, for her invaluable guidance, support and supervision throughout my M.Tech degree. I also thank the Faculty of IISc for providing me with the knowledge and understanding to pursue my studies. I appreciate the support of M. Sankaran (Director, URSC), Mr Sudhakar S (DD, URSC), Jagannath Das (GD, OSDG), Partha Bandyopadhyay (DH, CMSDD), and my colleagues at OSDG, URSC. I thank my friends, lab mates and colleagues Sheshadri, Dhanya, Archita and Prathamesh, for the memorable time we shared in the lab and outside. I also thank my family for their encouragement and support throughout my studies. I am especially grateful to my wife, Palmi, for her invaluable support and motivation.

Abstract

Serverless computing is an emerging paradigm where multi-tenancy, over-provisioning, short startup times, and execution speeds are important. FaaS is a popular realization of serverless computing, characterized by short-lived stateless functions triggered by events. Resource allocation, isolation, scalability, and orchestration are left to the service provider. In recent years, the research community has put effort into achieving these goals using Containers, Light Weight VMs, and, more recently, v8 Isolates (Web Assembly). However, Containers and VMs have bulky image sizes, and both suffer from cold start problems. While v8 isolates are yet to achieve bare metal performance. Although different execution contexts require isolation, sharing memory among function instantiations of the same workflow is desirable. In current serverless implementations, overheads of bootstrapping the process for function execution dominate the function execution time; this is a hurdle for applications requiring low latency and high throughput. In this research, we explore the possibility of developing a native function execution framework to exploit full hardware potential with the provision for shared memory and minimal process bootstrapping overhead. However, without sacrificing the security offered by virtualization technologies. Our objective is to optimize the platform overheads that degrade the performance of DAG execution in serverless architecture, especially when the DAG consists of short-lived functions. To achieve this, we design and implement Ataru-KVM VMM and Ataru Runtime. We benchmark our architecture against Firecracker, a widely used virtualization platform, prominently used for serverless worker instances and show that: 1. Ataru outperforms Firecracker significantly when the DAG functions have execution times in the order of tens of microseconds. 2. Ataru optimizes the utilization of the vCPUs assigned to it by dynamically suspending and resuming the vCPUs. 3. Due to its low memory footprint, Ataru achieves two order magnitude faster bootup times compared to Firecracker.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

1.1 Introduction 1

1.2 Motivation 2

1.3 Related Work 3

2 Programming Model **5**

2.1 Data Structure for DAG Representation 5

3 Architecture **8**

3.1 Ataru-KVM VMM 9

3.1.1 System IOCTLs 10

3.1.2 VM IOCTLs 10

3.1.3 vCPU IOCTLs 11

3.1.4 Device IOCTLs 11

3.1.5 Initialization Sequence 12

3.1.6 vCPU Management 13

3.1.7 ELF Format and Memory Image 13

3.1.8 Memory Layout 14

3.1.9 Dynamic Linking and Loading 17
3.2 Ataru Runtime 19
3.2.1 Changing Processor Mode 19
3.2.2 Stack Assignment 20
3.2.3 IRQ Chip Configuration 20
3.2.4 Booting Application Processors 23
3.2.5 Descriptor Tables 24
3.2.5.1 Global Descriptor Table 24
3.2.5.2 Interrupt Descriptor Table 25
3.2.6 Interrupts and Exceptions 25
3.2.7 System Calls 26
3.2.8 Bootup Sequence 27
3.2.9 Scheduler 27
3.2.9.1 LHP-LWP Problem 27
3.2.9.2 Lock Free Data Structure 28
3.2.9.3 Double CAS 29
3.3 Scaling using Ataru 30

4 Experiments and Results **32**
4.1 Experimental Setup 32
4.2 Results 34
4.2.1 Boot Time 35
4.2.2 Memory Requirements 36
4.2.3 Scaling 36
4.2.4 Micro-Benchmarking 37
4.2.4.1 SF: DAG with Single Function 38
4.2.4.2 C3: DAG with Chain of 3 Functions 40
4.2.4.3 C8: DAG with Chain of 8 Functions 42
4.2.4.4 P4: DAG with Parallelism 45
4.2.4.5 Scheduling Delay 49
4.2.5 Applications 50
4.2.5.1 Network Virtualization 51
4.2.5.2 Image Processing: Edge Detection 52
4.2.5.3 EKF: Localization Problem 55

5 Conclusion and Future Work **60**
5.1 Conclusion . . . 60
5.2 Achievements . . . 60
5.3 Future Work . . . 61

List of Figures

2.1 Example DAG 6

3.1 Ataru Architecture 8
3.2 KVM Programming Model 9
3.3 Guest Physical Memory Layout 15
3.4 Guest Virtual Memory Layout 16
3.5 Function Virtual Memory Layout 17
3.6 Example mmap of a Library 18
3.7 Block Diagram of a Typical APIC Implementation 21
3.8 Interrupt Command Register 23
3.9 Descriptor Register 24
3.10 Interrupt Stack Frame With Error and Without Error Code 26
3.11 STAR and LSTAR register description 26
3.12 Scaling Architecture 31

4.1 Experimental Setup for Ataru 33
4.2 Experimental Setup for Firecracker 33
4.3 Measurement Probe Placement for Firecracker (FC) and Ataru 34
4.4 Boot time vs pool_size with Ataru VMM pinned to single core 35
4.5 Cumulative Distribution of Execution Time of a DAG with Concurrent Clients . 37
4.6 SF: DAG with Single Function 39
4.7 SF : Performance Comparison of a DAG with single function 40
4.8 C3: DAG with Chain of 3 Function 41
4.9 C3 : Performance Comparison of a DAG with chain of 3 functions 42
4.10 C8: DAG with Chain of 8 Function 43
4.11 C8 : Performance Comparison of a DAG with chain of 8 functions 44
4.12 C8: Average Scheduler Delay Firecracker vs Ataru 45

4.13 P4: DAG with Parallelism of 4 46
4.14 P4: Performance Comparison of a DAG with Parallelism of 4 47
4.15 Function Execution Times (1vCPU,1pCPU vs 4vCPU,2pCPU) 48
4.16 P4: Firecracker vs Ataru Scheduler Delay 49
4.17 P4: Average Scheduler Delay Firecracker vs Ataru 50
4.18 Representative SFC from a NV Domain 51
4.19 DAG Time to Process 1000 Packets: Firecracker vs Ataru 52
4.20 Sobel Convolutional Kernels 53
4.21 Image Processing: Edge Detection DAG 53
4.22 Edge Detection of 128x128 54
4.23 Edge Detection, Time to Process 1000 128x128 Images 54
4.24 Robot Motion Model 56
4.25 EKF DAG 57
4.26 EKF: Localization Problem 59

List of Tables

3.1 PIC Ports 21
3.2 APIC Registers 22
3.3 Ataru GDT Entry Values 24

4.1 Memory Requirement Comparison 36

Chapter 1

Introduction

1.1 Introduction

Serverless computing has been on the rise with much attention from the research community with the prediction of improved performance [**?**] and deeper penetration into computing. Function as a Service (FaaS) is a popular rendition of serverless computing, the stateless functions execute in response to various types of events, such as HTTP requests, database triggers, or message queues. Traditionally, FaaS frameworks are enabled via Containers or VMs. Developers publish functions in a high-level language and define events that trigger their execution. The functions are then chained to create workflows. The service provider executes the functions inside a pre-packaged Container or VM. Spawning a Container or a VM for an initial request consumes time which is known as the cold start time. However, subsequent requests are serviced from the warm Container or VM, giving a faster response. Functions are dynamically scheduled across its infrastructure and thus exchange data through the network stack, further contributing to the overhead. Empirical studies indicate that platform setup overheads can dominate the function execution time by as much as 100X [**?**]. Applications such as button-click action on a web page, AI voice commands, and Data Analytics are now powered by commercial FaaS service providers. Such tasks being latency insensitive, lend themselves to current FaaS paradigms. However, the latency achieved by today's FaaS platforms is unacceptable for applications with real-time requirements such as remote driving vehicles, serverless robotic fleet management[**?**], serverless UAV swarms [**?**], remote surgeries, Network Virtualization (NV) etc.

With the advent of high-speed networks, 5G, and IoT, can Serverless pave the way for low latency applications on the cloud? Research indicates that there are several benefits for offloading these applications, such as Intellectual Property (IP) protection, lower power consumption, and scalability, [?] [?] in addition to the benefits of serverless computing. To solve this, we implement a lightweight Ataru Runtime and Ataru-KVM VMM (Virtual Machine Monitor) customized for serverless functions. The runtime exposes a minimal process abstraction and has a tiny memory footprint. Ataru executes the sequence of functions within the same worker node and thus allows for data exchange through a shared memory region.

1.2 Motivation

Serverless computing is a paradigm that enables developers to deploy and execute functions in response to events without managing the underlying infrastructure. Function as a Service (FaaS) is a widely adopted implementation of serverless computing that provides a platform for hosting and executing stateless functions. A typical FaaS platform consists of a front end that receives function requests, a scheduler that assigns functions to worker instances, and a pool of worker instances that run the functions. Depending on the platform design and requirements, the worker instances can be realized using different virtualization constructs, such as containers or virtual machines. The choice of virtualization construct affects the performance and resource utilization of the platform, as well as the overheads associated with function instantiation and execution.

One of the challenges of FaaS platforms is to support complex applications that require multiple functions to be executed in a coordinated manner. Such applications can be modelled as Directed Acyclic Graphs (DAGs) of functions, where each function represents a node and each dependency represents an edge. Existing FaaS platforms rely on external services, such as message queues or databases, to realize the DAGs and handle the communication and synchronization among functions. However, this approach introduces additional overheads and involvement of multiple components and network latency.

For applications with low function execution times with requirement of low latency and high throughput, these overheads are prohibitively expensive. To address this, we propose Ataru, a novel FaaS platform that supports DAG-based applications with low overheads and high performance. Ataru leverages lightweight virtualization based on Linux KVM subsystem to realize worker instances that can run multiple functions concurrently. Ataru also implements a novel DAG execution engine that orchestrates the execution of functions within the same worker instance using shared memory. Ataru aims to reduce the overheads of function instantiation and execution and improve the memory utilization and scalability of the platform. We present the design and implementation of Ataru and its evaluation using various benchmarks against an implementation based on process abstraction and semaphore synchronization mechanism in Firecracker, which is the fastest implementation possible to the best of our knowledge while still maintaining process abstraction.

1.3 Related Work

Hall et al. (2019) [?] benchmark the v8 engine isolates against native containers across various workloads. They found that web assembly performed consistently but slower than containers. Shillaker et al. (2020) [?] implement faaslet, based on web assembly. The Faasm functions execute inside faaslet, the applications compiled to web assembly. Web assembly code is interpreted using virtual machine runtime, thus compared to native, there is an overhead causing slowdown up to 1.5-2.5x. [?] [?] [?]. Ataru executes natively, thus avoiding any interpretation overheads.

Pfandzelter et al. (2020) [?] recognize that serverless computing will invariably include resource-constrained edge nodes. They develop tinyFaaS, a lightweight, single-node FaaS platform based on docker containers for edge environments. Oakes et al. (2017) [?] propose a package aware compute platform. Essentially, they suggest that to achieve faster bootup of applications, the large libraries must be quickly loaded or shared among the container instances. In Ataru, the functions and runtime are shared across instantiations since their pages are mmap'ed as Copy On Write (COW) by the VMM. Agache et al. (2020) [?] developed Firecracker, a lightweight VMM specialized for serverless workloads. It uses Linux kernel KVM virtualization infrastructure to provide MicroVM. To achieve lightweight VMM, only the disk, network and serial console are emulated. Manco et al. (2017) [?] developed a lightweight VM via unikernels for applications constructed using Tinyx, a tool to compile a stripped-down Linux VM. They also present LightVM, a virtualization solution based on Xen but modified to achieve faster bootup times. Instead of emulating a full-blown machine, Ataru only emulates a minimal process abstraction required for function execution. Belay, Adam, et al. [?] designed dune, which exploits VT-x technology to provide a process abstraction rather than a machine abstraction. The user processes run as non-root ring 0 code, and dune module catches all the vmexits. The virtualized environment exposed by dune module is one of the process, and the functionality of syscalls is exported as hypercalls. Wanninger, Nicholas C., et al. [?] implement virtines which use a similar idea to the dune sandboxing approach; however, they design the virtual context to be that of a function instead of a process. Ataru uses a similar idea; however, unlike Dune and Virtines, it can exploit parallelism with the knowledge of DAG and is customized for the serverless workload. Kotni, Swaroop, et al. (2021) [?] build Faastlane, which executes functions of a workflow as threads within a single process inside a container; it also allows shared memory architecture across functions belonging to the same task. Faastlane achieves isolation by using MPK (Memory Protection Keys) hardware functionality. Hedayati, Mohammad, et al. [?] propose three methods for compartmentalization of subdomains within an application, namely syscall-based page table switching, using VMFUNC functionality in recent intel processors, and using memory domains(PKU) feature. Authors prefer the MPK-based technique for performance reasons. However, MPK is limited because the domains could be manipulated in user mode by malicious code. Ataru, on the other hand, uses hardware virtualization and thus has superior isolation.

Chapter 2

Programming Model

We first outline the developer's view of the Ataru. A developer has a repository to which he can add functions and shared libraries, which must adhere to the custom syscall interface of the Ataru. An Application is defined by a Directed Acyclic Graph (DAG) describing the sequence of functions to be executed for a trigger. A trigger is an event which causes the execution of the DAG. For example, a trigger might be a packet reception from a specific IP and port. Upon execution completion of the DAG, the Ataru takes an action as defined by the developer. For example, an action might be to send a packet filtered by the DAG to a network interface. A developer provides a DAG, its library dependencies, a trigger, and an action for an application. Data flow across the functions is enabled through a shared memory region mapped to all the functions of the DAG.

2.1 Data Structure for DAG Representation

Fig. 2.1 shows an example for a generic application's DAG.

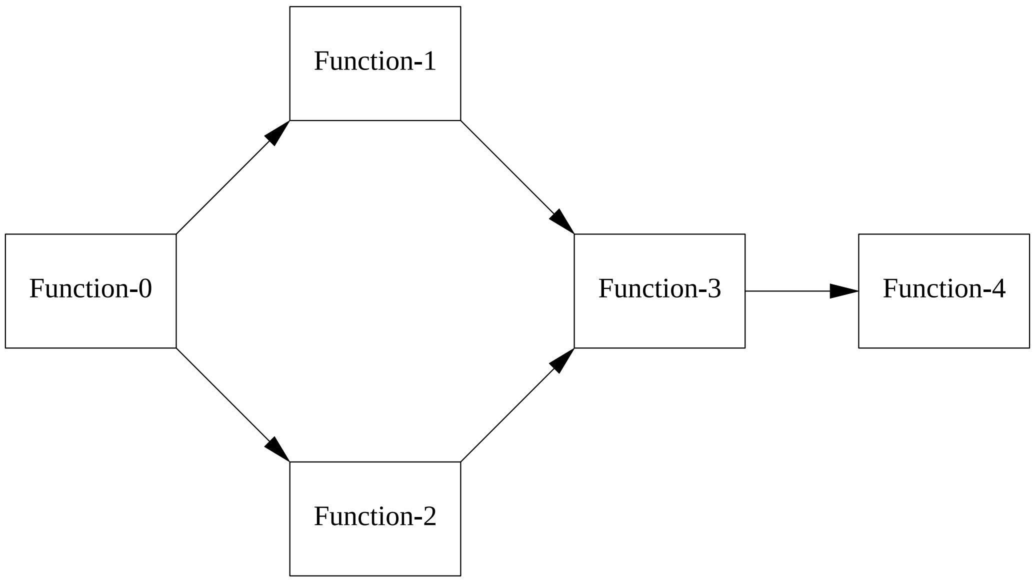


Figure 2.1: Example DAG

We represent Application's DAG by a linear array of integers. Below is the DAG for an application depicted in Fig 2.1.

```
num_nodes: 5
    +-----------------------------------------------------------------+
    | 0 | 1 | 1 | 2 | 1 | 0 | 2 | 3 | 4 | 5 | 5 | 1 | 2 | 3 | 3 | 4 |
    +-----------------------------------------------------------------+
Idx:  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
```

The array index[0. . . 4] stores the in-degree of each node in a graph of 5 nodes, corresponding to different functions. For instance, index[0] holds the in-degree of Function-0, which is zero, and index[1] holds the in-degree of Function-1, which is one. The array index[5. . . 9] stores the starting index of the adjacency list of each node in the graph, with index[10] being the sentinel value for the end of the list of Function-4. For instance, the adjacency list of Function-0 begins at index[11] and ends at index[12], indicating that Function-0 has two outgoing edges to Function-1 and Function-2.

Each DAG accesses a shared memory region for inter-function data transfer. The input and output pointers for each function invocation are derived from the input and output offsets specified by the developer and passed as function arguments. The developer can manipulate the offsets to reuse existing functions in different applications.

As an illustration of how shared memory can be used to facilitate the communication among functions in a DAG, consider the DAG in Fig 2.1 and assume that it represents an image processing pipeline where Function-0 partitions the image into smaller chunks and Function-1 and Function-2 perform operations on these chunks. Function-0 can write these chunks into separate memory segments within the shared memory region that is mapped to the worker instance and supply the offsets of these segments as function arguments to Function-0 and Function-1. This way, Function-1 and Function-2 can access the image chunks directly from the shared memory without requiring any additional data transfer.

Chapter 3

Architecture

Figure 3.1 illustrates the architecture of Ataru. The design of Ataru consists of two major components: Ataru VMM, and Ataru Runtime. Ataru VMM is a customized VMM that leverages the Linux KVM subsystem to set up Ataru Runtime for executing an application. Ataru targets short-duration functions with low latency and high throughput requirements. Thus, the design goal of Ataru is to provide a native serverless framework for the execution of the functions securely in a virtualized environment. Unlike traditional FaaS, where each request requires a re-initialization of the process, Ataru reuses the same process image for all requests in the same workflow. We use DAG and workflow interchangeably throughout this report.

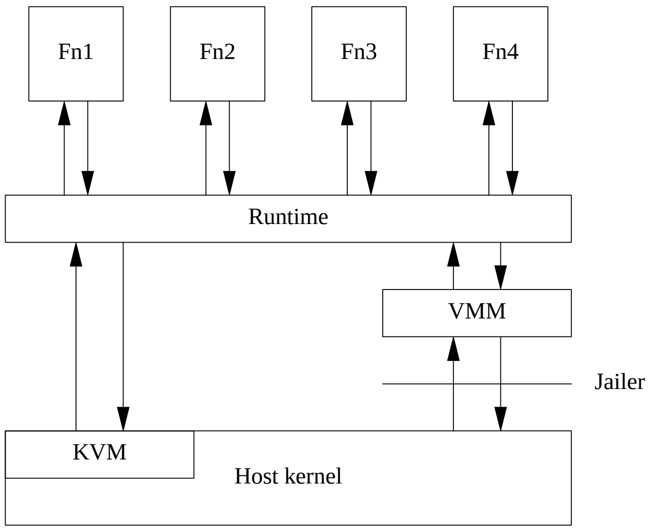


Figure 3.1: Ataru Architecture

3.1 Ataru-KVM VMM

Ataru VMM leverages the Linux KVM subsystem to create a minimal VM for runtime and function execution. Ataru VMM's responsibilities include vCPU, memory, and network management. Fig. 3.2 depicts the Linux KVM model for hypervisors. Each thread within a VMM process is responsible for a single vCPU. Execution of a VM proceeds transparently until an instruction is encountered or an interrupt is generated that requires the host to handle it. Such instructions or interrupts cause a vmexit, which transfers the control from the guest to the host. KVM handles vmexits via two mechanisms, soft vmexit and hard vmexit. A soft vmexit occurs when the guest executes a privileged instruction that the KVM subsystem can emulate without involving the VMM process. A hard vmexit occurs when the guest executes an instruction that requires the intervention of the VMM process, such as accessing an unmapped device or triggering an exception. In this case, the exit reason is transferred to the vCPU thread of the VMM process. The only hard vmexits that Ataru handles are those caused by IO read/write instructions, which are the main communication mechanism between the host and the Ataru Runtime. These instructions are used to transfer data and commands between the host and the guest, such as function requests and responses.

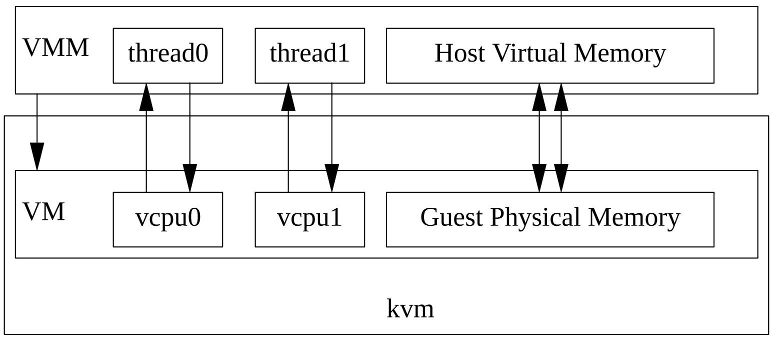


Figure 3.2: KVM Programming Model

Linux KVM Subsystem provides a set of ioctls (input/output control) to configure and query the parameters and state of a VM. KVM API defines ioctls into four categories, system ioctls, VM ioctls, vCPU ioctls and device ioctls. System ioctls are used to create and manage VMs. VM ioctls are used to create and manage vCPUs and memory regions. vCPU ioctls are used to run and control vCPUs. Device ioctls are used to create and manage emulated devices. Ataru-KVM VMM uses these ioctls to construct a VM that can execute Ataru Runtime.

3.1.1 System IOCTLs

These are used to fetch and modify global kvm subsystem attributes. In addition, these ioctls also create virtual machines

Some of the system ioctls used by Ataru are:

1. `KVM_CREATE_VM`: Creates a new VM and returns a VM fd to control the created new virtual machine. Ataru uses it to create a new VM worker instantiation for an application

2. `KVM_GET_VCPU_MMAP_SIZE`: Fetches the size of the shared memory used for the communication of the KVM_RUN ioctl. This shared memory communicates information in the Virtual Machine Control Block (VMCB). Ataru uses it to fetch exit reason, IO read/write size, address and data.

3. `KVM_GET_MSR_INDEX_LIST, KVM_GET_MSR_FEATURE_INDEX_LIST`: Used to fetch the list of supported msrs supported by the KVM. KVM allows for either kernel or VMM emulation of certain msrs or disable them completely. Ataru uses this ioctl to obtain the list of all kernel supported msrs and enable them for the guest them.

3.1.2 VM IOCTLs

These query and set attributes related to a virtual machine such as memory layout, creation of vCPUs and devices

Some of the system ioctls used by Ataru are:

1. `KVM_CREATE_VCPU`: This adds a vCPU to a virtual machine. Using this ioctl Ataru adds required number of vCPUs into the VM.

2. `KVM_CREATE_IRQCHIP`: Creates interrupt model in kernel. On x86 it specifically creates virtual IOAPIC and two PIC(nested). It also make sure all future vCPU have Local APIC. Ataru uses this ioctl to create an irqchip for the guest that enables communication and coordination among the vCPUs within Ataru.

3. `KVM_GET_IRQCHIP, KVM_SET_IRQCHIP`: Used to get/set the state of the kernel interrupt controller creates using `KVM_CREATE_IRQCHIP`.

4. `KVM_SET_USER_MEMORY_REGION`: This ioctl create the user memory region. It essentially creates the mapping from host physical to guest physical page tables. It can support slots number using 16 bit unsigned integer. Ataru uses this ioctl to mmap the runtime, shared memory and functions to guest physical address space.

3.1.3 vCPU IOCTLs

These query and set attributes to change how a single virtual cpu works. You should only use vCPU ioctls from the same thread that made the vCPU

1. `KVM_RUN`: This ioctl is used to run the virtual machine. This launches the Ataru Runtime until a hard vmexit is encountered.

2. `KVM_GET_REGS, KVM_SET_REGS`: Get/Set the general purpose registers of the vCPU. Ataru uses this to set the initial register state of the vCPU . On x86 this includes

```
struct kvm_regs {
        /* out (KVM_GET_REGS) / in (KVM_SET_REGS) */
        __u64 rax, rbx, rcx, rdx;
        __u64 rsi, rdi, rsp, rbp;
        __u64 r8,  r9,  r10, r11;
        __u64 r12, r13, r14, r15;
        __u64 rip, rflags;
};
```

3. `KVM_GET_SREGS, KVM_SET_SREGS`: Get/Set the special registers of the vCPU. Ataru uses this to set the segment registers. On x86 this includes

```
struct kvm_sregs {
        struct kvm_segment cs, ds, es, fs, gs, ss;
        struct kvm_segment tr, ldt;
        struct kvm_dtable gdt, idt;
        __u64 cr0, cr2, cr3, cr4, cr8;
        __u64 efer;
        __u64 apic_base;
        __u64 interrupt_bitmap[(KVM_NR_INTERRUPTS + 63) / 64];
  };
```

3.1.4 Device IOCTLs

These query and set attributes to change how a single device works. device ioctls are used from the same process (address space) that made the VM. Ataru does not emulate device and thus does not use these ioctls

3.1.5 Initialization Sequence

We enumerate the steps involved in the initialization and setup of Ataru. Ataru has to load the runtime, get the information about the DAG, locate and load the functions, request VM and required vCPUs and launch them.

1. `Setting up of Runtime`: Runtime is independent of the DAG it processes; thus, the runtime can be preloaded, and vCPUs can boot before any application is mapped onto it. The first page of the runtime contains metadata, which will be loaded with information about the DAG to be loaded later

2. `Setting up of Functions`: Once the runtime has been mmap'ed in the host address space, next the step is to get the DAG to execute. Once DAG has been supplied by the user, Ataru forks, and the parent looks for a new DAG requests, while the child starts loading the functions into host address space. During loading, Ataru also generates a list of library dependencies required for each of the functions.

3. `Resolving Dynamic Symbols`: Once all the library dependencies have been enumerated for each of the functions in a DAG, the position of the libraries within the virtual address space is fixed following which `ELF64_Rela` is enumerated on to get all the symbols to be resolved. Each symbol is searched linearly in the dependency list. If the symbol is found, it is then resolved.

4. `Register the Runtime` Once we have the runtime and function mmap'ed to host address space, it is time to register them to the guest's physical memory. First, we register the runtime into the guest's physical memory. Registration of the functions and DAG Initialization in the guest's physical memory is done parallelly to the boot process since both tasks are independent of each other, this allows us to optimize boot-up time.

5. `Setup vCPUs`: We request the required number of vCPUs and start the vCPUs from the entry point of the runtime. Initially, vCPU0 acts as a Bootstrap processor; eventually, other processors are sent Startup Inter-Processor Interrupt (SIPI) to start their boot process.

6. `Register the Functions`: Functions are registered to the guest's physical memory, and page tables are set up for each function. This is done parallely to boot process as explained earlier.

7. `Run the VM`: Once all the processors are booted and functions are registered, Ataru waits for the request from the client.

3.1.6 vCPU Management

Ataru VMM spawns a thread for each vCPU. Each vCPU is requested from the KVM and, by default, is in real mode. The power-on values for the vCPU registers are modified to inform the vCPU about its stack address, cpuid and the total number of vCPUs (`pool_size`) assigned to the DAG. The Ataru VMM only handles IO vmexits which is used as a communication mechanism between VMM and the runtime.

Following ports are used:

1. `0xfe`:This port enables syscall interception. Ataru can control the execution of system calls invoked by the functions to runtime. It is 16-bit port, allowing up to 65536 system calls to be handled. Currently Ataru-KVM VMM intercepts syscall number `0x0` to `0x200` thus it is possible to support a posix like syscall ABI in the future.

2. `0x3f8`: This is used for emulation of a serial port. Ataru Runtime uses it to have a printf support from the Ataru-KVM VMM.

3. `0x3f9`: Used by Runtime for signalling to Ataru-KVM VMM that it has booted and that it is waiting for the mapping of the functions

4. `0x3fa`: Used to halt the vCPU, used for debugging purposes

5. `0x3fb`: Used for printing regs at a certain instruction pointer in a program. Used for debugging purposes.

6. `0x3fc`: Used for fetching the cpu id for the vCPU. vCPU can use it to fetch its cpu id from the KVM VMM.

7. `0x3fe`: Ataru Runtime waits for the client request by writing an identifier and waiting on this port. Other identifiers are used for other auxillary messages.

3.1.7 ELF Format and Memory Image

Ataru uses a custom loader to load the program. To facilitate loading, the program is linked using a custom linker script. ELF (Executable and Linkable Format) defines format for program executable. The ELF file type is expressed by the first 4 bytes specifically 0x7f, 'E', 'L', 'F' at the start of the file. ELF file consists of Elf Header, Program Header and Section Headers.

`ELF Header`: This defines the number, size and location of program and section headers.

`Program Header`: This defines the segments of program along with their attributes such as read, write and executable permissions

`Section Header`: This defines logical partitioning of the program, string tables, symbol tables, relocation tables etc.

Ataru reads the executable and generates a memory image adhering to memory layout rules of Ataru. This memory image is then mmap'ed into guest as part of the DAG.

3.1.8 Memory Layout

Ataru uses 3 levels of page table addressing. Thus, it uses 2 MB page sizes for the guest virtual address space. Ataru uses 2 MB pages for two reasons: First, this allows for a smaller memory footprint of the page tables. Second, the host OS can use huge pages wherever possible in the nested page table; this reduces the number of vmexits due to page faults and leads to a gain in performance[**?**].

To save time spent on copying input to the process's memory, the Inter-Process Communication (IPC) occurs via a shared memory region allocated per workflow, and the the region is mapped to all functions at a predefined address during initialization.

The bootup time of a VM/Process is directly proportional to its image size[**?**]; thus low memory footprint is desirable. Ataru Runtime achieves a tiny memory footprint (<1 MB) by exporting the memory management responsibility to Ataru-KVM VMM and exploiting host kernel capabilities wherever possible. During initialization, the VMM mmap's the runtime and function memory images as a private Copy On Write (COW) pages; thus, each new instantiation shares the runtime and function pages; this reduces bootup time as well as elegantly solves the memory duplication problem which occurs in the traditional VMs.

Since the runtime's image is mmap'ed as private COW, the memory overhead incurred with scaling is extremely low. The runtime and functions are mmap'ed before any request is received; this reduces the overhead required to load pages during bootup. The VM and previously mmap'ed memory of runtime and functions are registered to guest physical memory. Following this, the control is transferred to the runtime for booting itself with the required number of vCPUs.

Fig. 3.3 shows the guest physical memory layout. The first 4 KB is dedicated to the metadata related to the DAG. After that, memory till the 4 MB boundary is reserved for runtime and page tables. Following this boundary, the functions and libraries are loaded. It is to be noted that even though we reserved 4 MB area for runtime, actual host physical memory allocation may not be 4 MB as host physical memory is lazily allocated in Linux.

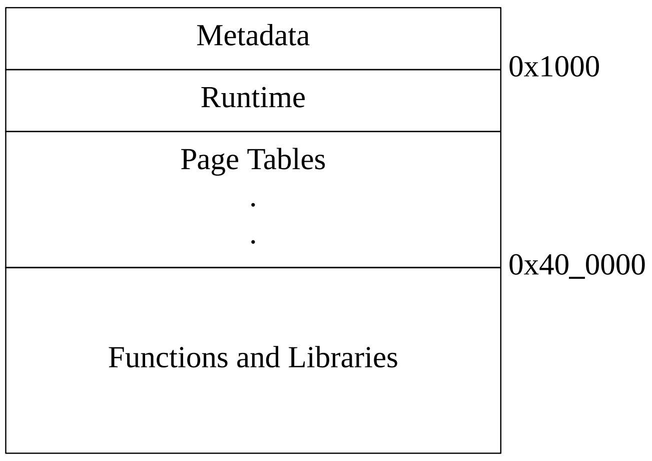


Figure 3.3: Guest Physical Memory Layout

Each function has its own virtual address space, therefore, its own page tables. Fig. 3.4 shows a virtual address space for a function. Each P3E entry spans 1 GB of memory. `0x0000_0000` to `0x4000_0000` is mapped as a 1 GB huge page for the runtime. `0x8000_0000` to `0xC000_0000` is reserved for function. `0xC000_0000` to `0x0001_0000_0000` is reserved for APIC. APIC-related registers, which are required to enable Inter Processor Interrupt (IPI) the mechanism are located in this region. The remaining P3E entries are reserved for libraries, one per each.

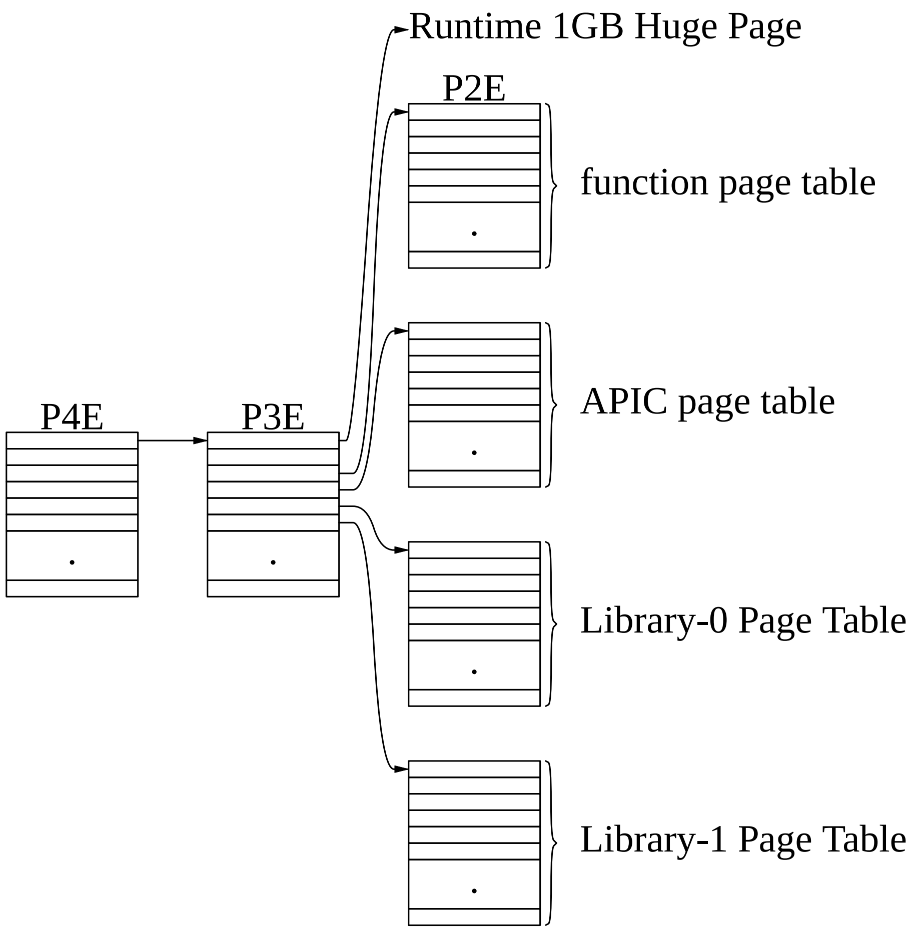


Figure 3.4: Guest Virtual Memory Layout

Fig. 3.5 shows the virtual memory layout spawned by the function page table as shown in Fig. 3.4. The shared memory region starts from `0x8000_0000`, which is used for IPC. The shared code page is a common code page automatically mapped to each function by the runtime at `0x8020_0000`. This shared code page is similar in functionality to Linux virtual dynamic shared object (vDSO). Following this from `0x8040_0000` to 1 GB boundary, the function provided by the developer, is mapped.

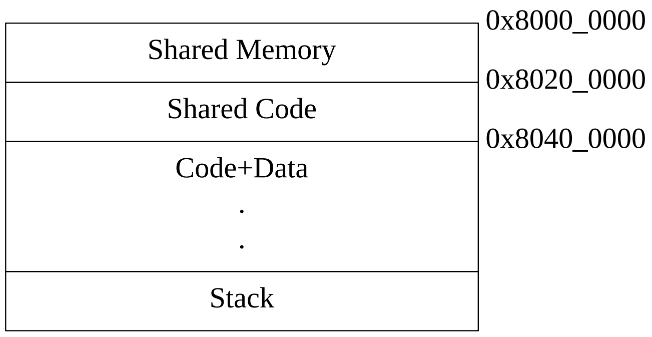


Figure 3.5: Function Virtual Memory Layout

3.1.9 Dynamic Linking and Loading

Ataru uses a custom loader to load functions and libraries. After the user-provided function is compiled, a disk image of the function's memory image is generated. During the Ataru VMM initialization sequence, the loader mmaps this disk image. It reads the header fields from the elf file used for dynamic linking and discovering the entry and stack address values. To facilitate dynamic linking loader uses a two-pass method. In the first pass, the loader recursively finds the shared library dependencies from the `.dynamic` section and assigns each library a `P3E` entry index so that the virtual address of the library is known. In the second pass, the loader resolves the symbols for all the libraries discovered. Fig 3.4 shows the library entries starting from p3e[4] onwards.

The relevant sections for dynamic linking are briefly described below:

`.dynstr`: This section holds strings for the name of the dynamic symbols

`.dynsym`: This section contains the dynamic linking symbol table.

`.rela.dyn`: This section holds the information for all the symbols that need to be resolved

`.dynamic`: This holds information for dynamic linking such as dependent libraries.

Utilizing the information in these sections allows the loader to resolve the dynamic symbols. The `r_info` field in the `Elf64_Rela` structure inside `.rela.dyn` section describes the type of linking to be performed. Ataru supports only 64 bit executable hence the relevant `r_info` type are `R_X86_64_JUMP_SLOT`, `R_X86_64_COPY`, `R_X86_64_64` , `R_X86_64_RELATIVE`, and `R_X86_64_GLOB_DAT`. These types correspond to different kinds of relocations, such as jump table entries, copy relocations, absolute addresses, relative addresses, and global data references.

An interesting feature required to be supported was the behaviour of global variables in a shared library. This behaviour is different across Operating Systems. Linux provides a separate instance of the global variable to each virtual address space it is mapped. Linux implements a separate instance of the global variable for each virtual address space where it is mapped.

Moreover, if a global variable `x` is declared in both the dependent shared library and the main program, then Linux will assign the same address for both.

External libraries can enable various functionalities, such as custom memory management, ramfs etc. However, these libraries must adhere to Ataru's constraint that a library can only access 1 GB of virtual memory since only one p3e entry is allocated per library. Since Ataru focuses on functions with low execution time, it can be reasonably assumed that any such function will also have low code size and dependency on libraries with small footprints; thus, in a real-world scenario, this constraint will not be prohibitive.

Each function maps the required libraries separately in guest physical memory, obviating any need for Runtime to handle COW. This does not affect the host's physical memory as the libraries are ultimately mapped to a single library in the host. Fig 3.6 illustrates this mapping of libraries from host physical to guest virtual.

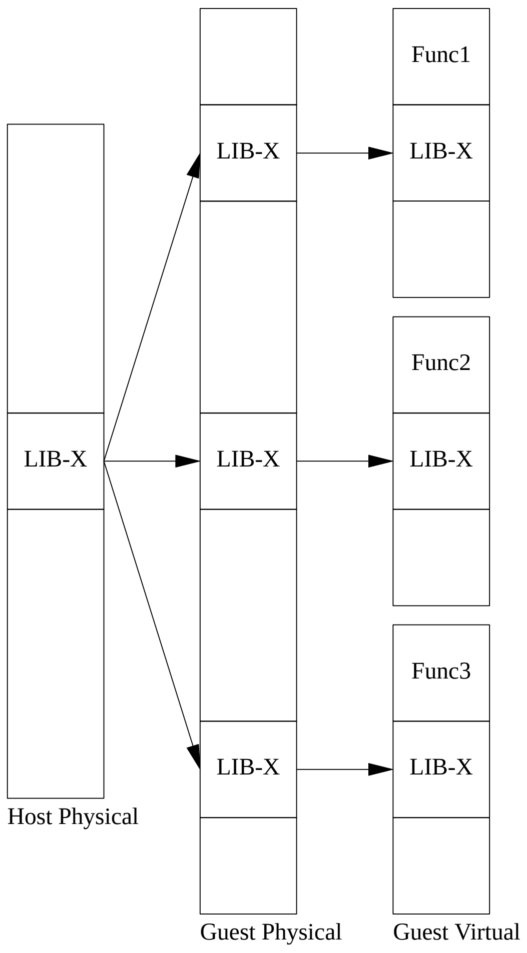


Figure 3.6: Example mmap of a Library

3.2 Ataru Runtime

Ataru runtime is designed to boot multiple vCPUs into a minimal processor state to support the execution of functions in separate virtual address spaces. A vCPU provided by the KVM is in real mode at startup. Therefore, it must be booted to long mode along with support for interrupts and exceptions, syscalls, paging, and IPI. The Ataru Runtime scheduler waits for the client requests once all the vCPUs for a DAG boot.

3.2.1 Changing Processor Mode

In 16-bit real mode, which is the processor's state after a RESET or INIT, the code-segment base-address is obtained by multiplying the CS-selector value by 16. This base address is then added to the Instruction Pointer (EIP) value to get the physical address in memory. Therefore, the processor can only access the first 1 Mbyte of memory in real mode.

Before running the newly created VM we initialize the `CS.selector` and `CS.base` as zero, and the general purpose registers are initialized as follows:

```
struct kvm_regs regs = {
    .rip = ENTRY_POINT,
    .rax = 2,
    .rbx = 2,
    .rcx = CPU_ID,
    .rsp = STACK_START,
    .rdi = STACK_START,
    .rsi = 0,
    .rflags = 0x2,
    .r10 = CPU_ID,
    .r11 = POOL_SIZE
};
```

The instruction pointer (`rip`) is set to the entry point of the Ataru Runtime, which is `0x2000`. The register `rcx` and `r10` hold the cpu id of the current vCPU, while the register r11 store the `pool size`, i.e. the total number of vCPUs in the VM. This is because Ataru boots multiple vCPUs.

The first task is changing processor mode from real to long mode. To achieve this following sequence is performed:

1. Clear the interrupt bit in the EFLAGS using `cli` instruction

2. Set the Physical Address Extension (PAE) and Page Global Enabled (PGE) bit in the cr4 register.
    - PAE: Enables virtual address translation upto 52 Bits to physical addresses
    - PGE: If set, address translations may be shared between address spaces
3. Update the cr3 register by pointing it to top level page table
4. Update Extended Feature Enable Register (EFER) MSR by setting Long Mode Enable (LME), No-Execute Enable (NXE) and System Call Extensions (SCE) bits
    - LME: Enables the processor to activate long mode. Long mode is not activated until software enables paging some time later
    - NXE: Enables the no-execute page protection feature
    - SCE: Enables the low latency `SYSCALL` and `SYSRET` system call instructions
5. Enter the compatibility mode by enabling paging and protected mode in the cr0 register
    - Protected Mode Enable (PE): Changes mode to protected mode
    - Paging (PG): Enables the paging
6. Update the Global Descriptor Table (GDT) register using `lgdt` instruction with the address of the descriptor table
7. Perform a long jump to the 64 bit code

3.2.2 Stack Assignment

The MSR KernelGSbase (`0xC0000102`) register stores the stack address of each vCPU, which is 4 KB in size and depends on the cpu id. The `swapgs` instruction is used to save the stack address in the register and to load it when switching from user mode to kernel mode during system calls. The stack area is pre allocated with a size of $64 * 4$ KB, which is the maximum number of vCPUs that Ataru can boot.

3.2.3 IRQ Chip Configuration

The `KVM_CREATE_IRQCHIP` ioctl creates an interrupt controller model in kernel, creating a Programmable Interrupt Controller (PIC) and IO Advanced Programmable Interrupt Controller

(IOAPIC). It also enables the Local APIC for the vCPUs. The APIC provides interrupt support and the local APIC accepts interrupts from the system and delivers to destination Local APIC of the core. Fig 3.7 shows the typical APIC implementation in x86.

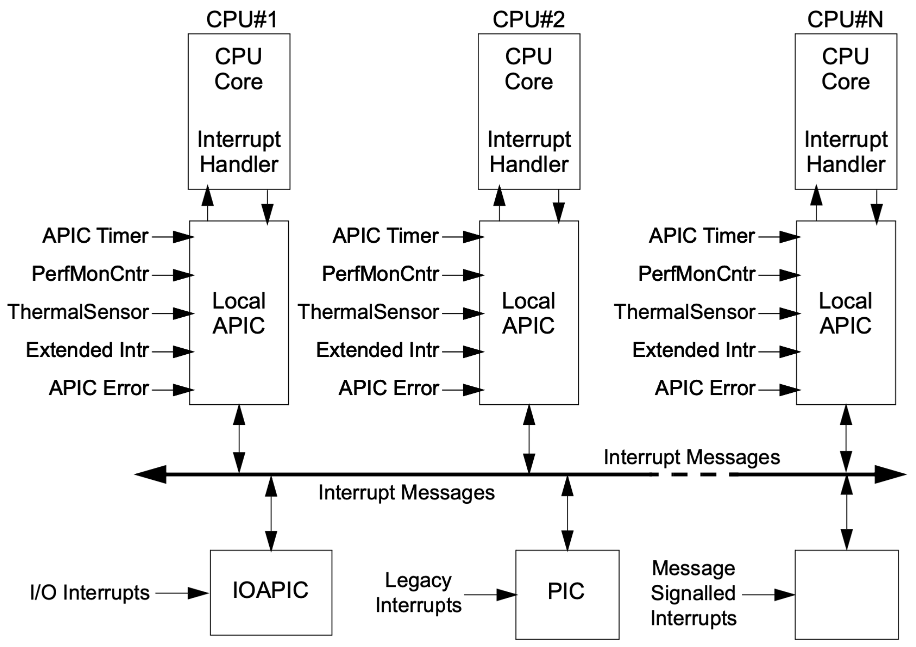


Figure 3.7: Block Diagram of a Typical APIC Implementation

Since Ataru does not use any external interrupt we disable PIC in the startup process.

There are two nested PIC which follow master slave architecture. PIC has a command and a data ports for its control. Table 3.3 lists the ports used in initialization of the PIC.

Description	Port Address
Command (Master PIC)	0x0020
Data (Master PIC)	0x0021
Command (Slave PIC)	0x00A0
Data (Slave PIC)	0x00A1

Table 3.1: PIC Ports

Procedure to disable PIC:

1. Start the initialization mode by writing `0x11` to command port of both master and slave

2. Remap the interrupts by writing `0xe0` for master and `0xe8` for slave to data port
3. Intimate master that there is a slave PIC at IRQ2 by writing `0x4` to master data port and `0x2` at slave data port
4. Configure PIC to use 8086 mode by writing `0x01` to the data port of both master and slave

After disabling PIC, next step is to enable Local APIC. Table 3.2 lists the registers used to configure APIC

Short Name	Description	Offset from 0xFEE0_0000
APIC_ID_REG	APIC ID Register	0x20
APIC_TPR_REG	Task Priority Register	0x80
APIC_EOI_REG	End of Interrupt Register	0xB0
APIC_LDR_REG	Logical Destination Register	0xD0
APIC_DFR_REG	Destination Format Register	0xE0
APIC_SPUR_INT_REG	Spurious Interrupt Vector Register	0xF0
APIC_ESR_REG	Error Status Register	0x280
APIC_ICR_LREG	Interrupt Command Register Low	0x300
APIC_ICR_HREG	Interrupt Command Register High	0x310

Table 3.2: APIC Registers

Procedure to enable APIC:

1. Write zero the APIC_ESR_REG to clear any prior errors
2. Enable Local APIC by writing 1 to bit position 8 of the APIC_SPUR_INT_REG
3. Write the `0xFF` in the vector field, which contains the vector that is sent to the CPU core in the event of a spurious interrupt
4. Disable Logical Addressing of the core by writing zero to APIC_LDR_REG
5. Enable flat model by writing 0xFFFF_FFFF to APIC_DFR_REG
6. Set the APIC_TPR_REG value to be zero to allow all the interrupts
7. Reset the interrupt bits by writing zero to APIC_EOI_REG

3.2.4 Booting Application Processors

Once the Boot Strap Processor (BSP) has booted, it must start the Application Processors (AP). This is achieved by the Startup Inter Processor Interrupt (IPI). Fig 3.8 gives the bit description of the APIC_ICR_LREG register.

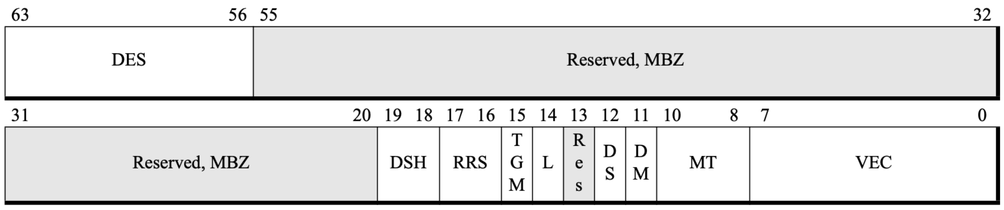


Figure 3.8: Interrupt Command Register

Relevant bit description for the register are as follows:

- VEC (7:0): Vector Address
- Message Type (MT, 10:8): Specifies the message type send to the processor
- Destination Mode (DM, 11): Logical or Physical destination
- Delivery Status (DS,12): Indicates interrupt delivery status
- Level (L,14): Assert/Deassert
- Trigger Mode (TGM,15): Edge/Level triggered
- Remote Read Status (RRS,17:16): Current remote read status
- Destination Shorthand (DSH,19:18): Whether shorthand notation is used

The SIPI requires a preceding INIT IPI. The INIT IPI is sent by writing `0xC4500` to the interrupt command register (ICR), and the SIPI is sent by writing `0xC0602` to the same register. The SIPI contains the page number of the entry point for the AP in the first 8 bits. Since the entry point is `0x2000`, the first 8 bits are `0x02` for the SIPI.

3.2.5 Descriptor Tables

3.2.5.1 Global Descriptor Table

System software running in protected mode must set up a global descriptor table (GDT). The GDT has entries for code segments and data segments for both kernel and user modes. The GDT can also have entries for gates and other system segments. System software can place the GDT anywhere in memory and should prevent non-privileged software from modifying the segment that contains the GDT.

GDT address is loaded into GDTR using instruction `lgdt` and passing the address of the location pointing to GDT along with the descriptor limit. Fig 3.9 specifies the format of the descriptor register.

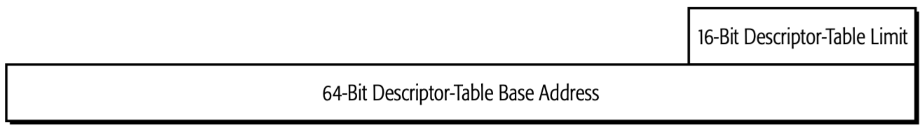

16-Bit Descriptor-Table Limit
64-Bit Descriptor-Table Base Address

Figure 3.9: Descriptor Register

The GDT must have a NULL descriptor as its first entry. It must also contain code-segment and data-segment descriptor entries for privilege levels 0 and 3. Additionally, it can hold other system-segment descriptor entries.

Entry Description	Value
Null Descriptor	0000_0000_0000_0000
Kernel Code Descriptor	00AF_9900_0000_FFFF
Kernel Data Descriptor	00CF_9300_0000_FFFF
User Data Descriptor	0000_F200_0000_0000
User Code Descriptor	0020_F800_0000_0000
Task State Segment	0000_8B00_7780_0067

Table 3.3: Ataru GDT Entry Values

3.2.5.2 Interrupt Descriptor Table

System software uses the interrupt descriptor table (IDT) to control how interrupts and exceptions are handled. System software chooses a specific IDT by loading a pointer to the IDT into the interrupt descriptor table register (IDTR). Like the GDT and Local Descriptor Table (LDT), system software can put the IDT anywhere in memory and should keep non-privileged software from changing the segment that has the IDT. The IDT can only have these kinds of gate descriptors: Interrupt gates, Trap gates, and Task gates. Section 3.2.6 elaborates on the IDT table for Ataru

3.2.6 Interrupts and Exceptions

When an exception or interrupt happens, the processor stops the current program and switches to a system-software routine that deals with the event. These routines are called exception handlers and interrupt handlers, or event handlers for short. Usually, the event handler can handle the interrupt without affecting the current program. The processor saves the return pointer of the current program when it switches to the event handler. The event handler also saves the state of the current program so that the processor can resume it after the event is handled. The processor uses the interrupt vector number to find the right entry in the IDT.

Exceptions and interrupts have three main sources:

- Exceptions are caused by software execution errors or other internal-processor errors. Exceptions are synchronous events because they are directly related to the instruction that was interrupted.

- Software interrupts are caused by executing interrupt instructions. Software interrupts are different from exceptions and external interrupts because they allow intentional activation of the interrupt-handling mechanism. Software interrupts are also synchronous events.

- External interrupts are caused by system logic in response to an error or some other event outside the processor. They are reported to the processor through external signals over the processor bus. External interrupts are asynchronous events that happen independently of the instruction that was interrupted.

In long mode, the processor uses the interrupt vector number times 16 as an offset to find the right entry in the IDT. The entry has a gate descriptor that has a segment-selector and a 64-bit segment-offset field. The gate descriptor's segment-offset field has the full virtual address of the interrupt handler. The gate descriptor's segment-selector field points to the code-segment descriptor for the target in either the GDT or LDT.

Ataru populates the descriptor table with gates referencing a shared interrupt handler. This handler normalizes the interrupt stack frame 3.10 by pushing a dummy error code for interrupts that lack one. It also pushes the interrupt vector number onto the stack frame for transferring control to the appropriate interrupt handler later.

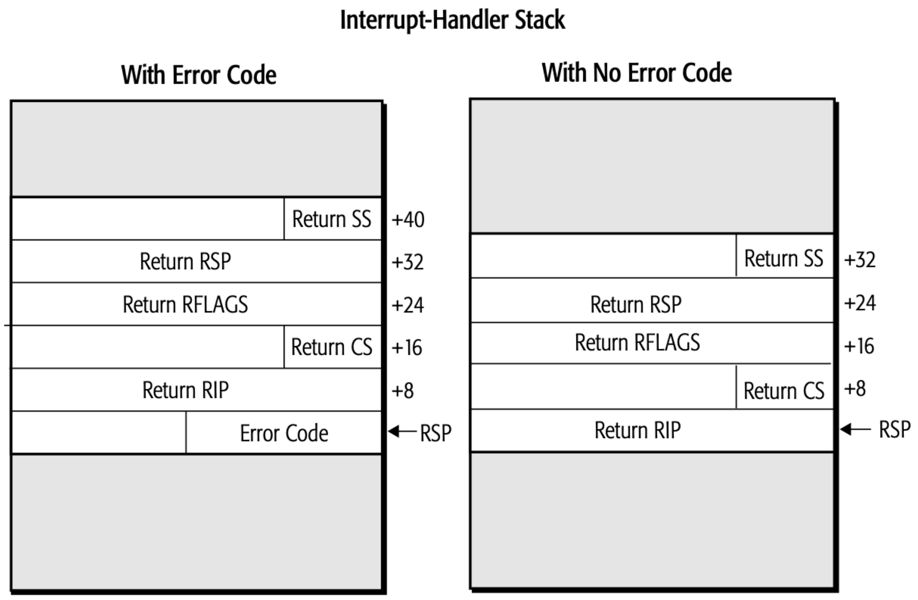


Figure 3.10: Interrupt Stack Frame With Error and Without Error Code

3.2.7 System Calls

The STAR and LSTAR MSR determine the target address of a `syscall` instruction and the CS and SS selectors of the procedures that are called and returned on `sysret`. Fig 3.11 shows the register description for the above MSRs.

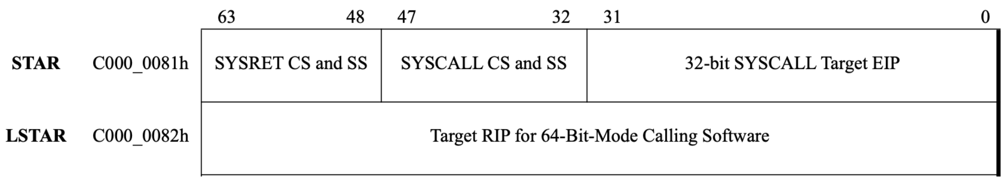


Figure 3.11: STAR and LSTAR register description

Ataru employs a common entry point for all syscalls. Upon syscall, `rip` is stored in `rcx`, and `r11` contains the `rflags`. Ataru redirects syscalls `0x0` to `0x200` to Ataru-KVM VMM, where they are handled individually. Similar to the Unix and Linux syscall interface convention, Ataru supports a maximum of 6 function parameters in registers `rdi`, `rsi`, `rcx`, `rdx`, `r8` and `r9`.

3.2.8 Bootup Sequence

The vCPU starts execution from `0x2000` program counter in real mode. It sets up the table in the `gdt`, then transitions from real mode to compatibility mode with paging enabled and then to long mode.

In long mode, it then sets up the other relevant data segment registers and enables the floating point and sse instructions. To support IPI, APIC is enabled. Interrupts and exceptions are enabled through the setting up of the IDT. Subsequently, APs are sent a SIPI signal to start their boot process.

The runtime defines one TSS segment and user level `GDT` table values to enable 64-bit user mode transition. To support `syscall` and `sysret` instructions, `STAR` and `LSTAR` registers are programmed.

Once all the vCPUs have booted, they wait on a port for input from the Ataru-KVM VMM.

3.2.9 Scheduler

Ataru Runtime's scheduler manages multiple vCPUs and orchestrates the function execution as the DAG dictates, optimally exploiting the parallelism. It is also responsible for halting and waking up the vCPUs as necessary.

The scheduler has the following responsibilities

- Scheduler has to be executed by multiple vCPUs to fetch the work to be done next
- If there is more work to be done than the number of free vCPUs, it must wake any available halted vCPU
- If there is no work to be done, it must halt itself
- As soon as DAG is finished, inform VMM about completion and look for new work

3.2.9.1 LHP-LWP Problem

Since the scheduler is executed by multiple vCPU, it is natural to implement them using locks. However, this may lead to LHP-LWP (Lock Holder Preemption and Lock Waiter Preemption) Problem. It occurs in a virtualized environment using spinlocks. The basic idea of a spinlock is that both the thread that acquires the lock and the threads that wait for the lock are not interrupted by other processes. However, in a virtualized environment, these threads run on top of virtual CPUs (vCPU) that can be paused by the hypervisor at any moment, causing the threads that wait for the lock on other vCPUs to spin and waste CPU cycles. This results in the well-known Lock Holder Preemption (LHP) and Lock Waiter Preemption (LWP) problem.

For example, let vCPU1 hold a lock for a scheduler data structure in order to fetch new work to be done, however just after fetching the lock vCPU1 may be interrupted, and the hypervisor may schedule vCPU2. When vCPU2 attempts to lock the same scheduler data structure, it will fail to do so and may waste CPU cycles by needlessly doing spinlock wait.

The solution employed in today's virtualized system is by using **pause intercept filtering**. The VMCB has a 16-bit value called PAUSE Filter Count. When VMRUN is executed, this value is copied into an internal counter. Every time a PAUSE instruction is executed, this counter is reduced by one until it becomes zero, and then a VMEXIT is triggered if the PAUSE intercept is turned on. If the PAUSE Filter Count is zero, and PAUSE Intercept is turned on, every PAUSE instruction will trigger a VMEXIT. There is a feature of **Advanced Pause Filtering** on newer processors in which when VMRUN is executed, and the PAUSE count value from the VMCB is copied into an internal counter. Then, for each PAUSE instruction, the hardware compares the number of cycles that have passed since the last PAUSE instruction with the PAUSE Filter Threshold. If the number of cycles is more than the PAUSE Filter Threshold, then the internal pause count is reset from the VMCB, and execution goes on. If the number of cycles is less than the PAUSE Filter Threshold, then the internal pause count is reduced by one. If the count value is below zero and the PAUSE intercept is turned on, a VMEXIT is triggered.

Thus by using **Advanced Pause Filtering**, the hypervisor is able to detect spin loops by the vmexits generated by the spinloops wasting CPU cycles. The scheduler lacks the information about which vCPU holds the lock, so it schedules another vCPU upon vmexit, hoping that it may be the lock holder; otherwise, it will also trigger vmexit, and the scheduler will switch to another vCPU.

3.2.9.2 Lock Free Data Structure

The solution employed by Ataru is to utilize lock-free data structures and to halt and wake the vCPUs as and when required in the course of execution of the DAG. Lock-free algorithms are non-blocking algorithms in which multiple entities eventually complete the operation without causing deadlock.

Ataru needs to keep track of halted vCPUs. However, it is impossible to HLT a vCPU and simultaneously update a data structure without adding a race condition. Specifically, the following snippet shows the problem.

```
   // rax has the cpuid of the vCPU
   // rdx is the memory location for book keeping
1: btsq rax, (rdx) // bit test and set
2: hlt             // HALT
```

It is possible that vCPU-1 is preempted after executing instruction-1; thus, it may falsely indicate that vCPU-1 is halted. Therefore, the kick is wasted if vCPU-2 kicks vCPU-1 while it is still running. Consequently, when vCPU-1 resumes execution, it may execute instruction-2 and halt indefinitely.

Ataru adopts a recovery-based solution. That is, the Ataru scheduler periodically verifies if the kicked vCPU has resumed execution, and if not, it re-kicks it.

3.2.9.3 Double CAS

The bitmap with the set bit of inactive vCPU is a data structure that requires concurrent access. The following technique is employed to achieve lock-free access to this bitmap which utilizes the Double CAS instruction. The operation of the double CAS can be expressed by

```
double DCAS(dst[2], rdx_rax[2], rcx_rbx[2]) {
  if (dst[0..1] == rdx_rax[0..1]) {
     dst[0..1]= rcx_rbx[0..1];
     ZF = 1;
  }
  else {
     ZF = 0;
     rdx_rax[0..1]= dst[0..1];
  }
}
```

The following trick involving DCAS allows a vCPU to fetch a cpuid and perform operations without worrying about halting progress if it is interrupted.

```
// e_bsf: extrack bit by bit scan forward
// bm_inactive_vCPUs: 64 bit bit map signifying halted vCPUs.
reflective_bm[0] = bm_inactive_vCPUs;
reflective_bm[1] = reflective_bm[0];
DCAS(&ss_reflective_bm, zero, reflective_bm);
 if((cpuid = e_bsf(&ss_reflective_bm[0])) >= 0) {
```

```
 // Perform task with cpuid
 reset_bit(cpuid, ss_reflective_bm[1])
}
```

Below are the steps involved in the scheduler algorithm. The $InCount$ is an array of integers where $InCount[x]$ represents the number of inbound edges to a function x.

1. If previously this cpu was executing a function y, reduce the $InCount[x]$ by 1 for all x dependent on y
2. For all x, for whom $InCount[x]$ is zero, add x into pending_work
3. Get new x from pending_work, wake up vCPUs equal to number of elements in pending_work and execute x
4. If the dag is finished, wait for next input else if there is no pending work, halt.

3.3 Scaling using Ataru

There is a proto process that has mmap'ed the Ataru Runtime as private COW. This proto-process forks for each new application. The newly forked process loads the corresponding DAG for the application. Moreover, new instantiations of the application are forked to perform horizontal scaling. Since the runtime and functions are already loaded in the memory, the cold start time can be reduced significantly. Fig 3.12 depicts the scaling architecture for Ataru. This figure shows two apps, namely `APP1` and `APP2`. `APP1` has 3 instantiations and `APP2` has 2 instantiations.

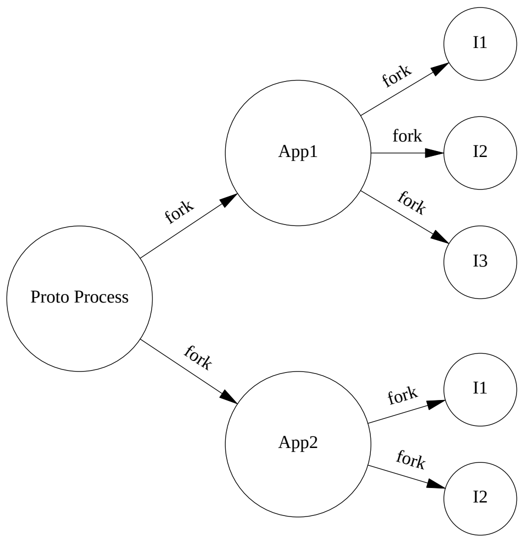


Figure 3.12: Scaling Architecture

Since the memory management in Ataru is performed by the Ataru-KVM VMM, the libraries and functions can be shared across the applications by mapping them to the same physical addresses. This can provide considerable benefit in memory utilization by avoiding duplication of code. Moreover, since the cache is tagged with physical address, shared code can also improve cache utilization by increasing cache hits and reducing cache misses.

Chapter 4

Experiments and Results

4.1 Experimental Setup

The experimental setup consists of an AMD Ryzen 9 5950X processor with 64 GB of RAM and Simultaneous Multi-Threading (SMT) disabled. Four cores (12 to 15) were isolated from the Linux scheduler using the isolcpus boot parameter, and their frequency was fixed to 2.8 GHz using the files in scaling files in `/sys/bus/cpu/devices/cpu12..15/cpufreq/`. The experiments were executed on these isolated cores. Firecracker was configured with a Linux kernel that was compiled with the recommended kernel configuration [?] options. Traffic was generated from a distinct machine that has a 1Gbps connection to the primary machine.

Procedure to isolate cores 12 to 15:

```
$ cat /etc/default/grub
...
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=domain,12,13,14,15"
...
$ sudo update-grub
```

Procedure to fix the frequency of the cores:

```
$ echo "2800000" > /sys/bus/cpu/devices/cpu{12..15}/cpufreq/scaling_min_freq
$ echo "2800000" > /sys/bus/cpu/devices/cpu{12..15}/cpufreq/scaling_max_freq
```

Fig 4.1 shows the experimental setup of Ataru, Machine `M2` is the traffic generator with a 1 Gbps link to machine `M1` Berkeley Packet Filter (BPF) Filter is used to filter out the packets only meant for Ataru. The packets are fetched by the Ataru-KVM VMM and transferred to the Ataru runtime via shared memory.

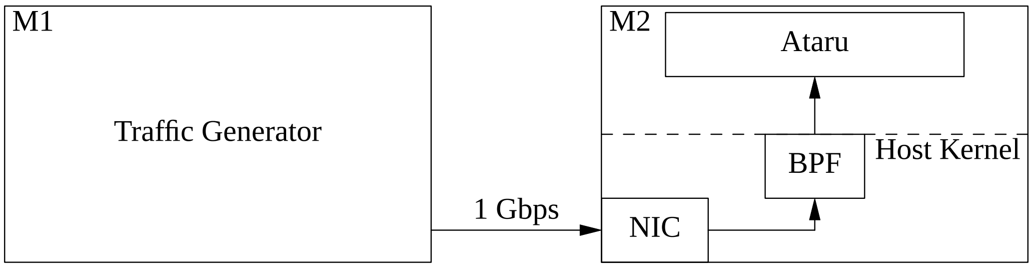


Figure 4.1: Experimental Setup for Ataru

Fig 4.2 shows the experimental setup of Firecracker, Machine `M2` is the traffic generator with a 1 Gbps link. A `TAP` filter is used to provide a Virtual NIC (vNIC) to Firecracker.

Procedure to vNIC capability by adding bridge and tap interface for firecracker:

```
In Host:
$ ip link add br0 type bridge
$ ip link set enp7s0 master br0
$ ip tuntap add tap_firecracker mode tap user ‘whoami‘
$ ip link set tap_firecracker up
$ ip link set tap_firecracker master br0

In Guest:
$ ip route add default via {{GATEWAY_IP}}
$ ip addr add {{IP}}/24 dev eth0
```

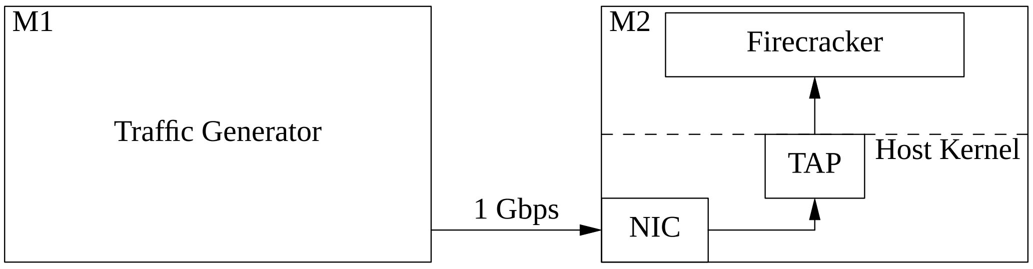


Figure 4.2: Experimental Setup for Firecracker

All of the experiments use a single-request processing model. A single request triggers the DAG, and a new request is only fetched after completing the previous request. The main performance metric is latency, so we do not use batching techniques to process multiple requests simultaneously. We assume that the required throughput can be achieved by horizontal scaling, i.e., adding more instances of the DAG.

The packet flow and the measurement probe locations for Ataru and Firecracker are illustrated in Fig 4.3. The packet enters the system via NIC and then goes to the platform, where the DAG processes it. The measurement probe captures the throughput. Other statistics captured are the minimum, maximum, average, standard deviation, skew and kurtosis for execution times of various entities such as DAG, functions, scheduler etc.

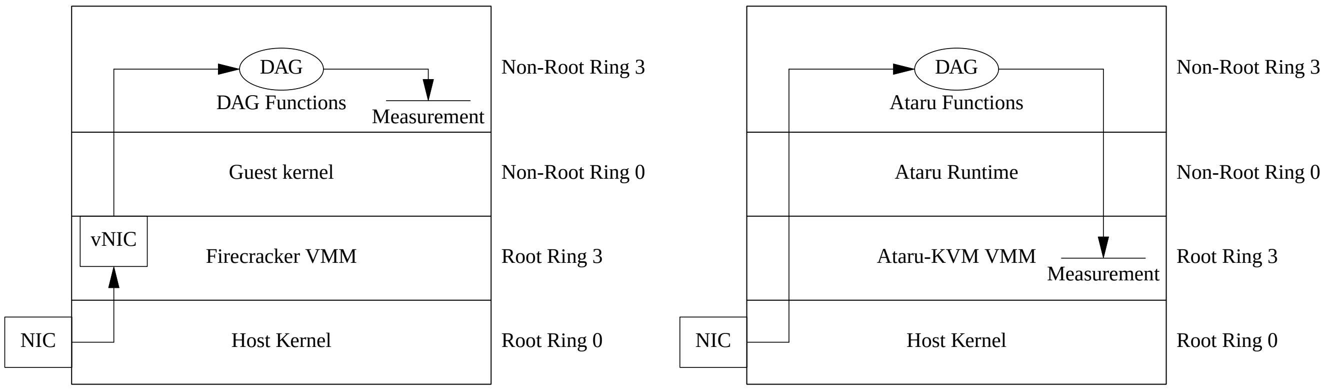


Figure 4.3: Measurement Probe Placement for Firecracker (FC) and Ataru

To demonstrate the advantages of Ataru, we designed an optimal process-based solution for executing a DAG in Firecracker. Using a semaphore synchronisation mechanism, we instantiate each function in the DAG as a process that blocks a semaphore waiting for a signal to start its execution. A master process receives the input and initiates the first function, which signals the now runnable functions according to the DAG structure upon completion. The kernel schedules the ready functions that are waiting on the corresponding semaphores until all the nodes in the DAG complete execution. To the best of our knowledge, this is the most efficient process-oriented implementation of a DAG execution model in Firecracker.

4.2 Results

We conduct a comparative evaluation of Ataru and Firecracker on metrics such as Boot-up time, Memory requirements, and performance on different synthetic DAGs. We also run three distinct applications on both platforms and analyze the results. Section 4.2.1 compares the Boot-up time of both platforms and discusses the factors that affect it. Section 4.2.2 illustrates the memory requirements of both platforms and explains how they differ. Section 4.2.3 demonstrate the scaling capability of Ataru. Section 4.2.4 demonstrates the performance benefit of Ataru over Firecracker on various synthetic DAGs and explores the reasons behind it. Finally, section

4.2.5 reports the results for various real-world applications and highlights the advantages and limitations of both platforms.

4.2.1 Boot Time

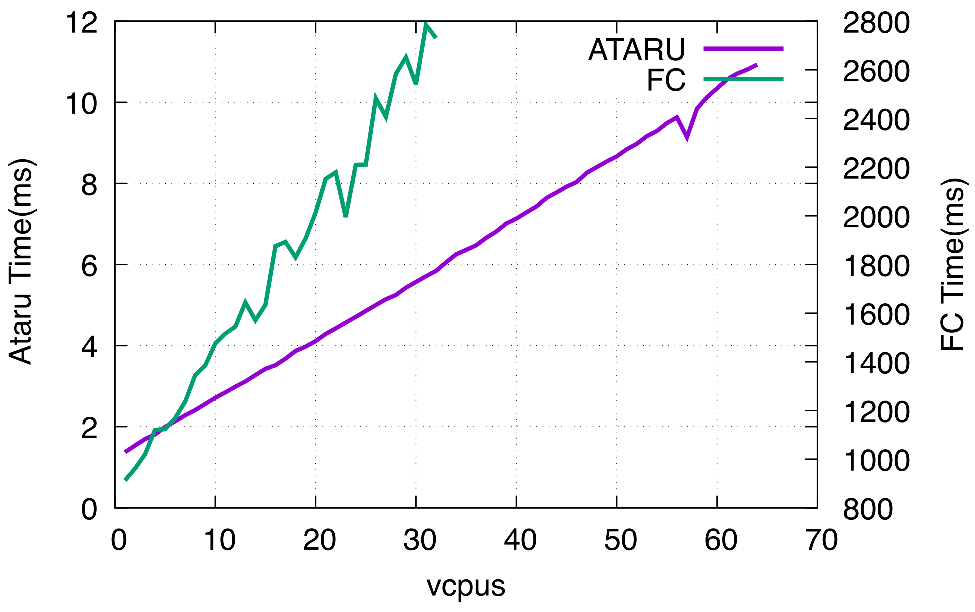


Figure 4.4: Boot time vs pool_size with Ataru VMM pinned to single core

Fig 4.4 shows the bootup time of the Ataru with the `pool_size`, which is the number of vCPUs allocated to a VM. In the experiment, we pinned the Ataru-KVM VMM to a single core. We see that the bootup time linearly increases with the number of vcpus. It is expected since each core has to complete its bootup process independently of others. Additionally, the bootup sequence requires certain data structures to be updated serially by each vcpu. Thus, the trend would remain linear even if more physical cores were present. Firecracker has a restriction that it can only boot 32 vCPUs, whereas Ataru is designed to boot 64 vCPUs. Consequently, the results of Firecracker are only shown till 32 vCPUs.

4.2.2 Memory Requirements

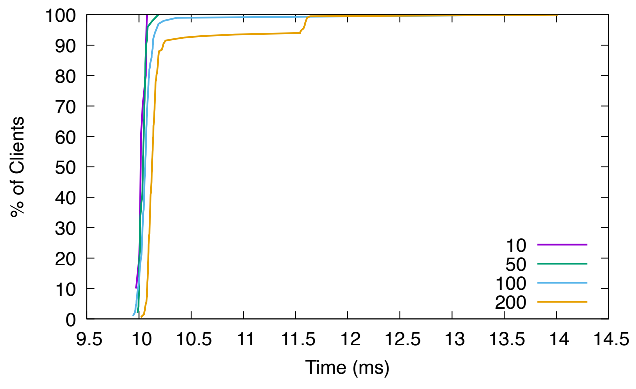
Platform	Ataru	Firecracker
Memory(Private)	<256 KB	<5.5 MB
Memory(RSS)	< 2.3 MB	<51 MB

Table 4.1: Memory Requirement Comparison

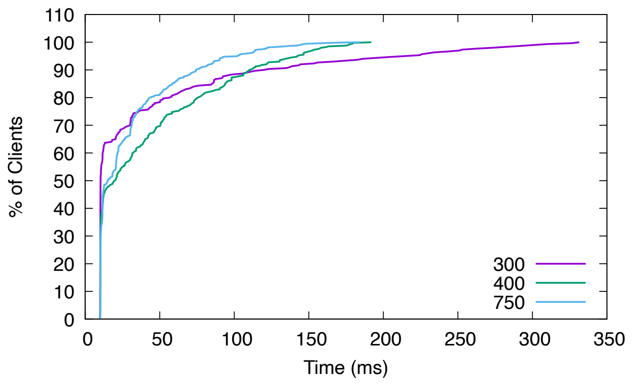
To estimate the memory required for Ataru VMM, we compute the sum of the private pages in `/proc/PID/smaps`. These are the pages that are only accessible by the process and will be released to the kernel when the process terminates (Memory(Private)). Therefore, they can be interpreted as a scaling factor for the memory overhead. Table 4.1 summarizes the results for a single vCPU scenario. The table also shows the resident set size memory (Memory(RSS)) for both the platforms. Ataru's low memory overhead is attributed to its customized minimal VMM+Runtime platform that contrasts with a full-blown Linux kernel of Firecracker.

4.2.3 Scaling

To evaluate the scalability of our system, we employ a DAG as depicted in Figure 4.13. For each client we deploy a 1vCPU, 1pCPU instance and randomly distribute each instantiation to one of the four isolated cores. This DAG has a function load of 10, which corresponds to a few microseconds of execution time. We initiate multiple concurrent clients and record the average execution time for 1000 requests. Figure 4.5 illustrates the cumulative distribution of the execution time for different levels of client concurrency, as indicated in the legend. In Figure 4.5a, we observe that the execution time remains stable for all clients in the absence of congestion. However, beyond a certain threshold, the execution time deteriorates as shown in Figure 4.5b. Congestion arises when the network throughput is lower than the Ataru processing capacity, which enables the system to maintain a stable execution time for a growing number of clients. However, when the number of concurrent clients exceeds a certain limit, the processing load surpasses the network throughput and causes a degradation in the execution time. The threshold of client concurrency at which congestion begins to affect the execution time is inversely proportional to the platform overheads. As demonstrated in section 4.2.4, Ataru achieves an order of magnitude higher performance than Firecracker in the short function regime, which implies that Ataru can also scale better for short functions. Furthermore, Ataru's low memory overhead enables it to handle thousands of concurrent clients. Firecracker, on the other hand, consumes an order of magnitude more memory than Ataru, which limits its scalability in terms of memory utilization with scaling. Therefore, Ataru also outperforms Firecracker in scaling with respect to memory consumption.



(a) Non-Congestion



(b) Congestion

Figure 4.5: Cumulative Distribution of Execution Time of a DAG with Concurrent Clients

4.2.4 Micro-Benchmarking

Ataru is optimized for short duration functions and has significantly higher performance compared to Firecracker when platform overheads become comparable to function execution time. To highlight various DAG configuration and function duration regime where Ataru excels and regimes where any gain is amortized by long function execution time we perform micro-benchmarking on the different synthetic DAGs.

The micro-benchmarking experiments use the same function for all the nodes in the DAG. The function has a synthetic workload that simulates the computation required by an application. We vary the workload of the function and measure the average throughput of processing single request for a given workload setting. We test for different function loads; specifically, we test for the following regimes:

1. Short Function Load Regime: Function load 1 to 100 in steps of 1
2. Low Function Load Regime: Function load 100 to 1000 in steps of 100
3. High Function Load Regime: Function load from 1000 to 10000 in steps of 1000

Translated to function execution time regimes we test from ~ 1 μs to ~ 10 ms

We measure two statistics for performance comparison:

1. Execution time for processing 1 request in the DAG

2. Execution Efficiency: Ratio of sum of all function execution times in the DAG to the total time required to process the DAG. As an example, execution efficiency close to 1 for a single vCPU indicates that the platform overheads are negligible and almost complete time was spent in executing the function instead of setting up the platform.

We measure the performance on a method to estimate π based on numerical integration of $\int_0^1 \frac{1}{1+x^2}$. This function mimics a realistic computational workload

We manipulate the computational demand of each function by varying the number of iterations inside the method. This enables us to benchmark the performance with increasing function load.

We benchmark the platforms for 4 configurations

1. Section 4.2.4.1: DAG with single function
2. Section 4.2.4.2: DAG with chain of 3 functions
3. Section 4.2.4.3: DAG with chain of 8 functions
4. Section 4.2.4.4: DAG with parallelism of 4

For each DAG configuration, we benchmark the following combinations of vCPU and Physical CPU (pCPU)

- `1vCPU,1pCPU`: This measures the performance when least compute resources are allocated
- `4vCPU,2pCPU`: This measures performance under over-provisioning scenario
- `4vCPU,4pCPU`: This measures performance when maximal compute resources are present

4.2.4.1 SF: DAG with Single Function

To estimate the overheads associated with launching a DAG, we benchmark a DAG with a single function. Fig 4.6 represents the DAG being executed, and Fig 4.7 shows performance comparison.

1. `1vCPU,1pCPU`: Result are shown in Fig 4.7a and 4.7b. The performance is comparable in all regimes between Ataru and Firecracker. This is expected since there is only one function. The only steps involved here are switching to guest mode executing the function and returning to the VMM. Since both platforms must perform this step, the execution times are similar.

2. `4vCPU,2pCPU`: Result are shown in Fig 4.7c and 4.7d. Only 1 vCPU is required for the DAG; thus, in over-provisioning, we see that Firecracker has a jitter in low function loads due to sub-optimal scheduling, whereas Ataru performs consistently better and without jitter.

3. `4vCPU,4pCPU`: Result are shown in Fig 4.7e and 4.7f. Since the number of vCPU matches the number of pCPUs in the case of Firecracker, the jitter is absent. However, scheduling overheads are still incurred by finding a runnable process in the DAG and switching context to it. Ataru performs similar to 1vCPU,1pCPU case and outperforms Firecracker in short function regime.

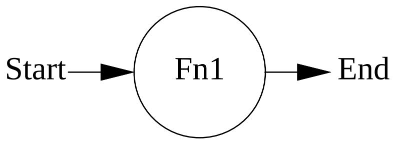
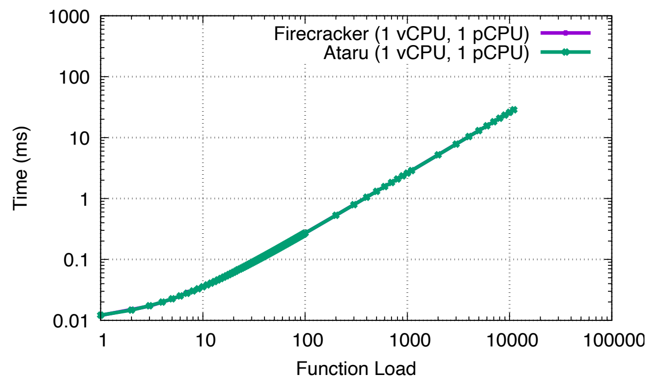
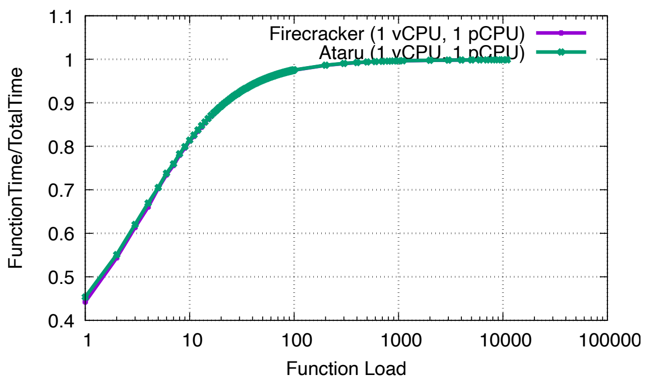


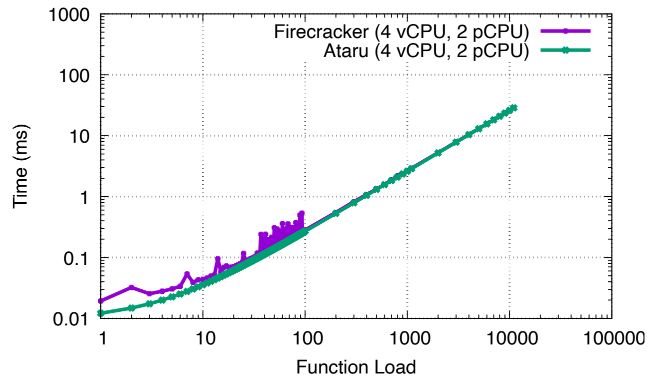
Figure 4.6: SF: DAG with Single Function



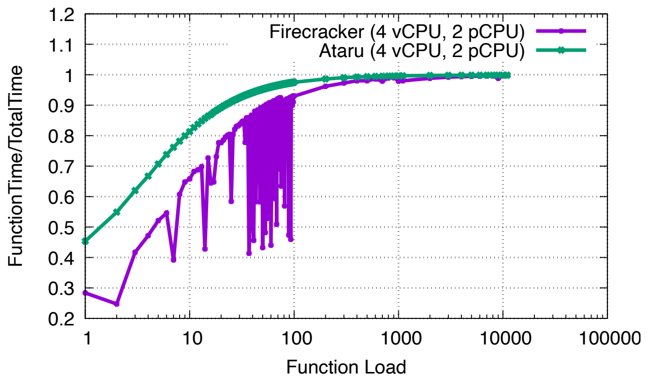
(a) 1vCPU,1pCPU: Single Request Execution Time



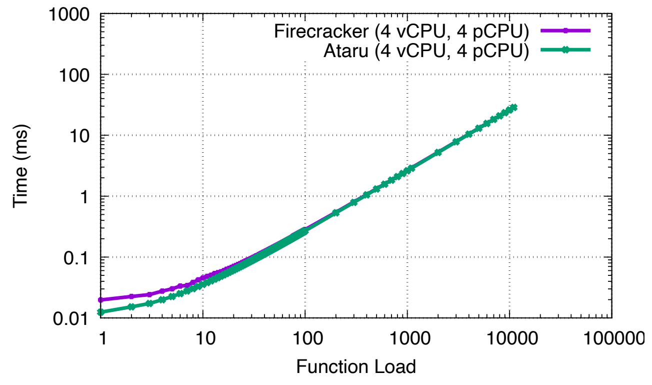
(b) 1vCPU,1pCPU: Execution Efficiency



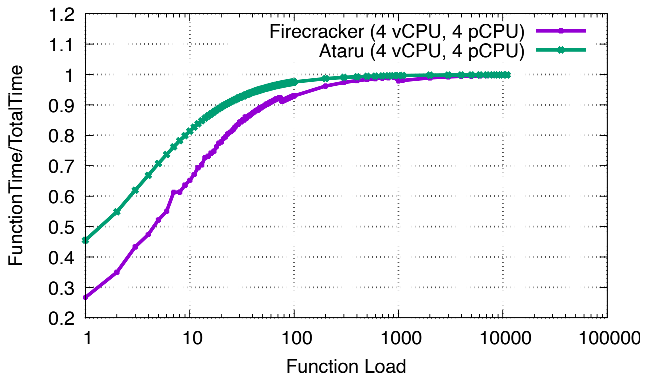
(c) 4vCPU,2pCPU: Single Request Execution Time



(d) 4vCPU,2pCPU: Execution Efficiency



(e) 4vCPU,4pCPU: Single Request Execution Time



(f) 4vCPU,4pCPU: Execution Efficiency

Figure 4.7: SF : Performance Comparison of a DAG with single function

4.2.4.2 C3: DAG with Chain of 3 Functions

To measure performance for DAG with a short chain, we benchmark a DAG with a chain of 3 functions. Fig 4.8 represents the DAG being executed, and Fig 4.9 shows a performance comparison.

1. `1vCPU,1pCPU`: Result are shown in Fig 4.9a and 4.9b. The Ataru exhibits superior performance in short-function load. The execution efficiency of Ataru is 70% and Firecracker is 45% at the lowest function load. Even at a function load of 10 corresponding to $\sim 100\ \mu s$ execution time, the Ataru maintains $> 90\%$, and Firecracker has 80% execution efficiency.

2. `4vCPU,2pCPU`: Result are shown in Fig 4.9c and 4.9d. Only 1 vCPU is needed for the DAG. Thus, we observe no benefit of allocating two physical cores. Even though Ataru's performance is comparable to 1vCPU,1pCPU case, we observe that Firecracker performance has jitter and degraded execution efficiency

3. `4vCPU,4pCPU`: Result are shown in Fig 4.9e and 4.9f. Since the number of vCPUs matches the number of CPUs, in the case of Firecracker, jitter is eliminated. However, scheduling overheads are still present as explained in case SF. Ataru, on the other hand, performs consistently and similarly to 1vCPU,1pCPU case.

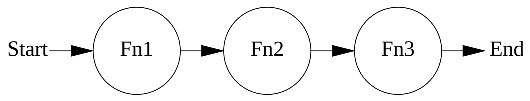
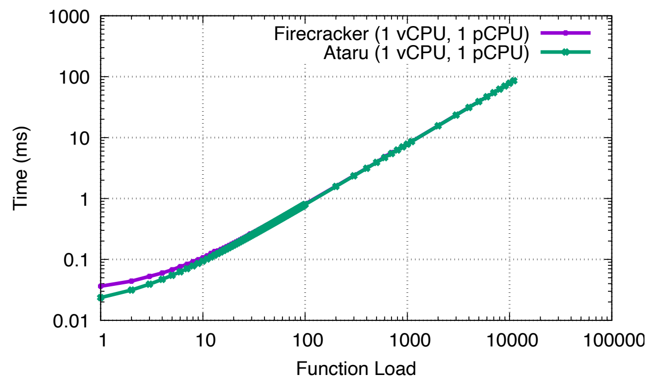
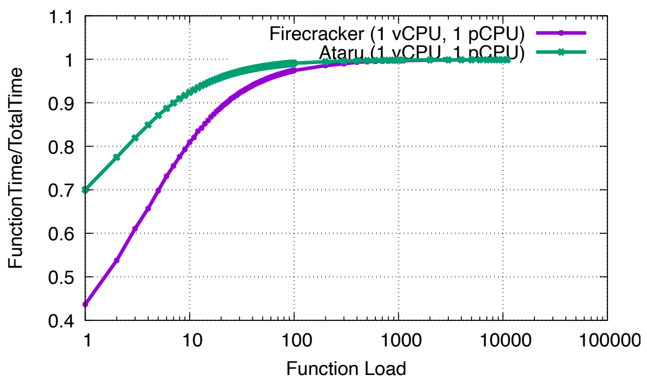


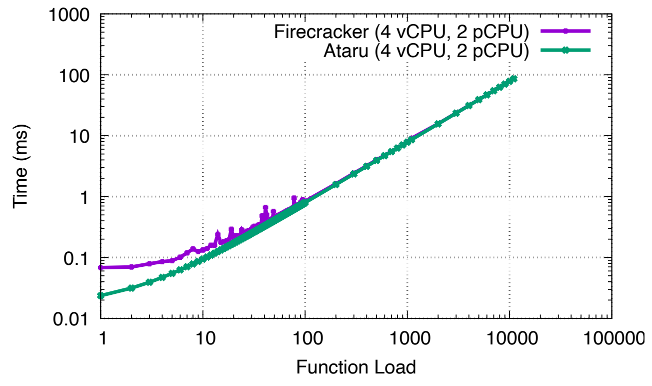
Figure 4.8: C3: DAG with Chain of 3 Function



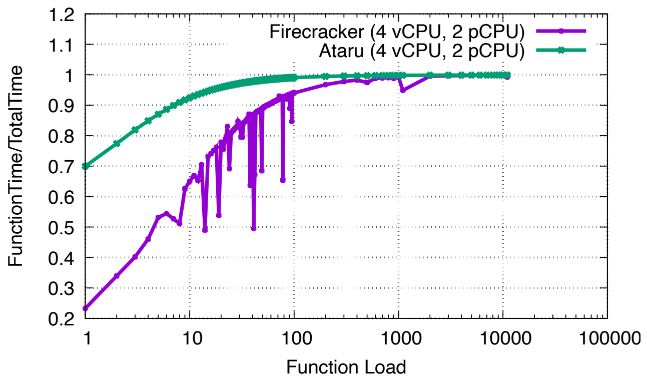
(a) 1vCPU,1pCPU: Single Request Execution Time



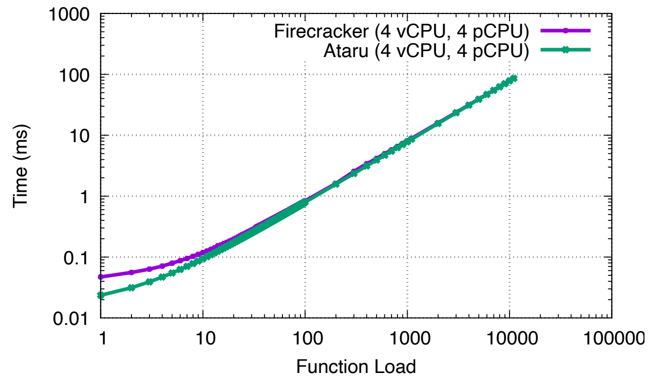
(b) 1vCPU,1pCPU: Execution Efficiency



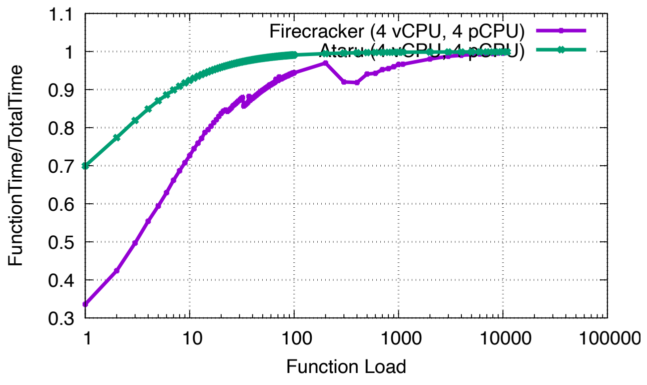
(c) 4vCPU,2pCPU: Single Request Execution Time



(d) 4vCPU,2pCPU: Execution Efficiency



(e) 4vCPU,4pCPU: Single Request Execution Time



(f) 4vCPU,4pCPU: Execution Efficiency

Figure 4.9: C3 : Performance Comparison of a DAG with chain of 3 functions

4.2.4.3 C8: DAG with Chain of 8 Functions

We benchmark a DAG with a chain of 8 functions to evaluate performance for DAG with a long chain. Fig 4.10 depicts the DAG being executed, and Fig 4.11 presents a performance comparison.

1. `1vCPU,1pCPU`: Result are shown in Fig 4.11a and 4.11b As in the C3 Case, the Ataru exhibits superior performance in short function load. The execution efficiency of Ataru is 85%, and Firecracker is 50% at the lowest function load. Even at a function load of 10 corresponding to $\sim 100\ \mu s$ execution time, Ataru maintains 95%, and Firecracker has 85% execution efficiency.

2. `4vCPU,2pCPU`: Result are shown in Fig 4.11c and 4.11d. Only 1 vCPU is required for the DAG; thus, we observe no benefit of allocating two physical cores. Even though Ataru performance is comparable to 1vCPU,1pCPU case, we observe that Firecracker performance has jitter and diminished execution efficiency worse than the C3 case, which shows poor scheduling in virtualized platforms.

3. `4vCPU,4pCPU`: Result are shown in Fig 4.11e and 4.11f. Since the number of vCPU matches the number of pCPUs, in the case of Firecracker, jitter is eliminated. However, there are still some scheduling overheads present as described in case SF. Ataru, on the other hand, performs consistently and similarly to 1vCPU,1pCPU case.

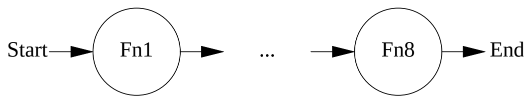
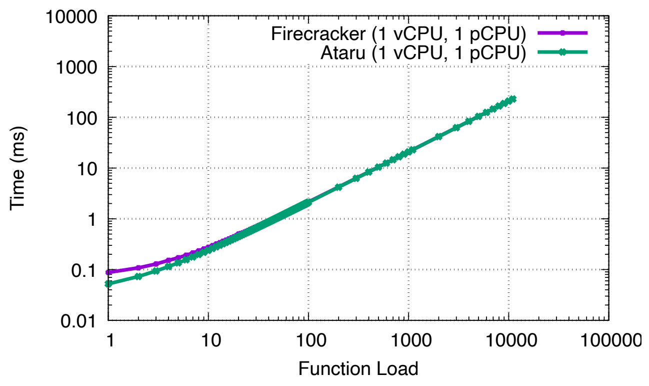
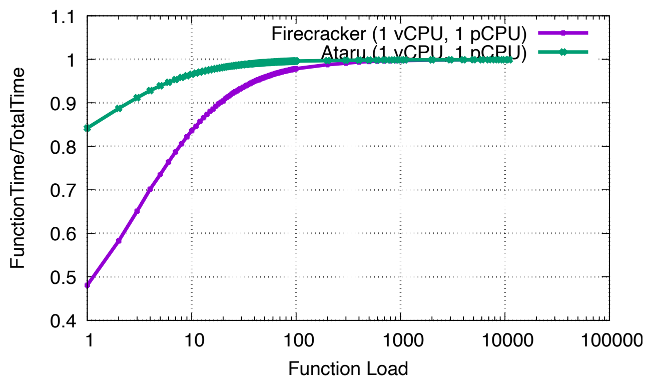


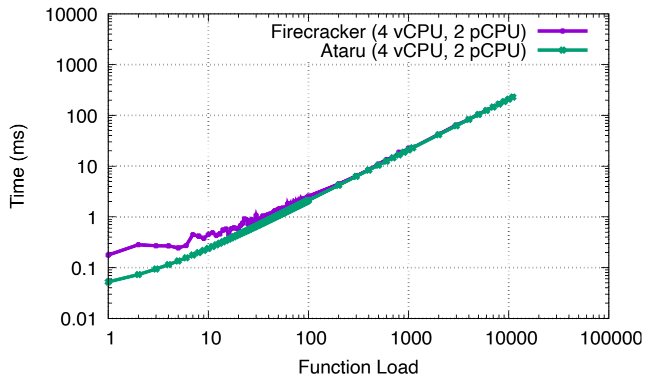
Figure 4.10: C8: DAG with Chain of 8 Function



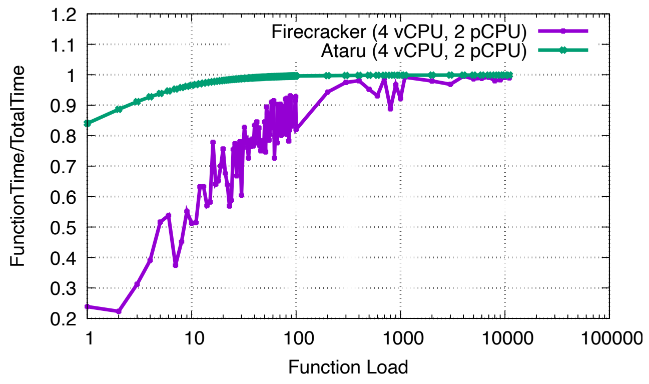
(a) 1vCPU,1pCPU: Single Request Execution Time



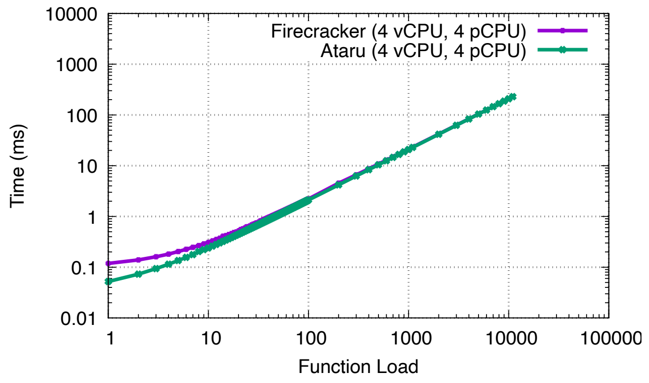
(b) 1vCPU,1pCPU: Execution Efficiency



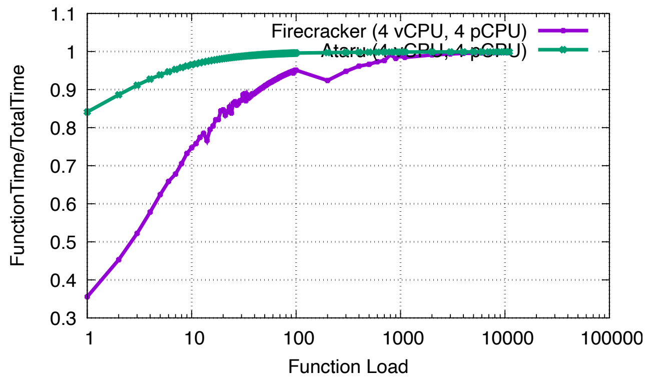
(c) 4vCPU,2pCPU: Single Request Execution Time



(d) 4vCPU,2pCPU: Execution Efficiency



(e) 4vCPU,4pCPU: Single Request Execution Time



(f) 4vCPU,4pCPU: Execution Efficiency

Figure 4.11: C8 : Performance Comparison of a DAG with chain of 8 functions

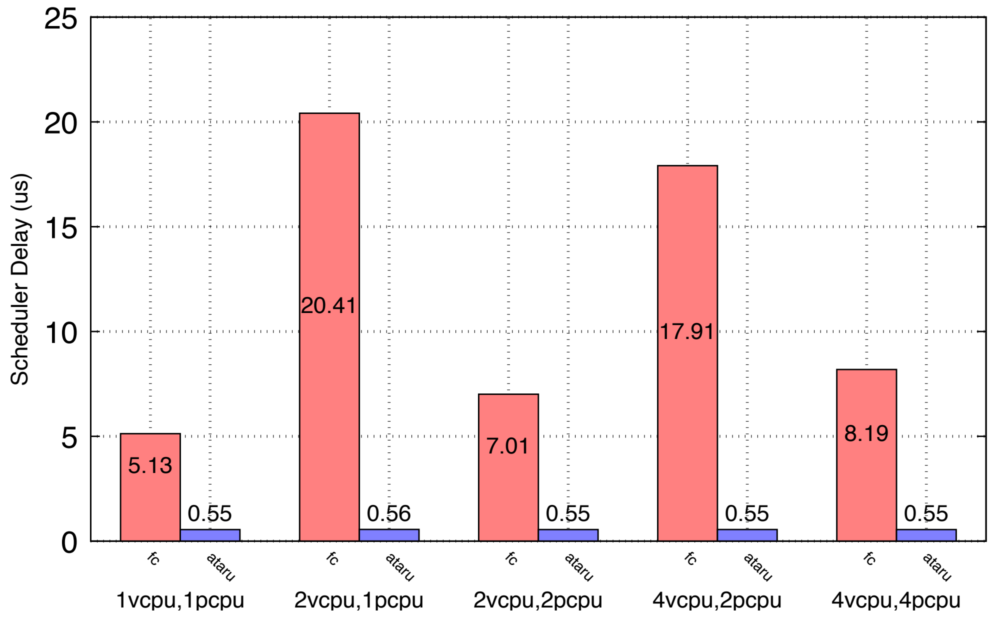


Figure 4.12: C8: Average Scheduler Delay Firecracker vs Ataru

Scheduler delay is depicted in Fig. 4.12. Firecracker exhibits an order of magnitude higher scheduling latency than Ataru across all scenarios. This is attributed to the decoupling of host and guest schedulers, which results in suboptimal resource utilization in Firecracker. In particular, overprovisioning scenarios (e.g., 2vCPU,1pCPU and 4vCPU,2pCPU) incur significant scheduling overheads due to the contention among vCPUs for physical cores. Moreover, scheduling latency increases with the number of allocated vCPUs even in scenarios with equal provisioning (e.g., 1vCPU,1pCPU, 2vCPU,2pCPU and 4vCPU,4pCPU), as more vCPUs entail more context switches and synchronization costs. In contrast, Ataru achieves an order of magnitude lower and consistent scheduling latency for all scenarios by leveraging a single vCPU to execute a linear chain of functions that does not exhibit any parallelism.

4.2.4.4 P4: DAG with Parallelism

To evaluate performance for DAG with parallelism, we benchmark a DAG with a single function followed by a fan out of 4 and then a barrier function. Fig 4.13 depicts the DAG being executed. and Fig 4.14 presents performance comparison.

1. `1vCPU,1pCPU`: Result are shown in Fig 4.14a and 4.14b As in the C3 and C8 Cases, the Ataru exhibits superior performance in short function load. The execution efficiency of Ataru is 80%, and Firecracker is 47% at the lowest function load. Even at a function load

of 10 corresponding to $\sim$ 100 μs execution time, Ataru maintains 95% and Firecracker has 83% execution efficiency.

2. `4vCPU,2pCPU`: Result are shown in Fig 4.14c and 4.14d. The DAG exhibits inherent parallelism; thus, more physical cores can enhance performance. We observe jitter in both Firecracker and Ataru as all 4 vCPU are used for DAG processing, and these 4 vCPUs have to be scheduled on two 2 pCPUs. Theoretically, if all the functions in the DAG have same load, maximum efficiency of 1.5 is expected. We observe this limit being achieved after 100 by Ataru and Firecracker achieves this limit after about function load of 1000.

3. `4vCPU,4pCPU`: Result are shown in Fig 4.14e and 4.14f. As discussed in the previous case, the execution time can benefit from the parallelism inherent in the DAG. The maximum theoretical efficiency that can be attained is 2.0 given that all functions have similar execution times. We observe that Ataru, at the lowest function load, has an efficiency of 1.2 and Firecracker has 0.6. Ataru is significantly superior in short-function regimes compared to Firecracker. In can also be seen that the theoretical maximum efficiency of 2.0 is reached earlier by Ataru.

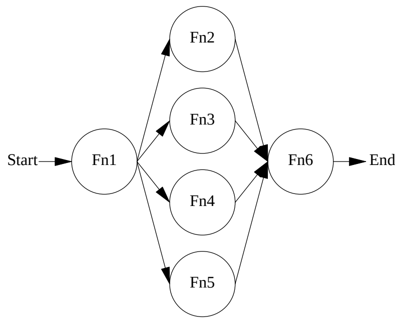
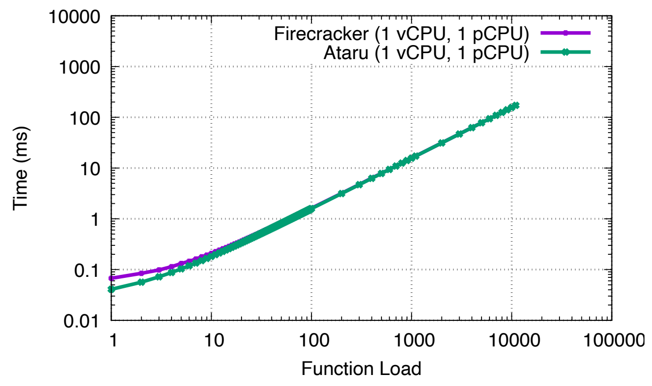
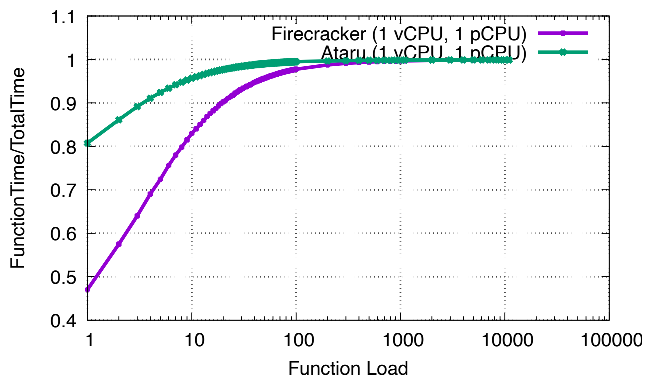


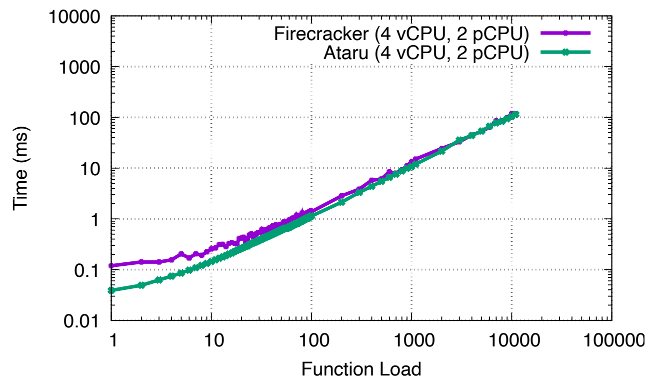
Figure 4.13: P4: DAG with Parallelism of 4



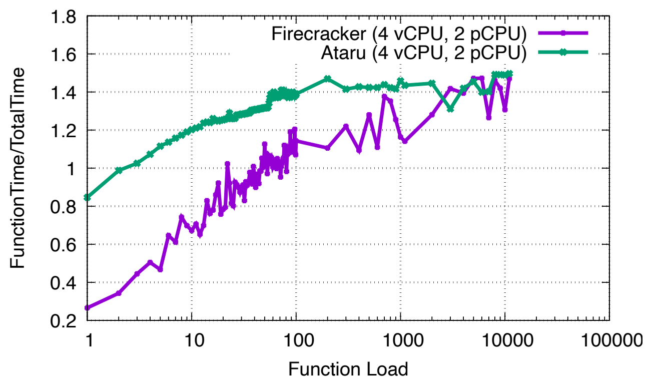
(a) 1vCPU,1pCPU: Single Request Execution Time



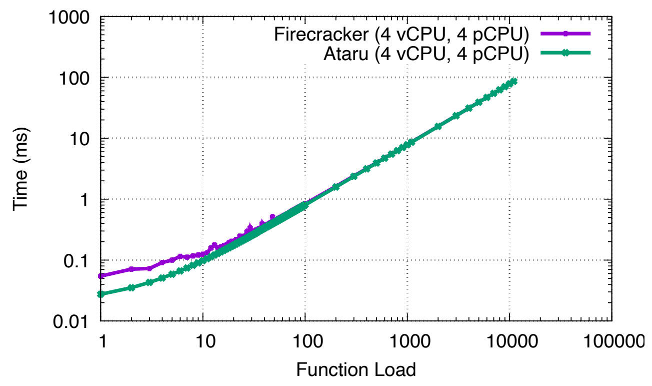
(b) 1vCPU,1pCPU: Execution Efficiency



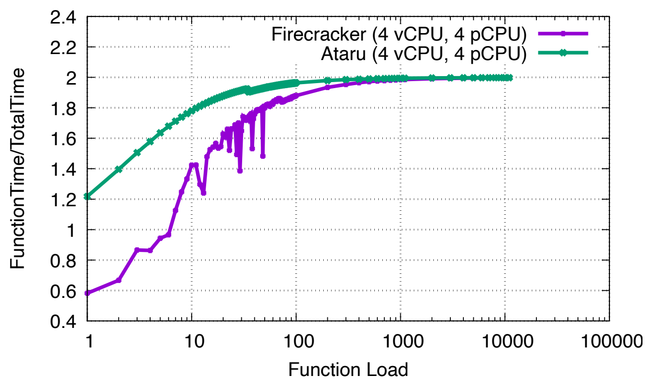
(c) 4vCPU,2pCPU: Single Request Execution Time



(d) 4vCPU,2pCPU: Execution Efficiency

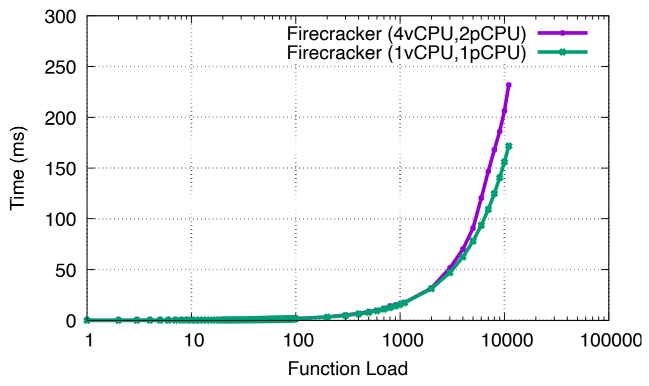


(e) 4vCPU,4pCPU: Single Request Execution Time

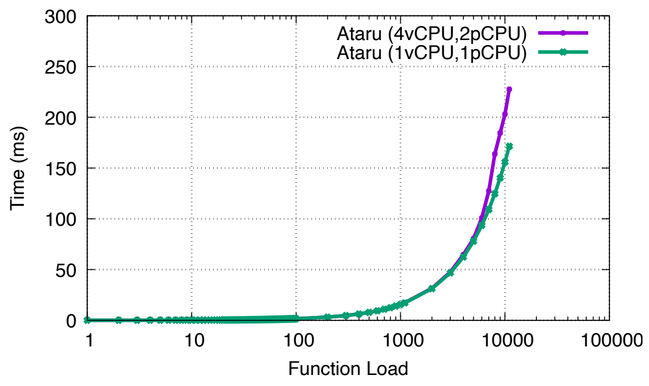


(f) 4vCPU,4pCPU: Execution Efficiency

Figure 4.14: P4: Performance Comparison of a DAG with Parallelism of 4



(a) Firecracker

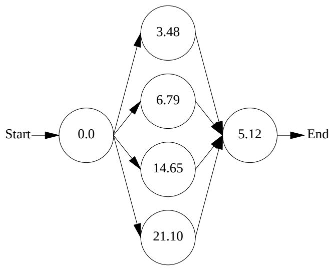


(b) Ataru

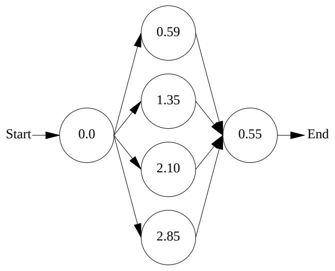
Figure 4.15: Function Execution Times (1vCPU,1pCPU vs 4vCPU,2pCPU)

In Fig 4.15, we present the results of measuring the total function execution time under two different configurations: 1vCPU,1pCPU and 4vCPU,2pCPU. The comparison reveals a significant increase in the function execution time for the 4vCPU,2pCPU configuration as the function load becomes higher, compared to the 1vCPU,1pCPU configuration. This phenomenon can be explained by the fact that in a virtualized environment with a TSC passthrough, a higher function load implies a higher chance of vCPU preemption, and scheduling in of another vCPU which leads to an inflation of the observed function execution time compared to the true function execution time.

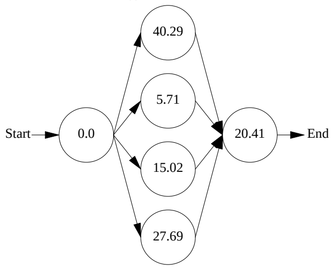
4.2.4.5 Scheduling Delay



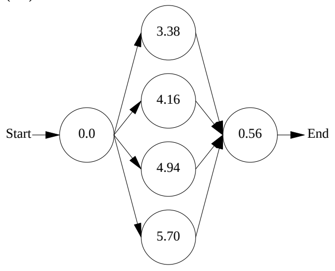
(a) 1vCPU,1pCPU: Firecracker



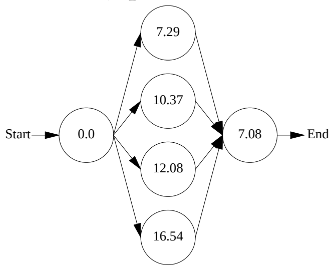
(b) 1vCPU,1pCPU: Ataru



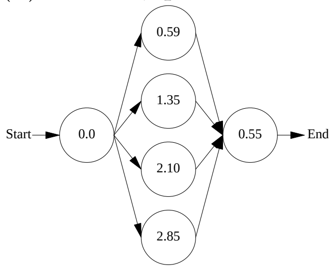
(c) 2vCPU,1pCPU: Firecracker



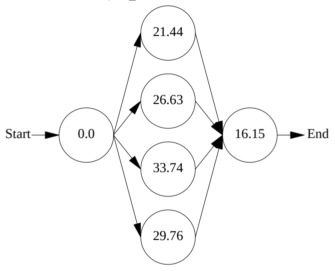
(d) 2vCPU,1pCPU: Ataru



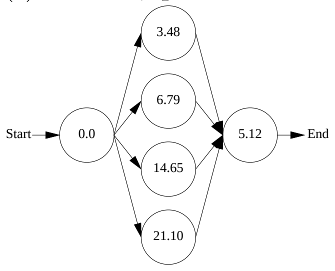
(e) 2vCPU,2pCPU: Firecracker



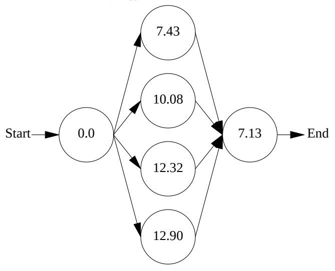
(f) 2vCPU,2pCPU: Ataru



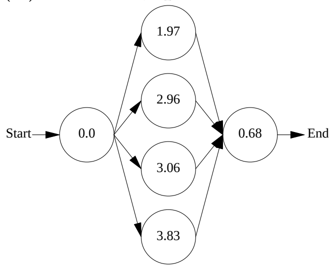
(g) 4vCPU,2pCPU: Firecracker



(h) 4vCPU,2pCPU: Ataru



(i) 4vCPU,4pCPU: Firecracker



(j) 4vCPU,4pCPU: Ataru

Figure 4.16: P4: Firecracker vs Ataru Scheduler Delay

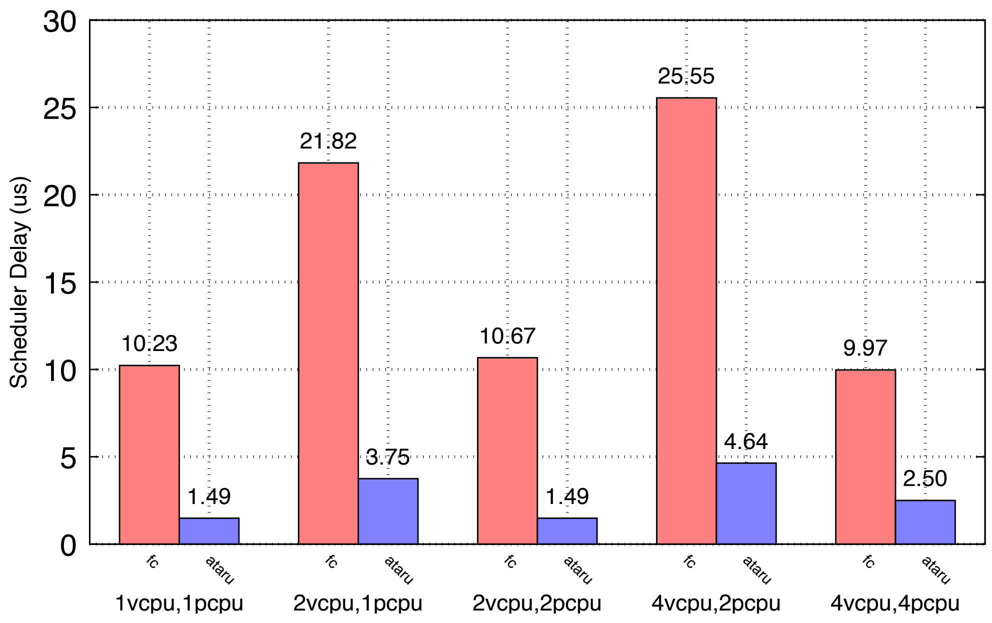


Figure 4.17: P4: Average Scheduler Delay Firecracker vs Ataru

Fig. 4.16 depicts the average scheduler latency per function in the DAG graph, while Fig. 4.17 illustrates the corresponding histogram plot. The first scheduled function is assumed to have zero latency. The latency of each function is relative to the maximum latency of its predecessors in the DAG. Firecracker exhibits higher scheduler latency than Ataru for all functions. Ataru schedules the DAG as a linear chain in the 1vCPU,1pCPU scenario, as only one vCPU is active. However, this incurs higher latency, as parallel functions could theoretically execute concurrently and their sequential execution is considered as a scheduling delay. In the 4vCPU,4pCPU scenario, Ataru achieves a scheduler latency of $< 2\mu s$ instead of $0.55\mu s$ as observed in chain of 8 functions because the delay encountered in waking up the vCPUs.

4.2.5 Applications

To evaluate the performance of Ataru for real-world applications, we deploy three divergent applications in both Ataru and Firecracker. Section 4.2.5.1 presents the results for a network virtualization application, Section 4.2.5.2 presents the results for an image processing application, and Section 4.2.5.3 presents the results for a robotics application.

We employ 5 configurations to assess the performance:

1. `1vcpu,1pcpu`: We allocate 1 vCPU and pin it to 1 pCPU (physical CPU). This measures the performance under the minimum compute resource scenario.

2. `2vcpu,1pcpu`: We allocate 2 vCPU and pin them to 1 pCPU. This measures the performance under the scenario of physical core sharing among 2 virtual cores. It simulates the case of over-provisioning in cloud environments.

3. `2vcpu,2pcpu`: We allocate 2 vCPU and pin them to 2 distinct pCPUs. This measures the performance under the scenario of exploiting the parallelism in a DAG. It also evaluates the effectiveness of the guest scheduler to utilize more cores.

4. `4vcpu,2pcpu`: We allocate 4 vCPU and pin them to 2 distinct pCPUs. This measures the performance under a more extreme version of `2vcpu,1pcpu` case. It evaluates how well the guest scheduler manages the 4 vCPUs, especially when the DAG requires fewer vCPUs than available.

5. `4vcpu,4pcpu`: We allocate 4 vCPU and pin them to 4 distinct pCPUs. This measures the performance under the maximum compute resource scenario and how well the platforms exploit resources.

4.2.5.1 Network Virtualization

For this application, we chose a representative Service Function Chain (SFC) from an NV domain is shown in Fig. 4.18. The SFC comprise a chain of Firewall, IDS, and Encryption functions derived from the NV literature [?][?][?].

In this experiment, we generate 256 byte packets at an interval of 1 ms using `nping` traffic generator utility. These packets are routed to the SFC in Fig 4.18. We choose an interval of 1 ms, which is sufficiently small enough to prevent overloading the vNIC and degrading the performance of Firecracker. Hence, to compare the performance, instead of measuring the total execution time, which will be identical given a sufficiently low traffic rate, we measure only the time needed to execute the DAG, excluding the network time.

Fig 4.19 shows the average time required to process 1000 packets by the DAG over 30 runs.

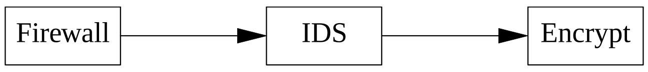


Figure 4.18: Representative SFC from a NV Domain

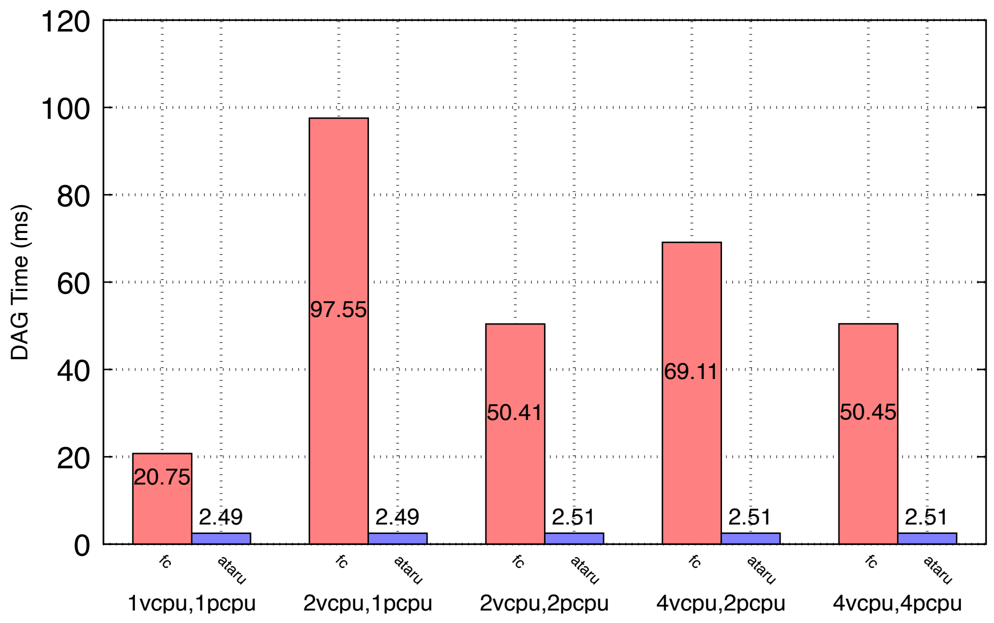


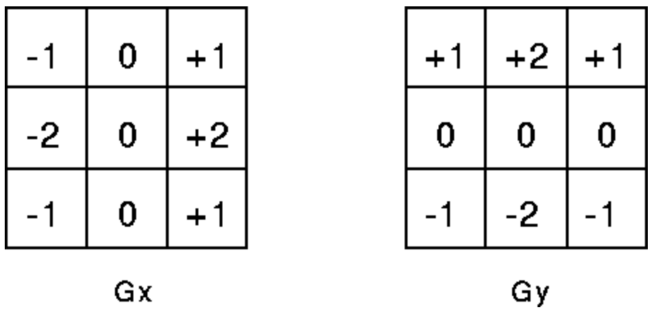
Figure 4.19: DAG Time to Process 1000 Packets: Firecracker vs Ataru

The process abstraction of Ataru is minimalistic and only contains the page table management information, which leads to an order of magnitude performance improvement over FC, where platform overheads dominate function execution times. In a linear SFC, increasing the number of vCPUs does not decrease processing time, as functions have to be executed sequentially. FC, on the other hand, experiences significant overheads due to over provisioning of resources, mainly caused by scheduling delays. The scheduling delay arises from the mismatch between the kernel scheduler that allocates physical cores to vCPUs and the guest scheduler that assigns functions to vCPUs. These schedulers are independent and may make suboptimal decisions. Ataru Runtime's scheduler optimizes the vCPU utilization by waking and kicking the vCPUs that can progress at maximum speed and halt the idle vCPUs that have no work to fetch, resulting in consistent performance across all configurations, demonstrating optimal resource utilization in case of over provisioning.

4.2.5.2 Image Processing: Edge Detection

Edge detection is a technique that aims to identify the boundaries of objects and regions in an image based on discontinuities in pixel intensity values. Edge detection is a fundamental operation in image processing, and computer vision, as it can be used to detect and extract salient features from an image. By applying an edge detection algorithm to an image, the amount of data to be processed can be significantly reduced while retaining the essential structural information of the image. This can facilitate the subsequent task of interpreting the semantic content of the original image. Many video analytics and image processing applications in the cloud use edge detection as a preprocessing step. In this experiment, we implement the Sobel edge detection algorithm, which the DAG represents in Fig 4.21.

The Sobel edge detection algorithm is a technique that uses two 3x3 convolutional filters (Fig 4.20) to compute an image's gradient magnitude and direction. One filter is obtained by rotating the other filter by 90 degrees. The filters are designed to respond maximally to edges that have orientations that are orthogonal to each other. The filters can be applied separately to the image, and the resulting gradient components (G_x and G_y) can then be combined. A common way to combine the gradient components is to use Eqn 4.1, which computes the absolute value of their sum.



-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 4.20: Sobel Convolutional Kernels

$$|G| = |G_x| + |Gy| \tag{4.1}$$

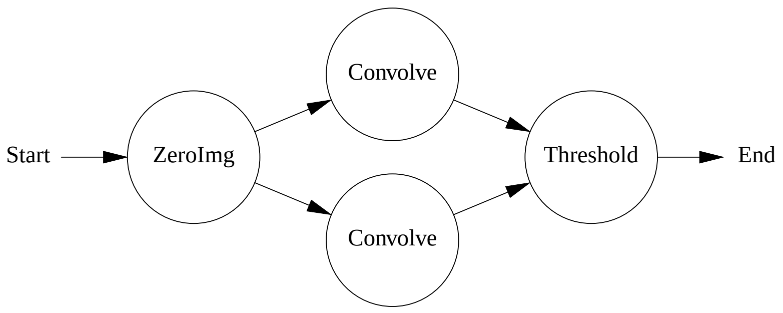


Figure 4.21: Image Processing: Edge Detection DAG

Each block in the DAG represented in Fig 4.21 are briefly described below:

1. `ZeroImg`: Zero the output image
2. `Convolve`: Convolve with the convolutional kernel
3. `Threshold`: Perform thresholding operation in the resulting image

A sample output of the Sobel edge detection algorithm is shown in the Fig 4.22.

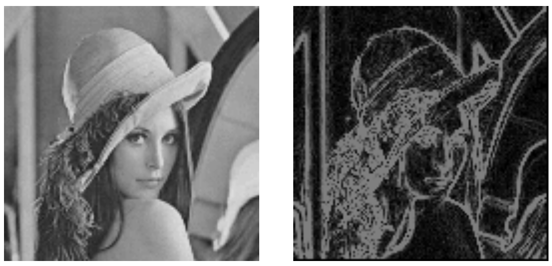


Figure 4.22: Edge Detection of 128x128

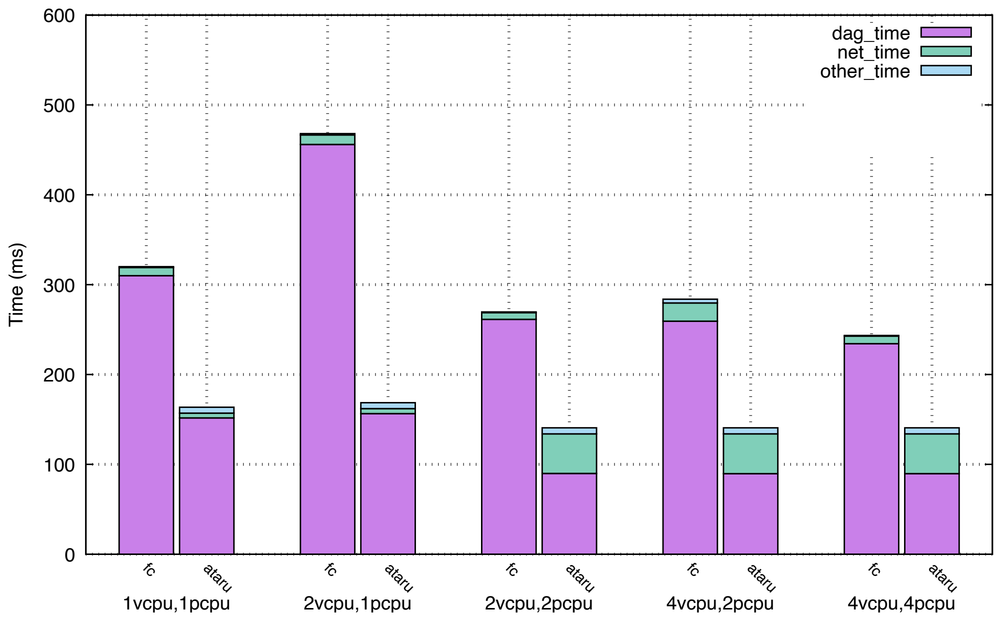


Figure 4.23: Edge Detection, Time to Process 1000 128x128 Images

The performance comparison of FC and Ataru for the edge detection application is depicted in Fig 4.23. Since the two convolution functions can be executed concurrently, the minimum DAG execution time can be achieved in configurations higher than 2vCPU,2pCPU. Ataru has a performance advantage over FC in the 1vCPU, 1pCPU configuration due to its minimal process abstraction. FC suffers from significant overheads as described in the section 4.2.5.1 in the 2vCPU, 1pCPU configuration, while Ataru maintains consistent performance similar to the 1vCPU, 1pCPU configuration. FC shows slight performance improvement in the 2vCPU, 2pCPU configuration, while Ataru reduces the DAG execution time by half. In the 2vCPU,2pCPU, 4vCPU,2pCPU, and 4vCPU,4pCPU configuration, the net_time of Ataru increases considerably due to network bandwidth saturation as Ataru started to execute faster than the request rate, indicating that Ataru was waiting for the data from the network.

4.2.5.3 EKF: Localization Problem

The development of several Cloud Robotics platforms was motivated by two factors: the limited on-board computational resources of mobile robots that operate under energy constraints and the rapid progress in cloud computing technologies that enable access to high-performance computing resources over the network. Applications such as Simultaneous Localization and Mapping (SLAM) and Navigation are suitable candidates for cloud-based implementation [**?**] [**?**]. In this example, we demonstrate a problem involving a robot navigating a terrain with sensors that measure the distance and angle to various landmarks. The robot could be an autonomous vehicle that uses computer vision to detect trees, buildings, and other features. The robot can turn using the steering angle of the wheels, assuming the differential drive is present, resulting in a nonlinear motion model that involves a pivot point at the rear axle and a forward displacement. The robot also has a sensor that estimates the distance and angle to known targets in the terrain leading to a nonlinear observation model that requires square root and trigonometric operations to compute the position from the distance and angle measurements. Fig 4.24 shows the pictorial representation of the wheeled robot with steering angle α with turn radius as R, turn angle as β and wheelbase as w.

The experiment implements an EKF algorithm, as shown by equations 4.2 and 4.3. The algorithm is implemented as DAG represented by Fig 4.25. The algorithm receives the time stamp, landmark locations and sensor measurement, which the DAG then processes.

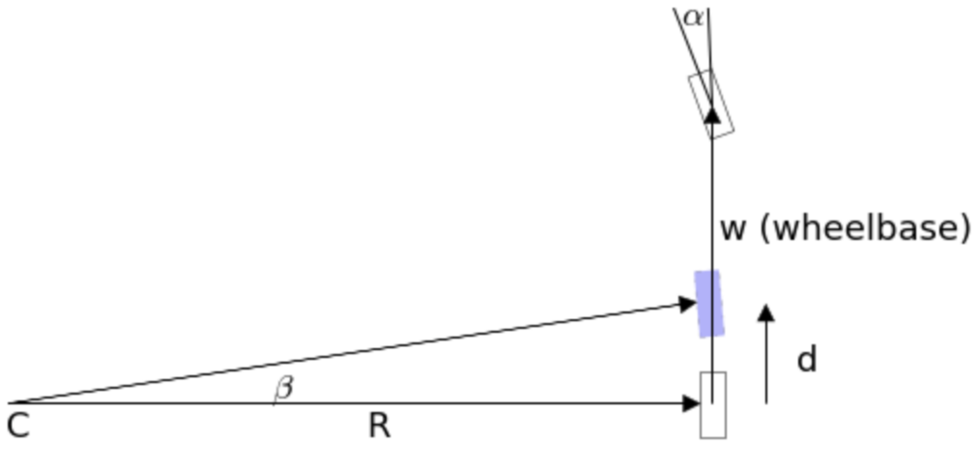


Figure 4.24: Robot Motion Model

$$\begin{aligned}
\hat{\boldsymbol{x}}_{k|k-1} &= f\left(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k\right) \\
\boldsymbol{P}_{k|k-1} &= \boldsymbol{F}_k \boldsymbol{P}_{k-1|k-1} \boldsymbol{F}_k^T + \boldsymbol{Q}_k \\
\tilde{\boldsymbol{y}}_k &= \boldsymbol{z}_k - h\left(\hat{\boldsymbol{x}}_{k|k-1}\right) \\
\boldsymbol{S}_k &= \boldsymbol{H}_k \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T + \boldsymbol{R}_k \\
\boldsymbol{K}_k &= \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T \boldsymbol{S}_k^{-1} \\
\hat{\boldsymbol{x}}_{k|k} &= \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \tilde{\boldsymbol{y}}_k \\
\boldsymbol{P}_{k|k} &= \left(\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k\right) \boldsymbol{P}_{k|k-1}
\end{aligned} \tag{4.2}$$

$$\begin{aligned}
\boldsymbol{F}_k &= \left.\frac{\partial f}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k} \\
\boldsymbol{H}_k &= \left.\frac{\partial h}{\partial \boldsymbol{x}}\right|_{\hat{\boldsymbol{x}}_{k|k-1}}
\end{aligned} \tag{4.3}$$

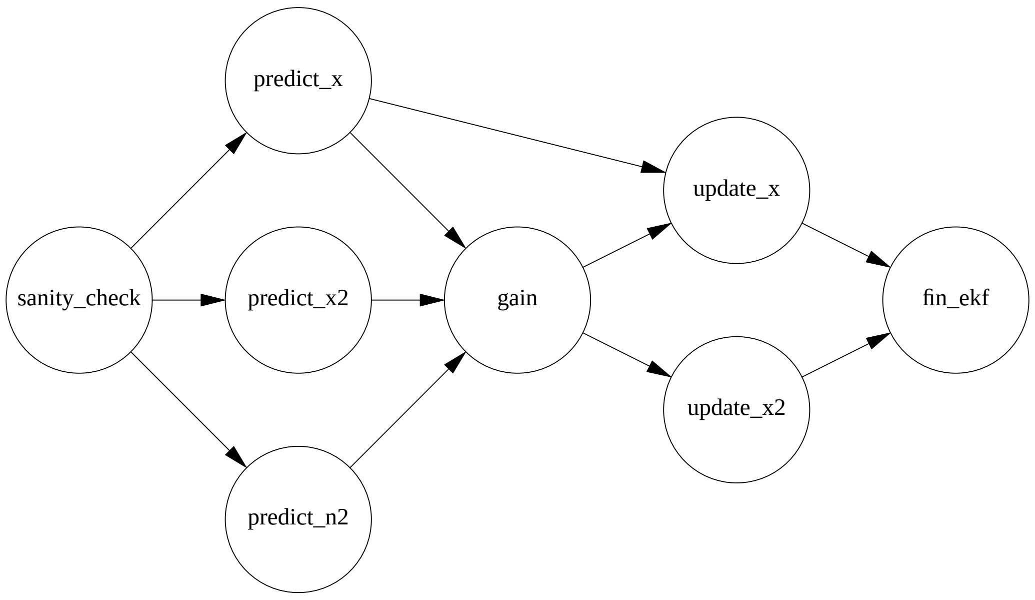


Figure 4.25: EKF DAG

Each block in the DAG represented in Fig 4.25 are briefly described below.

1. `sanity_check`: Check for the sanity of the inputs, and threshold the steering angle if near zero
2. `predict_x`: Calculate $\hat{\boldsymbol{x}}_{k|k-1}$, $\boldsymbol{H}_k$ and $h\left(\hat{\boldsymbol{x}}_{k|k-1}\right)$
3. `predict_x2`: Calculate $\boldsymbol{F}_k \boldsymbol{P}_{k-1|k-1} \boldsymbol{F}_k^T$
4. `predict_n2`: Calculate $\boldsymbol{Q}_k$
5. `gain`: Calculate $\boldsymbol{K}_k$
6. `update_x`: Calculate $\hat{\boldsymbol{x}}_{k|k}$
7. `update_x2`: Calculate $\boldsymbol{P}_{k|k}$
8. `fin_ekf`: Act as barrier and finalization of outputs

Eqn 4.4 to 4.11 shows the specific matrices used for this example. Sensor noise $\boldsymbol{R}_k$ and input noise M to determine process noise VMV^T is chosen appropriately.

The sensor noise covariance matrix $\boldsymbol{R}_k$ and the input noise covariance matrix M are selected accordingly. The process noise covariance matrix is obtained by multiplying VMV^T, where V is the linearization matrix of the motion model as shown by Eqn 4.11

$$\begin{aligned} \beta &= \frac{d}{w}\tan(\alpha) \\ R &= \frac{d}{\beta} \end{aligned} \tag{4.4}$$

$$\begin{aligned} x &= x - R\sin(\theta) + R\sin(\theta+\beta) \\ y &= y + R\cos(\theta) - R\cos(\theta+\beta) \\ \theta &= \theta + \beta \end{aligned} \tag{4.5}$$

$$\boldsymbol{F} = \begin{bmatrix} 1 & 0 & -\frac{w\cos(\theta)}{\tan(a)} + \frac{w}{\tan(a)}\cos\left(\frac{tv}{w}\tan(a)+\theta\right) \\ 0 & 1 & -\frac{w\sin(\theta)}{\tan(a)} + \frac{w}{\tan(a)}\sin\left(\frac{tv}{w}\tan(a)+\theta\right) \\ 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

$$\mathbf{P}^{-} = \mathbf{FPF}^{\top} + \mathbf{VMV}^{\top} \tag{4.7}$$

$$\boldsymbol{h} = \begin{bmatrix} \sqrt{(p_x-x)^2 + (p_y-y)^2} \\ \arctan\left(\frac{p_y-y}{p_x-x}\right) - \theta \end{bmatrix} \tag{4.8}$$

$$\boldsymbol{H} = \begin{bmatrix} \frac{-px+x}{\sqrt{(px-x)^2+(py-y)^2}} & \frac{-py+y}{\sqrt{(px-x)^2+(py-y)^2}} & 0 \\ -\frac{-py+y}{(px-x)^2+(py-y)^2} & -\frac{px-x}{(px-x)^2+(py-y)^2} & -1 \end{bmatrix} \tag{4.9}$$

$$\mathbf{V} = \frac{\partial f(x,u)}{\partial u} \begin{bmatrix} \frac{\partial \dot{x}}{\partial v} & \frac{\partial \dot{x}}{\partial \alpha} \\ \frac{\partial \dot{y}}{\partial v} & \frac{\partial \dot{y}}{\partial \alpha} \\ \frac{\partial \dot{\theta}}{\partial v} & \frac{\partial \dot{\theta}}{\partial \alpha} \end{bmatrix} \tag{4.10}$$

$$\boldsymbol{V} = \begin{bmatrix} t\cos\left(\frac{tv}{w}\tan(a)+\theta\right) & \frac{tv}{\tan(a)}\left(\tan^2(a)+1\right)\cos\left(\frac{tv}{w}\tan(a)+\theta\right) - \frac{w\sin(\theta)}{\tan^2(a)}\left(-\tan^2(a)-1\right) + \frac{w}{\tan^2(a)}\left(-\tan^2(a)-1\right)\sin\left(\frac{tv}{w}\tan(a)+\theta\right) \\ t\sin\left(\frac{tv}{w}\tan(a)+\theta\right) & \frac{tv}{\tan(a)}\left(\tan^2(a)+1\right)\sin\left(\frac{vv}{w}\tan(a)+\theta\right) + \frac{w\cos(\theta)}{\tan^2(a)}\left(-\tan^2(a)-1\right) - \frac{w}{w}\left(\tan^2(a)+1\right) \end{bmatrix} \tag{4.11}$$

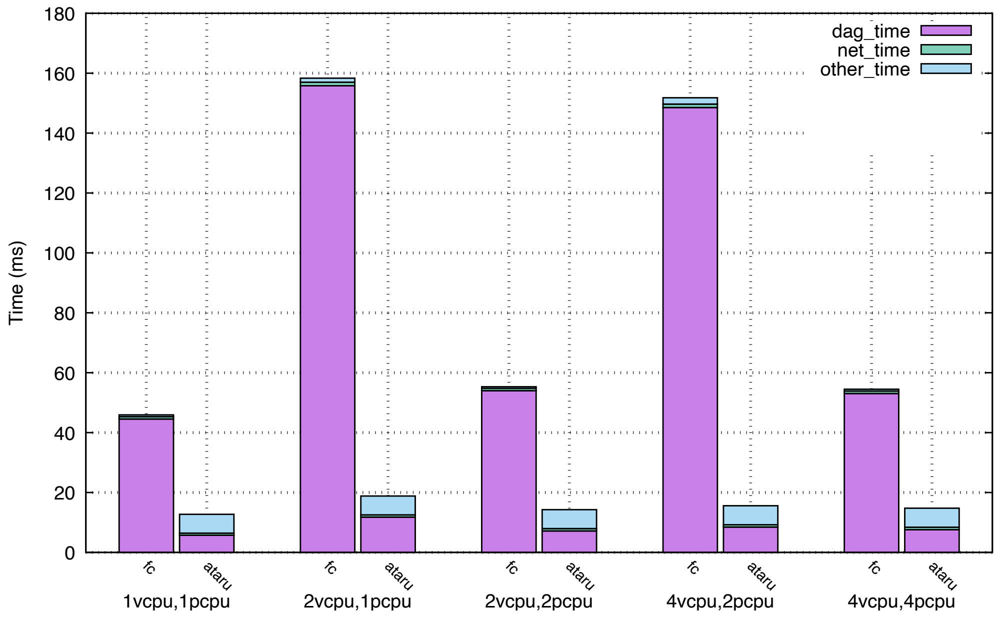


Figure 4.26: EKF: Localization Problem

Figure 4.26 presents a comparative analysis of FC and Ataru for the EKF Application. The EKF DAG depicted in Figure 4.25 demonstrates the presence of exploitable parallelism in the DAG. Ataru outperforms FC by a factor of 3 in the 1vCPU,1pCPU scenario. FC exhibits degraded performance in the 2vCPU,1pCPU and 4vCPU,2pCPU scenarios due to suboptimal scheduling as discussed in Section 4.2.5.1. Despite the parallelism inherent in the DAG that could benefit from multicore availability, there is no improvement in execution time because platform overhead dominates the execution time, thus rendering the gain due to additional cores insignificant.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Current virtualized platforms incur high platform overheads that dominate the function execution time, which hinders their adoption for short duration functions. A virtualization construct with minimal platform overhead that can satisfy the requirements of low latency functions while still offering the security and isolation of virtualization technologies is needed. In this thesis, we develop Ataru, a low-latency serverless platform that supports the execution of a DAG of functions. Ataru is designed to minimize the platform overhead by enabling the threaded execution of functions and efficient memory management. Since Ataru is designed for short functions, it executes the DAG within the same worker instance, avoiding overheads incurred in existing serverless platforms that use message queues or external databases for DAG realization. Ataru also solves the memory deduplication problem in virtual machines by delegating the memory management task to the VMM, thus allowing code sharing across worker instances and improving memory utilization. To perform a fair comparison, we implemented the fastest possible implementation to execute a DAG in Firecracker, a state-of-the-art serverless platform while still maintaining address space isolation offered by process abstraction. We have shown that Ataru outperforms Firecracker in terms of platform overhead, DAG execution time, and vCPU management. We have also demonstrated the performance of Ataru on synthetic benchmark and three real-world applications that require low latency and high throughput.

5.2 Achievements

We accomplish the following objectives in this dissertation

1. Implemented a low-overhead virtualized platform with multiprocessor capability. The

platform consists of two components: a Ataru-KVM VMM and Ataru Runtime. Ataru-KVM VMM is specifically tailored to the Ataru Runtime as a guest that can orchestrate function execution according to the DAG

2. Implemented an architecture for executing functions in a process abstraction; thus allowing for address space isolation of functions

3. Addressed memory deduplication problem by enabling the Ataru VMM to mmap the functions in the guest physical address space.

4. Addressed the LWP-LHP problem by designing a scheduler that halts and kicks the vCPU as needed during the execution of the DAG. The solution to inherent race conditions discussed in the section 3.2.9 is resolved using a recovery-based approach

5. Designed a custom dynamic linker to resolve the symbolic links when linking against a library for Ataru

6. To perform a fair comparison we designed the best implementation possible in Firecracker for executing a DAG while still preserving the isolation of the process abstraction.

7. Compared the performance of Firecracker and Ataru for various benchmarks and applications

8. We demonstrate the performance superiority of our platform Ataru over Firecracker in the function execution duration regime of few tens of microseconds.

5.3 Future Work

Currently, Ataru processes one request at a time, which causes a hard vmexit for each request; this can be optimized by implementing an asynchronous communication mechanism between Ataru-KVM VMM and Ataru Runtime, similar to virtio networking, which will enable concurrent processing of multiple requests and reduce the vmexit overhead. Another possible optimization for the boot-up time is to use snapshot of the vCPU state and reuse it for each instantiation avoiding the initialization cost of the vCPU for each new DAG. Some potential directions for future research are live Ataru migration and extending Ataru to support stateful function execution. Live Ataru migration would enable moving Ataru instances across hosts without disrupting the service. Stateful function execution would allow functions to preserve their state across invocations and enable more complex application scenarios.

Bibliography

[] Firecracker Guest Config guest configs. https://github.com/firecracker-microvm/firecracker/tree/main/resources/guest_configs. Accessed: 2023-05-03.

[] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged {CPU} features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, 2012.

[] Swarnava Dey and Arijit Mukherjee. Robotic slam: a review from fog computing and mobile edge computing perspective. In *Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services*, pages 153–158, 2016.

[] Vincenzo Eramo, Alessandro Tosti, and Emanuele Miucci. Server resource dimensioning and routing of service function chain in nfv network architectures. *Journal of Electrical and Computer Engineering*, 2016, 2016.

[] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 225–236, 2019.

[] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor:{Intra-Process} isolation for {High-Throughput} data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, 2019.

[] Devon Hockley and Carey L. Williamson. Benchmarking runtime scripting performance in webassembly. 2022.

[] Guoqiang Hu, Wee Peng Tay, and Yonggang Wen. Cloud robotics: architecture, challenges and applications. *IEEE Network*, 26(3):21–28, 2012. doi: 10.1109/MNET.2012.6201212.

[] Justin Hu, Ariana Bruno, Brian Ritchken, Brendon Jackson, Mateo Espinosa Zarlenga, Aditya Shah, and Christina Delimitrou. Hivemind: A scalable and serverless coordination control platform for uav swarms. *ArXiv*, abs/2002.01419, 2020.

[] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of {WebAssembly} vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating {Function-as-a-Service} workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820, 2021.

[] Daniel M. Lofaro, Arvin Asokan, and Edward M. Roderick. Feasibility of cloud enabled humanoid robots: Development of low latency geographically adjacent real-time cloud control. *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 519–526, 2015.

[] Pedro García López, Aleksander Slominski, Michael Behrendt, and Bernard Metzler. Serverless predictions: 2021-2030. *ArXiv*, abs/2104.03075, 2021.

[] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[] Pankaj Mendki. Evaluating webassembly enabled serverless approach for edge computing. *2020 IEEE Cloud Summit*, pages 161–166, 2020.

[] Kenji Nishimiya and Yuta Imai. Serverless architecture for service robot management system. *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11379–11385, 2021.

[] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE*

37th International Conference on Distributed Computing Systems Workshops (ICDCSW), pages 395–400. IEEE, 2017.

[] Tobias Pfandzelter and David Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24. IEEE, 2020.

[] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A scalable low-latency serverless platform. *arXiv preprint arXiv:1911.09849*, 2019.

[] Byounghee Son, Euiseok Nahm, and Hagbae Kim. Voip encryption module for securing privacy. *Multimedia tools and applications*, 63:181–193, 2013.

[] Andrea Tomassilli, Nicolas Huin, Frédéric Giroire, and Brigitte Jaumard. *Energy-efficient service chains with network function virtualization.* PhD thesis, Inria Sophia Antipolis; Université Côte d'Azur; Cnrs; Concordia University, 2016.

[] Nicholas C Wanninger, Joshua J Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C Hale. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 644–662, 2022.