Reducing Hypervisor Overhead for Virutal Interrupt Delivery in KVM/ARM Platform

Roja Eswaran

October 24, 2023

Abstract

With the recent hardware support for virtualization, low power consumption, low cost, the ARM architecture is getting a lot of attention from the embedded, server, and networking market. Unlike x86 architecture, ARM SoC doesn't have the sophistication of MRIOV, SRIOV, VT-d so the I/O virtualization overhead is very evident. The I/O virtualization overhead comes from two sources: DMA setup and Payload copy, interrupt delivery. we intend to reduce the overhead due to interrupt delivery to reap the virtualization hardware benefits the ARM platform has to offer.

our work focus on design, implementation, evaluation of KVM based Direct Interrupt Delivery (DID) in ARM to deliver timer interrupts and interprocessor interrupts(IPI) directly to VM without hypervisor's intervention thus significantly reducing the virtualization overhead caused to interrupt delivery.

1 Introduction

With the increase in the use of ARM platforms in edge, IoT, mobile, server, and networking areas due to its recent hardware support for virtualization capabilities, it's very important to analyze the virtualization overhead to complement the added hardware features. The main source of virtualization overhead comes from CPU, Memory, and I/O workloads. The main source of I/O virtualization overhead comes from two sources: setting up DMA operations and copying DMA payloads and interrupt delivery overhead. As of now, ARM architecture doesn't support SRIOV and MRIOV, Our interest is to reduce I/O virtualization overhead which comes from interrupt delivery.

DID on the ARM platform aims to deliver the interrupts directly to the target VM without intervention by the hypervisor. Our goals include delivering all interrupts to targeted running VM from sources such as the timer, IPI, external devices directly. There should be no VM exits when the interrupts are delivered to VM.

We should take advantage of ARM's hardware support to achieve our goals. First, Interrupt Translation Service (ITS) is used to route the physical and virtual interrupts from external physical devices to the targeted VM in the form of Locality-Specific Peripheral Interrupts (LPI), see Section 2.6. Second, we are still investigating on direct delivery of timer interrupts and IPI to the targeted VM.

This is the first work to focus exclusively on direct interrupt delivery in the ARM Platform. The previous work [4–6] optimized KVM according to added hardware virtualization features such as CPU Mode extensions, Virtualization Host Extensions (VHE), Virtual Generic Interrupt Controller(VGIC). Especially with VHE [2], ARM shows superior VM performance versus x86 [4]. Previous DID work on x86 [11] reduces VM exits by a factor of 100 and decreases the interrupt invocation latency by 80 percent. Hence, it's evident that DID in ARM would be an excellent performance booster to the architecture.

2 Background

In this section, we provide brief background on ARM's virtualization hardware support features and I/O virtualization overhead.

2.1 Exception Levels

The x86 platform has two privilege levels called user and kernel level across Intel's root and Non-root mode. In ARM, Privilege levels AKA exceptions levels have separate hyp mode(EL2) which has its own set of features more privileged than previous user mode (EL0) and Kernel-mode (EL1) to take care of underlying hardware. EL3 is more privileged than EL2 where the firmware runs, see Figure 1. The older KVM/ARM design [5] split hypervisor called KVM-Split Mode, across two different exception levels EL1 and EL2. The lowvisor on EL2 does very minimal processing and delegates the bulk of work to highvisor which runs on EL1 along with the host kernel as it leverages existing

Linux functionalities such as scheduler, interrupts, timer related functions to avoid redundancy. As a result, when there is an interrupt, VM running on EL1 exits and returns to lowvisor which then enters to highvisor to execute the appropriate interrupt handler function and returns to lowvisor again. Finally, lowvisor injects the virtual interrupt to VM and resumes VM execution. Splitting hypervisor across two exception level doubles the context switch rate. This problem is solved with the introduction of hardware support - Virtualization Host Extensions (VHE) [2] where Host Kernel and Hypervisor run on same exception level - EL2, thus reducing overhead due to context switch.

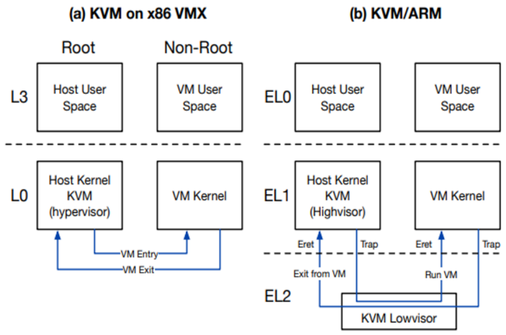

Figure 1: Exception Levels x86 VS ARM

2.2 Interrupt Types

In the ARM platform, Interrupts AKA exceptions can be either wire-based signals or Message Signaled Interrupts (MSI/MSI-X) [1]. Private Peripheral Interrupt (PPI), private to a specific core is wire-based interrupts. PPI on one core is not visible to others, so the same interrupt numbers can be used across different cores, for example, Timer Interrupts. Shared Peripheral Interrupts (SPI) can be either wire-based or message-based peripheral interrupts, it's up to the distributor, see Section 2.3, to decide the target core. SPI and PPI only differ in their allocation of interrupt numbers by having a single interrupt line per SPI, there is a line per PPI per processor. Software Generated Interrupts (SGI) are mainly used for inter-processor communication and can be generated by write operation on SGI register in the distributor. Finally, Locality Specific Peripheral Interrupts (LPI) is message-based peripheral interrupts that can be generated with the support of Interrupt Translation Service (ITS) or by writing on GICR-SETLPIR register in Redistributor.

2.3 Generic Interrupt Controller

ARM platform uses Generic Interrupt Controller (GIC) for interrupts. GIC has three components, see Figure 2. 1) Distributor - holds all routing and priority information and provides routing configuration for SPI. 2) Redistributor provides configuration setting for PPI or SGI and each core has its redistributor. Redistributor presents the CPU interface with the highest priority pending interrupts. 3) CPU interface - per core has its own EOI and ACK register for acknowledgment and deactivation of interrupts. Like APICv support for virtualization in x86, ARM's GIC has an additional component called virtual CPU interface which allows VM to access interrupt controller registers directly without trapping to the hypervisor. But delivering the interrupt involves hypervisor in writing the list register on virtualization control interface to register pending virtual interrupt for a corresponding physical interrupt.

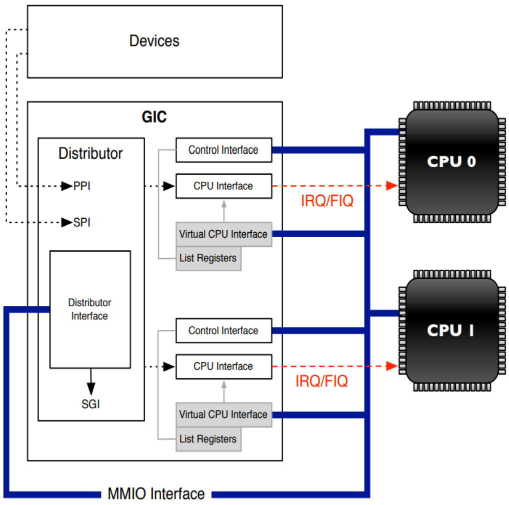

Figure 2: Generic Interrupt Controller with Virtualization hardware support

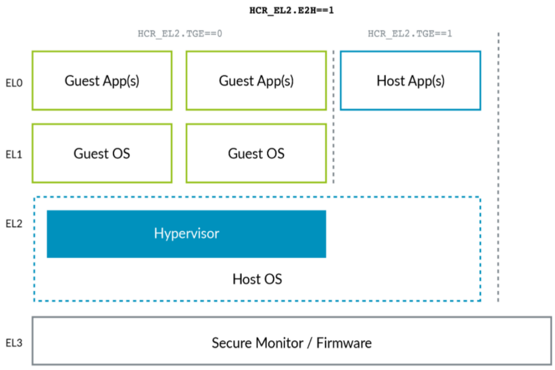

Figure 3: Host Kernel runs along with Hypervisor in EL2 with VHE support

2.4 Timer Virtualization

First, in x86, whenever the guest attempts to program the timer for the next event, the hypervisor takes control and we have VM exit. Unlike x86, the ARM platform provides a dedicated virtual timer for EL1, thus the guest can read/write its timer register without the hypervisor's intervention. But the expiration of the virtual timer generates a physical interrupt that passes the control to the hypervisor, so we have VM exit when the interrupt is delivered. Second, in x86, when the interrupt handler completes, VM tries to write on the EOI register which leads to another VM exit. ARM platform dedicates a separate EOI register for VM to deactivate interrupts without trapping to the hypervisor.

2.5 Virtualization Host Extension

Traditionally host kernel runs on EL1 and to have virtualization control, EL1 should depend on the hypervisor running on EL2 causing additional context switches. The problem is solved by VHE, see Figure 3 which allows the unmodified host OS to run along with the hypervisor. VHE is controlled by Hypervisor Control Register (HCR-EL2). E2H bit in the register decides whether VHE can be enabled or not during boot-time. EL0 is now shared between guest and host applications, EL2 uses TGE bit in the control register to decide whether the application belongs to host/guest to allot them appropriate resources. When the interrupt arrives while executing VM, the hypervisor takes control causing VM exit. In a traditional system without VHE, the hypervisor needs to enter EL1 to run interrupt handlers but in VHE such overhead is eliminated. without VHE there are at least four context switches between VM exit and VM entry, but with VHE it is reduced to two.

2.6 Locality Specific Peripheral Interrupt

LPI interrupts are very different from the other interrupt types in its life cycle and model. External device interrupts can be delivered as LPI with the help of ITS or direct write on one of the re-distributors register. In x86, the posted interrupt mechanism help in delivering virtual interrupt directly to VM without hypervisor's intervention, similarly in ARM LPI is used for the same effect. Direct delivery of virtual interrupt is supported from GIC version 4, only LPIs can be routed with ITS, the other interrupt types(PPI, SPI, SGI) don't have any effect on ITS, see Figure 4. ITS being the critical component of LPI comprises of additional data structures in the memory such as Device Table(DT), Event Table(ET), Interrupt Translation Table(ITT), Collection Table(CT), and Virtual Processing Element Table(vPET) all initialized before enabling ITS by the software. The arrived interrupt source has device ID and Event ID associated with it. Each DT provides Device Table Entries (DTE) indexed using device ID returns the base address points to ITT in the memory. The ITT is a collection of Interrupt Translation Entries (ITE) indexed using event ID defines physical and virtual interrupts. For physical interrupts, ITE returns 1) output physical Interrupt ID (pINTID), 2) ICID for identifying entries in CT to determine the target core to which the physical interrupt must be delivered. For virtual interrupts, ITT returns 1) output virtual interrupt ID (vINTID), 2) vPEID that identifies the entry in vPET to determine the target redistributor which then gives target core, 3) a doorbell to use if the vCPU is not scheduled.

3 Related Work

To the best of our knowledge, no one has tried direct interrupt delivery on the ARM platform before. In x86, DID work [11] removes most of the

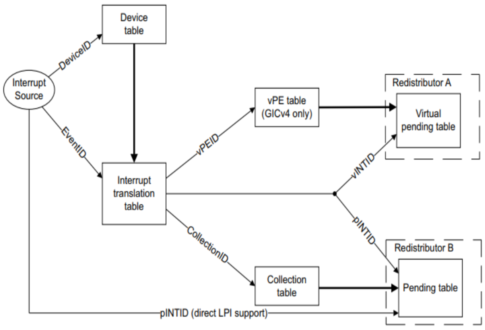

Figure 4: Physical and Virtual LPI Delivery Flow

VM exits due to interrupt dispatches and EOI notification for SRIOV devices, para-virtualized devices, and timers. They achieve this by leveraging IOAPIC's interrupt remapping table to avoid misdelivery of direct interrupts and employs a self-IPI mechanism to inject virtual interrupts, which enables direct EOI writes without causing priority inversion among interrupts.

Directvisor [3] which leverages intel's posted interrupt mechanism to deliver timer interrupts and IPI directly to VM without hypervisor's intervention. Physical interrupts always cause VM exits to let hypervisor gain control over the hardware resources. For timer interrupts, directvisor configures local APIC to deliver posted interrupt notification vector instead of the regular timer interrupt vector when the timer expires. Directvisor also disables VM exits due to read/write operations to local APIC's initial count register, divide configuration register, and HLT instruction. It also removes emulation overhead due to HLT, being the sensitive instruction, and lets the guest idles on the physical core directly. For IPI, directvisor removes VM exit due to write in per-CPU interrupt command register and as it dedicates each guest with dedicate core for vCPU, vCPU directly sends IPI to each other without hypervisor's emulation with the help of posted interrupt notification, see Figure 5.

The ARM platform doesn't have sophisticated hardware and extremely different from x86, we believe our work would have a major impact in reducing overhead in I/O interrupt delivery path.

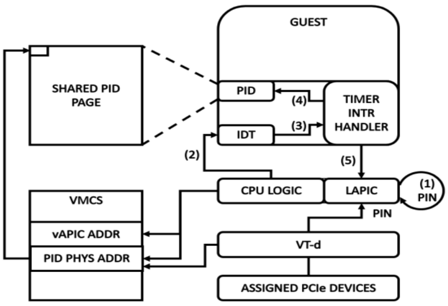

Figure 5: Delivering Timer interrupt and IPI directly with VT-d

4 Proposed Direct Interrupt Delivery Scheme

4.1 Overview

In ARM Platform, we don't have to worry about VM exits due to timer register read/write and EOI, ACK operations as VM has required hardware support to achieve above operations without hypervisor's intervention. The big challenge is delivering the interrupts to target VMs without getting caught to hypervisor. It's important to analyze interrupts from three sources, interrupt from external devices, emulated or virtual devices and local interrupts. In this section, we will discuss how these interrupts can be delivered with zero or minimum VM exits.

4.2 Disabling VM exits and Polling overhead due to WFI

The traditional polling overhead due to Wait for Interrupt (WFI) emulation by the hypervisor is 164us. Whenever vCPU is expecting any interrupts, it executes WFI to get into the low power or idle mode. As WFI is privileged instruction, it causes a trap to EL2. Now hypervisor emulates WFI instruction by continuously polling on physical core and wakeup vCPU once new interrupt arrives. If we dedicate a physical core for a vCPU, disable VM exits on WFI and let the physical CPU idles in EL1 context, we can avoid VM exit and polling overhead by the hypervisor.

4.3 External Device Interrupt

For external devices such as NIC, even though ARM shows the same bandwidth rate for Bare-Metal and VM, the CPU utilization is very high in VM(see Section 5). As ARM has the support of IOMMU, one way to reduce such overhead is with device passthrough of NIC to VM with VFIO drivers. But VFIO drivers get along only with PCI devices to achieve the passthrough. The Jetson, being the embedded SoC integrates a Realtek RTL8211FDI Gigabit Ethernet controller. The on-module Ethernet controller that supports 10 or 100 or 1000 Gigabit Ethernet, unfortunately, is not a PCI device [8]. As VFIO device platform driver and QEMU/VFIO doesn't support physical embedded device especially from Realtek for now, passing through NIC to VM is not attainable in Jetson. Traditional interrupts from NIC are delivered in the form of SPI which involves VM exits during the interrupt delivery, see Figure 6 . Whenever the packet arrives, NIC asserts physical interrupt(1), GIC delivers SPI to the VM's vCPU running on physical cores(2) which causes a VM exit. The hypervisor(3) then acknowledges the physical interrupt that makes the interrupt active and inserts virtual interrupt to the list of pending interrupts for the target vCPU(4). When this virtual interrupt has sufficient priority, the hypervisor writes into the list register - vINTID and pINTID and HW bit to tie virtual interrupt with physical interrupt. When vCPU is running, it takes the pending interrupt from the virtual CPU interface(5) and executes the corresponding interrupt handler function. Once vCPU completed the handler function, it deactivates the virtual interrupt and tied physical interrupt by writing on EOI register(6).

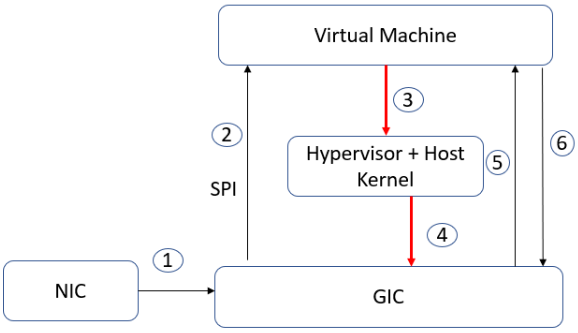

Figure 6: NIC delivers interrupt in the form of SPI

LPI along with ITS helps us achieve direct delivery of interrupt to target VM, if VM is not running the control is passed to the hypervisor to execute the doorbell interrupt. Any interrupts other than LPI should get trapped to hypervisor while delivering to VM.By delivering interrupts from NIC as LPI instead of SPI, we can completely remove hypervisor from critical path, see Figure 7.

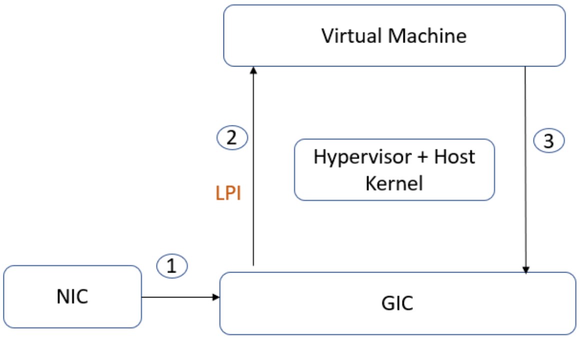

Figure 7: NIC delivers interrupt in the form of LPI

4.4 Timer Interrupt

Timer interrupts are critical for all the process as it determines how long the process owns CPU, once the process expired its time slice, it gives up CPU, OS should program the timer chip to make sure the process doesn't exceed their time slice for upcoming events. The big challenge in ARM is delivering the timer interrupt without getting caught by the hypervisor. The timer interrupt can be taken place either in the form of PPI or SPI, see Figure 8. The guest can program its virtual timer register without trapping(1). Once the virtual timer expires, a physical interrupt is generated in the form of SPI or PPI(2). The physical interrupt causes VM exit(3) and traps to the hypervisor. The hypervisor then determines the interrupt belongs to VM and program the corresponding virtual CPU interface with vINTID and pINTID(4). When the target VM is scheduled, VM acknowledges the pending virtual interrupt and starts executing the interrupt handler(5). Once VM finished its task, it deactivates both virtual and underlying physical interrupt by writing on EOI register on virtual CPU interface(6). Our kernel-level latency test, see Section 5, shows that the invocation latency of VM is four times higher than what we experienced with Bare-Metal so, we are still investigating the efficient way of delivering timer interrupt without hypervisor interference.

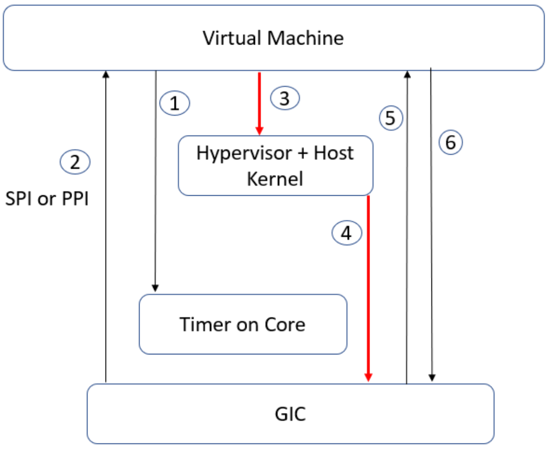

Figure 8: Timer Interrupt delivered as SPI or PPI

4.5 Virtual Device Interrupt

Two cores communicate with each other with the help of IPI to maintain coherency in states. In ARM, SGI is used for generating IPI by writing on the SGI register on Redistributor. To generate SGI targeting on EL2, write must be performed on ICC-SGI1R-EL1 register. The interrupt Routing Mode(IRM) field in the SGI register gives the routing information for the CPU interface. Setting IRM-1: interrupt is sent to all the cores except for the originating core, IRM-0: interrupt is sent to aff3.aff2.aff1.Target List, where the target list is encoded as 1 bit for each affinity 0 nodes under aff1. This means that the interrupt can be sent to a maximum of 16 PEs, which might include the originating PE. Once the SGI is generated by the redistributor, it has to again go through the distributor to reach the target core.

The interrupts generated by virtual devices are known as virtual interrupt. Figure 9 shows the IPI delivery between vCPUs. Let's consider a VM with two vCPU, vCPU 0 and vCPU 1, VCPU 0 wishes to send IPI to vCPU 1. When vCPU 0 attempts to write on the SGI register(1) there is a trap to the hypervisor(2). The hypervisor emulates a virtual distributor which holds all the routing related information and returns to VM(3). The virtual distributor then writes to list register(4) of the CPU on which the target vCPU 1 runs. The vCPU 1 should see the virtual IPI when it is scheduled to run next time.

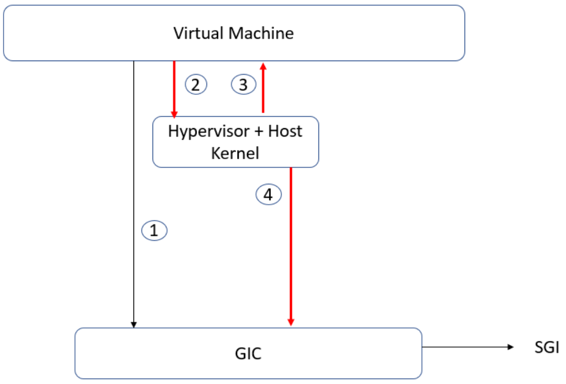

Figure 9: SGI delivery between two vCPU

Figure 10 shows pCPU 0 sending IPI for vCPU 1 that runs on pCPU 1 for maintenance purposes. CPU 0 sends physical SGI targeting CPU 1. The physical interrupt always causes VM exit(1) and the control is passed to the hypervisor. The hypervisor realizes the interrupt belongs to vCPU 1 and writes on the virtual CPU interface of CPU 1(2). When the vCPU 1 is scheduled again, it should receive virtual IPI when the priority hits(3). In this case, there is no need for virtual distributor emulation as CPU 0 holds control over physical GIC yet hypervisor is still involved in both cases. We are still investigating the effective way to achieve IPI delivery without any emulation or hypervisor overhead.

5 Performance Evaluation

5.1 Evaluation Methodology

As the ARM servers available on market are very expensive, we decided to conduct our experiment on NVIDIA's Jetson Xavier NX [9]. Jetson has 6-core NVIDIA Carmel ARMv8.2, 64-bit CPU, 6 MB L2 + 4 MB L3, and also has the support of

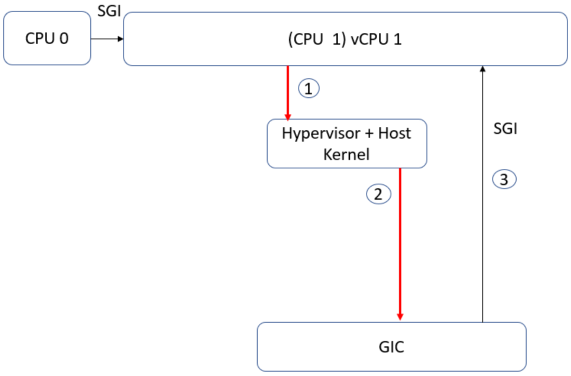

Figure 10: SGI delivery between two pCPU and vCPU

VHE, gigabit ethernet. The Host kernel is the customized NVIDIA's Jetpack of version 4.9.140-tegra and the guest runs Vanilla VM of kernel version 5.9.9. We disabled GUI in both host and guest, all the 6 cores must be running at max frequency as GUI and on-demand CPU scaling governors reported higher timer interrupt latency values. We tested Network Performance using ping and iperf. To evaluate timer interrupt latency, we created our kernel-level timer interrupt latency test [7] using high-resolution timer [10].

5.2 Network Performance

We used iperf 2.0.10 to measure the network bandwidth. The server is Raspberry-pi 4 model B which has gigabit ethernet support, the client is Jetson, both are connected using LAN which supports 1Gbps. While running as Bare-Metal or VM , Jetson shows the same bandwidth of 942Mbps but the CPU utilization measured using mpstat shows Bare-Metal has 7.6% and VM utilized 25.4% which is almost as three times as Bare-Metal.

Ping tool is used to measure the network latency. With the same experimental setup, Bare-Metal has average latency of 0.509 ms while VM has 1.632 ms almost as twice as Bare-Metal.

5.3 Kernel-Level Timer Interrupt Latency Test

Host and Guest have their dedicated core for the latency test to avoid interference from the scheduler and hypervisor. The timer interrupt latency test was run for 20 seconds with an inter-interrupt interval of 200 us. We plotted our output latency values with cumulative distribution function, the median timer interrupt latency value for bare-metal and VM is 3.9us and 14.8us respectively. These values substantiate our problem statement and eliminating the factors that cause this overhead in interrupt delivery would significantly improve the performance of VM, see Figure 11.

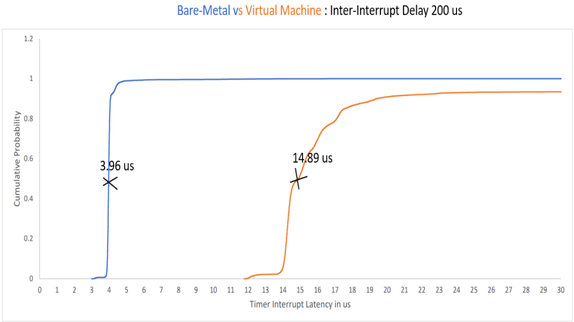

Figure 11: Kernel Level Timer Interrupt Latency Test values plotted against Cumulative Probability

6 Conclusion

We have thoroughly studied the state-of-art interrupt delivery mechanism, but we haven't found any work reducing the delivery overhead due to I/O virtualization. The groundwork on literature and our experiment on Jetson helps us understand there exists a real problem with interrupt delivery mechanism even though the ARM platform has various virtualization hardware features. We believe our work would have a potential impact on ARM platforms as embedded SoCs can't afford to interrupt latencies while running critical real-time tasks.

References

[1] Arm Limited. Exception Types ARM. https://developer.arm.com/architectures/learn-the-architecture/exception-model/exception-types.

[2] Arm Limited. Virtualization Host Extensions-ARM. https://developer.arm.com/

architectures/learn-the-architecture/aarch64-virtualization/virtualization-host-extensions.

[3] Kevin Cheng, Spoorti Doddamani, Yongheng Li, Kartik Gopalan, and Tzi-Cker Chiueh. Directvisor: Virtualization for bare-metal cloud. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, New York, NY, USA, 2020. ACM.

[4] Christoffer Dall and Shih-Wei Li and Jason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *2017 USENIX Annual Technical Conference (USENIX ATC '17)*, page 304–316, 2017.

[5] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. Arm virtualization: Performance and architectural implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*, page 304–316, 2016.

[6] Christopher Dall and Jason Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 333–348, 2014.

[7] Kevin Cheng and Roja Eswaran. Kernel Level Timer Interrupt Latency Test. https://github.com/osnetsvn/Latency_Test.git.

[8] NVIDIA Limited. Jetson Product Datasheet. https://developer.nvidia.com/jetson-xavier-nx-product-design-guide-v10/.

[9] NVIDIA Limited. Jetson with Virtualization Host Extensions. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[10] Thomas Gleixner. High Resolution Timer. https://lwn.net/Articles/167897/.

[11] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 1–15, New York, NY, USA, 2015. ACM.