arm

A layered approach to high performance device virtualization

Revere-AMU System Architecture

Architecture and Technology Group
February 2019

Version B

Contents

Non-Confidential Proprietary Notice

LES-PRE-20349

Release information

Date	Version	Changes
2018/Aug/04	A	First version
2019/Feb/01	B	Public version

Executive Summary

This document proposes a layered system architecture, enabling an efficient and smooth integration of IO and accelerator devices in a virtualized environment.

After providing an overview of various existing and future device driver use models that enable both scalability and performance, this paper explains the need for standards – various layers – to ensure the development of an ecosystem (software and hardware re-use). Finally, this white paper describes 'Revere-AMU', an example architecture that implements all the identified layers and a prototype implementation of the architecture.

This document is aimed at Systems and System-on-Chip (SoC) architects, device and accelerator architects, device drivers and Operating Systems (OS)/virtualization frameworks developers.

Arm aims to enable the development of standard IO virtualization frameworks. Arm is seeking feedback on the current layered proposal and is open to discussion and collaboration.

Introduction

Computing systems increasingly require devices, such as accelerators and I/O, that offload computationally expensive operations or interface outside of the system.

These devices need a defined interface to software: the hardware/software interface. The design of a hardware/software interface is critical for performance and for ease of software deployment.

Current trends are significantly increasing the complexity of a state-of-the-art hardware/software interface. For example, virtualization has brought significant economic benefit to the cloud by enabling multi-tenancy, and similar benefits are now being translated to other markets, such as networking and automotive. In spite of the benefits, virtualization adds significant complexity, for example if the device must service multiple isolated virtual machines.

Optimizing performance with device assignment

Optimal performance requires appropriately minimizing the software overhead for accessing the device. Minimizing the access latency is especially important for small tasks (fine grained offload).

Device assignment means that the software using the device directly reads/writes memory mapped registers on the device, and/or directly reads/writes data structures in memory that are accessed by the DMA facilities of the device. One of the benefits is performance since the system call overheads are removed. In the case of virtualization, device assignment to a virtual machine is often required for performance.

Reducing software cost and the need for standards

The advent of device assignment means that the volume and complexity of software impacted by a hardware/software interface design is increasing. The development cost of this software is often dominant when bringing an innovative new System-on-a-Chip to market. Standardization in the hardware/software interface is fundamental to reducing software cost and time-to-market. Existing standards are available and important (notably the PCI Express software architecture and associated standards), but they are not complete in light of current and upcoming use-cases.

A further benefit of device assignment is simplified deployment. When the device does not provide services to the operating system, a fully standard software framework can handle the device assignment without the need for a driver.

In the case of virtualization, device assignment in a standard way enables generic and standard code in the hypervisor.

Device driver models

PCI Express[1] provides a software architecture for device discovery and assignment. Devices – referred to as a PCI Express End Point – can be integrated on the same SoC (using only the PCI Express software architecture) or on a discrete chip (using other parts of the PCI standard). Recent versions of the PCI Express standard include the Single Root I/O Virtualization (SR-IOV) capability that provides efficient hardware support for a large number of *Virtual Functions* (VF) to support virtualization.

The PCI Express software architecture enables standard software frameworks that can be reused across many systems. For this reason, Arm's Server Base System Architecture[2] requires the use of PCI Express for device enumeration/discovery to enable interoperability.

In the traditional way of exposing devices to software, a single device driver has exclusive access to the physical device resources and provides access to other software using system calls. The benefit is that the driver fully abstracts the hardware/software interface, but at performance cost. When performance is important, a device can be directly assigned to the software that is using it.

[1] *PCI Express Base Specification*. (Rev. 4.0 Version 0.9) PCI-SIG.
[2] *ARM® Server Base System Architecture*. (DEN0029B 5.0)

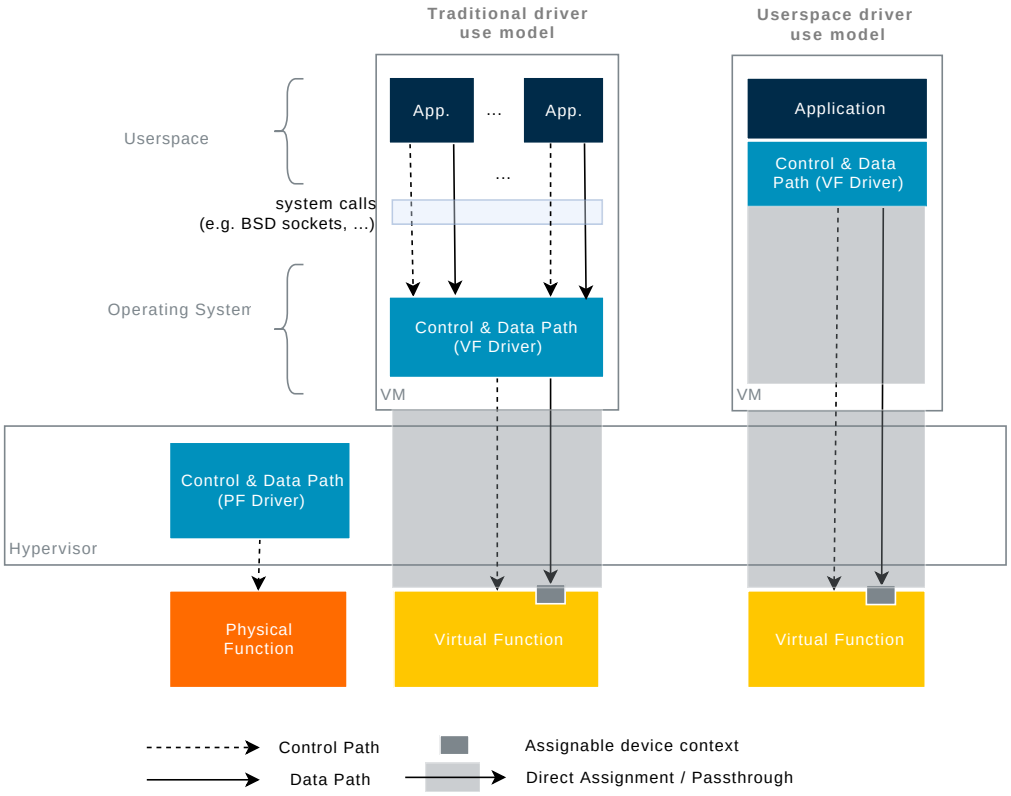

Figure 1: In virtualized environments, the traditional driver use model enables multiple applications to access the physical resources through system calls. With the userspace driver use models, an entire VF is assigned to userspace to provide near-native performance to one application.

Traditional driver use model

When the VF is assigned to the guest OS, the VF driver runs in the guest OS and userspace applications issue system calls to exercise the control and data paths. A system call API is defined per device class, for example, the Berkeley Software Distribution (BSD) sockets API for networking devices. This driver use model is referred to as **Traditional driver use model** in Figure 1. It provides scalability – multiple userspace applications access the device control and data paths – and software re-use but with a limited performance due to system call overhead.

Userspace driver model

In order to achieve a near-to-native performance, the VF is assigned to userspace applications (through generic, OS-provided mechanisms). The VF driver – now executing in userspace – has direct access to the control and data paths of the device. This model is referred to as **Userspace driver model** in Figure 1. The scalability of this model is limited since the granularity of assignment is an entire PCI Express VF and each VF can only be assigned once.

Control-mediated driver

Neither the traditional nor userspace driver models provide both performance and assignment scalability at the same time. In order to enable fine-grained assignment to userspace applications, an assignable *context* – smaller than a Function – is defined. The assignable hardware context provides the device data path. The mediating software provides the control path, including controls for assignment of the data-path. This is the **control mediated** device model.

There are several approaches for the mediation of the Control Path. Figure 2 describes an implementation in which the control mediation is realized in the guest OS. The Control Path driver (executed in the guest OS) handles the control path while the data path drivers (executed in userspace) have direct access to assignable *contexts*.

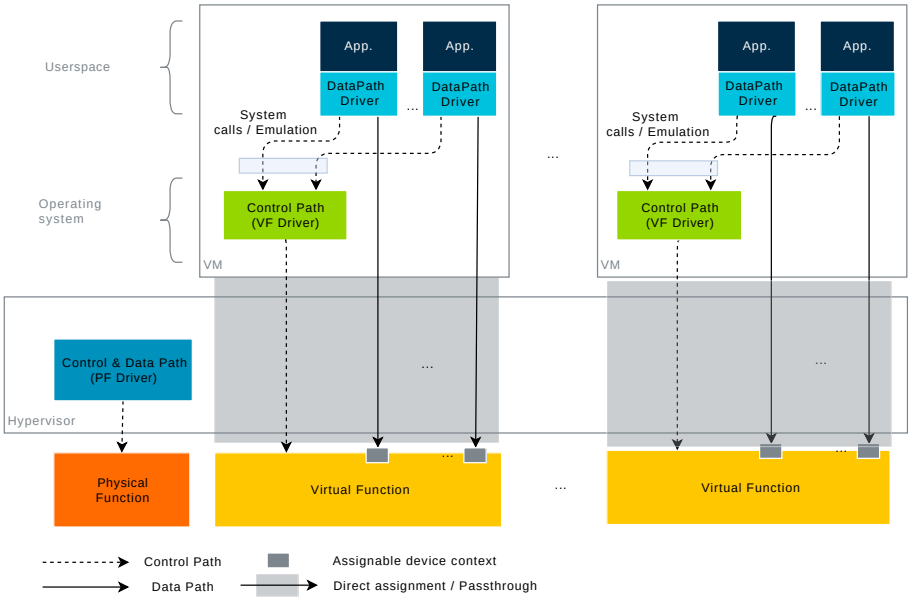

Figure 2: Control mediation in the guest OS with an entire (control and data path) Function assigned to the virtual machine.

PCI Express defines the Process Address Space ID (PASID) feature that enables sharing of a single Function across multiple processes while providing address space isolation. A context implements appropriately-aligned data path resources in the Function Memory Mapped IO (MMIO) space and uses PASID to configure a separate translation scheme per context.

Several enhancements to the control mediated model are possible:

- Contexts are dynamically mapped/unmapped to Functions. This provides increased flexibility in provisioning resources across the system. The mechanisms for doing the mapping are not defined by PCI Express so this must be done in a device specific way.
- Over-provisioning of contexts. A control mediated driver can over-provision the assignment of contexts to userspace software using a trap and emulate scheme. Over-provisioning enables scalability to an unlimited number of userspace clients, with performance cost.
- Standardization of the device context. Sufficient additional architecture would enable the assignment of a context by generic software. With a standard context, the control-mediated model shifts much of the device-specific functionality from the OS level to userspace.

Control mediation in the hypervisor

While the control mediated model provides both performance and scalability, the cost of the physical control interfaces – that have to be duplicated per VF – can be significant. For example, in cases with thousands of virtual machines and/or containers. At the same time, it might be acceptable from both a performance and security standpoint for a hypervisor driver to mediate on control path operations.

Hypervisor control mediation is shown in Figure 3. With this model, the VF driver uses either paravirtualization or full emulation of the control path in the hypervisor. This model provides a flexible and fine grained assignment capability while minimising the overhead of the VF. The VF is only used to control the translation scheme per VM.

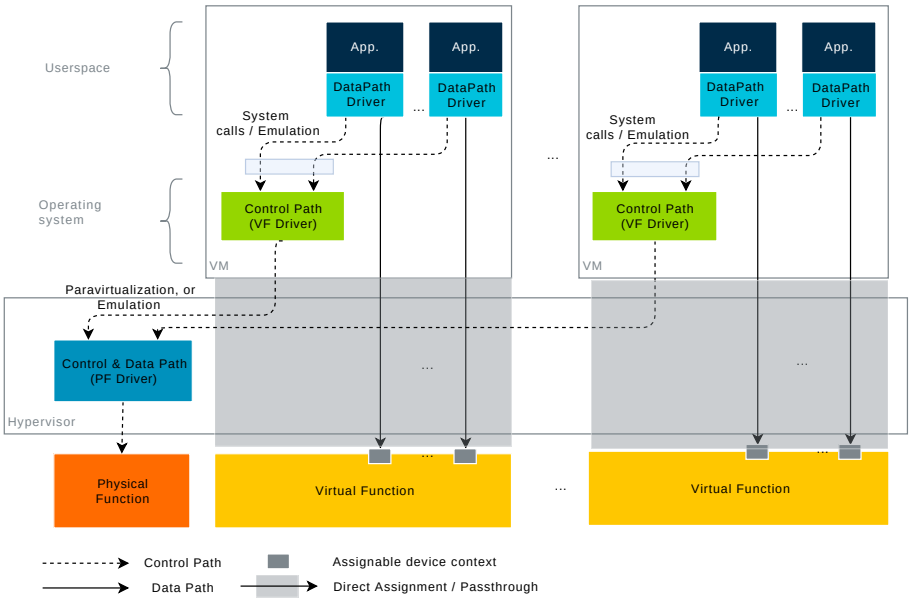

Figure 3: Control mediation in the guest OS with lightweight (no replicated control path) Functions assigned to the virtual machine.

Figure 4 shows an optimization to the control mediated model, described in the recently introduced Scalable IOV (S-IOV) specification.[3] S-IOV dispenses with the VF entirely by using the PASID to control the translation scheme for each VM but has not been incorporated back into the PCI standard.

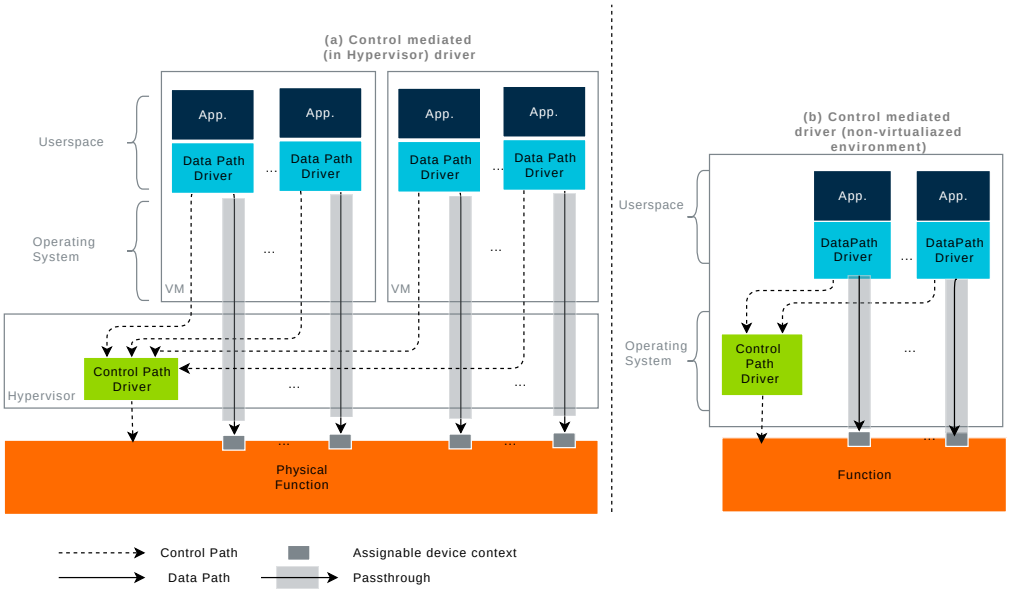

Figure 4: Fine grained assignable contexts used without SR-IOV in (a) virtualized environments and (b) non-virtualized environments.

[3] *Intel® Scalable I/O Virtualization*. (1.0 337679-001) Intel®.

All approaches described above present various characteristics that are summarized in the following table:

Table 2: Summary of the various driver use models in virtualized environments

Use models	Description	Scalability	Performance	Cost
Traditional driver model with SR-IOV	Entire VF is assigned to VM. Userspace accesses to the control and data path are mediated by a privileged driver	Scales with the number of processes	Low (system calls)	Control interface replicated per VF
Userspace driver model	Entire VF is assigned to VM and to userspace processes	Limited to the granularity of the VF	High (Near native performance)	Control interface replicated per VF
Control mediated (with VFs)	Control and data paths separated. Control path is mediated by privileged driver, data path contexts are directly assigned.	High scalability with fine grained assignment	High (Near native performance) for the data paths	Control interface replicated per VF
Control mediated (with lightweight VFs)	VF does not implement the control interface. Control is fully mediated. Data path contexts are directly assigned.	High scalability with fine grained assignment	High (Near native performance) for the data paths	No Control interface replicated per VF
Control mediated (without VFs)	Control is fully mediated. Data path contexts are directly assigned.	High scalability with fine grained assignment	High (Near native performance) for the data paths	No VF overhead

Requirements and the need for standards

A state of the art hardware/software interface must meet a large and growing set of requirements, summarized in Figure 5. Designing to meet these requirements is complex and the architect must study an increasing number of software stacks to ensure that the hardware/software interface design meets the assumptions of that software.

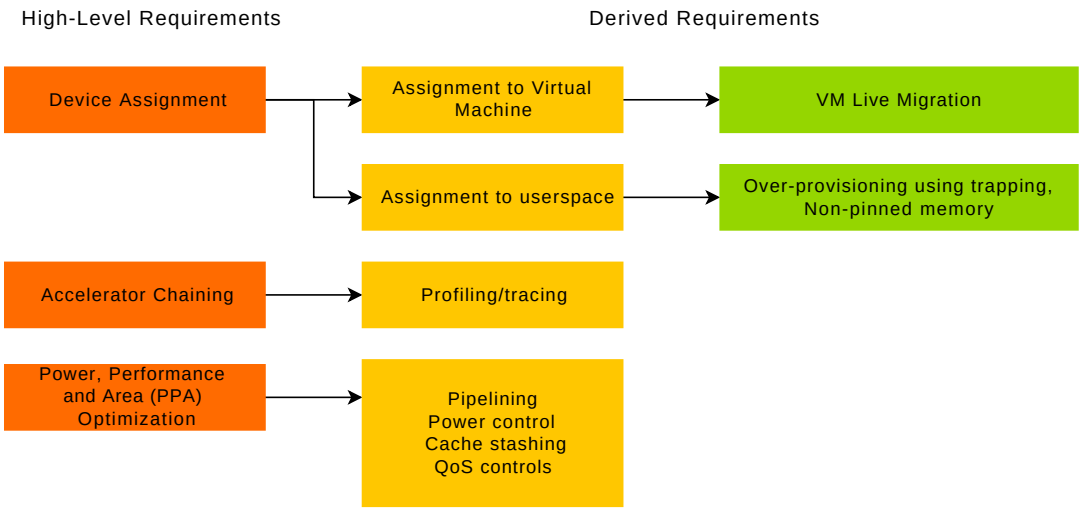

Figure 5: Requirements and derived requirements for a hardware/software interface

What existing architecture is available to avoid 'reinventing the wheel' each time?

For a small number of device classes, there is an existing standard to build to that meets these requirements. For example: the NVMe standard defines a complete hardware/software interface for a storage device.

For all other devices, the PCI Express[4] software model offers a starting point that is also a requirement in many markets. The PCI Express software model gives sufficient architecture for software to enumerate devices in the system and also perform device assignment at the granularity of a PCI *Function*.

Need for additional standardization layers on top of PCI Express

However, there is still a significant amount of work in designing an interface on top of PCI Express that meets the requirements shown in Figure 5, and writing the necessary enabling software. It is also critical to use the PCI Express features in the way that existing software expects.

This work is rarely differentiating and the prevalence of fragmented solutions creates an opportunity for new layers of standardization that substantially reduce the burden when creating a new device type.

[4] *PCI Express Base Specification*. (Rev. 4.0 Version 0.9) PCI-SIG.

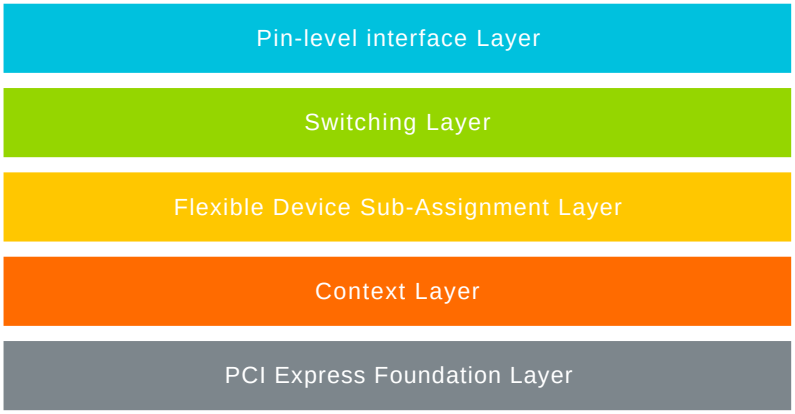

Figure 6: Set of standardization layers on top of PCI Express to enable low friction integration of accelerators and the implementation of standard software frameworks.

PCI Express Foundation Layer

The PCI Express software architecture enables standard enumeration and discovery of devices. Direct assignment of PCI Functions to a virtual machines or userspace driver does not require device-specific OS or hypervisor modifications.

The PCI Express software model and SR-IOV are widely supported by existing hypervisor, OS and other system software.

Flexible and Fine grained Device Sub-assignment Layer

While PCI Express and SR-IOV enables direct assignment of functions to software, recent market evolutions require more scalability and flexibility in the assignment schemes. For example, current systems must support thousands of VM/containers and require features such as resource over-provisioning and live migration.

An optional *Flexible and fine grained device sub-assignment Layer* defines an abstract device context and assignment controls to VMs and to userspace processes. The architecture at this layer specifies:

- an assignable device context, smaller than a PCI Express Function
- controls to support a flexible mapping of contexts to PCI Express Functions
- controls to manage assignment (for example: quiesce, reset, configure translation scheme)

The benefits are:

- support for a fine grained assignment scheme orthogonal to SR-IOV (can be used with or without SR-IOV)
- support a wide range of driver use models (non-virtualized/virtualized, mediated/control mediated drivers, ...)
- support for resource over-provisioning and VM live migration
- standard, generic software support for the above feature set.

Context Layer

An optional *Context Layer* defines the device context itself. Design of a context includes data structures to exchange data between the software and hardware, as well as signaling mechanisms (doorbell, interrupts, ...).

Many devices use a variation on a message queue abstraction: N message queues from the software to the hardware, and M from hardware to software. The depth of the queues is used to pipeline and hide the interface latency. A message contains an implementation specific command or response, and optionally pointers to buffers in shared memory.

An optional Context Layer defines:

- Efficient data structure exchange mechanism to exchange messages with devices
- Flexible message descriptor formats enabling performance optimizations (For example: cache stashing)

The benefits are:

- Standard software for communicating between software and hardware
- Performance for a standard message queue will be optimized across many implementations

For instance, an existing standard format (such as virtio's virtqueues) could be used as a Context Layer.

Switching Layer

The *Switching Layer* specifies mechanisms to establish communication between software and hardware contexts. Once a session between two contexts is established, messages can be exchanged. Sessions are established at run-time and can be dynamically changed by privileged software to support over-provisioning and live migration use cases. Another benefit of this layer is the support of direct message exchange between hardware contexts (without CPU intervention) known as accelerator chaining.

An optional Switching Layer defines:

- sessions that establish connections between contexts (hardware/software, hardware/hardware, software/software)
- controls of priorities to support QoS
- controls to generate cache stashing hints of the messages exchanged
- mechanisms to enable profiling and tracing

The benefits are:

- standard software for the establishment of sessions at run-time
- standard mechanisms enabling privileged software to implement dynamic load balancing, live migration and resource over-provisioning
- standard mechanisms enabling overall system performance optimizations (QoS, Cache stashing, profiling capabilities)
- standard approach to system observability (tracing capabilities)

Pin-level Interface Layer

The pin-level interface defines a bus protocol to enable a future hardware IP ecosystem of reusable components targeting different performance points and accelerators. The pin-level interface is completely invisible to software, making it entirely optional to implement the microarchitecture in this way.

An optional Pin-level Interface Layer defines:

- a bus protocol (including packet format, flow control scheme, stashing hints)
- a mechanism to enable an IO/accelerator device to initiate DMA transactions (qualified with the appropriate attributes to control the stage 1 and 2 address translations).

The benefits are:

- Standard and efficient bus protocol enabling the development of an ecosystem of standard IO and accelerator building blocks that can be re-used.
- Compliance with PCI Express Function requirements. For example, Bus Master Enable (BME) support.

Revere-AMU Architecture

Revere Accelerator Management Unit (Revere-AMU)[5] is a layered system architecture that implements all the layers described in the previous section. It provides a hardware and software framework to enable low-friction deployment of accelerators and I/O devices, in both embedded and server-class systems (such as SmartNICs, Crypto, or Neural Network accelerators).

PCI Express Foundation Layer

A Revere-AMU device is presented to software as a PCI Express[6] device, which enables the reuse of all the existing software infrastructure to discover, configure and assign PCI Functions.

Context Layer

The software interface – Accelerator Message Interface for software (AMI-SW) – corresponds to the smallest assignable context. A message abstraction is built on top of architected ring buffers allocated in shared memory. Software can notify the device when data is ready for consumption by writing to a memory mapped device register, commonly referred to as a doorbell. The device can notify software through MSI-X interrupts, specified by PCI Express. One AMI-SW can support variable numbers of TX and RX ring buffers that can be monitored through digest registers.

AMI-SWs can be independently reset and quiesced. Their state can also be saved and restored by software through the management interface, which enables use cases, such as over-provisioning and live migration.

Messages can contain both in-band data and referenced data through pointers in the descriptor. In-band data is useful to enable a message-based control path with accelerators, where message data is short-lived and bound in size. Referenced data enables use cases where data is generally large and needs to be allocated ahead. Figure 7 shows an example of a Revere-AMU message. Revere-AMU architects several message formats, which vary in terms of the message size, the amount of metadata that is transported and the features they enable.

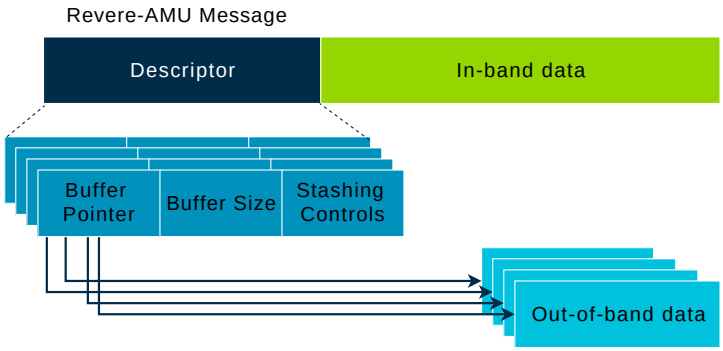

Figure 7: A Revere-AMU message contains an architected descriptor and, optionally, in-band data. The descriptor may contain a number of buffer pointers and associated stashing controls.

Flexible and Fine grained Device Sub-assignment Layer

A Revere Accelerator Management Unit (Revere-AMU) Function can be assigned to virtual machines by leveraging PCI Express SR-IOV capability. Revere-AMU contains a control path context (referred to as Management Interface),

[5] *Revere-AMU System Architecture*. (ARM IHI 0078A ALP-02) Arm Ltd.
[6] *PCI Express Base Specification*. (Rev. 4.0 Version 0.9) PCI-SIG.

and one or more data path contexts (referred to as AMI-SW). The data path context can directly be sub-assigned to userspace applications using PCI Express PASID capability.

Revere-AMU defines a standard way for the control path context in the PF to dynamically apportion data path contexts between Functions. This is shown in Figure 8.

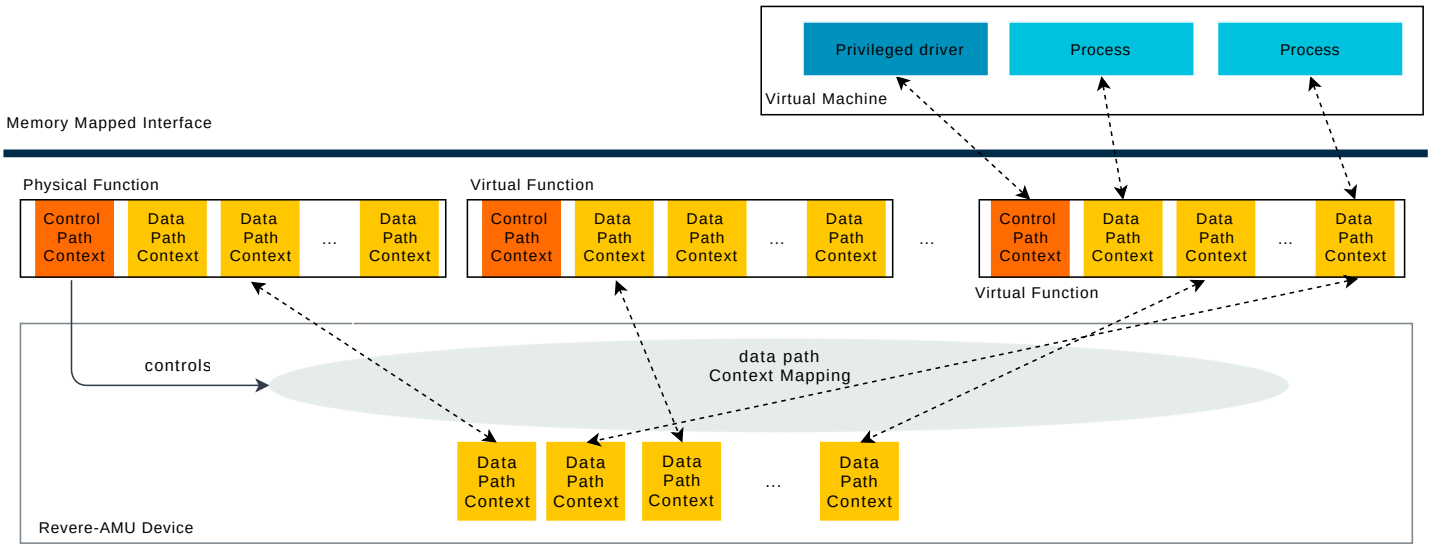

Figure 8: A Revere-AMU device is presented to software as a PCIe device. It contains a single Physical Function and an arbitrary number of Virtual Functions, exposing identical interfaces to software. A Function has a single instance of a control path and may have multiple instances of a data path that can be sub-assigned to userspace applications. The mapping of data path contexts to Function is dynamic and under the control of the PF driver

Device configuration through the Management Interface

Each Revere-AMU Function exposes a management interface, which comprises a set of discovery/control/configuration registers and a set of configuration queues. The management interface is used to control the configuration of the device. While the Physical Function has access to the global configuration, such as the establishment of connections between contexts or AMI mappings, Virtual Functions are limited to accessing per-VF state, such as the configuration associated with any AMIs mapped to them. Figure 9 summarises the controls exposed by the management interface.

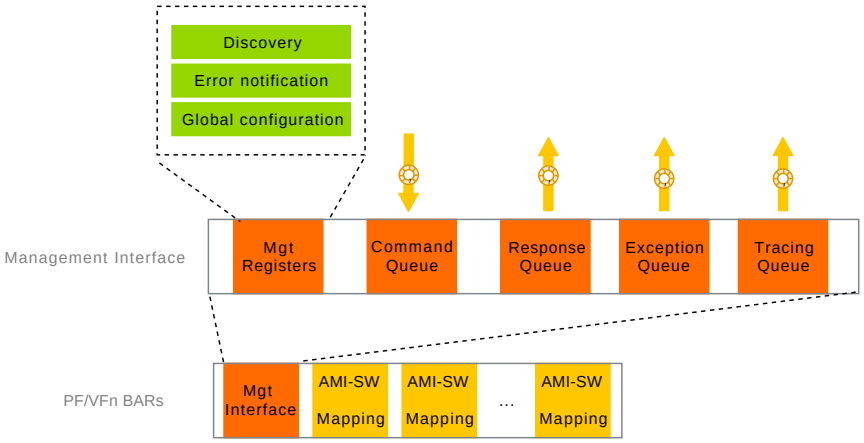

Figure 9: Each Revere-AMU Function contains a management interface, comprising a set of memory mapped registers and a management AMI. A management AMI has four queues associated with it to request configuration changes and enable other features such as tracing and exception notifications. While the PF can modify the global configuration of the device, VFs are limited to managing per-VF own configuration.

Management registers can be directly accessed by the Function driver, and the controls they expose include the ability for Function drivers to discover the device capabilities, be notified of errors, configure the management AMI and enabling/disabling profiling for the local Function. Physical Function drivers can also enable/disable the AMU globally. Certain command messages are reserved for the PF driver to use. In particular, those that configure the mapping of AMIs to Functions and set up/tear down sessions between AMIs.

Switching Layer

Accelerator Message Interfaces for hardware (AMI-HW)

As mentioned, the AMU implements the generic portion of the device, which deals with things like triggering interrupts, managing contexts or terminating the hardware/software interface.

The device-specific portion of the device is implemented by Accelerator Hardware Agents (AHA). An AMU is associated with one or more AHAs. Figure 10 shows where AHAs fit in a Revere-AMU device.

AHAs have their own contexts, represented by AMI-HWs, with their associated state (including the configuration to independently assign AMI-HWs to processes). AHAs may also maintain a set of device-specific state globally and/or on a per AMI-HW basis.

Communication abstraction through Accelerator Sessions (ASN)

Software needs a communication path with AHAs. One of the responsibilities of the AMU is to manage the exchange of *messages* between software and accelerators. The Accelerator Management Unit (AMU) manages the connections and performs routing between software and AHAs, through what is referred to as Accelerator Sessions (ASN) as shown on Figure 10.

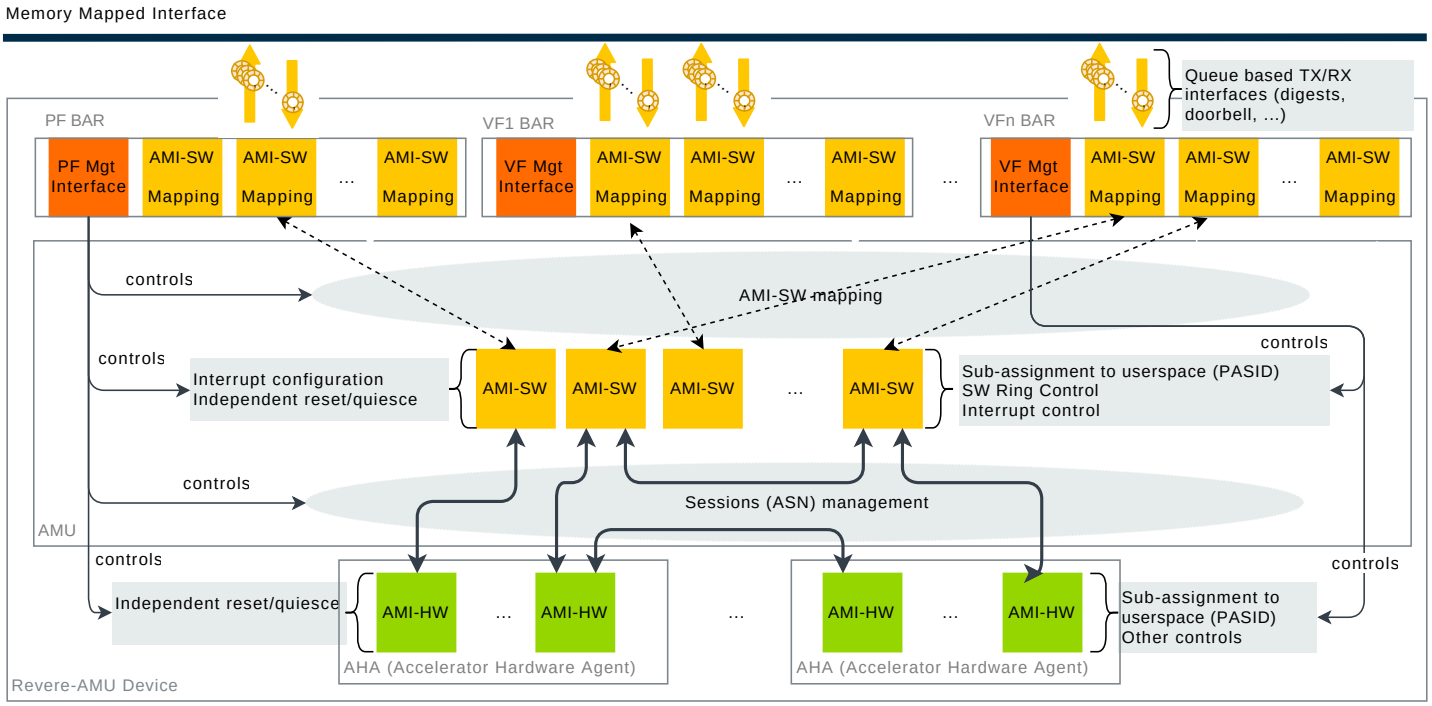

Figure 10: The Accelerator Management Unit (AMU) implements the generic portion of the device, and is exposed to software through device contexts called AMI-SW. Accelerator Hardware Agents (AHA) implement the device-specific portion of the device. AHAs also expose a number of device contexts, called AMI-HW. An AMI-HW contains a small set of architected state and might also be associated with device-specific state.

Software and hardware agents exchange messages through Accelerator Sessions (ASNs), which are unidirectional, ordered channels (i.e in which messages are linearly ordered). ASNs are established to transport messages between a sender and receiver. The PF driver controls the establishment of ASNs and enforces a security/safety policy by determining which ASNs can be created.

Accelerators and I/O devices, in many cases, do not exist in complete isolation. Certain use cases require the chaining of functionality provided by different accelerators and I/O devices. For example, a network adapter, may transport encrypted packets. The same system may feature a crypto accelerator which can also be leveraged by software to encrypt/decrypt those packets.

Pin-level interface layer

Revere-AMU defines an optional pin-level interface – the AMU AHA Interface (AAI) – which defines a protocol layer and transport layer based on AXI4STREAM.

Examples and prototypes

Arm developed a virtual platform (Fast Model)[7] including models of the Revere-AMU and a set of associated AHA. This platform is being used for the development of software drivers and application prototypes.

Software drivers prototypes

Several Linux drivers have been developed to demonstrate the feasibility of various driver use models enabled by Revere-AMU. For instance, the following prototypes of drivers and kernel modules have been developed:

Device specific drivers

- a Revere PF kernel driver that handles Revere-AMU's configurations with the capability to spawn a number of VF and establish a set of sessions during initialization.
- various examples of kernel drivers exposing a network device (*netdev*)
  - Arm Revere Network Adapter (RNA) device driver – a net driver for Revere Network Adapter (see Revere Network Adapter)
  - Arm Revere mediated AMI network device driver – a net driver for the mediated Revere AMI
- a userspace driver – a DPDK Poll Mode Driver (PMD)

Generic drivers and frameworks

All drivers mentioned above leverage generic and standard code from:

- a new kernel module – Revere Generic – that provides a set of generic functions to control and manage Revere-AMU devices
- a new kernel driver for a Revere mediated device that exposes one AMI as a PCI Express device vfio/mdev
- the mainline VFIO (*vfio_pci* and *vfio/mdev*) framework.

The following diagrams describe the various software architectures developed for various experiments:

[7] See https://developer.arm.com/products/system-design/fast-models

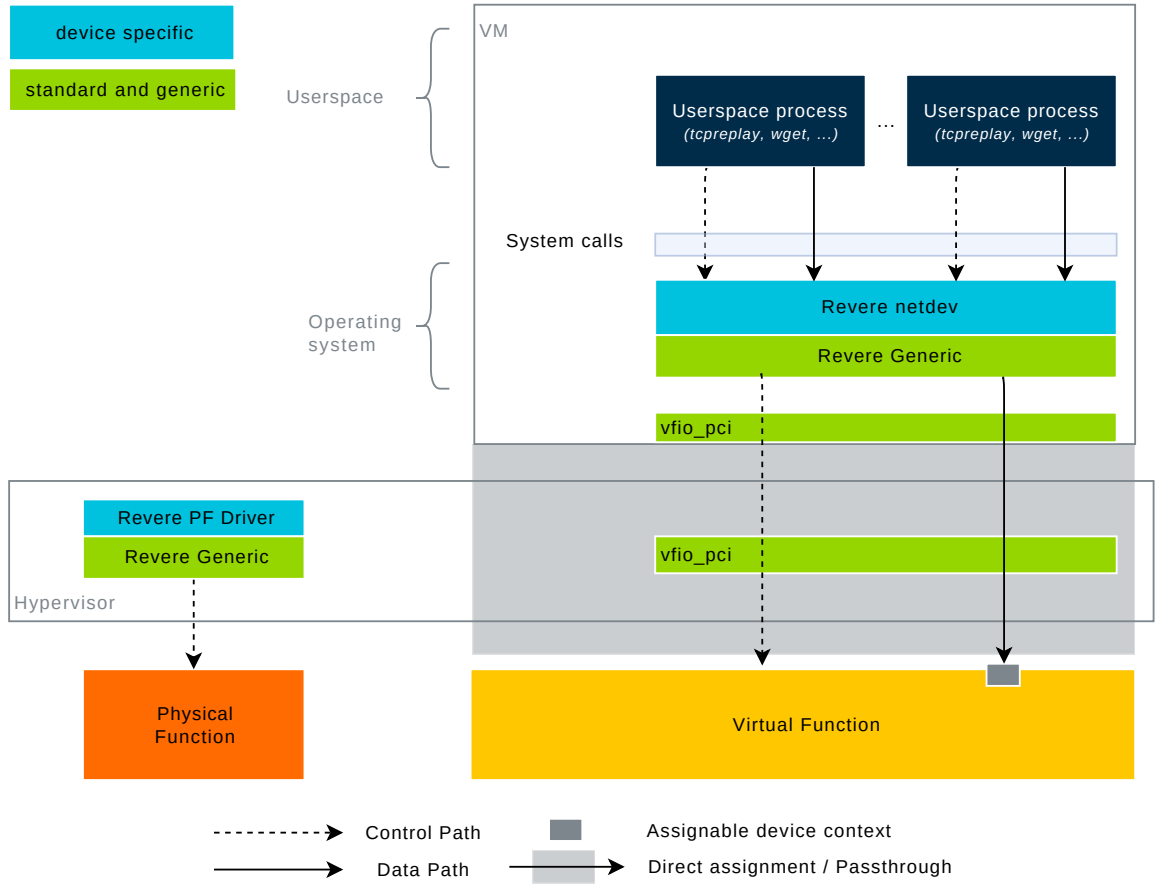

Figure 11: Traditional network driver running in guest OS. The entire *Function* is assigned to a VM using vfio_pci.

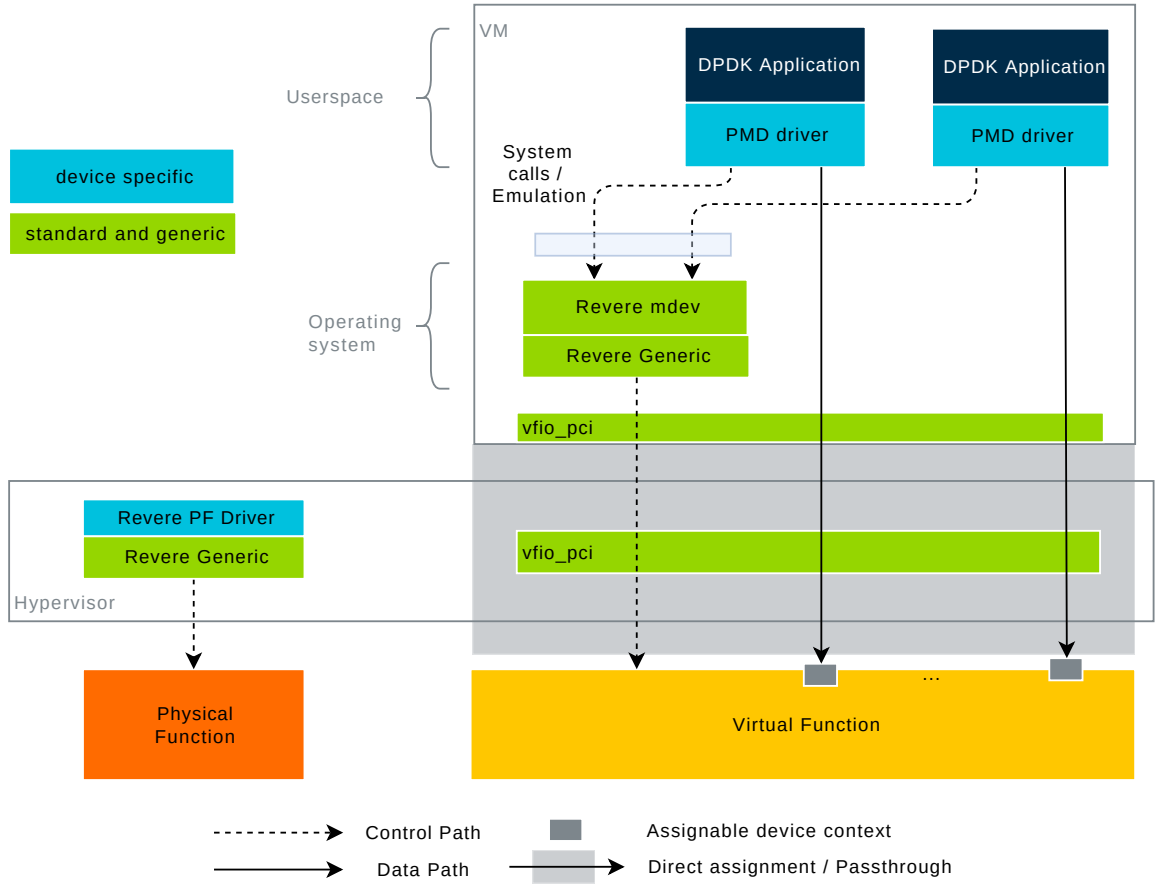

Figure 12: Control mediated drivers in the guest OS. The entire *Function* is assigned to a VM using vfio_pci. Revere mdev exposes an AMI to the userspace application and emulate all the control path.

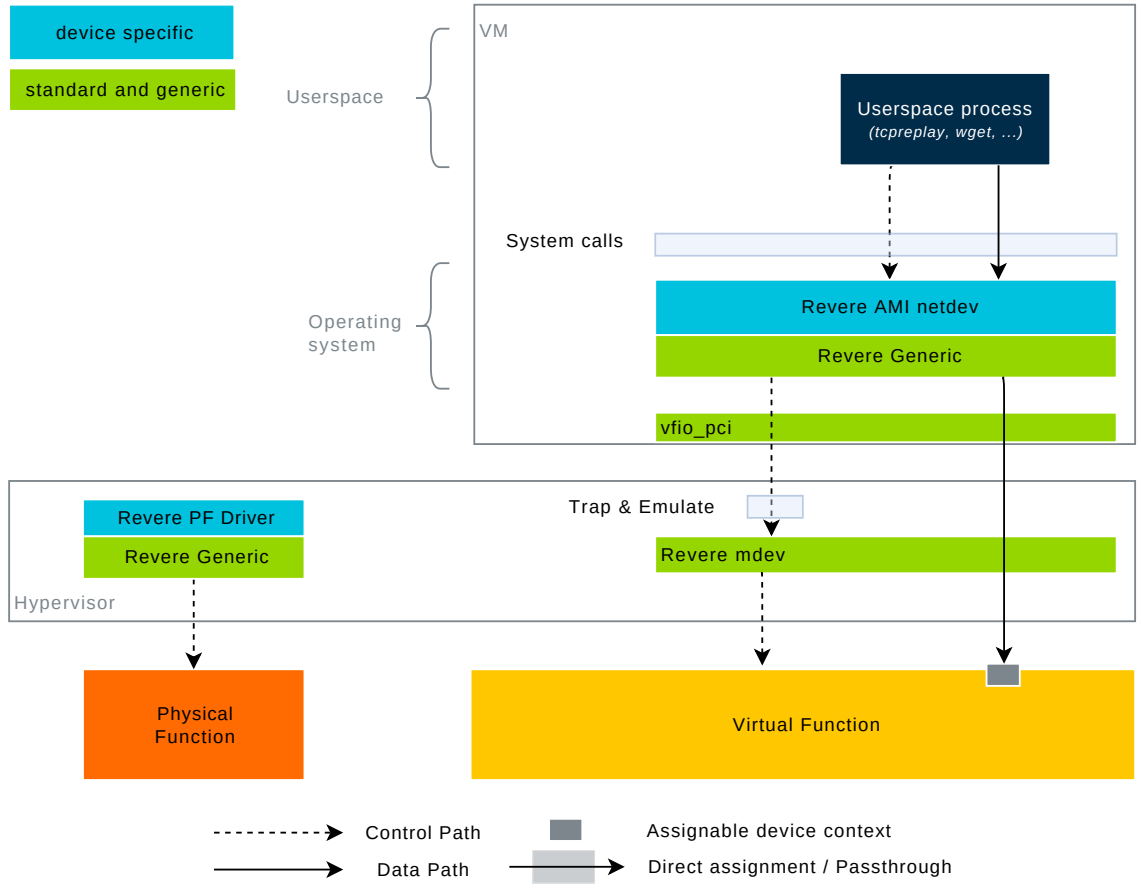

Figure 13: Control mediated drivers in the Hypervisor. Revere mdev exposes a PCI device to the guest OS, which can directly access an AMI. Revere mdev emulates the control path in the Hypervisor.

Revere Network Adapter prototype

With the fast deployment of virtual computing, the importance and processing cost of virtual switching is growing.

Virtual switching encompasses a large number of use cases including inter-VM communication (usually achieved through a network device abstraction) and encapsulation/decapsulation of overlay protocols, which enable Virtual Machines disaggregated across several hosts in different physical locations to appear to the network as if they were part of the same LAN.

Nowadays, virtual switching functionalities are frequently offloaded to smart Network Interface Controller (NIC) devices.

Arm developed a model of a simple configurable NIC referred to as Revere Network Adapter (RNA) plugged in the Revere-AMU hardware and software framework. Figure 14 describes the high-level architecture of the RNA AHA that supports the following set of functionalities:

- Queue-based configuration interface
- Generic flow match/action processing pipeline supporting tunneling/overlays and multi-queue/RSS.

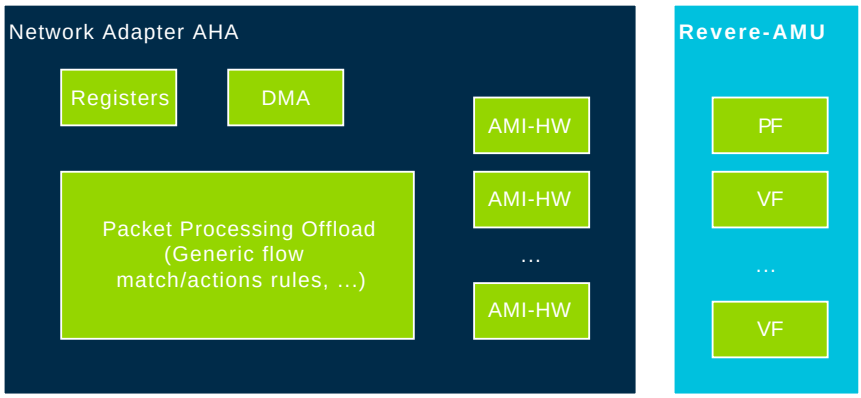

Figure 14: Revere Network Adapter prototype

Within this environment, Arm demonstrated Hypervisor to Hypervisor, Hypervisor to Guest OS and Guest OS to Guest OS communications with Virtual Extensible LAN (VXLAN).

Revere-AMU implementations

Revere-AMU has been selected as the hardware / software interface for two projects of the DARPA's Electronics Resurgence Initiative (ERI):[8] Software Defined Hardware (SDH) and Domain-Specific System on Chip (DSSoC). For these projects, Revere-AMU is implemented as an integrated End Point.

Revere-AMU endpoints can be integrated on a chip or instantiated on a discrete accelerator chip and can also leverage the Cache Coherent Interconnect for Accelerators (CCIX)[9] protocol, as shown on Figure 15.

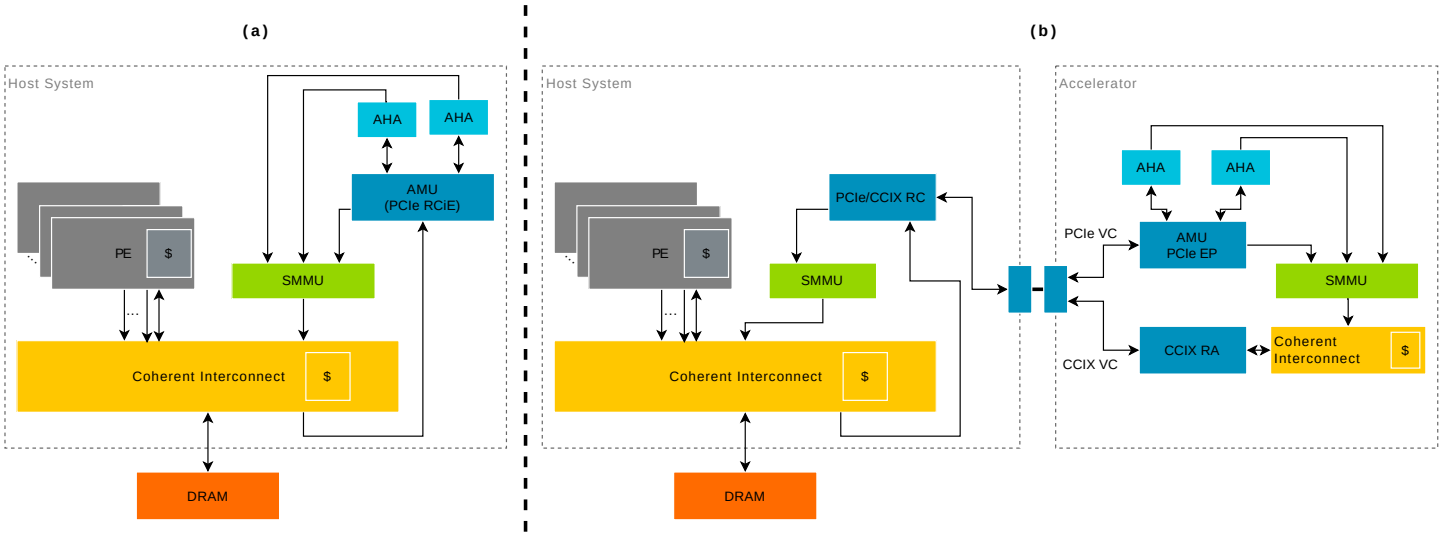

Figure 15: (a) Integrated Revere-AMU endpoint. (b) Revere-AMU instantiated on a discrete accelerators connected through PCI Express and CCIX

[8] See https://www.darpa.mil/work-with-us/electronics-resurgence-initiative
[9] *Cache Coherent Interconnect for Accelerators*. (Master 6.1)

Conclusion

This paper describes the importance of standardization when interfacing accelerators and IO devices and proposes a set of standard layers.

Revere-AMU is an example that covers all the proposed system architecture layers. It enables the implementation of standard software and hardware frameworks for interfacing accelerators and IO devices. It supports a wide range of device driver use models, including the support for flexible fine-grained assignments that can scale to a large number of Virtual Machines / userspace applications.

A virtual platform (Fast Model)[10] is available for the development of software driver and application prototypes. In addition, a RTL implementation of Revere-AMU is being developed to be integrated in several of DARPA's ERI[11] projects.

Arm is going to engage in discussions with several open-source communities and standards bodies, to enable the driver use-models presented in this white paper to become standard.

Arm is always interested in collaborations with stakeholders in the industry, to improve the eco-system.

[10] See https://developer.arm.com/products/system-design/fast-models
[11] See https://www.darpa.mil/work-with-us/electronics-resurgence-initiative

Glossary

AHA

Accelerator Hardware Agent

AMI

Accelerator Message Interface

AMI-HW

Accelerator Message Interface for Hardware

AMI-SW

Accelerator Message Interface for Software

AMU

Accelerator Management Unit

ASN

Accelerator Session

Cache Stashing

Ability of an agent in the system to request that a line is allocated in (or stashed) to a cache local to another agent in the system.

CCIX

Cache Coherent Interconnect for Accelerators

Context

A context is a view of memory combined with associated state. For software running on an Armv8 PE, the context is identified by a VMID, an ASID and the PE architectural state for a thread.

DARPA

Defense Advanced Research Projects Agency

Device

A granule of hardware resources (for example: accelerators, I/O) in the system that is visible to software and can be managed independently. In the context of an AMU, a device is a combination of an AMU and all AHAs associated with that AMU. A device has memory mapped registers, may generate interrupts and may have DMA capabilities.

DMA

Direct Memory Access performed by a device.

DPDK

Data Plane Developer Kit

ERI

Electronics Resurgence Initiative

HW

Hardware

I/O

Input/Output

Message

A message is a logical entity comprising a command and associated data.

MMIO

Memory-mapped I/O

MMU

Memory Management Unit

MSI-X

Message-signaled Interrupts Extended

OS

Operating System

PASID

Process Address Space ID

PCI Express

Peripheral Component Interconnect

PE

Processing Element

PF

Physical Function

PMD

Poll Mode Driver

RCiEP

Root Complex Integrated Endpoint

RTL

Register-Transfer Level

SMMU

System Memory Management Unit

SR-IOV

Single Root I/O Virtualization

SW

Software

VF

Virtual Function

VM

Virtual Machine