Technical Disclosure Commons

Defensive Publications Series

January 10, 2019

Fast & Scalable I/O for Emulated HPUX

Maheshwara Aithal
Hewlett Packard Enterprise

Dinesh Thanumoorthy
Hewlett Packard Enterprise

Ranjith VK
Hewlett Packard Enterprise

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Aithal, Maheshwara; Thanumoorthy, Dinesh; and VK, Ranjith, "Fast & Scalable I/O for Emulated HPUX", Technical Disclosure Commons, (January 10, 2019)
https://www.tdcommons.org/dpubs_series/1869

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Fast & Scalable I/O for Emulated HPUX

Abstract

HPE has positioned containerized solution called c-UX (code named Kiran) which runs HPUX in emulated (Itanium hardware emulation on x86) mode as a futuristic solution for the margin rich UNIX business. The value of containerized HPUX is that it allows customers using legacy HPUX applications to continue running on x86 hardware. Significant effort has been expended to increase the effectiveness of hardware resource utilization on c-UX. The next step in fully optimizing I/O in c-UX environment is to provide truly scalable high-performance, by enabling a single I/O device to provide DMA for multiple VMs. This scalability challenge can be solved using Single Root I/O Virtualization (SR-IOV) technology, delivering near-native I/O performance for multiple c-UX instances, while also providing memory and traffic isolation for security and high availability, accelerating live migrations, and reducing the cost and complexity of I/O solutions. Network and Storage adapters from various vendors can be used to realize SR-IOV on c-UX, which otherwise was not possible on native HPUX due to hardware and firmware limitations. This paper talks about an innovative mechanism to enable SR-IOV on emulated HPUX OS using Virtual Function I/O framework (VFIO) available in Linux. Disclosed is an approach of achieving highly scalable performance in c-UX application by allowing the guest OS direct access to parts of the I/O subsystem of the host and handle various aspects of the communication like DMA and interrupts. It also throws light on the network I/O performance gains achieved using this method.

Problem statement

The c-UX is a solution that can emulate Itanium System on x86-64. It is a user space application that emulates all hardware components of Itanium required to run HPUX OS, but with inherent performance issues. However, certain I/O workloads on c-UX can make use of direct device access or device assignment technique to deliver the highest possible I/O performance. Scalability challenges remain though, since these I/O virtualization solutions require a dedicated physical I/O port for each guest OS instance. Currently, there is a need to optimize I/O performance by enabling efficient and scalable I/O device sharing across various c-UX instances. Single Root I/O Virtualization (SR-IOV) can be used to support large numbers of direct assigned VMs per adapter to dramatically improve scalability, performance and slot utilization, while also reducing I/O-related power consumption and cabling requirements. The challenge here is to implement a comprehending solution for emulated HPUX on Linux environment that is scalable and performing. As such, SR-IOV ecosystem requires support from the OS and system firmware (UEFI), which is currently unsupported on HPUX - is being realized on c-UX via PCI pass through method on Linux without any kernel changes and with a new device driver on guest OS.

Competitive approaches

Other techniques include VMDq, VirtIO model, DPDK, device emulation model etc. With VMDq, the functionality will be limited as we can only share Tx/Rx queues. Non-VFIO models need specialized kernel drivers that have to either go through the full development cycle, be maintained out of tree, or make use of the UIO framework, which has no notion of IOMMU protection, limited interrupt support, and requires root privileges to access things like PCI configuration space. The hardware assisted SR-IOV consumes approximately 40% less cycles than the para virtualized driver VirtIO for both sending and receiving.

Proposed solution

SR-IOV defines a method to share a physical function of the I/O port of a device without software emulation. This process creates a number of VFs (Virtual Functions) per physical port of the I/O device. Each VF acts as an independent but lightweight PCIe device, capable of being configured for data traffic movement. There are also PFs (Physical Functions) which contain full PCIe functions and can be used to configure SR-IOV enabled devices. In this solution, VFs are directly assigned to guest OS, using VFIO in Linux by emulating certain parts of the VF and managing other aspects of device handling like MMIO access, DMA and interrupts by bypassing the host kernel, thereby achieving near native performance.

VFIO (Virtual Function I/O) is an open source device agnostic framework, for exposing direct device access to user space, in a secure, IOMMU protected environment. An IOMMU is a hardware block, comparable to the MMU of the CPU, which intercepts any memory access from the device and translates the memory addresses, from virtual addresses to physical addresses. Initially, both VF and PF will be claimed by the corresponding host drivers. Then, the VF is unbind from host driver and bind it to the VFIO module. Using the standard VFIO APIs, the VF's MMIO and DMA regions are mapped to guest memory securely, allowing the HPUX guest to claim the VF interface, while the x86_64 host can handle and forward incoming interrupts. The VF driver – a new para-virtualized driver on HPUX guest communicates directly with the hardware to perform data movement operations via APIs exposed to c-UX and rely on the services of the PF driver to handle operations that can have global impact, like configuration, initialization and control of the secondary fabric.

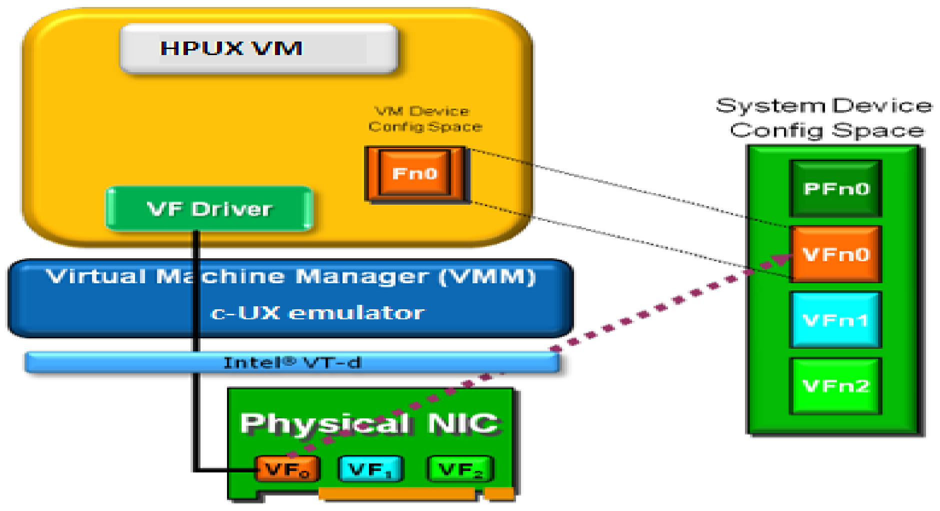

Fig.1 Example showing VF being exported to HPUX running on c-UX

Handling Configuration/MMIO space access: In c-UX, a DIO emulator framework is developed that is furnished with VF device specific information like host hardware path etc. which is eventually parsed and used to populate the VF device on guest. VFIO abstracts PCI devices as different regions like PCI configuration space, MMIO and I/O port BAR spaces, MSI-X or IRQ spaces etc. We've used the VFIO APIs to query such information and emulate the configuration space of VF, so that it can be discovered on guest PCI environment. During the guest boot up, c-UX intercepts and emulates attempts by the guest OS to access PCI config space and MMIO space of the device. Emulation of the VF registers is achieved by hooking the interface specific I/O handlers directly to the existing facilities of c-UX. The host BAR address space of the VF interface will be mapped to the guest address space using the standard HPUX WSIO functionality. This way, using the VFIO standard interface, PCI devices behind an IOMMU is unmapped from the host operating system, and subsequently map to c-UX virtual address space. On HPUX, VF is loaded based upon the Device ID presented to the guest as part of the PCI configuration information from the c-UX. The VFs have a unique Device ID using which the appropriate driver can be loaded. The new HPUX VF driver will claim the exposed device on guest, followed by the initialization, configuration and port link up. The communication between VF and PF is implemented in HPUX utilizing a set of hardware mailboxes and doorbells within the I/O controller for each VF.

Handling DMA: In this case, the IOMMU is programmed with the Linux host user virtual address of the entire c-UX application, to map the HPUX guest memory with the host physical memory space, enabling the device to do DMA directly to the guest space, eliminating the buffer copy overhead. By configuring the IOMMU this way, the I/O device will operate under the illusion that it is directly accessing the physical memory of the Linux host, while it has actually been mapped to the memory space of HPUX running on c-UX. The IOMMU will prevent the hardware components behind IOMMU from accessing memory beyond the mappings that have been assigned in the page tables thereby protecting the c-UX system from accesses to memory from the device. Multiple devices can be shared to the same guest by pooling them together by creating VFIO containers, which handles the DMA mapping/unmapping and not the device.

Handling Interrupts: MSI-X is the most efficient way to spread interrupts from one device among multiple cores. VFIO provides standard interfaces to configure and interact with the MSI-X/IRQ signals of the device. The device signals the host driver using interrupts. And in this design, the guest is interrupted using eventfd, a file descriptor for event notification. Eventfd creates an eventfd object that can be used as an event wait/notify mechanism by the emulator running in the user space and by the kernel to notify guest operating system of events. During the MSI-X initialization on guest, the HPUX vector details are gathered by intercepting the MMIO writes using the device MMIO handlers on c-UX. For every vector on guest, a corresponding MSI-X vector is allotted on the device via VFIO, and each vector is assigned a handler (pthread in this case) on the host, along with an eventfd for notification. When an IRQ is being triggered, the eventfd wakes up the handler, which will notify the HPUX guest with the corresponding IRQ details that are already stored in the handler.

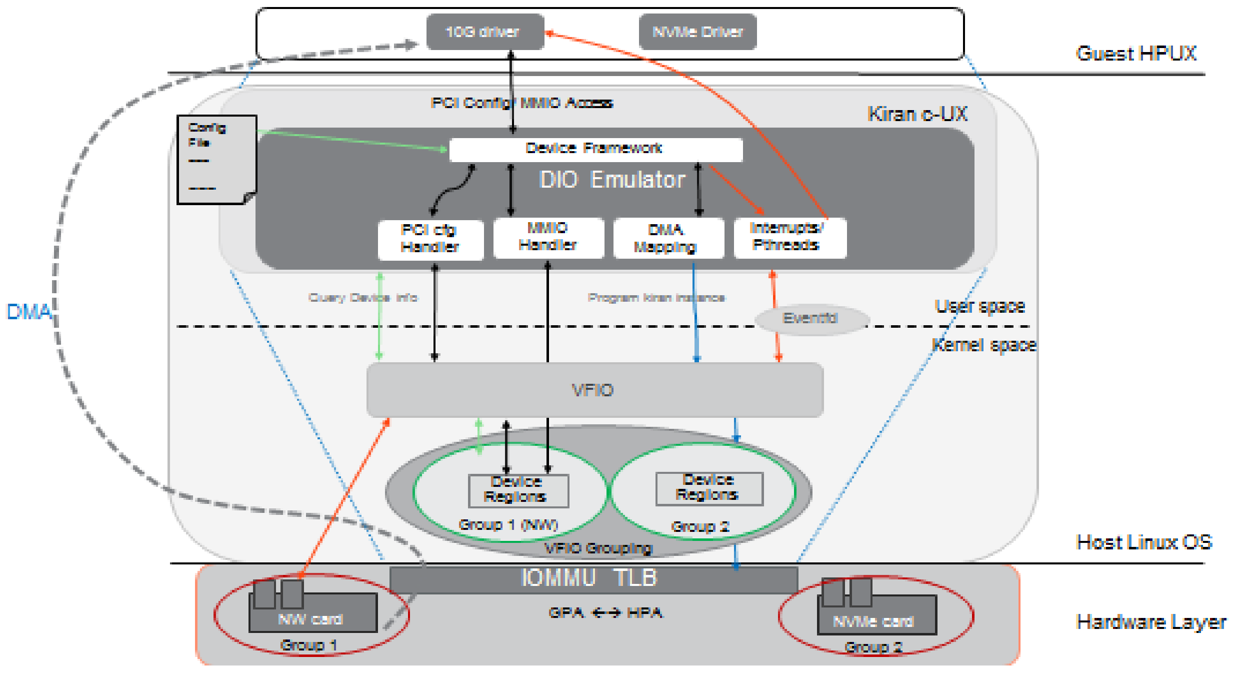

Fig.2 Describing interaction between various modules via DIO emulator on a Linux system

The SR-IOV technology via VFIO delivers the best performance when implemented within multi-core processor environments. By avoiding the Rx side of the copy with DMA, this method results in better CPU utilization on virtualized servers. It can scale I/O to multiple VMs at the cost of less additional CPU overhead per VM, without sacrificing throughput. This is completely a user space solution without any kernel changes involved.