

First Workshop on I/O Virtualization (WIOV '08)

San Diego, CA
December 10–11, 2008

I/O ARCHITECTURE

Summarized by Mike Foss (mikefoss@rice.edu)

■ *Towards Virtual Passthrough I/O on Commodity Devices*

Lei Xia, Jack Lange, and Peter Dinda, Northwestern University

Lei Xia delivered the first presentation of the workshop, explaining how one might use a model-based approach to allow virtual passthrough I/O on commodity devices. The current approaches to allow high-performance I/O in guest operating systems are limited. In one approach, the virtual machine provides full emulation of the device in order to multiplex it to each guest operating system; however, this requires significant overhead in the VM. To reduce the performance penalty, a guest might bypass the virtual machine altogether in direct-assignment I/O. However, this approach is less secure, since a guest could affect the memory of other guests or the VM itself. Some devices are multiplexed in the hardware and allow each guest to directly access the device while preserving security, but this feature is not available on commodity I/O devices, nor do these devices currently allow migration of guests.

Xia introduced virtual passthrough I/O (VPIO), which allows a guest to have direct access to the hardware for most operations and also allows a guest to migrate. VPIO assumes that there is a simple model of the device that can determine (1) whether a device is reusable, (2) whether a DMA is about to be initiated, and (3) what device requests are needed to update the model. VPIO also assumes that the device can be context-switched, that is, that the device can deterministically save or restore the state pertaining to a guest operating system. For the best performance, the goal of VPIO is to have most guest/device interactions complete without an exit into the VM.

Under VPIO, each access to the device must go through a Device Modeling Monitor (DMM). The purpose of DMM is twofold: (1) It saves enough state about the guest and the device that a guest could migrate to a new VM, and (2) it ensures that the VMM enforces proper security. It also keeps track of a hooked I/O list, which is a set of I/O ports that require VM intervention if accessed by a guest. Unhooked I/O ports may be used by the guest directly. The device is multiplexed by performing a context switch on the device (restoring the guest-specific state into the device). Currently, if the DMM disallows the guest to continue with an operation (e.g., in the case of a DMA to an address out of bounds), the DMM delivers a machine-check exception to the guest. If the device issues an interrupt, it may not be clear to which guest to forward the interrupt, as in the case of receiving a packet on a NIC. Currently, Xia's team is working on finding a general solution to this problem.

Xia's team did implement a model of an NE2000 network card and had it running under QEMU. The model was under 1000 lines of code, and only a small fraction of I/Os (about 1 in 30) needed VM intervention. The remaining challenges for this project include the following: moving more of the model into the guest in order to reduce the cost of a `vm_exit`; handling incoming device input, such as interrupts without a clear destination guest; and obtaining a device model from hardware manufacturers.

■ *Live Migration of Direct-Access Devices*

Asim Kadav and Michael M. Swift, University of Wisconsin—Madison

Asim Kadav presented the second paper of WIOV, explaining how to migrate direct-access I/O devices from one virtual machine to another. While direct, or passthrough, I/O offers near-native performance for a guest OS, it inhibits migration, because the VM does not know the complete state of the device. Furthermore, the device on the destination machine may be different from that on the source machine. Asim proposed to use a shadow driver in the guest OS in order to facilitate migrating guests that take advantage of passthrough I/O.

The challenge of the shadow driver is to simultaneously offer both low constant overhead and short downtime during migration. The shadow driver listens to communication between the kernel and the device driver via taps. In its passive mode, the shadow driver keeps track of the state of the driver. It intercepts calls by the driver, tracks shared objects, and logs any state-changing operations.

During migration, or active mode, the shadow driver is responsible for making sure that migration occurs without the need to modify the existing device driver or hardware. First, the shadow driver unloads the old device driver and monitors any kernel requests during the period where there is no driver. Next, it finds and loads the new driver into the appropriate state.

Asim's team modified Xen and Linux in order to implement a prototype shadow driver. The shadow driver implements taps by substituting functions in the kernel/driver interface with wrapper functions. These wrappers were generated by a script that would accommodate any network device. Asim showed that the shadow driver method worked and only cost a percentage point of both extra CPU overhead and network throughput during passive mode. Migration to a new virtual machine took four seconds, but most of the time was spent in the initialization code of the network driver. Asim also showed that migration between heterogeneous NICs was possible by enabling the lowest common denominator of features between the participating NICs. No device driver or hardware modifications were needed in order to use the shadow driver.

- **Scalable I/O—A Well-Architected Way to Do Scalable, Secure and Virtualized I/O**

Julian Satran, Leah Shalev, Muli Ben-Yehuda, and Zorik Machulsky, IBM Haifa Research Lab

Muli Ben-Yehuda presented the final paper of the first session of WIOV, a position paper on how I/O should be scaled for any system. The current device driver model presents a unique problem in the OS for several reasons. First, communication with the hardware consists of only register transfer and DMA operations. Furthermore, each driver is vendor-specific and must be maintained by the vendor. They are often the source of bugs in the OS. These problems are pronounced in virtualized systems. Muli proposed to virtualize the entire I/O subsystem rather than each driver or device, in order to enhance the scalability and security of I/O in virtual machines.

The scalable I/O architecture consists of device controllers, I/O consumers, and host gateways. A device controller (DC) is responsible for communicating directly with the device. It implements I/O services and can serve many I/O consumers simultaneously. It also protects devices from unauthorized access. An I/O consumer is any process on the host that wishes to access the device. The I/O consumer accesses the proper DC by first sending the request through a host gateway. The host gateway (HG) is in charge of granting protected I/O mechanisms to all the I/O consumers on the host. It can be thought of as an elaborate DMA engine that provides a DMA to virtual memory. The HG and DC are connected by shared memory or any I/O interconnect in general, which is completely abstracted away from the I/O consumer.

Protected DMA (PDMA) is the mechanism by which the HG and DC communicate. The HG generates a memory credential whenever an I/O consumer wishes to use the DC. This credential is later validated by the HG whenever the DC accesses the consumer's memory.

The scalable I/O protocol grants several benefits over current I/O mechanisms. I/O consumers may submit I/O pro-

grams to the DC, which gives a high-level I/O interface to consumers. Furthermore, the I/O subsystem is now isolated from the rest of the operating system, which may improve performance and robustness. A programmable I/O interface also allows for enhanced flexibility and scalability.

Another benefit of scalable I/O is that memory pinning becomes unnecessary. Memory pinning is expensive and puts an error-prone burden on the programmer. In scalable I/O, devices ignore pinning and assume that the memory is always present. In the unlikely case that the desired memory is not present, the device takes an I/O page fault, and the DC and HG communicate in order to resolve the page fault.

STORAGE VIRTUALIZATION

Summarized by Asim Kadav (kadav@cs.wisc.edu)

- **Block Mason**

Dutch T. Meyer and Brendan Cully, University of British Columbia; Jake Wires, Citrix, Inc.; Norman C. Hutchinson, University of British Columbia; Andrew Warfield, University of British Columbia and Citrix, Inc.

Block Mason by Dutch Meyer addresses the problem of developing agile storage systems for virtualization by proposing a high-level declarative language to manage blocks. Dutch began by describing the file system interface as basically a block interface but with accessibility issues in practice, as the kernel hides it. However, in virtualized interfaces the block layer becomes more important since shared storage can leverage significant functionality from the block layer. At block level one can add many features such as compression, encryption, deduplication, or even advanced gray box techniques. The key idea of this talk is to provide a user-level framework for building reusable modules at the block level that one can connect to perform more complex tasks. Block Mason helps developers build fine-grained modules and assemble and reconfigure them to build high-level declarative verifiable block manipulation. The implementation of Block Mason was done in Xen using the blktap interface. In user mode, a new scheduler, parser, and driver API were implemented. There were also some minimal updates to blkback.

Dutch further detailed the implementation, discussing the basic building blocks (elements/modules) and connectors (ports/edges). An element example would be as simple as recording I/O requests. Any details of elements can be added to configuration files. The connectors are the ports, identified by names. Block Mason also supports live reconfiguration of the modules built by draining outstanding requests and initializes new ones as they arrive. The architectural support implemented includes message passing and dependency tracking.

As an example of a service using these various constituents, Dutch suggested migrating storage from a local disk to another storage device. The two subservices that are using the

Block Mason interface in this example are I/O handling and background copying, implemented using low-level Block Mason constituents. More complex modules such as cloud-backed disks were also described briefly. Block Mason can be also used to perform other tasks such as correctness verification. Block Mason will be integrated into the new blktp2 interface in Xen. Future work will include developing declarative languages to perform block tasks.

Dan Magenheimer from Oracle commented on how powerful Block Mason is and inquired about the things that can be done with Block Mason. Dutch answered that, using Block Mason, one can build simple features and aggregation of simple features such as disk encryption. Himanshu from Microsoft asked about synchronization issues with Block Mason. Dutch explained synch issues with the copy example. In response to a question about whether synchronous write would work properly, Dutch explained that only one port is used to perform I/O in his example. Another questioner asked about block failures and their handling by Block Mason. Dutch replied that one can trap failures and perform recovery actions and explained it in his disk copy example. Muli Ben-Yehuda posited that this may be similar to using pipes, but Dutch said that pipes would give the same expressiveness but coarse-grained modules.

- **Experiences with Content Addressable Storage and Virtual Disks**

Anthony Liguori, IBM Linux Technology Center; Eric Van Hensbergen, IBM Research

Eric Van Hensbergen gave a talk on his research on how to reduce redundancy in virtual disk images using content addressable storage. The motivation here is that virtualization causes lot of image duplication with many common files, libraries, etc. In a cloud scenario, the problem is even more severe, with many thousands of disk images on the server. The first part of his talk consisted of analyzing the existing duplication at file and block levels. The results had filtered out duplicates due to hard links and sorted the results to obtain self- and cross-similarity separately. Eric showed considerable overlap in terms of the same blocks in various Linux distributions for their 32/64-bit versions. There are also similar overlaps in different distributions of the same operating system (Linux) and in different versions of the same distribution. All results show considerable overlap in the binaries that can be exploited. Even in analyzing different images created from different install options, there is a large degree of overlap (duplication). These results are also the same for Windows (factoring out swap/hibernation files). Analyzing the deduplication efficiency, the results show a slightly higher efficiency for 8k blocks, but this is primarily due to error associated with partial blocks and the discounting of zero-filled blocks. A disk-based scan was able to identify approximately 93% of the duplicate data.

The second part of the talk compared existing solutions and their solution. The common existing solution is to use

Copy-On-Write (COW) disks. The problem with the COW approach is that there is a drift to higher disk usage with application of the same updates to different disks. This is because, as the same updates are applied to similar images, since updates are applied one after another the images get out of sync. Eric's solution is to use an existing Content Addressable Storage (CAS) system (Venti) as a live backing store and use a filesystem interface on top of it to present to virtual disks. The hypervisor was modified to use these virtual disks. Further, Eric gave some background on CAS and then described some related work including Foundation (CAS to archive VM for backup), Mirage (file-based CAS), and Internet Suspend Resume (CAS to access personal computing environments across a network by using virtualization and shared storage). He also cited a work from Data Domain in which a filesystem-based approach is used to leverage deduplication in backups.

In terms of implementation, Eric reused an existing block-based CAS, vbackup in Plan 9. QEMU/KVM ran the virtual machines and provided a hook via vdiskfs that uses vbackup as the underlying store.

The evaluation consisted of measuring block utilization before and after same updates on two similar disk images. There was also an additional micro-benchmark using the dd command. The results show better results with CAS and compressed CAS than those with COW. The performance, however, takes a hit and the boot time to bring up the system using CAS is much higher. In terms of the micro-benchmark, CAS performs much worse, running at 11 Mbps compared to 160 Mbps (without CAS) in raw mode. To summarize the evaluation, the space efficiency is great but performance is bad, since Venti is single-threaded and synchronous and also was configured with a small cache for these experiments. Their future work includes reworking CAS for live performance, experimenting with flash disks for index storage, and building in support for replication and redundancy.

A questioner asked about the case of dirty blocks and Eric replied that he was using a write buffer to avoid using them for CAS; however, dirty blocks can be used as a single large cache for all virtual machines. Another person from the audience pointed out a related work from CMU about finding similarities using encryption system. When asked whether this was even the right approach to the problem, Eric said he didn't know, but it was easy to implement and took only two weeks. Jake Oshins from Microsoft wondered whether it would be advantageous for the file system to know what blocks are being deleted. Eric said it would definitely help and pointed out a related work that addresses this.

- **Paravirtualized Paging**

Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel, Oracle Corporation

Dan's talk covered a new type of cache, called hcache, aimed at resolving memory issues in virtualization. Mem-

ory is cheap but, currently, memory systems are running unutilized and are being wasted. Most recent work on virtualization has concentrated on efficient CPU and I/O utilization, but little work has been directed toward memory utilization. Described in the talk was a proposed approach to resolving memory issues in virtualization by allocating a separate pool in the virtual machine's memory space, called transcendental memory. Dan then focused on the basics of physical memory concepts in a single machine and virtualization servers and discussed the common memory issues there. In a single machine with a single operating system, the memory is basically a huge page cache that is never optimized, and a lot of idle memory in page cache is simply wasted. This is because the operating system has no way of determining which areas of page cache are being utilized and which are not. The pages are moved out of page cache using a page cache replacement algorithm (PRFA); even PRFA cannot determine the correct working set of page cache, resulting in many false-negative page evictions, making the matter worse.

The situation is no different in virtualization servers, where the physical memory is still used inefficiently. Memory allocations to guest virtual machines are either by static partitioning of memory or by dynamic partitioning of memory. Neither of them is helpful. Static partitioning has problems such as fragmentation of memory and memory holes resulting from machine migration. There is also almost no load balancing of memory by the hypervisor in static partitioning. The second method, dynamic partitioning (also known as ballooning), uses a balloon driver in the guest virtual machines. Ballooning tunnels pages across balloon drivers to transfer memory from one virtual machine to another to perform load balancing. This scheme also has many issues, since OS/virtual machines rarely voluntarily give up memory and always demand more memory. There are further difficulties in determining which virtual machine is the neediest here. Also, ballooning is not instantaneous for large or fast changes in balloon size.

The solution here aims to answer unanswered questions such as how to reclaim I/O without increasing disk I/O. Also, the problems of ballooning and memory just mentioned can be alleviated by using the solution described. The solution provided is to reclaim all idle memory into a pool called the transcendental memory pool. All guests access memory via the hypervisor using transcendental APIs, which cause memory operations that are synchronous, page-oriented, and copy based. This memory pool is subdivided into four subpools: private ephemeral, private persistent, shared ephemeral, and shared persistent. The private ephemeral memory is labeled as "hcache." The false evictions now fall into hcache and have a low cost now. Also, any memory in hcache can be thrown away quickly, resulting in faster memory allocations to virtual machines so that operations such as ballooning can be done quickly. Dan also pointed out that very minimal changes are needed

to implement this solution. He further described hswap, which is a persistent, private cache that works like a pseudo swap device. It helps to balloon fast, as ops from pool are faster and there is no thrashing as memory is allocated from the pool. He further pointed out that shared ephemeral/persistent pools can act as shared memory for inter-VM communication; this is a part of future work.

Transcendental memory can also be used in a single OS, because API is clean, as a useful abstraction (NUMA memory, hot-swappable memory, or SSDs). It can also be used as a cache for network file systems. In conclusion, transcendental memory is a new way to manage memory for single operating system and virtualization servers and reduces many of the existing memory issues.

To the question of whether one needs contiguous memory for transcendental memory, Dan replied that the transcendental memory solution works even with fragmented memory.

DEVICE VIRTUALIZATION

Summarized by Jeff Shafer (shafer@rice.edu)

■ **GPU Virtualization on VMware's Hosted I/O Architecture** *Micah Dowty and Jeremy Sugerman, VMware, Inc.*

Micah Dowty presented a paper on how to virtualize a GPU. In this talk he introduced a taxonomy of GPU virtualization strategies and discussed specifics of VMware's virtual GPU.

A GPU can provide significant computation resources, and both desktop and server virtualized environments seek to take advantage of these resources. Virtualizing a GPU poses many different challenges, however. There are multiple competing APIs available, and these APIs are complicated with hundreds of different entry points. In addition, the APIs and GPUs are programmable. Every GPU driver is also a compiler, and each API includes a language specification. Hardware GPUs are all different, covering a wide range of architectures that are often closely guarded secrets that change frequently between product revisions. Finally, the hardware state of the GPU chip and associated memory is large, covering gigabytes of data in a highly device-specific format including in-progress DMAs and computation.

There are several potential options to virtualize a GPU, as presented in the taxonomy. These strategies include capturing application API calls at a high level and proxying them to another domain for execution (API remoting), providing a virtual GPU for each guest with which the native software stack communicates (device emulation), and various pass-through architectures to tunnel GPU commands down to the actual hardware. Each technique has various tradeoffs in terms of performance and isolation.

The VMware virtual GPU uses a combination of techniques, most notably device emulation and API remoting, to provide accelerated GPU support in a virtualized environment on

top of any physical GPU. With this architecture, interactive graphics applications can now be run at a usable performance level, whereas it was not possible to run them in a virtualized environment before. Future work will focus on new pass-through techniques as well as the development of virtualization-aware GPU benchmarks that stress, not the raw GPU hardware performance, but, rather, the API-level paths that are at issue in a virtualized system.

One audience member asked about the challenges involved in migrating virtual machines across different GPUs. Dowty replied that migration requires reading all the state from the GPU and memory, but this is not always possible considering that some state is generated by the graphics card itself and is not always accessible to the driver or API. GPU vendors have a lot of flexibility in implementing new technologies (such as SR-IOV) to make virtualization and migration simpler and more complete.

- **Taming Heterogeneous NIC Capabilities for I/O Virtualization**

Jose Renato Santos, Yoshio Turner, and Jayaram Mudigonda, Hewlett-Packard Laboratories

Jose Renato Santos from HP Labs presented a paper on a network I/O virtualization (IOV) management system that can translate high-level goals into low-level configuration options. In addition, methods for efficient guest-to-guest packet switching were discussed.

In recent years, different vendors have provided many mechanisms for I/O virtualization, such as software virtualization, multi-queue NICs, and SR-IOV multifunction NICs. In the process, however, they have created significant challenges for managing networks of heterogeneous devices, each with different hardware and software approaches to virtualization. Configuration becomes more complex and fragile with increasing diversity in IOV mechanisms. What is needed is a higher-level abstraction for I/O configuration, where users specify logical networks and a mapping of virtual interfaces to logical networks, and then the system selects and configures the appropriate mechanism.

This configuration can be done statically or dynamically. Although a static system may be simpler, consider a case where there are more guests than hardware NIC contexts available to support them. Then a dynamic management system that looks at current workload levels may be needed for optimal assignment. In addition to a new configuration mechanism, a spectrum of methods for efficient intranode guest-to-guest packet switching were also discussed, including switching in software, on the NIC, or in external network devices. All these techniques have tradeoffs in terms of CPU, I/O bandwidth, link bandwidth, and memory use, and this must be considered by the high-level management tool depending on constraints input by the user.

One audience member asked how frequently the system can change its configuration based on these high-level policy guidelines. Jose answered that this is an open question, but

certainly not on every packet. There are several concerns involving maintaining packet ordering and minimizing setup/teardown overheads.

- **Standardized but Flexible I/O for Self-Virtualizing Devices**

Joshua LeVasseur, Ramu Panayappan, Espen Skoglund, Christof du Toit, Leon Lynch, and Alex Ward, Netronome Systems; Dullloor Rao, Georgia Institute of Technology; Rolf Neugebauer and Derek McAuley, Netronome Systems

Rolf Neugebauer spoke about some of the limitations of the SR-IOV standard for virtualizing complex network devices and proposed a new approach, software configurable virtual functions, to provide increased flexibility for virtualization.

In today's networking environment, multi-queue NICs are an emerging standard, and some include hardware support for virtualization. Hypervisors allow assignment of PCI device functions to virtual machines by virtualizing the PCI configuration space. Moreover, modern chipsets include I/O MMUs to provide DMA isolation and address translation. The SR-IOV (Single Root I/O Virtualization) standard ties these three trends together and allows endpoints such as network cards to be enumerated as PCI virtual functions. Because this is performed in hardware through the device configuration space, however, the SR-IOV has limits to its flexibility.

The new Software Configurable Virtual Functions (SCVF) is proposed for highly programmable network devices such as the Netronome NFP3200. SCVF is built on the same base PCI Express technologies and provides isolated access to virtual functions using IOMMUs. Rather than using the hardware-based configuration space and device support to provide virtualization, however, it performs device enumeration by host OS software. In SCVF, the PCI configuration space is not used to enumerate virtual functions. Rather, SCVF simply presents a PCI device to the host OS. The OS loads a card driver for the physical device function. This driver acts as a privileged control driver and implements a virtual PCI bus on which SCVFs are enumerated as full PCI devices. The operating system recognizes the virtual PCI bus, and then everything "just works." When implemented in Linux, the kernel loads the physical function control driver just as for other devices, and no changes were required to Linux or Xen. The existing software stack is used for hot-plugging, device discovery, and PCI device assignment.

An audience member asked how, after an interrupt is generated, it is routed and virtualized. Rolf answered that the host sets up a list of MSI interrupts via the card driver, which emulates MSI, and then relies on the hypervisor to route those interrupts to CPU cores normally. Thus there are two levels of PCI virtualization: their driver, and then the Xen back-end/front-end interrupt virtualization.

■ **SR-IOV Networking in Xen: Architecture, Design and Implementation**

Yaozu Dong, Zhao Yu, and Greg Rose, Intel Corporation

Greg Rose presented a paper on the SR-IOV specification and its application to network devices to provide direct I/O in a virtualized environment.

SR-IOV (Single Root I/O Virtualization and Sharing) is a PCI-SIG standard released in September 2007 for sharing device resources on virtualization-capable hypervisors or kernels. It specifies how a single physical function (PF) device should share and distribute its resources to many virtual functions (VFs). It is not networking-specific but, when properly employed in a network device, should provide the full native I/O bandwidth to a virtualized guest operating system and improve scalability over emulation/paravirtualization as more virtual machines are added.

Greg presented a network architecture that includes the SR-IOV NIC, Xen hypervisor, and individual guest domains. Domain 0 runs the physical function device driver and accesses the physical functions of the NIC, while each guest domain runs a virtual function device driver and accesses a virtual function of the NIC. The physical function

device driver is responsible for controlling all of the virtual function capabilities and providing configuration services. It maintains administrative control of all the Tx/RX queues, and thus it has ultimate responsibility for device security. The virtual function device driver, in contrast, is similar to a normal NIC driver. It serves as an I/O engine in the virtual machine to “pump packets” to the NIC and depends on the physical function driver for most configuration and notification of events.

The presentation concluded with a demo of a functional SR-IOV NIC running in the lab and a discussion of future work. Areas that need further effort include handling a physical function driver reset (such as one caused by a power state transition), because that reset also affects all the virtual function drivers that depend on it. In addition, network-specific management tools are needed to set parameters such as replication, loopback, MAC addresses, and more. One audience member asked where the packets go when two virtual machines on the same host are communicating. Greg replied that the packets are going down to a layer-2 switch fabric in the physical NIC and then going back up to the other guest. Domain0 never sees the intra-VM packets.