Scalable I/O - a Well-Architected Way to Do Scalable, Secure and Virtualized I/O

Julian Satran satran@il.ibm.com
Leah Shalev leah@il.ibm.com
Muli Ben-Yehuda muli@il.ibm.com
Zorik Machulsky machulsk@il.ibm.com

IBM Haifa Research Lab, Haifa, Israel

Abstract

Today in both virtualized and non-virtualized systems the entire I/O functionality is based on device drivers. They are central to any system structure; both anecdotal and informed evidence indicates device drivers as a major source of trouble in the classical OS and a source of scaling and performance issues in virtual I/O, due to "trusted intermediary" required for the shared I/O. We propose an architecture which virtualizes the entire I/O subsystem rather than each I/O device, and provides device-independent I/O at higher level of abstraction than the traditional I/O interfaces. In our suggested architecture the system robustness is increased by isolating drivers; efficient and scalable virtualization becomes possible by a complete separation of the I/O and compute function and introducing a protection model that does not require a trusted intermediary for I/O.

1. Introduction

Today in both virtualized and non-virtualized systems the entire I/O functionality is based on device drivers. They are central to any system structure; both anecdotal and informed evidence indicates device drivers as a major source of trouble in the classical OS and a source of scaling and performance issues in virtual I/O.

In conventional I/O, the I/O functionality is achieved through a combination of simple hardware with only two basic operations supported by the processor/memory complex - namely register transfer and DMA. The structure used to be appealing as the platform hardware support for the I/O was reduced to a minimum. Unfortunately, it brought on very large amount of vendor-specific device driver code, which can no longer be thoroughly tested for each system.

Virtualization brings with it another challenge (and an opportunity for paradigm change). The solutions proposed until now are based on the conventional device driver architecture; they use either dedicated devices directly attached to a specific OS or a "trusted intermediary" for the shared I/O. This position paper suggests an alternative.

We propose an architecture which virtualizes and isolates the entire I/O subsystem rather than each I/O device, and provides device-independent I/O at higher level of abstraction than the traditional I/O interfaces. Similar types of architecture have been used on very large machines, based on substantial specialized hardware (the Channel of the IBM mainframes [1] and I/O processors of the large Control Data and Cray machines). However, those machines had a monolithic structure and scaling them was expensive. In our suggested architecture scaling becomes possible by a complete separation of the I/O and compute function and introducing a protection model that does not require a trusted intermediary for I/O.

2. Existing I/O Virtualization Approaches

The traditional device driver model was extended to support I/O virtualization in several different ways:

- Legacy device drivers are used in fully virtualized systems, based on device emulation [2]. The hypervisor traps the device accesses (such as memory-mapped IO operations) and converts them to operations on a real device, which may be different from the emulated device. This approach does not require changes to the guest OS, at the cost of significant performance degradation due to frequent context switches between the VM and the hypervisor.
- Para-virtualized (``virtual'') I/O drivers are hypervisor-aware I/O drivers, installed in the guest OS [3, 4, 5]. A paravirtualized driver communicates with the real device driver running outside the guest; the level of abstraction is raised from low-level bus operations (such as MMIO) to device-level operations (such as "send a packet"). The performance of paravirtual solu-

tions is significantly better compared to device emulation, but still far from native [6].

- Direct access (also known as pass-through access) device drivers provide guest access to real hardware. A device is dedicated to the guest, which interacts directly with the device, without a software intermediary. This significantly improves performance compared to device emulation or para-virtualized drivers. However, this approach in fact does not virtualize the I/O. The vendor-specific device driver is executed within the guest, which preserves the traditional driver-related problems (such as high development cost and stability issues). Direct access also poses challenges to many aspects of virtualization, in particular it significantly complicates live migration [7, 8, 21].
- Self-virtualizing devices [9, 10, 18, 20] allow direct access interface to multiple VMs; logically, several devices of the same type are packaged together. Since the device driver within the quest OS interacts with the hardware directly, a self-virtualizing device (as a direct-access device) does not really virtualize the I/O. In addition, even though the hardware allows I/O sharing, the supported number of the virtual devices (and accordingly the number of VMs sharing the device) is typically very low, since the self-virtualization support has significant impact on hardware size per virtual interface. Accordingly, this solution scales poorly, while the device cost is significantly increased.

3. Scalable I/O Architecture

Our architecture virtualizes the entire I/O subsystem, and separates I/O execution environment from the VM. The architecture resembles para-virtualization approach [5], but takes it several steps farther: it provides higher level of I/O abstraction, conceals I/O interconnect type, and makes I/O performance improvements possible.

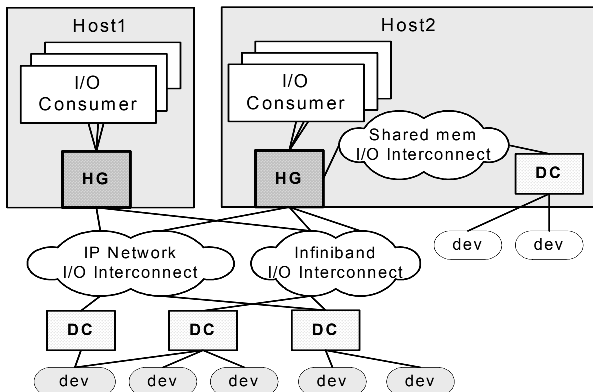

Figure 1. Scalable I/O Architecture

Device Controllers (DCs) implement the I/O services; they control the I/O devices and isolate all device specific code. DCs are logically distinct entities, implemented as separate physical or virtual machines. They can reside on a separate machine in case of a scale-out environment, on a different processor in an SMP, on a separate partition, or even as a separate CPU thread in case of a low cost system. They also can be implemented in hardware, bundled with the device, e.g., as device firmware.

A single DC can provide I/O services to multiple IO consumers simultaneously. An I/O consumer is a “client” of the I/O services provided by DC. DCs execute I/O requests, oblivious to the nature of the requester (i.e., whether it is a VM or a user application). In order to allow secure device sharing, DCs protect the devices from unauthorized access. Device protection is achieved by appending to every I/O request a device capability/credential (issued by a separate I/O management entity), which the DC then enforces. The device credentials involve cryptographic hashes with keys shared between the issuer and the enforcer, and cannot be forged by the I/O consumers. The mechanism is similar to the OSD security protocol [14].

Any user-space or kernel-space application that holds a valid device credential can access the DC services. Such an application is called an *I/O consumer*; it accesses the I/O services using a thin interface layer (an I/O access library). The API may be either legacy API (such as the BSD sockets API in case of network access), or a new API that can take full advantage of the protocol flexibility. Different APIs can be provided for the same class (and for the same device) without any modifications at the device side.

Since the security model does not require a trusted intermediary, I/O consumers can bypass kernel/hypervisor, using techniques similar to existing user-space interface techniques such as Virtual Interface Architecture (VIA [19]) or Infiniband [11].

The I/O consumers interact with DCs through a *Host Gateway* (HG), which can be thought of as an enhanced Infiniband Host Channel Adapter (HCA). The main function of the Host Gateway is to provide a generic protected I/O mechanism to all I/O consumers on the host, i.e., protect consumers from memory access by rogue DCs. An HG is an elaborate DMA engine which provides DMA access to regions of virtual memory of applications and/or VMs, as described in *Section 3.6*. It is similar to DMA engines included in many modern I/O adapters (such as Infiniband adapters), although the memory protection mechanism is different. The DMA operations are secured by address protection code – a cryptographically enforced memory credential, which is generated by the HG when an I/O request is passed from a consumer to a DC, and then validated by the HG when the DC accesses the consumer memory. The mechanism is called *Protected DMA* (PDMA).

PDMA allows remote DMA over different types of interconnect; it allows also local DMA between partitions, which is handled as remote DMA over a shared memory

interconnect. The interconnect network (whenever it is present) is completely hidden from the I/O consumers.

Scalable I/O architecture suits ideally scale-out systems. By splitting the function between compute and I/O nodes we provide the OS a virtualized view of the I/O, allowing great flexibility in tailoring the system characteristics to the application: compute performance, I/O performance, and energy efficiency tradeoff. For example in a cluster environment, an HPC system can be built by coupling high performance compute nodes with low cost nodes for the I/O, while a high performance database server can be built by using high performance nodes as intelligent disk device controllers.

3.1 Scalable I/O Protocol

Interaction between the I/O consumers and the I/O device controllers is achieved using the Scalable I/O protocol, depicted below.

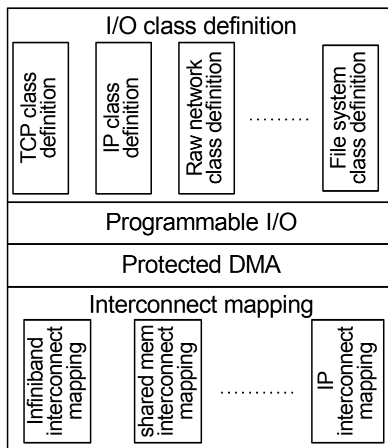

Figure 2. Scalable I/O Protocol Stack

The protocol allows execution of remote I/O programs. The protocol is asynchronous; it allows the consumer to submit multiple I/O programs, to be executed in background. An I/O program consists of one or more I/O commands, and includes general information such as target device ID, unique program ID, and a device credential (which grants appropriate device access rights).

The general structure of an I/O program is common to all device classes, however each device class can "customize" the commands, creating a well-defined class definition. For example, we defined and prototyped a TCP class, which allows efficient offloading of the TCP/IP stack to a DC.

The Protected DMA mechanism enables remoting of the I/O program execution, i.e., shipping the I/O programs from the host to a DC, followed by performing DMA operations (access to consumer memory) requested by the DC. PDMA does not depend on the device class; any class-specific information is passed by the PDMA layer transparently. PDMA is only concerned with I/O commands that access the consumer memory. The PDMA layer appends memory protection bits (memory credentials) to the I/O programs, and validates these credentials during subsequent DMA operations.

PDMA is transport-independent, and can be mapped over a variety of interconnects. Our prototype implements mappings over TCP, Infiniband [11], HiperSockets [12], and a shared memory queues mechanism; in general any cluster interconnect or partition-to-partition interconnect can be used to carry PDMA. The PDMA mapping to any interconnect provides the same reliability guarantees and the same RDMA semantics, which makes it possible to hide the nature of the interconnect from the higher levels.

3.2 Execution Flow of a Scalable I/O Program

A high-level description of an I/O program execution flow is provided below.

The I/O consumer first obtains from the management infrastructure the appropriate device credentials. This operation can be encapsulated within the I/O access library and hidden from the application.

Next, the I/O consumer invokes (through the I/O access library) a simple or compound I/O operation. The I/O access library builds a class-specific I/O program and submits it to the HG (for execution on an appropriate DC). The HG can be built to support submission of I/O programs through it directly from user space, without kernel or hypervisor involvement.

The HG generates memory credentials (which are valid during the lifetime of the program), and appends them to the program, forming a PDMA I/O request. The HG then decides which interconnect interface to use, and passes the PDMA request to the appropriate transport implementation (mapped over the selected interconnect).

The DC receives the I/O program encapsulated within the PDMA I/O request from the interconnect. The DC interprets the program, and invokes the appropriate class handler to execute the I/O commands. The DC enforces the device credentials provided with the program, and only allows execution of an I/O operation if the credentials pass validation successfully. In addition to protection of I/O device resources, the credential may be used also to protect DC resources, for example to limit the amount of DC memory used by the I/O programs executed on behalf of a particular IO consumer.

Whenever the DC needs to access the consumer I/O buffers, it invokes asynchronously PDMA memory access operations at the HG, specifying the memory credentials which were provided with the I/O program. The HG validates the memory credential before executing a PDMA memory access request.

After the I/O program execution is done, the DC sends to the consumer a status message. This message notifies the consumer of program completion, and also notifies the HG

that the memory credentials associated with this program are no longer valid.

3.3 High-Level I/O Interface

Many of the problems of the existing I/O virtualization approaches are the result of the low level of I/O abstraction in the commodity systems. A typical device normally provides a low-level logical interface (such as "send a network packet"), implemented by the vendor-specific device driver using very-low-level hardware primitives (bus-level memory-mapped I/O and DMA operations). The existing virtualization techniques are based on either implementing the former (para-virtualization), or emulating the latter (full virtualization), both with significant overhead.

In contrast with the existing virtualization techniques, we choose to raise the abstraction level, and to "offload" higher layers of the I/O stacks. For example, handling network devices in Scalable I/O is conceptually different than conventional virtualization. While the current virtualization approaches create a Virtual NIC that provides layer 2 services, we provide network (layer 3) or transport (layer 4) services instead.

The high-level interface significantly reduces interaction between the VM and its virtualized I/O implementation, and thus significantly reduces the overhead of I/O virtualization. In addition, it allows a more efficient implementation of the I/O stack in modern multicore CPUs, by isolation the I/O subsystem in its own environment. The high-level interface allows us to allocate dedicated resources for demanding high-level I/O tasks and avoid the very high cost of blind CPU sharing (cache pollution, privilege-level boundary crossing and locking). I/O stacks that consume a substantial amount of computation on conventional architectures (e.g., the TCP/IP stack or a file system) can perform much better in a confined environment (e.g., an isolated core or thread), than in shared environment [16].

Unlike with other types of offload, such as a TOE (TCP offload engine), in our approach the I/O stacks are not moved to an inferior platform (such as an embedded platform using a weak CPU and/or an accelerator with slow access to the main memory). Also, multiple DCs can be used, thus the isolated subsystem does not become a bottleneck.

We prototyped this concept for TCP/IP stack in two different environments: using a dedicated CPU on an SMP, and using a dedicated node in an Infiniband-connected cluster. Significant performance improvement was achieved in both cases. A detailed evaluation and performance analysis is under preparation.

3.4 Programmable I/O Interface

The Scalable I/O model allows execution of I/O programs on the DCs. An IO program can be written using a limited command set. For example, it can be a simple sequence of basic I/O commands, similar to the *compound operation* in NFSv4 [13]. It also can be a sequence of basic I/O operations and condition testing – similar to the channel programs in mainframe Channel I/O [1] (this model was implemented in our TCP class prototype). A more sophisticated I/O program model can allow sandboxed execution of code snippets (written in a full fledged programing language) at the DC – similar to remote data processing mechanisms such as MapReduce [17]. Unlike the Channel model, we do not require a trusted intermediary; unlike the other remote data processing mechanisms, we include an effective protection mechanism for devices, DC resources and host memory.

Executing I/O programs rather than simple I/O requests has several benefits. First, it provides significant performance improvement for any processing besides bulk data transfers. Multiple control operations or control and data operations can be combined, which reduces the interaction overhead. Command combining also reduces latency, as it eliminates the round-trip delay otherwise associated with execution of a sequence of dependent operations. Complex I/O programs can be used for analytical processing of data (executed near the data), to offload significant amount of work from the host, to decrease amount of data transferred to or from the host, and ultimately to build more scalable systems.

An additional advantage of programmable I/O is flexibility to support future requirements. Without programmability, it would be necessary to define comprehensive command sets which would then need to be extended frequently as new requirements evolve (as exemplified by SCSI). Instead, we propose to use a limited command set together with a means to combine the commands, to allow far greater flexibility.

3.5 Making Remote Execution Efficient

A naive implementation of communication between the I/O consumers and the I/O subsystem can be very inefficient. Our goal was not only to avoid the performance impact, but moreover to improve performance. Even though the architecture introduces a new layer (communication channel between the I/O subsystem and I/O consumers), the overall performance can be improved due to significant reduction of direct and indirect costs imposed by the traditional I/O structure. In many scenarios, the savings due to subsystem isolation can outweigh the cost of the (lightweight) communication, as was shown previously for TCP stack isolation [15, 16].

The communication overhead is kept low by using direct access from the consumer to HG. Additionally, the amount of interaction with the I/O subsystem is relatively low due to high-level command interface to I/O devices (as

described in *Section 3.3*), and due to programmable I/O (as described in *Section 3.4*).

Given an efficient communication channel, subsystem isolation allows significant decrease in several types of performance costs:

- The overhead of OS/hypervisor boundary crossings is significantly reduced by a direct access interface to the I/O communication channel, as described above.
- The cost of sharing the CPU between I/O subsystem (driver, kernel services and higher level functions such as a protocol stack) and the rest of the system is eliminated, as each DC uses its own resources. Using dedicated CPUs prevents cache pollution. It also allows to reduce the amount of context switches due to interrupts, since the isolated device controller can work in polling mode most of the time.
- The overhead of DMA access setup (registration of I/O buffers or copying I/O data) is decreased by using Protected DMA mechanism with "dynamic pinning" which allows on the fly memory registration, as explained in *Section 3.6*.

3.6 Efficient and Protected Memory Access

The setup of DMA operations is a complex and expensive operation on traditional systems, and even more so on virtualized systems. On many systems, the setup time can be high enough to warrant a copy of consumer data to or from statically pre-registered memory areas used for I/O.

While CPU accesses to memory go through virtual memory translation, a DMA operation usually accesses physical memory directly[1]. In order to synchronize physical memory accesses between CPUs and devices, consumers need to take extra care when initiating and completing I/O operations. A consumer must "pin" the portion of memory it is about to access. Pinning locks a shared resource – physical memory – for exclusive access by that consumer, for an unbounded amount of time. Once the consumer is done using that portion of physical memory, the consumer must "unpin" the memory and release it back to the system.

Memory pinning wastes both CPU cycles and memory. It wastes cycles, since it requires context-switching into privileged software (such as an OS or a hypervisor), and it wastes memory since once a portion of physical memory is pinned it is no longer available to the rest of the system. In addition to being wasteful, it is also complex to use and places an additional burden on the developer. Device driver errors, such as unpinning memory before DMA is completed, can have devastating effect on the whole system.

DMA setup is even more complicated when the DMA addresses undergo translation, either by an IOMMU that provides access into one or more I/O address spaces, or by an adapter that allows DMA to consumer virtual addresses. A separate translation table has to be maintained (either in the host or in the adapter or IOMMU). In these cases, in addition to pinning physical memory, it is also necessary to register the memory at the DMA translation tables, and to keep both CPU and I/O page tables in sync.

Our goal is to avoid the need for explicit memory pinning and registration. We define a mechanism that allows DMA access to consumer virtual addresses, using existing MMU translation tables for virtual-to-physical translation during the DMA operations.

Unlike Infiniband, PDMA does not require memory preregistration and using memory keys for protection. Unlike Address Translation Services (ATS) as defined by the PCI-SIG IOV workgroup ([10]), PDMA does not perform translation based on device identity, and does not hand the translated addresses to the devices. Instead, the PDMA request carries secured information on the address space of the application or VM which initiated the original I/O request. The DMA address and the address space ID are protected using the memory credentials mechanism; the credentials are generated on the fly when the I/O consumer generates an I/O request, which eliminates the need in pre-registration or maintaining the DMA translation tables.

The mechanism does not require explicit pinning. A "dynamic pinning" protocol ensures that a mapped page is not unmapped while a DMA access is in progress, and most of the complexity is moved to a rare unmap operation.

The dynamic pinning protocol introduces the concept of an I/O page fault. HG raises an I/O page fault when an I/O device tries to access a non-present memory page on behalf of a consumer. Privileged software is responsible for handling the page fault and restarting the I/O access by HG. The PDMA protocol allows the HG to slow down the DC if too many I/O page faults are handled concurrently (although this is assumed to be rare). The I/O devices at the DC do not need to be aware of the page faults at the host, as the PDMA protocol takes care of this.

4. Summary and Conclusions

The proposed architecture inherently simplifies I/O virtualization and improves I/O efficiency, security and scaling in virtualized and non-virtualized environments. The I/O access libraries provide the OS and the applications with a generic view of I/O, while the device specific details managed by the, logically remote, Device Controller. As a result, the networked nature of modern I/O is hidden from the OS and is visible only to the interface elements (Host Gateways). Security is achieved by using a capability/credential mechanism to access both memory and devices, with the I/O infrastructure being responsible for enforcing credentials (some I/O components will protect memory while others will protect devices or

[1] Bus addresses may be different from physical addresses on some architectures, however we ignore simple bus-to-physical address transformations such as a constant offset when discussing untranslated DMA access to physical address space.

resources within devices). Since access to devices and resources within devices is protected, there is no need any more for virtualizing device adapters; device/resource access is done in the same way from a user process, an OS, or a OS hosted by a VM.

5. Acknowledgements

The Scalable I/O architecture was developed by a team that, in addition to the authors and in various stages, included Ton Engbersen, Scott Guthridge, Orran Krieger, Vadim Makhervaks, Ilan Shimony, T Basil Smith and John Tracey – all with IBM at the time the work was conducted. The security mechanisms underwent a deep review with a team from the IBM Zurich Research Lab led by Michael Waidner.

Additional people worked on different prototype implementations and a large group of people contributed to this work through their review, comments and valuable insights. We are grateful to all of them. We especially thank those that pointed to us various implementation alternatives for different processors and subsystems (Jimi Xenidis, Hubertus Franke, Ed Seminaro, Greg Still, Alan Benner, Tom Bradicich – all from IBM). We would like to express our special thanks to Micky Rodeh (IBM) for making this work happen and the management team from IBM Research, and especially the IBM Research Laboratory in Haifa for standing by us for several years.

6. References

[1] R. Cormier, R. Dugan, and R. Guyette. System/370 Extended Architecture: The Channel Subsystem. IBM Journal of Research and Development, 27(3), p 206-218, May 1983.

[2] J. Sugerman, G. Venkitachalam, B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, p.1-14, June 2001.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, Oct. 2003.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, p. 225-230, July 2007.

[5] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. In *ACM SIGOPS Operating Systems Review, Volume 42 Issue 5*, July 2008.

[6] J. Santos, Y. Turner, J. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference*, 2008.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, p.273-286, May 2005.

[8] E. Zhai, G. D. Cummings, Y. Dong. Live Migration with Pass-through Device for Linux VM. In *Proceedings of OLS '08: The 2008 Ottawa Linux Symposium*, July 2008.

[9] H. Raj, K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the 16th international symposium on High performance distributed computing*, June 2007.

[10] PCI-SIG I/O Virtualization specifications, http://www.pcisig.com/specifications/iov/

[11] InfiniBand™ Trade Association, InfiniBand™ Architecture Specification Release 1.2, October 2004.

[12] M. E. Baskey, M. Eder, D. A. Elko, B. H. Ratcliff, and D. W. Schmidt. zSeries Features for Optimized Sockets-Based Messaging: HiperSockets and OSA-Express.In *IBM Journal of Research and Development.46, No. 4/5*, p. 475-485, July/September 2002.

[13] B. Pawlowski, D. Noveck, D. Robinson, R. Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*).

[14] M. Factor, D. Nagle, D. Naor, E. Reidel, J. Satran. The OSD security protocol. In *Proceedings of 3rd International IEEE Security in Storage Workshop (2005).*

[15] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, I. Shimony. Loosely Coupled TCP Acceleration Architecture. In *Proceedings of 14th IEEE Symposium on High-Performance Interconnects (HOTI'06)*, p. 3-8, Aug. 2006.

[16] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, A. Foong. TCP onloading for data center servers. In *IEEE Computer*, November 2004.

[17] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, p. 137-150, 2004.

[18] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel, Concurrent direct network access for virtual machine monitors, in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 306-317.

[19] Philip Buonadonna, Andrew Geweke, David Culler. An Implementation and Analysis of the Virtual Interface Architecture, In *Proceedings of SuperComputing '98.*

[20] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines, in *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2006, p. 3

[21] W. Huang, J. Liu, M. Koop, B. Abali, and D. Panda. Nomad: migrating os-bypass networks in virtual machines, in*VEE '07: Proceedings of the 3rd international conference onVirtual execution environments*. New York, NY, USA: ACMPress, 2007, pp. 158-168.