Taming Heterogeneous NIC Capabilities for I/O Virtualization

Jose Renato Santos Yoshio Turner Jayaram Mudigonda

Hewlett Packard Laboratories, Palo Alto, CA

{*joserenato.santos,yoshio.turner,john.janakiraman*}*@hp.com*

Abstract

The recent emergence of network interface cards (NICs) with diverse hardware features for I/O virtualization poses an important challenge for virtual machine environments, particularly in the area of system management. In this paper, we make the case for developing a high-level network I/O virtualization management system that can translate user-relevant policy specifications into the hardware and software-specific configurations that are needed on each particular hardware platform. As a first step toward this goal, we describe and classify configuration options that are presented by a wide variety of mechanisms for NIC hardware support for virtualization, and discuss workload and policy considerations that should be factored into configuration decisions. In addition, we propose new mechanisms for intra-node guest-to-guest networking that leverage NIC hardware switching support, and we present a unified system architecture for network I/O virtualization that exposes the configuration options that we identified to high-level management layers.

1 Introduction

Emerging network interface cards (NICs) provide hardware support for virtualization which enables the NIC to be shared efficiently and safely by multiple guest domains. Specifically, these NICs provide numerous descriptor queues where each queue can be assigned a distinct Ethernet MAC identifier and be dedicated to handle the traffic for a particular guest domain. This allows the addresses of guest domain data buffers to be posted directly to the NIC, avoiding the overheads of traditional software-based I/O virtualization, particularly for the packet receive path.

These emerging NICs exhibit considerable variety – in the software interface they provide to expose the multiple queues, and in the switching and traffic management functionality they may provide to complement the use of multiple queues. For example, some NICs appear to the virtualization software (e.g., the hypervisor, or a device driver domain) as a single PCI device with multiple contexts, while others present as multiple PCI devices (one device per context). For data protection between guest domains, some NICs have built-in memory address translation while others rely on host IOMMUs or software isolation mechanisms. Some NICs have advanced layer-2 (or higher) switching capabilities, possibly including advanced firewalling and/or traffic shaping capabilities. This diversity is likely to persist as the industry searches for the ideal feature set to satisfy different customer needs. Moreover, as we discuss later, the right mix of hardware and software features to use will continue to be workload and system dependent, even if NIC functionality were to stop evolving.

The growing feature sets and diversity in modern NICs pose a significant challenge for virtual machine environments like Xen. To enable Xen to use each new feature requires large modifications to the network I/O virtualization software architecture and to the system management tools. More importantly, the user is burdened with the task of modifying guest virtual machine configurations and possibly driver domain configurations to actually make use of the new NIC hardware feature. The resulting configuration is brittle since it is customized to the hardware features of the particular physical machine on which the guest will execute, and is therefore poorly matched with other valuable functions such as live migration that are provided through virtualization. We claim that this friction imposes a barrier to innovation and adoption of new network I/O virtualization mechanisms, particularly in complex data center environments.

We advocate that an attractive goal which would solve this problem is to develop a high level network I/O virtualization management system. We envision that this manager would run as an agent in a privileged management domain, for example Domain 0 in Xen. The manager would relieve users of the need to make decisions and configurations that are customized to the underlying hardware capabilities. Instead, the manager would allow users to specify policies at a high level and then determine the appropriate low-level configurations specific to the particular hardware environment that would

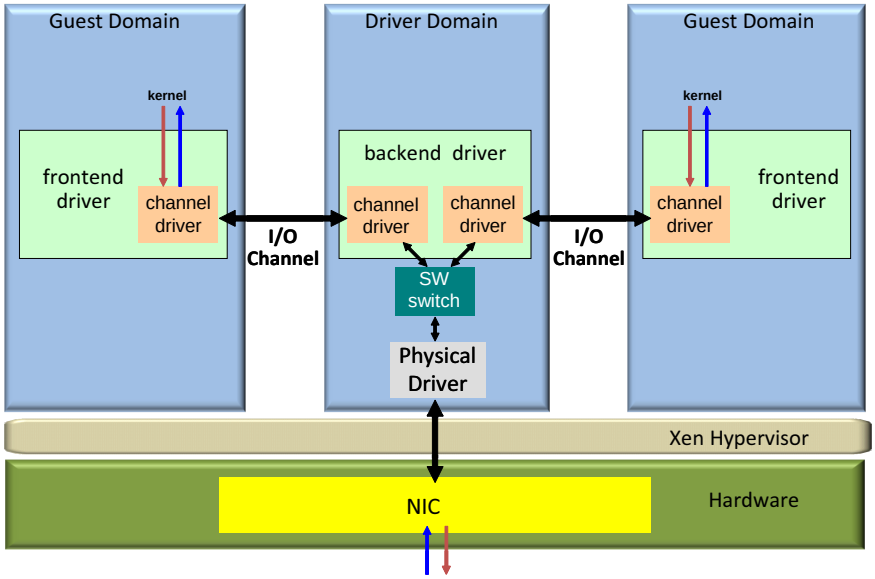

Figure 1: I/O virtualization with traditional NIC

implement the policies. Thus, the manager would provide a clean separation between user-relevant policies, and the hardware and software mechanisms that are used to implement the policies. For example, the user could indicate which networks (i.e., which physical networks or VLANs) each guest should be able to access, instead of specifying which software bridge to use or whether to use a particular NIC descriptor queue. For another example, the user could specify firewall or traffic shaping rules to apply to the guest's traffic, and the manager would automatically translate the rules to the appropriate settings in the software or the NIC.

In this paper, we present some early steps toward this ambitious goal of providing a high-level network I/O virtualization management system. In particular, we identify a large configuration space for the network I/O virtualization subsystem, and we illustrate several important constraints and trade-offs that must be considered to determine the best configuration settings. We organize the configuration space into four basic I/O virtualization functions: NIC virtualization, packet switching, data transfer, and traffic management. Each function can be implemented either in hardware or in software, and by different software components such as guest device drivers, driver domains, or the hypervisor. We discuss how factors such as performance, resource availability, and high-level network management policies influence or constrain the choice of where each function should be implemented on a particular hardware platform. For example, the number of descriptor queues on a NIC limits the number of guest domains that can be assigned to dedicated queues. As another example, if the system administrator policy requires using firewall rules, and the NIC does not support this capability in hardware, then packets should be switched in software instead of using the NIC switching hardware.

In addition to examining the configuration space, we propose some new mechanisms and a software architecture that would expose the configuration options to the manager (or a user, until the manager is developed). The new mechanisms we propose are inspired by the novel separation of the packet switching and data transfer functions that we listed above. In particular, we propose *envelope switching* which performs hardware-based intra-host guest-to-guest packet switching in a NIC but carries out the associated packet data transfer in software (or by another DMA engine) to relieve the I/O bus bottleneck. To support safe software-based packet data transfer between guests, we propose and discuss trade-offs for three potential extensions to the Xen grant mechanism that can be used with hardware-based or software-based packet switching. Finally, we present an architecture that can be configured to support all the types of NIC hardware extensions and software mechanisms that we describe in this paper.

2 Network I/O Virtualization Functions

A network I/O virtualization subsystem provides four main functions:

- **NIC virtualization:** Provide guests with virtual network interfaces (vNICs) enabling then to share a single physical network interface card (NIC) which provides access to the external network.
- **Packet switching:** Switching of packets between local guests for intra-node traffic as well as between guests and the physical NIC.
- **Data transfer:** Capability of transferring packet data between local guests.

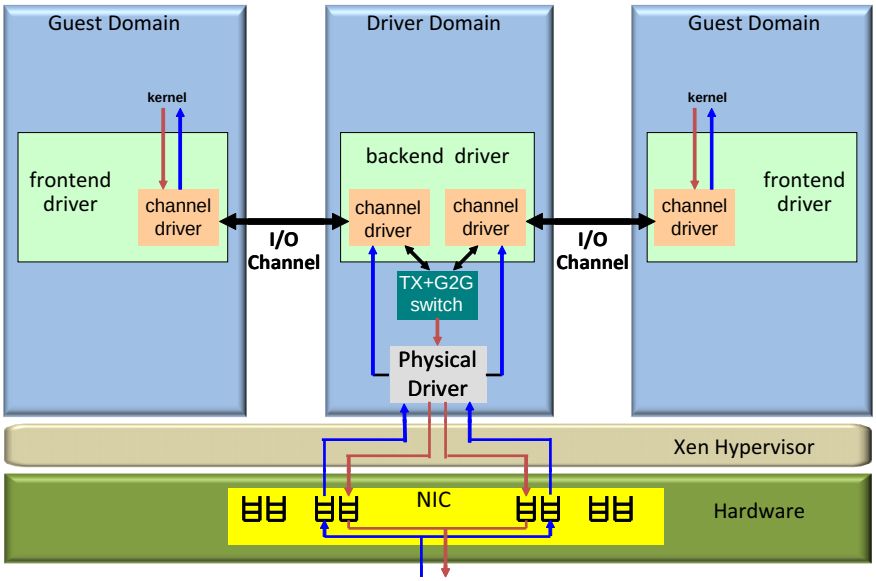

Figure 2: I/O virtualization with multi-queue NIC

- **Traffic management:** In contrast with the other functions this is an optional function, but usually strongly desired by system administrators. Comprises network security mechanisms such as firewalls, VLAN isolation, etc., and mechanisms to provide I/O bandwidth scheduling such as traffic rate control, etc.

3 Configuration Choices

All of these four functions have traditionally been provided by a software virtualization layer either inside the hypervisor (e.g. VMware, KVM) or in a special virtual machine called driver domain (e.g. Xen, Microsoft Hyper-V). More recently, NIC hardware has begun to provide a variety of sophisticated support for I/O virtualization [4, 1, 2]. Ian Pratt classified NICs in four different types based on their level of hardware support for virtualization[6]: **Type 0:** traditional NIC without support for virtualization (Figure 1), **Type 1:** multi-queue NIC (figure 2), **Type 2:** Direct I/O NIC (Figure 3) **Type 3:** Direct I/O NIC with hardware switch (Figure 4). We expect that in the future NICs will also provide hardware support for QoS and security functions such as traffic shaping and firewalls [5].

The main challenge in exploiting this wide variety of powerful hardware is to design a software architecture that, by being highly configurable, facilitates the implementation of high-level policies and efficient use of the underlying hardware. The rest of this section examines the important design and usage trade-offs such an architecture must take into account.

3.1 Traffic Management

Guests belonging to mutually distrusting customers, with different performance requirements, can be co-located on the same physical machine. Such systems often require rate control to enforce I/O bandwidth scheduling, security-related network filtering mechanisms (e.g. firewall), and subnet isolation mechanisms such as VLANs; and these mechanisms must be enforced outside of guest control. The mechanisms must be co-located with the switching function as they need to intercept every packet. Therefore, if a *type-3* NIC lacks hardware support sufficient to implement the desired high level policies, the system must fall back to software switching and the guest cannot use the direct I/O functionality. Instead, the NIC would need to be configured as a multi-queue NIC with all virtual interfaces allocated to the driver domain. The software bridge in the driver domain would be configured to enforce the desired traffic management mechanisms.

Ideally, a language and syntax would be developed that would enable users to submit high-level specifications of traffic management requirements to the I/O virtualization management subsystem. Additionally, there must be a way to specify or probe device capabilities and to test whether the capabilities on a particular system can satisfy the specified requirements. Finally, the manager needs to be able to configure the hardware and software to exploit the matching capabilities in the appropriate way. All of these operations are challenging open areas for investigation.

Admission control mechanisms and policies are often used to prevent new workloads from being deployed on a system that would cause violations of service level objectives. Often, the mechanisms used to enforce traffic management policies impose a reduction in the achievable performance. For example, a mechanism that re-

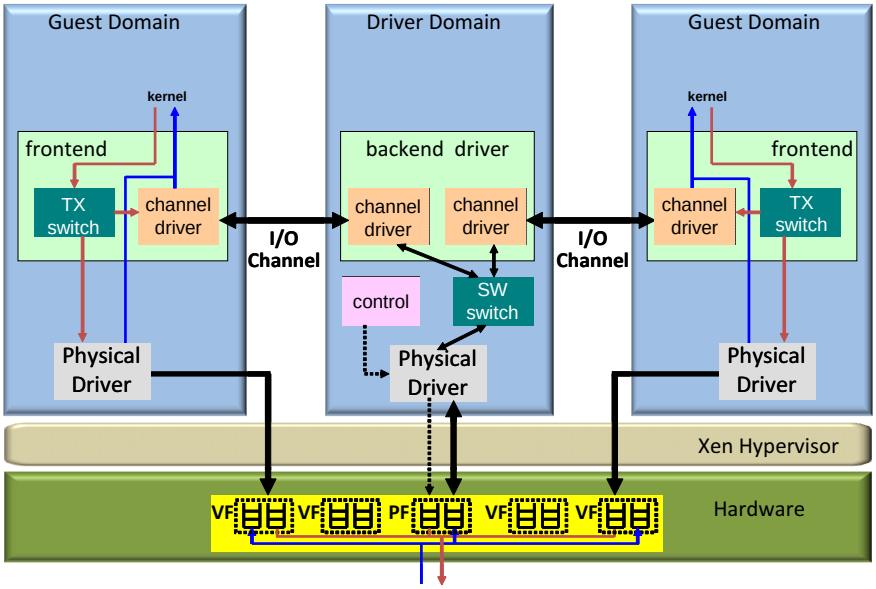

Figure 3: Direct I/O

quires each packet to be examined in software will likely reduce the maximum packet rate that can be sustained. Therefore, the performance impact of traffic management mechanisms must be factored into admission control decisions. This is another open area of investigation.

Apart from the QoS and security requirements that are applicable to all the traffic, the I/O subsystem configuration involves the following important factors that are specific to guest to guest (intra-machine) traffic.

3.2 Switching

We define switching as the process of identifying the receiving guest. Switching can be performed in the NIC, in the virtualization software, or in the source guest itself. These options are not mutually exclusive – some pairs of guests can communicate through the NIC, while others communicate through the virtualization software, and still others communicate via direct guest-to-guest software channels. We next describe the trade-offs involved in choosing these options.

- **Switching in the NIC.** Local guest to guest packet switching can be done by a *type-3* NIC unless, as described above, the traffic management policies cannot be enforced by the hardware. Offloading the switching to hardware reduces the amount of CPU resources consumed by the I/O virtualization subsystem, but it increases the load on the I/O bus/fabric (e.g., PCI Express). As we describe later, the severity of this trade-off can be reduced, though not eliminated, by decoupling the switching mechanism from the data transfer mechanism.

- **Switching in the virtualization software.** It may be preferable to do packet switching in software (e.g. in the hypervisor or driver domain) if the I/O subsystem is overloaded and becomes a scarce resource. This option may become more attractive in the future as the number of available CPU cores increases making CPU cycles a cheap resource when compared to a shared I/O subsystem. Software switching is also required in cases where the number of guests exceeds the number of virtual interfaces supported in hardware.

- **Switching in the source guest.** Packet switching can also be done in the guest[3]. A guest can have direct shared memory channels with other guests and forward packets directly to the appropriate destination guest, avoiding the cost of executing code in a third party privileged entity such as the hypervisor. This may be even more beneficial when using the driver domain model which usually has higher cost than when using the hypervisor. Even when using a *type-3* direct I/O NIC it may be beneficial to bypass the hardware switch and route packets to other guests directly in the guest virtual device driver, to reduce load on the I/O bus/fabric and avoid the latency of the NIC switch. Direct guest to guest channels can provide good performance but are not appropriate if traffic management rules such as firewalls have to be enforced by the virtualization layer. Using direct guest to guest channels exclusively may not be the best choice for systems with large numbers of guests, since each guest would pay a cost to process events across a large number of channels. Also, the required number of channels scales as the square of the number of guests. Therefore, an attractive configuration for a large number of guests is a combination of direct guest channels for guest pairs with high traffic intensity and switching outside the guest for the remaining guest pairs.

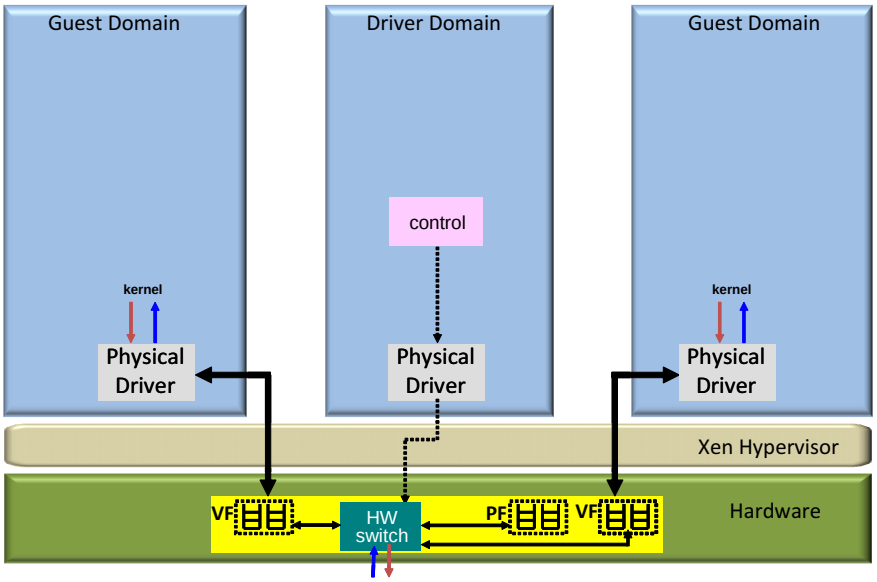

Figure 4: Direct I/O with HW switch

3.3 Data Transfer

Although physical switches transfer data while switching packets, it is useful to separate data movement from the switch function in a virtual machine environment. For local guest to guest communication, the packet data and the destination data buffer are both located in the memory subsystem. Therefore, the data transfer can be performed using a direct memory copy that avoids transferring packet data through switching intermediaries. The switching mechanism only needs to forward metadata containing handles that point to the packet data. We call this approach envelope switching.

Envelope switching can improve the efficiency of the I/O virtualization subsystem. For example, in the current version of Xen (3.3), packets are switched using the software bridge in the driver domain, and packet data is copied from the sender guest memory to the destination memory in the backend driver (which is also located in the driver domain). Recent work[7] has shown that it is more efficient to do the data copy in the destination guest than in the driver domain. This is because in a SMP system the guest and the driver domain are likely to be executed in different CPUs and doing the copy in the receiving guest has the advantage of bringing the data to the guest CPU cache. The guest CPU that is likely to access the data later and thus can benefit from a cache hit. Enabling the data transfer to be done later after the packet is routed and delivered to its destination thus allows the use of more efficient mechanisms.

Envelope switching can also be used with a hardware switch in the NIC. The switch just needs to forward the metadata to the destination virtual interface and software can do the copy after the "envelope" is received at the destination guest. The advantage of using envelope switching instead of regular hardware switching is that it avoids the cost of transferring the data through the I/O bus (or PCIe fabric) from the source memory to the NIC and then back to the destination memory. Envelope switching should enable higher data rates for intra node traffic as the I/O bandwidth is usually lower than the memory system bandwidth.

We observe that hardware envelope switching does not require special support in the NIC, except for the already existing switching capability between virtual interfaces. The guest can transfer the metadata with data pointers as a small regular ethernet packet with a special type used to distinguish envelopes from regular packets. The receiver could then detect that the packet contains metadata and use a safe data copy mechanism that transfers the data from the source buffer at the sender to the local receiver buffer. The receiver should also send a small packet back to the sender after the data copy is complete to notify that the buffer can be freed.

Data transfers need to be protected and constrained to memory that belongs to the guest for which the I/O is performed to provide safe isolation between virtual machines. If using a *type-3* NIC with traditional switching in hardware, an IOMMU is required to ensure DMA operations are safe. If however envelope switching is used, a safe data copy mechanism provided by the hypervisor is required instead. In Xen, the *grant mechanism* enables safe data copies in software.

In general, the choice of using *envelope* or *traditional* can also be a configuration choice and depend on which resource is higher demand, the I/O bus or CPU. This choice can even be dynamic based on workload conditions. In this case, it is important to have a memory protection mechanism that is unified across hardware and software. In Xen, this can be accomplished by having both grant table entries and IOMMU table en-

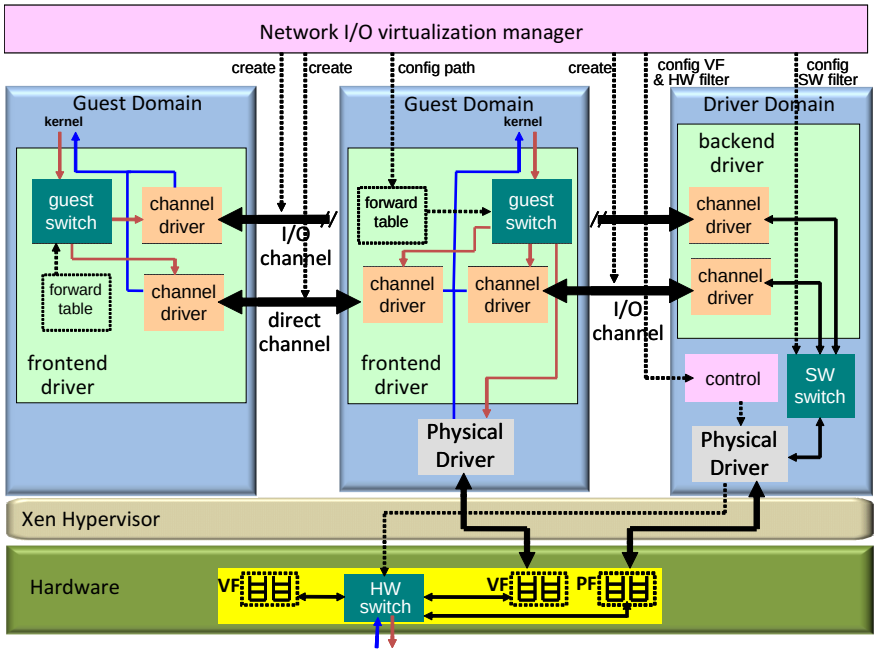

Figure 5: Flexible Network I/O Virtualization Architecture

tries represent the same address space and have the same translation in both tables. Therefore the guest could use the same pseudo-physical address regardless if the data transfer is done by hardware or by software.

We note however that the current Xen grant mechanism needs to be extended to be used with envelope switching. The grant mechanism requires that the source domain specify which domain is granted access to its page. Although this works fine with regular switching where the data copy is performed by the driver domain, it creates difficulties for envelope switching since the destination domain is not known until the switching function is performed.

To enable safe data copy at the destination guest with envelope switching we propose *guest destination mapping*. In this approach, the transmitting guest device driver performs a switching function to map the packet destination to the domain id of the destination guest if it is local. The guest domain is provided with a read-only hash table that maps each destination MAC address to the corresponding destination guest domain id. This allows the transmitting guest to issue a grant to the correct destination guest domain. This approach requires minimal hypervisor changes and can be used with hardware envelope switching. Although it requires additional support in the guests, these changes can be contained in the virtual device driver.

We considered and rejected two alternatives to our guest destination mapping approach. As discussed in [7] the grant mechanism can be extended to enable grant transitivity. With this mechanism a domain with special privilege such as the driver domain could transfer the right to access a granted page from one guest to another guest. This could be accomplished by creating a special grant which instead of specifying a physical page owned by the domain specifies a grant provided by another domain. The hypervisor would then check if both the original and the indirect grant are valid when the destination domain requests a data copy using the grant. The main advantages of this approach is that it requires no support in the guest and the extension to the hypervisor grant mechanism is relatively simple. However, this approach cannot be used with envelope switching in hardware.

The second alternative that we considered was to modify the grant mechanism to allow grants with keys. These special grants would be associated with keys instead of using domain ids. The key would be transferred with the grant reference to the destination domain. The hypervisor would then allow the grant to be used by any domain which had a valid key, but the grant would be automatically revoked after its first use to prevent unauthorized use. The main advantage of this mechanism is that it can be used with hardware envelope switching, but has the disadvantage that significant modifications to the grant mechanism would be required in both the hypervisor and guests.

4 A Flexible Network I/O Virtualization Architecture

The right choice of all the configuration options we described must be determined based on high level policies, hardware capabilities, resource availability, workload conditions, etc. To enable this vision, we need a flexible network I/O virtualization architecture that al-

lows all possible configurations that could be chosen by a network I/O virtualization manager as illustrated in Figure 5.

The architecture shown in Figure 5 has the flexibility to do packet switching in any of the modes described in 3.2. When a guest is created or migrated to a different machine, a network I/O virtualization manager running in a privileged domain (e.g Xen domain 0) configures the I/O virtualization mechanisms based on system policies and resource constraints of the current machine using a control plane such as the xenstore/xenbus mechanism in Xen. The manager can also change the configuration while the guest is running depending on workload condition changes. The manager creates I/O channels between guests and a driver domain and/or between guest pairs depending on the selected configuration. If a direct I/O option is selected and a virtual function (VF) is assigned to the guest, the manager configures the NIC virtual function through a control agent running in the driver domain. In addition the manager informs the frontend driver which network interface should be used for the direct I/O path. In this case the frontend driver configures itself as a bonding device integrating the physical device driver interface and one or more I/O channels as a single virtual network interface that is exposed to the guest kernel. The manager also configures the software switch in the driver domain and the hardware switch in the NIC (if used) with the desired traffic management policies. The manager also exports a shared memory forwarding table to each frontend driver, which implements a hash table that maps destination MAC address to output channel and destination guest domain id. This table lists virtual interfaces that are co-located in the same physical machine which are allowed to communicate, i.e. that belong to the same logical network or VLAN. This table serves two purposes. First it is used by a guest switch to select the route to reach a particular virtual interface given its MAC address. For example if the NIC has a hardware switch, the forwarding table could indicate if a packet to a co-located guest should use a particular I/O channel or the physical interface depending if both source and destination guests have a direct I/O path to the NIC or not. If the destination MAC address is not local then it will not be present in the forwarding table and the frontend will choose a default route for external traffic which could be an I/O channel to a driver domain or a physical interface depending on the configuration. In addition, the forwarding table also exports the domain ids associated with the local virtual interfaces which is necessary to support envelope switching. Using the domain id the sending guest can selectively grant I/O buffer access to the receiving guest, which then can access the transmitted data after the packet is delivered, regardless if the packet is transmitted over an I/O channel or over a physical interface.

By changing the default route to external traffic and relevant forwarding table entries it is possible to dynamically switch a guest from using direct I/O mode to using driver domain mode, or vice-versa. This flexibility can be used to dynamically change which guests are assigned to a direct I/O NIC context based on workload changes, when the number of guests exceeds the number of hardware contexts. More importantly, this flexibility enables live migration of guests to machines with heterogeneous hardware capabilities, similarly to the techniques described in [4] and [8]. For example, if the source machine has a NIC with direct I/O capability but the destination machine does not, the frontend driver can switch to driver domain operating mode after migration. On migration the manager can map the same high level policy requirements to the new resource constraints of the destination machine and appropriately configure the I/O virtualization mechanisms transparently to the guest operating systems (except for the mechanisms supported in the frontend driver).

5 Conclusion

In this paper we elaborated the challenges posed to users and developers of virtual machine environments by the emergence of diverse hardware support for network virtualization in modern NICs. We believe that our work is a promising first step toward providing a high-level management layer that would shield users from this complexity, but much future work remains to achieve this goal. While our system architecture exposes several configuration options to higher level management, it may need further extensions as new hardware or software techniques for I/O virtualization are invented. In addition, big open questions include how policies should be specified, and what techniques can be developed to automatically map policies onto mechanism configurations. We hope that our work can stimulate further investigations in this important problem area.

Acknowledgement

We would like to thank Ian Pratt, Steven Smith and Keir Fraser for helpful discussions during the design of Xen Netchannel2 which inspired some of the ideas described in this paper.

References

[1] BESTLER, C. Virtual networking via direct assignment, neterion. In *6th Xen Summit* (Jun 2008).

[2] CHINNI, S., AND HIREMANE, R. Virtual machine device queues. `softwaredispatch.intel.com/?id=1894`. Supported in Intel® 82575 gigE and 82598 10GigE controllers.

[3] GOPLAN, K., AND REDHAKRISHNAN, P. Mechanisms to improve inter-vm network communication performance. In *6th Xen Summit* (Jun 2008).

[4] MANSLEY, K., LAW, G., RIDDOCH, D., BARZINI, G., TURTON, N., AND POPE, S. Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains. In *Proceedings of Euro-Par 2007 Workshops: Parallel Processing* (2007).

[5] NEUGEBAUER, R. Network topology offload, netronome. In *6th Xen Summit* (Jun 2008).

[6] PRATT, I. Support for smart nics. In *4th Xen Summit* (Apr 2007).

[7] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND PRATT, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference* (June 2008).

[8] ZHAI, E., CUMMINGS, G., AND DONG, Y. Live migration with pass-through device for linux vm. In *Open LInux Symposium* (2008), pp. 261–267.