Challenges for Inter Virtual Machine Communication

Carl Gebhardt and Allan Tomlinson

Technical Report
RHUL–MA–2010–12
01 September 2010

Abstract

Past research in virtualisation technology has mainly focused on increasing isolation of co-resident virtual machines. At the same time network intensive applications, such as web services or database applications are being consolidated onto a single physical platform. The isolation properties of virtualisation, however, demand a strict separation of the shared resources. Co-resident virtual machines are therefore forced to fallback to inefficient network emulation for communication. Many inter virtual machine communication methods proposed recently, introduced shared memory, customised libraries or APIs. This is not only unpractical but can also undermine a system's integrity; moreover transparency and live migration is commonly neglected. Therefore in this paper we discuss the challenges and requirements for inter virtual machine communication and examine available solutions proposed by academia and industry. We also discuss how the current evolution of virtualisation and modern CPUs pose new challenges for inter virtual machine communication. Finally, we consider the possibility of utilising previously unused CPU capabilities to accommodate an inter virtual machine communication mechanism.

1 Introduction

One of the outstanding properties of virtualisation is it's ability to isolate co-resident Operating Systems (OS) on the same physical platform. While isolation is an important property from a security perspective, co-resident virtual machines (VMs) often need to communicate and exchange a considerable amount of data. Despite ever growing accomplishments to improve VM performance, research has shown that network virtualisation performance is still poor [1]. As demonstrated by Menon et al. [2], performance can be improved if carefully designed. The resulting improvement, however, is still not comparable to inter-process communication on a non virtual platform [3]. Currently many efforts are undertaken to improve network virtualisation performance as well as inter VM communication (IVMC). An example of IVMC is two processes on the same physical machine, but in different VMs, which want to exchange data in some form. The processes in the VM therefore have to communicate via the standard network interface, as if they did not share the same physical host. This clearly inflicts unnecessary performance penalties for both VMs: data has to be encapsulated, addressed, transmitted and checked via the network stack as well as the virtualisation layer.

Moreover, the cost and improvement of I/O virtualisation has received a lot of attention recently [4–7]. Compared to CPU or memory virtualisation, I/O device virtualisation is still considered costly and presents a performance bottleneck. Though the secure and efficient I/O hardware sharing is a popular topic in academic and commercial research, only minor attention is given to secure and efficient IVMC itself. I/O device communication is closely related to IVMC, and as discussed in the following, they are mutually dependant.

To address the issue of IVMC performance, many solutions propose the use of shared memory to tunnel the isolation boundaries. Shared memory seems like the obvious solution, but as we will demonstrate, it has certain drawbacks with regard to security, handling and transparency. Also, even if shared memory is being used, the performance of different implementations varies considerably. Based on our analysis, we discuss in this paper the open challenges for IVMC and propose a possible hardware supported solution.

The rest of the paper is structured as follows. Section 2 outlines the requirements for IVMC and provides the required background knowledge. Different proposed IVMC mechanisms are compared in Section 3, followed by an evaluation in Section 4. In Section 5 we discuss the possibility of utilising vacant CPU capabilities to facilitate IVMC mechanism and conclude in Section 6.

2 Preliminaries

In the following we focus on communication methods based on para-virtualisation. Apart from VMware's own VM communication interface, almost all efforts in academia are based on XEN and para-virtualisation. This is, we believe, a shortcoming in research of IVMC. XEN is not the only open source para-virtualisation project, but very popular because XEN's source code is freely available and its concepts are widely understood.

At the same time there are IVMC mechanisms for language based VMs [8], and OS system level virtualisation [9]. The concepts and level of isolation provided by such solutions differ from machine virtualisation and for this paper we focus on machine virtualisation.

2.1 Requirements

Enabling VMs to communicate requires a fine balance — on the one hand, enforcing isolation is a critical building block for VM security and must remain intact; on the other hand, enforcing isolation can inflict a serious performance overhead if VMs need to communicate across isolation boundaries. In contrast to I/O device sharing and its exchange of data, IVMC does not operate on a hardware communication endpoint. Consequently, a different security model and protection strategy is necessary:

2.1.1 Information leakage and illegitimate use

From a security perspective, evaluating and distinguishing between legitimate and illegitimate communication can be a further performance impediment. Performance considerations prohibit examining all VM communication. Leaving communication unregulated could therefore potentially result in information leakage.

2.1.2 Isolation and illegitimate access

Isolating shared resources is a fundamental building block of virtualisation security and must remain intact. Unfortunately, in order to allow efficient IVMC, isolation is sometimes weakened. For example, a shared communication buffer between two VMs could potentially be exploited by a malicious VM to access memory pages it is not entitled to.

2.1.3 Data integrity

For VMs the communication path is not transparent. Data might travel unprotected via an untrusted path where it could be altered or intercepted. Especially, if IVMC is used to signal important events, its integrity must be guaranteed.

2.1.4 Denial of Service

Communication can take place between VMs on different trust levels. Hence it is important to prevent an untrusted VM from exhausting all available resources and starving others.

Not only security and its protection strategies have to be considered when designing IVMC, but also unique virtualisation features such as user level transparency and live migration must be preserved:

2.1.5 Guest OS modifications

Changes to guest OSes running inside the VM should be avoided to maintain transparency and backward compatibility.

2.1.6 Migration

Communication management is necessary to determine the location of the communicating VMs. The communication management has to determine whether VMs are co-resident on the same physical platform, but also has to maintain the communication channel when a VM is migrated to another physical platform.

2.2 XEN Background

One possible software approach to I/O virtualisation is the para-virtualised solution implemented by XEN. It reduces the cost and complexity in contrast to a full device emulation significantly. Rather than fully emulating a device in software, para-virtualisation modifies the guest OS to implement a stub or front-end driver. This modified driver operates on a simpler device model and implements communication mechanisms to exchange information with the back-end driver. The back-end driver on the other side, is a complete and fully privileged device driver with direct access to the device's hardware. Therefore, the back-end driver either resides in the hypervisor itself, in the management domain or in the driver domain. The para-virtualised model is used by XEN for para-virtualised guests [10], KVM with virtio [6] and in VMware, if VMware tools are installed [11]. From a security perspective, device drivers have proven to be error prone [12]. Therefore, the device driver should preferably be isolated within an Isolated Driver Domain (IDD) [13]. As a consequence, the hypervisor and the driver domain have to be part of the Trusted Computing Base (TCB). In the current version of XEN the TCB also includes Domain 0, which contains all the drivers including a complete Linux distribution and the amount of code seriously impacts any trust assumptions.

The XEN front and back-end driver communicate with each other by using two ring buffers: one for packet transmission and one for packet reception. The ring buffers are implemented in XEN via *grant tables* and *event channels*. XEN *grant tables* are a mechanism to share (*shared pages*) or transfer (*page flipping*) memory pages between VMs. A VM notifies the hypervisor to grant a different VM access to its own memory page. The hypervisor keeps the grants in a *grant reference table* and passes the *grant reference* onto the other VM and signals this via an event channel. According to the *grant rights*, a page can then be written, read or exchanged. *Grant tables* are generally faster than *bounce buffers*, as the VM can directly access the memory page via Direct Memory Access (DMA) [10]. The hypervisor ensures that a VM can only access a page it holds the corresponding rights for. However, *grant tables* can only be used to share or transfer data between a privileged and an unprivileged domain. Hence, this mechanism alone is not sufficient for IVMC between unprivileged domains.

3 Comparison of Inter VM Communication

In the following section we provide an overview over comparable IVMC mechanisms, which were known to the authors at the time of writing.

3.1 XEN virtual network

The default method of VM communication is to route via the standard network interface. This offers the highest amount of isolation because VMs communicate as if they did not share the same physical platform. At the same time it provides the lowest performance and least amount of data integrity protection, as data could potentially be altered or intercepted during transport.

The default virtual network implementation in XEN creates a network bridge and attaches it to the physical network interface. For each VM, XEN then creates a virtual network interface and connects it to that bridge. This process is transparent to VMs, but also leaves Domain 0 in charge of the network interface. Additionally, as all communication is relayed via Domain 0, this implicitly demands the domain to be trustworthy.

Data exchange between the virtual network interface and the back-end is implemented via an I/O channel on the basis of the previously discussed grant tables [2]. In order to perform the grant table page sharing or transfer, a hypercall and trap into the hypervisor is triggered. However, the interaction with the hypervisor, page flipping and excessive domain switching negatively impacts performance.

3.2 XenLoop

XenLoop [14] is based on a Linux kernel module which is integrated into the network layer of each guest OS. The module intercepts and inspects every outgoing network packet via the Linux netfilter hook. Moreover, the module contains a mapping table of all VMs that are running on the same physical platform. If two machines on the same host need to exchange data, the modules in each machine set up a shared memory data channel – effectively bypassing Domain 0's data path and control. During the channel setup XenLoop determines the server and client roles depending on the domain ID. However, all VMs are regarded as equally trustworthy. To prevent data leakage, XenLoop foresees that both communicating VMs scrub out memory content of the transfer buffer. Using XenLoop along with XEN's standard net back and front mechanism allows fallback to standard TCP/IP communication in case of VM migration.

3.3 XenSockets

XenSockets [3] is an one way communication pipe between two VMs. It implements an IVMC mechanism based on the Unix domain socket principle. The implementation of XenSockets is based on statically shared memory buffers. It omits the traditional XEN page flipping mechanism and uses shared memory for message passing instead. This is realised by the sender creating a memory page and then re-mapping that page into the receiver's accessible address space. Each communication endpoint therefore implements two types of shared memory pages: a descriptor page, used for control information and a buffer page, used for data transfer.

In order to prevent a possible Denial of Service (DoS), pages must always be shared by the less trusted VM and mapped by the more trusted VM. This way the less trusted domain cannot exhaust the resources of the more trusted VM, e.g. by requesting a large number of XenSockets without tearing them down.

Moreover, the teardown of XenSocket has to guarantee communication integrity and therefore includes a shutdown procedure to synchronise the sender and receiver VM. The authors of XenSockets must also assume that one domain is more trustworthy than its communication partner. The client or sender VM has to be trusted and to behave correctly, as it could potentially block the communication with the server or receiver VM.

XenSocket's POSIX socket Application Programming Interface (API) results in a clear and simple interface, but requires applications to be recompiled against the new API and thus breaking compatibility.

3.4 XWAY

XWAY [15] defines a virtual device in the XEN device model, along with a device driver inside each VM. To establish a communication channel, the device drivers in each VM have to set up shared memory and an event channel. XWAY relies here on the communication interfaces provided by XEN: grant tables, xenstore and event channels, but omitting the costly page flipping mechanism for performance reasons. The two XWAY device drivers exchange references to the shared memory and event channel through a special helper daemon running inside each VM.

XWAY offers legacy compatibility, hence rather than introducing new libraries or APIs, XWAY hides behind the standard TCP socket API. This allows standard applications to benefit from IVMC without the need to recompile against new APIs. If an application requests a socket, instead of creating a TCP socket, an XWAY socket is created. As it is initially not apparent if the new socket is used for inter VM or network communication, the XWAY switch has to determine the communication endpoint. By doing so, the XWAY switch design also allows a flexible live migration.

However, the helper daemon synchronises and manages communications inside a potentially untrusted domain. Consequently, it has to be trusted and remain trusted throughout the VMs life-cycle.

3.5 Inter-OS Communication on Highly Parallel Multi-Core Architectures

Youseff et al. [16] also base their work on the XEN grant table, but allow an arbitrary sharing between VMs of different trust levels. Other than using a POSIX socket interface as proposed by XenSockets or XWAY, the authors decided to expose shared memory to the VM and its applications directly. This clearly inflicts less overhead but leaves the burden of type-safety and correctness to the developers. Potentially, this implementation renders the system vulnerable to DoS attacks. An untrusted application could exhaust the platforms resources by acquiring all available system memory. As a consequence, all components have to be equally trusted.

3.6 Virtual Machine Aware Communication Libraries for High Performance Computing

The VM aware communication libraries proposed by Huang et al. [17] are based on shared memory and implement a socket style API for XEN. The authors mainly target High Performance Computing (HPC) with their work and therefore implement a Message Passing Interface (MPI) library[1]. Message Passing Interfaces are common in HPC environments. By doing so, the library supports IVMC as well as the VMM-bypass I/O which has been part of authors previous work [4]. IVMC is set up as following: firstly, a process inside a VM invokes the IVMC aware libraries to allocate memory pages as a communication buffer with the remote endpoint. Secondly, the libraries call into the IVMC aware kernel driver to grant the target VM access to those pages. The shared memory itself is realised through the XEN grant table

[1] MVAPICH2 in particular

mechanism, hence the reference handles are passed onto the IVMC libraries. After creating shared memory between two libraries, a socket-like API can be used by the communicating processes. This also requires the application to be recompiled against the new API.

The authors embrace their previous work of VMM-bypass I/O with the IVMC libraries, and demonstrate that a communication channel can be kept intact even during VM migration.

3.7 A High-efficient Inter-Domain Data Transferring System for Virtual Machines

Inter-Domain Data Transfer System (IDTS) [18] provides bidirectional communication between VMs. IDTS implements a shared memory tunnel without a network stack. Therefore, it follows the XEN design of a split driver model and implements a front and back-end driver. The back-end driver along with the connection management logic resides in Domain 0, while the simple front-end driver is placed in the VM. IDTS also exploits xenstore for shared memory setup and passing grant table information between the communication endpoints. Moreover, a XEN event channel is established between the endpoint for notification. The front-end driver only offers a basic interface without supporting TCP/IP. Hence applications wishing to use the front-end driver have to be modified in order to support the interface.

The advantage of IDTS lies in its ability to establish bidirectional communication, which allows the role of sender and receiver to be associated more flexibly, as well as allowing an one-to-many communication. A disadvantage however, is that IDTS requires all tunnels and their management to be routed via Domain 0, which does not allow a direct VM-to-VM communication.

3.8 VMware VMCI

VMware uses a Virtual Machine Communication Interface (VMCI) [19] to enable virtual machines to communicate. VMCI allows guest-to-guest, or guest-to-host, on the same physical platform but no communication over the network. The VMCI library exports a socket style API similar to processes running inside a VM. The underlying communication relies on a virtual VMCI device which is located on the hypervisor level and a VMCI driver inside the VM. VMCI offers both a connection-less as well as a connection-oriented path. Note that VMCI explicitly allows guest-to-host communication but prohibits any network connection. Its main disadvantage is however, that applications have to be recompiled against the new communication library.

4 Evaluation

Based on the comprehensive study of existing IVMC mechanisms, we compiled a performance comparison as shown in Figure 1 and Table 1. Data is based on information provided by the authors of the previously discussed IVMCs. However, we ignore latency, as most authors did not provide sufficient data for comparison. Figure 1 compares the bandwidth depending on the message size. Unfortunately, comparable performance results could not be obtained for [16] and [19], hence they are not represented in Figure 1. As the figure indicates, the performance of the standard TCP/IP model is independent of the message size, but consistently below 1 Gbps. IDTS, XenSockets and the VM aware communication libraries perform exponentially better with larger message sizes. XenLoop and XWAY, which offer more transparency, perform linear with growing message sizes.

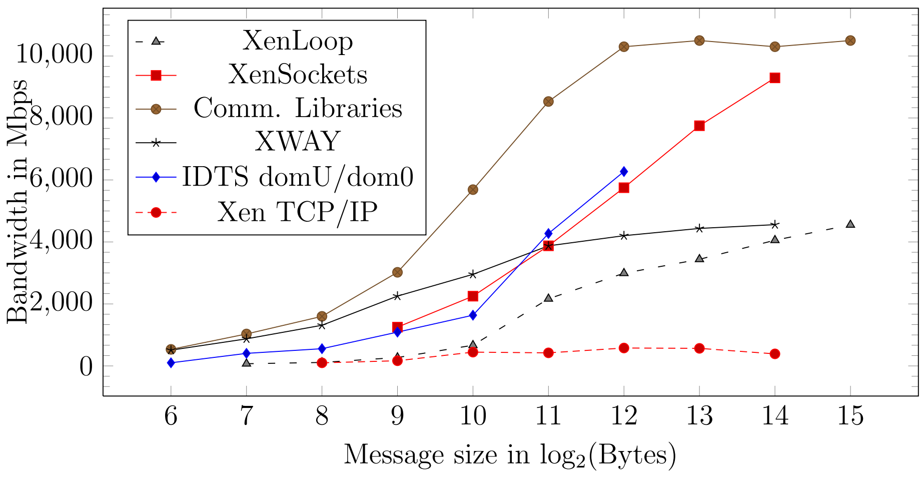

Figure 1: Benchmark comparison, based on published data.

Table 1 seeks to compare performance against the level of isolation and user level transparency. Moreover, the table indicates whether the IVMC is capable of supporting live migration.

The above communication methods have been designed with either transparency or performance in mind. However, most attention has focused on improving the performance of IVMC [3, 17]. Little consideration has been given to security, integrity or the trustworthiness of inter VM channels. Only XenSockets discusses the graceful teardown of a channel, while only XenLoop highlights the need for memory scrubbing. Both are vital in order to maintain the integrity of a communication channel and data. Additionally, some of the solutions rely on modifications to the guest OS by introducing new APIs and libraries.

Inter VM communication should not rely on inherently insecure and unprotected shared media such as the main memory. For instance, DMA capable devices on the same physical platform could be used to read or alter arbitrary memory regions, and thus manipulate a communication channel. Emerging hardware support for I/O sharing such as AMD DEV [20] or Intel's VT-d [21] technology promise to mitigate the problem on the hardware side.

Todays CPU design typically implements multiple cores and current development suggests that future processors will integrate even more cores on a single chip. Also, developments in machine virtualisation indicates that future platforms will have multiple VMs operating at the same time. This is not only apparent from a consolidation point of view to reduce energy consumption in large data centres, but also on commodity hardware such as laptops and desktop machines. Users are now becoming accustomed to having several different OSes on a single platform: in a corporate environment this can allow separation of corporate trusted compartments from private untrusted compartments [22].

Table 1: Performance Comparison.

IVC	Performance Mbps/msg size[1]	Isolation	Transparency	Migration support
XEN virt. network	980 Mbps/4k	Highest	Highest	Yes
XenLoop	3000 Mbps/4k	High	High	Yes
XenSockets	5800 Mbps/4k	High	Low	No
XWay	3900 Mbps/4k	Modest	High	Yes
Inter-OS comm.	2250 Mbps/1k	Low	Low	No
Comm. Libraries	10400 Mbps/4k	High	Modest	Yes
IDTS	5500 Mbps/4k	High	Modest	No
VMCI	N/A	High	Low	No
note1: values are rounded and differ depending on the message size.				

Combining all of the above: corporate, private, driver, management VMs, policy VM, etc., could easily result in a large number of VMs on a single platform, creating the need for a secure and efficient way of communicating. Thus IVMC is likely to be unavoidable and the need to securely interconnect multiple VMs will become increasingly important. Unfortunately none of the existing solutions meet all requirements and thus IVMC remains an open research question. In the following section, we will discusses a speculative solution to the problem space.

5 Hardware assisted IVMC

With the increasing use of virtualisation in the 1990's, hardware vendor's such as Intel and AMD adapted their CPU design to support this technology. However, current hardware architecture does not naturally support inter VM communication. Moreover as discussed in the following, there are capabilities in modern CPUs which are not utilised at all. For instance, out of the four x86 protection modes available, only two have been used up until recently. Utilising the unused protection modes is certainly possible, but inflicts a significant amount of development as well as breaking legacy applications. With the introduction of hardware assisted virtualisation technology, even more protection modes have been introduced and are not utilised. Consequently, we propose to make full use of the existing protection mechanisms available in the new, hardware assisted, virtualisation CPUs at this early stage of hypervisor development.

5.1 Ring Protection Background

The ring protection scheme is based on a 2-bit privilege level, enabling the CPU to determine four different separate levels of privilege - from ring 0, with the most privileges, to ring 3, with the least privileges. The Current Privilege Level (CPL) is stored in the lower two bits of the segment selector. Depending on which ring level code is being executed, the program has access to different CPU functionalities and features. Traditionally, an OS kernel, including device drivers, runs with the highest privileges and no restrictions in ring 0. Applications are placed inside ring 3, the least privileged level. Rings 1 and 2 are generally unused. Figure 2 illustrates the privileges in a non-virtualised context.

In ring compression [23], a hypervisor executes the guest OS at a lower privilege level, for instance ring 1. However, ring de-privileging is not straightforward, and a hypervisor can be implemented in different ways. Technical details and the challenges caused by ring de-privileging are beyond the scope of this paper.

To overcome the limitation of ring de-privileging and allow any unmodified guest OS to run without restrictions at its intended privilege level, CPU manufactures introduced hardware support for virtualisation. Hardware assisted virtualisation allows the creation of a Hardware Virtual Ma-

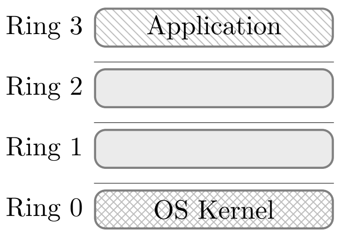

Figure 2: Traditional ring protection scheme.

chine (HVM) under the control of a hypervisor and supported by hardware extensions to the CPU. HVM introduces a special mode of operation which allows the guest OS kernel to execute at its intended privilege level. HVM knows two modes of operation, privileged and non-privileged mode. The non-privileged mode is a modified view of the CPU to accommodate virtualisation. In this mode VMs can now run at their intended privilege level, all privileged and some unprivileged CPU instruction will trap into the privileged mode. The privileged mode remains the same as if running a native OS kernel, however a hypervisor is placed into the privileged mode to handle the traps from VMs. As outlined in Figure 3, the hypervisor in root mode has also four different privilege levels at its command.

Because the authors are more familiar with the Intel architecture as well as limited access to AMD documentation, we want to focus on Intel terminology in the following. However the concepts and ideas are potentially equally applicable to any x86 architecture and AMD's virtualisation technology in particular.

5.2 HVM Protection Ring Background

In Intel terminology the privileged mode is labelled VMX root mode whereas the un-privileged mode is called VMX non-root mode [24]. The Virtual-Machine eXtension (VMX) is new hardware enhancement introduced to support virtualisation [25]. It is important to emphasise, that the VMX root mode implements the protection scheme, illustrated in Figure 2, whereas the VMX non-root mode can be regarded as a new mode of operation with reduced privileges. Both modes support all four rings, where the guest OS is presented its expected ring model and the hypervisor is able to use multiple privilege levels on its own [24]. In the following we use the terminology *H0* to *H3* for the rings in VMX root mode that are available to the hypervisor. Figure 3 outlines the use of VMX root and VMX non-root mode.

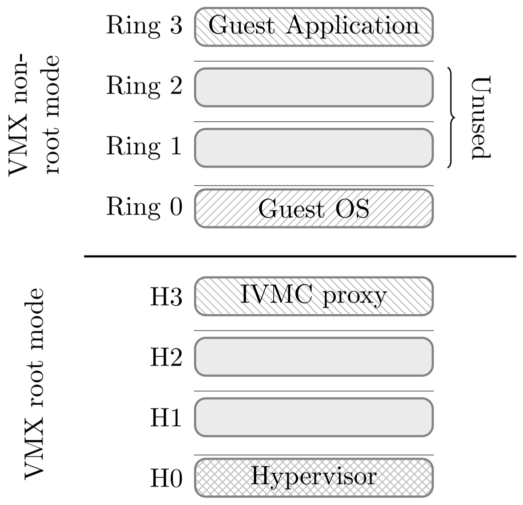

Figure 3: VMX root mode/non-root mode.

5.3 Inter VM Communication Implementation

Bearing in mind that the VMX root mode operates in the same way as an unmodified CPU, we can use the ring model to construct a hypervisor system which consists of multiple components on different levels of privileges. Within the root mode, those components are isolated from each other by hardware mechanisms and not visible to any non-root mode code. Most virtualisation management contexts, for example handling VM exits and other vital emulation functions, have to remain in VMX root mode ring 0 (*H0*). However, other functionality could be outsourced to different privilege levels. To allow the hypervisor to communicate with outsourced modules on different rings, the hypervisor itself has to implement communication mechanisms, for instance via message parsing or trusted libraries. Preferably, this would be something as simple as a traditional syscall interface for a H0 to H3 communication.

We have seen earlier that co-resident VMs are forced to communicate with each other using standard network interfaces and protocols as if they are resident on different physical machines. Those protocols were designed to operate on unreliable media, and are not optimal for communication on the same platform. Therefore they inflict additional overhead, e.g. in form of integrity checks on multiple layers. Additionally, the network hardware used in VMs is often complicated when emulated in software.

To implement IVMC, we envision a proxy agent, which is located on the previously vacant *H3* ring. This proxy can support multiple end-points and orchestrate the communication between VMs on different trust levels. Instead of using a management domain to do the emulation and setup of channels, this can be handled by the proxy agent instead. Firstly, this would de-privilege the management domain, stripping it of those functions and secondly it would improve performance as the number of costly VM entries and VM exits to and from the management domain would be reduced.

Furthermore, the proxy agent is not part of the hypervisor itself, it is in that sense an application. The proxy on *H3* is isolated by hardware protection mechanisms from the hypervisor on *H0*. Therefore, from the hypervisors's point of view, the proxy could be any untrusted code.

The major performance benefits however, are based on the fact that, once the CPU has entered the VMX root mode, a majority of the functions can be handled in this mode without returning back to a VM and shifting data back and forth between different VMs. This essentially trades computational expensive world transitions for less costly context switches. Nevertheless, new protocols are still necessary to circumvent the remaining performance bottlenecks, for instance redundant integrity checks which were required on unreliable media.

6 Conclusion

In this paper we evaluate and compare existing IVMC mechanisms, based on their performance, transparency, isolation properties and live migration capabilities. However, we find that many of the investigated solutions succeed in only one of those areas. Moreover, we propose in this paper to fully utilise the available hardware protection mechanisms to implement IVMC for virtualised systems.

References

[1] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing Performance Overheads in the Xen Virtual Machine Environment," in *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. New York, NY, USA: ACM, June 2005, pp. 13–23.

[2] A. Menon, A. L. Cox, and W. Zwaenepoel, "Optimizing network virtualization in Xen," in *ATEC '06: Proceedings of the annual conference*

on USENIX '06 Annual Technical Conference. Berkeley, CA, USA: USENIX Association, June 2006.

[3] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin, "XenSocket: A High-Throughput Interdomain Transport for Virtual Machines," in *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware.* New York, NY, USA: Springer, November 2007, pp. 184–203.

[4] 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, June 2006, pp. 3–3.

[5] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for I/O virtualization," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, June 2008, pp. 29–42.

[6] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, July 2008.

[7] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick, and E. Wahlig, "Utilizing IOMMUs for Virtualization in Linux and Xen," in *OLS '06: The 2006 Ottawa Linux Symposium.* Linux Symposium Inc, July 2006, pp. 71–86.

[8] M. Wegiel and C. Krintz, "XMem: type-safe, transparent, shared memory for cross-runtime communication and coordination," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation.* New York, NY, USA: ACM, June 2008, pp. 327–338.

[9] Z. Shan, Y. Yu, and T. Chiueh, "Confining windows inter-process communications for OS-level virtual machine," in *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems.* New York, NY, USA: ACM, March 2009, pp. 30–35.

[10] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, "Xen 3.0 and the Art of Virtualization," in *Proceedings of Linux Symposium 2005.* ACM, July 2005.

[11] J. Sugerman, G. Venkitachalam, and B.-H. Lim, “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor,” in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2001, pp. 1–14.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An empirical study of operating systems errors,” in *SOSP ’01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, October 2001, pp. 73–88.

[13] K. Fraser, H. Steven, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, “Safe Hardware Access with the Xen Virtual Machine Monitor,” in *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.6391

[14] J. Wang, K.-L. Wright, and K. Gopalan, “XenLoop: a transparent high performance inter-vm network loopback,” in *HPDC ’08: Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA: ACM, June 2008, pp. 109–118.

[15] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim, “Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen,” in *VEE ’08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, March 2008, pp. 11–20.

[16] L. Youseff, D. Zagorodnov, and R. Wolski, “Inter-OS Communication on Highly Parallel Multi-Core Architectures,” http://www.cs.ucsb.edu/research/tech_reports/reports/2008-17.pdf, University of California, Santa Barbara, Tech. Rep., published 2008.

[17] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda, “Virtual machine aware communication libraries for high performance computing,” in *SC ’07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, November 2007, pp. 1–12.

[18] D. Li, H. Jin, Y. Shao, and X. Liao, “A High-efficient Inter-Domain Data Transferring System for Virtual Machines,” in *ICUIMC ’09: Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*. New York, NY, USA: ACM, January 2009, pp. 385–390.

[19] VMware, “VMCI Sockets Programming,” http://www.vmware.com/pdf/ws65_s2_vmci_sockets.pdf, published 2008.

[20] AMD, “AMD I/O Virtualization Technology (IOMMU) Specification 1.2,” http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf, AMD Corporation, Tech. Rep., February 2007.

[21] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, “Intel Virtualization Technology for Directed I/O,” *Intel Technology Journal*, vol. 10, no. 03, pp. 179–192, August 2006.

[22] C. Gebhardt and C. I. Dalton, “LaLa: A Late Launch Application,” in *The Fourth Annual Workshop on Scalable Trusted Computing.* Chicago, Illinois, USA: ACM, November 2009.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles.* New York, NY, USA: ACM, October 2003, pp. 164–177.

[24] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith, “Intel Virtualization Technology,” *Computer*, vol. 38, pp. 48–56, May 2005.

[25] Intel, “Intel Virtualization Technology Specification for the IA-32 Intel Architecture,” Intel Corporation, Tech. Rep. C97063-002, April 2005.