V4VSockets: low-overhead intra-node communication in Xen

Anastassios Nanos, Stefanos Gerangelos, Ioanna Alifieraki * and Nectarios Koziris

Computing Systems Laboratory,
National Technical University of Athens
{ananos,sgerag,nkoziris}@cslab.ece.ntua.gr
ioanna-maria.alifieraki@postgrad.manchester.ac.uk

Abstract

Nowadays, the cloud computing paradigm has given rise to new and unconventional application deployments on elastic compute infrastructures. For instance, IaaS providers are willing to support a diverse set of computing workloads, ranging from service-oriented deployments to HPC applications. As a result, the underlying systems software has to be generic enough to support lightweight, efficient execution for a wide range of applications. In this work, we examine communication methods within a single VM container that ease data exchange between co-located VMs without sacrificing upper-layer API compatibility.

We present V4VSockets, a generic, socket-compliant framework for intra-node communication in the Xen hypervisor. The transport layer is based on V4V, a simple hypercall-based mechanism to transfer data. Our framework resides within the hypervisor, providing a dispatch logic to communication, contrary to the common Xen concept of decoupling the control and data plane using a privileged VM.

V4VSockets improves intra-node data exchange in terms of both latency and throughput by a factor of 4.5. To demonstrate the applicability of V4VSockets, we spawn a VM with a GPU device assigned to it and deploy a remote GPU acceleration benchmark on co-located VMs. Preliminary results show that V4VSockets boosts the transfer throughput by a factor of 7 (at best) while adding an overhead of 15% compared to native execution.

1. Introduction

Modern cloud data centers provide flexibility, dedicated execution, and isolation to a vast number of service-oriented applications (i.e. high-availability web services, core network services, like mail servers, DNS servers etc.). These infrastructures, built on clusters of multicores, offer huge processing power; this feature makes them ideal for mass deployment of compute-intensive applications.

In the HPC context, applications often scale to a large number of nodes, leading to the need for a high-performance interconnect to provide low-latency and high-bandwidth communication. In the cloud context, distributed applications are executed in a set of Virtual Machines (VMs) that are placed in physical nodes across the cloud data center. These VMs suffer communication overheads [5, 10, 13] and are unaware of their physical placement – this presents a problem, because application instances running on the same physical node but on different VMs are not able to exploit locality. Additionally, the advent of Software Defined Networks and Network Function Virtualization (commonly referred to as SDN/NFV) has given rise to examine approaches where lightweight VMs exchange data with co-located peers to perform network processing operations. We build on this trend and explore intra-node communication mechanisms to achieve efficient, low-overhead data exchange between co-located VMs.

We introduce V4VSockets, a socket-compliant, high-performance intra-node communication framework for co-located VMs. V4VSockets simplifies the data path between co-located VMs: this is achieved by creating a peer-to-peer communication channel between a VM and the hypervisor. V4VSockets eliminates the overhead of page exchange/mapping and enhances throughput by moving the actual copy operation to the receiver VM. Our framework improves security by operating in a shared-nothing policy; pages are not shared between VMs – the hypervisor is the only one responsible for transferring data to the peer VM.

The contribution of this paper can be summarized as follows:

- We introduce V4VSockets, an efficient, socket-compliant, high–performance intra-node communication framework (Section 3).
- We evaluate V4VSockets using generic micro-benchmarks and compare it to conventional communication paths (Section 4). We find that V4VSockets outperforms the generic method of intra-node communication and scales efficiently with a large number of VMs both in terms of throughput as well as latency.

* Currently with the University of Manchester

CloudDP'15, April 21-24, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/2744210.2744215

- We discuss the applicability of our framework, presenting a real-life use case: we deploy an HPC application stencil over rCUDA [8], a remote GPU execution stack over V4VSockets.

The rest of this paper is organized as follows: first, we lay groundwork in Section 2 by presenting the basic concepts of intra-node communication frameworks, briefly discussing the essentials of Xen and related research. Section 3 describes the design and implementation of V4VSockets, while Section 4 presents a detailed evaluation of its performance, with regards to latency, throughput, and scaling. We discuss how V4VSockets enables VM GPU sharing by deploying rCUDA over our framework (Section 4.2) and conclude, presenting possible future endeavors (Section 5).

2. Background and Motivation

In this section, we briefly describe design choices for intra-node communication in a virtualized environment, providing background information on the key communication components of Xen. We mostly focus on the control and data handling in the network stack. Additionally we discuss related research, presenting various approaches and their trade-offs.

Overview of the Xen Architecture

Xen [1] is a popular hypervisor, based on the ParaVirtualization (PV) concept. Data access is handled by privileged guests called *driver domains* that enable VMs to interact with the hardware based on the *split driver model*. Driver domains host a *backend* driver, while guest VM kernels host *frontend* drivers, exposing a per-device class API to guest user– or kernel–space.

In Xen, memory is virtualized in order to provide contiguous regions to OSs running on guest domains. This is achieved by adding a per-domain memory abstraction called *pseudo-physical* memory. So, in Xen, *machine memory* refers to the physical memory of the entire system whereas pseudo-physical memory refers to the physical memory that the OS in any guest domain is aware of.

To efficiently share pages across guest domains, Xen exports a *grant* mechanism. Xen's grants are stored in *grant tables* and provide a generic mechanism for memory sharing between domains. Network device drivers are based on this mechanism in order to exchange control information and data. Two guests setup an *event channel* between them and exchange events that trigger the execution of the corresponding handlers. *I/O rings* are ring buffers, a standard lockless data structure for producer-consumer communication. Through I/O rings, Xen provides a simple message-passing abstraction built on top of the grant and event channel mechanisms.

Xen PV Network I/O:

The common method of VM communication in Xen is through the PV network architecture. Guest VMs host the *netfront* driver, which exports a generic Ethernet API to kernel-space. The driver domains host a hardware specific driver and the *netback* driver, which communicates with the frontend using the event channel mechanism and injects frames to a software bridge.

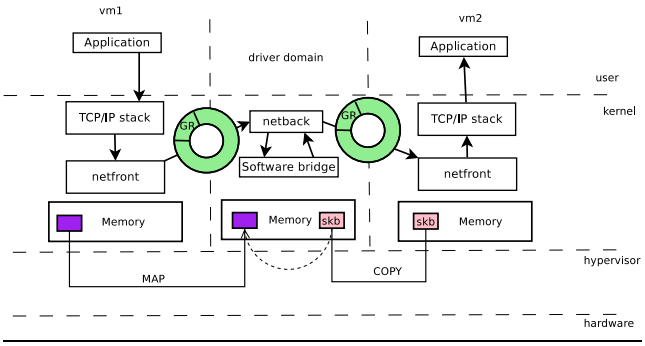

Figure 1. Generic intra-node communication in Xen.

Data flow in and out of the VM using the grant mechanism, while notifications are implemented using event channels, the virtual IRQ mechanism that Xen provides. Less-critical operations are carried out by Xenstore (interface numbering, feature exchange etc.).

To communicate with each other, VMs that co-exist in the same VM container have to cross through the software bridge in a driver domain. Figure 1 presents the data path of two VMs exchanging data. Data movement is realized using shared pages that are set up using the grant mechanism. Each transmission request contains a grant reference and an offset within the granted page. This allows transmit and receive buffers to be reused, preventing the TLB from needing frequent updates. To receive packets, the guest domain inserts a receive request into the ring, indicating where to store a packet, and the driver domain places the contents there.

Related Work

A major source of intra-node communication overhead is the complex data path between co-existing VMs. Network traffic between peer VMs is redirected via the driver domain, resulting in a significant performance penalty. Packet transmission and reception involves traversal of the TCP/IP network stack and the invocation of multiple Xen hypercalls. Several optimizations have been proposed regarding this limitation: *shared memory* techniques, provided by the Xen hypervisor are exploited to facilitate data exchange between VMs. Using a pool of shared pages for direct packet exchange seems a lot more efficient than traversing the network communication path via the driver domain. XenSockets [14] and IVC [2] provide a basic, one way communication channel using socket semantics, introducing a new address family type. XenLoop [12], on the contrary, intercepts calls to local VMs through the Linux netfilter mechanism and establishes a full-duplex data channel between peers to efficiently exchange data. In XWay [3] the authors define a new virtual device that establishes direct communication between VMs, bypassing the driver domain completely. MMNet [9] eliminates copies by mapping the entire kernel address space of a VM into the address space of its communicating peer.

Apart from intrusive methods like the above, there has been great effort in optimizing the existing network stack in Xen:

for instance, Menon et al. [4] improve network performance by introducing copies instead of page remapping and using advanced memory features such as superpages and global page mappings.
Additionally, the CPU scheduler in the hypervisor has a major influence on the latency of communication between co-located VMs. If the CPU scheduler is unaware of communication requirements of co-located VMs, then it might make non-optimal scheduling decisions that increase the inter-VM communication latency.
We build on this strand of the literature, but instead of optimizing the data path through the driver domain, we bypass it, using the hypervisor as the network medium. In the following sections we describe the design of our framework and show that when VMs exchange data, crossing the hypervisor seems much more efficient than installing a direct shared memory path or optimizing driver domain intervention.

3. Design and Implementation

In this section we describe V4VSockets, a highly efficient intra-node communication framework in the Xen platform. Our approach builds on V4V, part of the XenClient project [11]. V4V is an abstract mechanism provided by the Xen hypervisor that supports basic communication primitives.
We describe V4VSockets, a generic socket layer for the V4V transport mechanism that enables applications running on VMs' userspace to communicate with other VMs coexisting on the same VM container. V4VSockets consists of a device driver to expose the socket API to user-space and the V4V transport mechanism provided as an extension to the Xen hypervisor. An analogy to the TCP/IP protocol suite is shown in Figure 2.
In what follows, we discuss the V4VSockets architecture, briefly presenting the design overview, as well as implementation details.

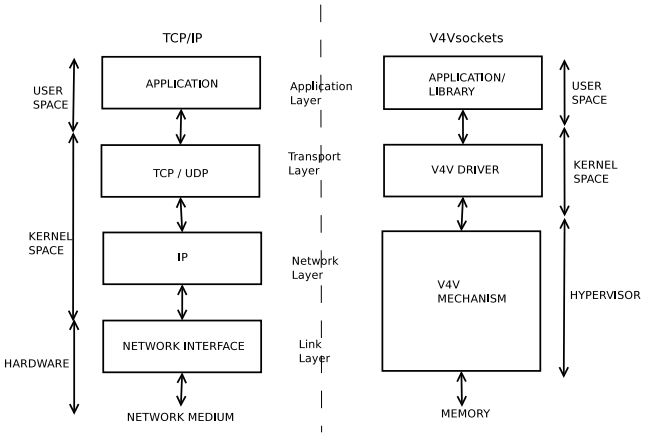

Figure 2. V4VSockets and TCP/IP

3.1 Design Overview

V4VSockets is built as a full-stack protocol framework that supports peer-to-peer communication between co-located VMs. Contrary to the common approach of decoupling communication to a privileged VM, leaving only security issues to be handled by the hypervisor, we choose to bypass the intermediate layer and use the hypervisor as the control *and* the data plane. Data flow between two VMs without the intervention of a third VM, providing better isolation and scalability. To provide an architectural overview, we briefly describe how the operations are realized in each layer.
Application layer: One of the most important aspects of our design is the API compatibility with generic concepts, namely, the socket interface. Specifically, we aspire to provide a low-overhead socket communication framework to applications running in co-located VMs without the need to refactor, re-implement or recompile them.
Thus, in V4VSockets, the application layer refers to the common socket-layer calls (`socket()`, `bind()`, `connect()` etc.) which forward the relevant actions and arguments to the transport layer.
The *Transport layer* in our approach resides in the VM kernel. Essentially it implements the socket calls and the communication primitives of the communication protocol over the network medium (Xen in our case). Specifically, the transport layer handles the virtual connection semantics between peer VMs that need to communicate, is in charge of fragmenting and sending upper-layer packets by issuing hypercalls to the hypervisor (network layer), and provides a notification mechanism to the VM's user-space for receiving packets, as well as error control.
The *Network/Link layer* resides in the hypervisor, providing encapsulation of upper-layer messages to packets that will be transmitted to their destination, according to V4V semantics, and packet delivery. This layer is in charge of transmitting the packet to its destination, which in our case consists of a memory copy. Thus, in the case of a packet send, the hypervisor places data into the relevant memory space of the receiver and notifies the Transport layer about an incoming packet.
In the following section we describe the V4VSockets framework in detail and present essential parts of the implementation, focusing mainly on the data exchange mechanism.

3.2 Implementation details

The core part of our framework is the transport layer, implemented as a VM kernel module, where all calls from userspace are translated into `V4V` hypercalls and issued to the hypervisor (the network/link layer). `V4V` provides basic support for communication through the following hypercalls:

- `register/unregister`: This call is used when a new socket is created and provides the necessary memory space to transfer data.
- `send`: This call is used when a send call is issued, providing the relevant structures to the hypervisor that, in turn, realizes the transfer.
- `notify`: When data is placed correctly, or when there's a notification that needs attention, the VM kernel issues

this call and the hypervisor handles all necessary steps to complete the operation (e.g. receive calls).

The intriguing part of V4VSockets is mainly focused on the data path; we base our framework on the `v4v_ring` structure (Figure 3), containing a static, pre-allocated ring buffer that essentially simulates the network medium. This buffer follows the generic producer-consumer concept, with two pointers `rx` and `tx` that are altered by the VM and hypervisor respectively.

This buffer is allocated in the VM kernel and registered to the hypervisor with the `bind()` system call. Essentially, this translates into a `register` hypercall and, thus, the machine frames that comprise the ring buffer are stored and mapped in Xen, forming a shared memory region between the VM kernel and the hypervisor.

The `accept()` system call initializes a receive operation: the application listens to a specific port for incoming packets.Once data have been written to the ring, the hypervisor updates the `tx` pointer. Following a `recvmsg()` system call, the VM kernel copies data from the ring space to a local staging buffer. Additionally, it updates the `rx` pointer (to free up space in the ring) and copies the received packet to user-space.

The `sendmsg()` system call initiates a send operation: the VM kernel creates an iovector from the userspace arguments, packs the data into a `V4V` message and issues the `send` hypercall. The hypervisor copies the data into the ring of the receiver VM, updating the `tx` pointer.

An example of a data exchange between two peer VMs is shown in Figure 3.

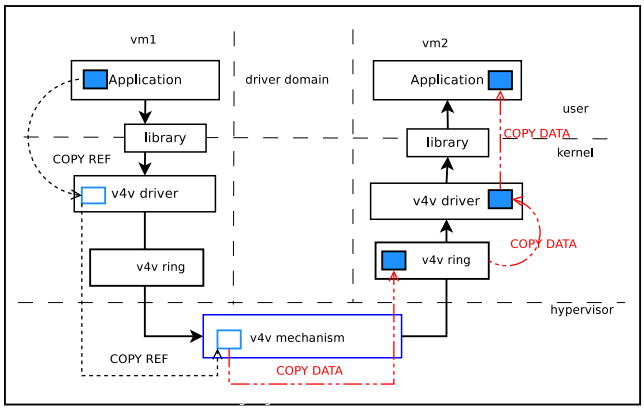

Figure 3. V4VSockets overview.

In V4VSockets, userspace applications issue generic socket calls using a custom address family constant instead of `AF_INET`. To keep compatibility for applications that have this constant hard-coded, we wrap the initial `socket()` system call around a library that re-issues the call with our custom address family. The rest of the calls (e.g. `bind()`, `listen()`, `accept()` etc.) use the socket descriptor provided by the initial call; as a result, the API remains intact.

4. Performance evaluation

In this section we describe the experiments we performed to analyze the behavior of V4VSockets and identify specific characteristics of our approach compared to the common setups used for intra-node communication in virtualized environments.

We setup a host machine with 2x Intel Xeon X5650 (Chipset 5520) and 48GB RAM (@1333MHz) and perform two basic experiments using microbenchmarks, in order to illustrate the merits and shortcomings of our approach without the noise of application-specific communication patterns. We also perform a third real-life experiment using a CUDA application from the GPGPU domain.

We setup the host as a VM container and spawn up to 16 single core VMs (VM_1 VM_2, ...,VM_{16}). We use NetPIPE [6] as a microbenchmark, in order to compare V4VSockets to the default TCP/IP over netfront/netback case. We deploy the microbenchmark between VMs (16 separate instances, VM_1 to VM_2, VM_3 to VM_4 and so on).

4.1 Microbenchmark evaluation

Figure 4 and Figure 5 plot the respective measurements when two VMs (VM_1 to VM_2) exchange messages. Figure 4 shows that the latency achieved by V4VSockets for a 2 Byte message is improved by 81% compared to the generic case. Specifically, the latency of the split driver model is 86 us, while V4VSockets completes the same task at 16 us. This is mainly due to the processing overhead of the TCP/IP stack, as well as the inefficient data path through the driver domain (Section 2), which is bypassed in our optimized transport mechanism.

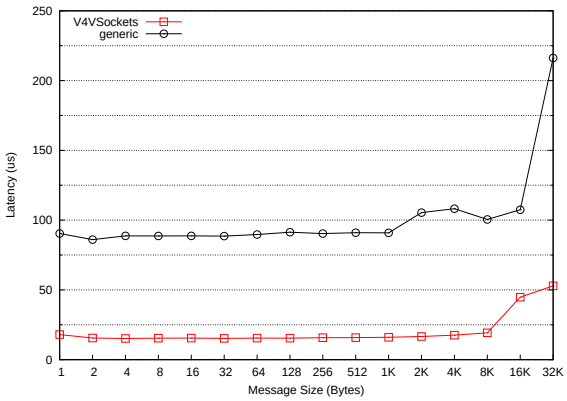

Figure 4. V4VSockets latency

In terms of throughput (Figure 5), V4VSockets outperforms the default case as well. V4VSockets peaks a maximum throughput of 2299 MB/s, 4.5x better than the split driver, which performs poorly at 501 MB/s for 1 MB messages.

To examine how V4VSockets scale with a various number of VMs exchanging messages we measure the system's throughput for 2, 4, 8 and 16 VMs communicating in pairs (Figure 6). The aggregate throughput increases proportionally to the number of communicating VMs. For instance, two VMs are able to exchange 512KB messages at $\approx$ 2 GB/s, while 16 VMs achieve 8x aggregate throughput for the same message size ($\approx$ 16 GB/s).

Based on our approach (Section 3), V4VSockets performs three data copies when transferring messages across: VM_1–

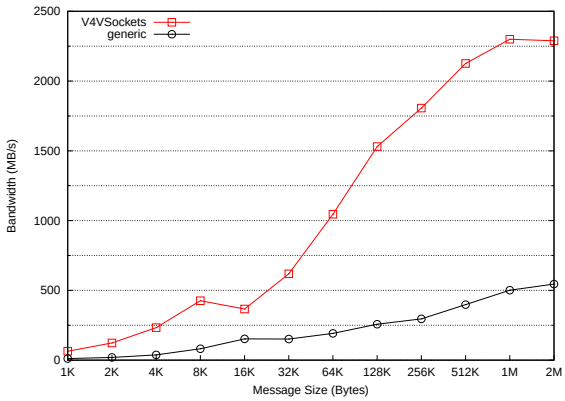

Figure 5. V4VSockets throughput

to–Xen, Xen–to–VM_2-kernel, VM_2-kernel–to–VM_2-userspace. This is a design choice: VM_1 notifies through a system call and a hypercall that there is a message for VM_2. Xen copies data from VM_1 userspace into VM_2 and notifies the kernel; when the kernel wakes up, data are already in the processor's cache, and thus, data flow directly to VM_2 userspace. As a result, we are able to reach more than half of the system's memory bandwidth[1], bringing memory-copy-like bandwidth measurements to VM–to–VM message exchange.

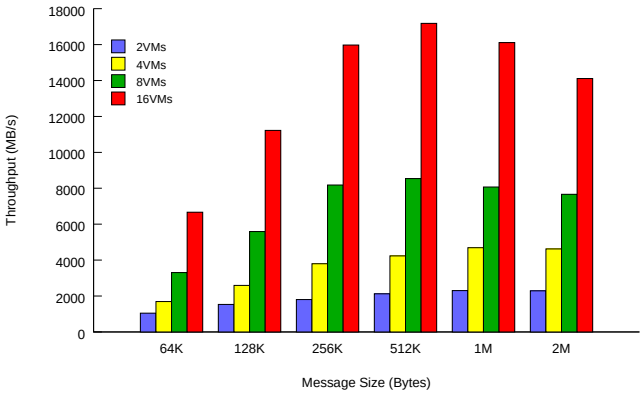

Figure 6. V4VSockets aggregate throughput

To validate that the system sustains acceptable performance when a large number of VMs put pressure on the memory bus, we examine the effect our data exchange mechanism has on latency. When 16 VMs exchange small messages in pairs, the round trip latency remains as low as 16 us, verifying the scalability of our approach.

4.2 GPU-enabled VMs

In this section, we demonstrate the merit of V4VSockets on a real-life benchmark from the GPU applications domain. We apply our efficient transport mechanism to rCUDA [8], a framework for enabling remote and transparent GPU acceleration. Building on existing frameworks and V4VSockets, we enable VMs to benefit from a GPU-equipped VM residing in the same VM container, without complicated setups, or disruptive and expensive techniques such as IOV.

[1] We performed a stream microbenchmark and measured 27 GB/s as the maximum memory bandwidth.

We use two VMs, VM_1 acting as the rCUDA server, and VM_2 as the client. In order to provide GPU access to the rCUDA server domain, we assign the GPU device to this VM using PCIe passthrough. Finally, we consider two cases: the generic transport mechanism using TCP/IP over the split driver model (*rCUDA generic*) and V4VSockets (*rCUDA over V4VSockets*). As a baseline, we perform the exact same experiment without the intervention of rCUDA, directly on VM_1 (*passthrough*)[2].

We use a common HPC application stencil, the single-precision matrix-matrix product, provided in CUDA by the NVIDIA samples [7].

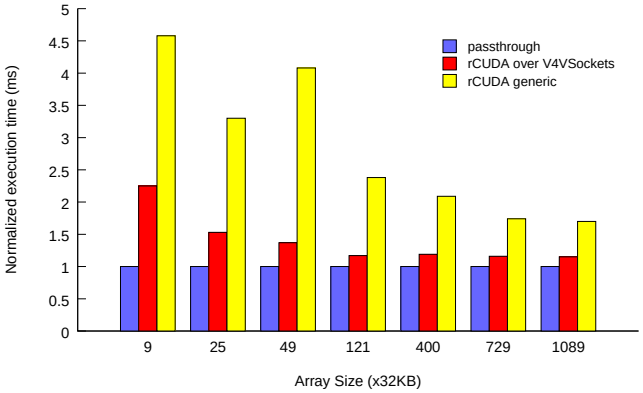

Figure 7. Matrix product total time of execution

This experiment includes the following procedure: two copies of the input matrices from node's main memory to GPU device memory, the product execution on the GPU and finally one copy of the output matrix back to main memory. The normalized total time of execution of the matrix-matrix product benchmark is depicted in Figure 7. The X axis presents the array size multiplied by 32 KB.

We observe that V4VSockets performs really close to the baseline case. For an input matrix size of 1089 x 32 KB (2112 x 4224 `float` type elements) V4VSockets adds a 15% overhead compared to running locally, whereas the generic case adds a 70% overhead. This is essentially our goal: through V4VSockets and rCUDA, VMs can seamlessly share a GPU device in a single VM container with a minimum overhead compared to the generic case. Given that full stack HPC applications use large matrix sizes, our framework can provide the necessary bandwidth to offload GPU execution with the minimum overhead due to remote execution.

To elaborate more on the impact of V4VSockets to the improvement in the execution time, we plot the throughput achieved when copying one of the input matrices from the machine's main memory to the GPU device memory (essentially this is a `cudamemcpy()` call) in Figure 8. We observe that the peak throughput is 3.79 GB/s in our baseline experiment, while the respective throughput in the remote V4VSockets case is 2.46 GB/s. However, for a matrix size

[2] We validate the measurements in a non-virtualized environment with an identical GPU device. We did not observe significant difference in terms of performance compared to the passthrough VM execution.

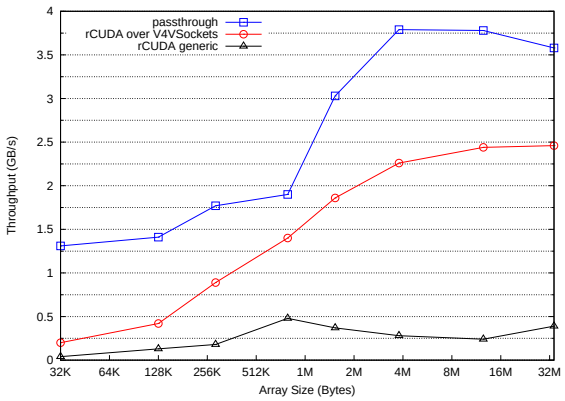

Figure 8. Matrix product transfer throughput

of 34 MB (2112 x 4224 `float` type elements) V4VSockets outperforms the generic case by a factor of 7 (0.35 GB/s).

5. Conclusion

In this work, we present the design and implementation of V4VSockets, a framework for efficient, low-overhead intra-node communication in co-located VMs. V4VSockets bypasses the driver domain completely and uses the hypervisor as the transport medium kernel and a backend on the hypervisor. V4VSockets is open-source software, available online at `https://github.com/HPSI/V4VSockets`.

Experimental evaluation of our prototype has revealed the following: (a) V4VSockets is able to saturate the memory bus, outperforming the generic case of intra-node communication through netfront / netback drivers both in terms of throughput and latency; (b) V4VSockets scales efficiently with a large numbers of VMs; (c) through V4VSockets, VMs are able to share a single GPU device using rCUDA, taking advantage of the efficient, low-overhead, data exchange channel our system provides.

We believe that VM containers with efficient intra-node communication features can host a wide variety of applications, especially in the device sharing context and the concept of SDN/NFV. Achieving data exchange rates equivalent to memory copies can disrupt the SDN world, leading to a large number of deployment options. Additionally we plan to extensively evaluate our approach in accelerator sharing scenarios in order to provide seamless FPGA/GPU sharing techniques for lightweight VMs.

Acknowledgments

The authors would like to thank the members of CSLab for the stimulating conversations that lead to this work, especially dr George Goumas, dr Konstantinos Nikas and Nikela Papadopoulou. Additionally, many thanks to the anonymous reviewers for their useful comments and suggestions.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. A. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proc. of the 19th ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[2] 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*, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. .

[3] K. Kim, C.-Y. 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 *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 11–20, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4. .

[4] 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*, pages 2–2, Berkeley, CA, USA, 2006. USENIX.

[5] A. Nanos, G. Goumas, and N. Koziris. Exploring I/O virtualization data paths for MPI applications in a cluster of VMs: a networking perspective. In *5th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '10)*, Ischia - Naples, Italy, 8 2010.

[6] NetPIPE - Network Protocol Independent Performance Evaluator. `http://linux.die.net/man/1/netpipe`.

[7] NVIDIA CUDA samples. `https://developer.nvidia.com/cuda-downloads`.

[8] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato. A complete and efficient cuda-sharing solution for HPC clusters. *Parallel Computing*, 40(10):574–588, 2014.

[9] P. Radhakrishnan and K. Srinivasan. Mmnet: An efficient inter-vm communication mechanism. In *XenSummit*, 2008.

[10] 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*, pages 29–42, Berkeley, CA, USA, 2008. USENIX.

[11] V4V implementation. `http://lists.xen.org/archives/html/xen-devel/2013-05/msg02711.html`.

[12] 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*, pages 109–118, NY, USA, 2008. ACM. ISBN 978-1-59593-997-5. .

[13] L. Youseff, R. Wolski, B. Gorda, and R. Krintz. Paravirtualization for HPC Systems. In *In Proc. Workshop on Xen in High-Performance Cluster and Grid Computing*, pages 474–486. Springer, 2006.

[14] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin. Xensocket: A high-throughput interdomain transport for virtual machines. In R. Cerqueira and R. Campbell, editors, *Middleware 2007*, Lecture Notes in Computer Science. 2007.