YASMIN: Efficient Intra-Node Communication Using Generic Sockets

Michalis Rozis, Stefanos Gerangelos, and Nectarios Koziris

National Technical University of Athens
Computing Systems Laboratory
Athens, Greece
{mrozis,sgerag,nkoziris}@cslab.ece.ntua.gr

Abstract. Nowadays, virtual machines are becoming widely used and their range of applications include a large number of scientific fields. From HPC to IaaS, communication between co-located VMs is a critical factor of efficiency. In our paper, we examine communication methods between VMs located in the same physical node, optimizing communication cost without sacrificing upper-layer API compatibility. We present *YASMIN (Yet Another Shared Memory Implementation for Intra-Node)*, a generic socket-compliant framework for intra-node communication in the Xen hypervisor. We build on the concept of Vchan, a Xen library for intra-node communication between different VMs and we use Xen granting and signaling mechanisms to provide an efficient communication framework. The key of our design is the transport layer which runs underneath the `AF_VSOCK` protocol family, implemented as a dynamically inserted module. We are able to achieve 4.4x higher bandwidth rate and 65% lower latency without the need of application binary recompilation.

Keywords: intra-node, virtualization, sockets, Xen, networking, shared memory

1 Introduction

The advent of High-Performance-Computing (HPC) systems and the increasing needs for better control, isolation and resource management have made Virtual Machines (VMs) a significant part of modern data centers, HPC scientific applications and enterprise service platforms [1][2][3]. The key reason that make VMs such a critical factor of modern computing systems is the ability to execute intense applications and services providing a secure, isolated environment of execution, improving system utilization and communication cost between applications [6]. Today, power consumption is becoming an important topic for data center providers [5]. Virtual machines provide the capability of more efficient system utilization which results in energy cost reduction [4]. For these reasons, investing in virtualization technologies is a major trend for different applications and service providers.
Due to the benefits that virtualization provides to infrastructure providers, the same

concept is also exploited in network facilities. With the exploding data traffic through vast network infrastructures, middleboxes, i.e hardware network devices, are a fundamental part of today's networks. Although there are many advantages in using hardware middleboxes, there are also many reasons for shifting to *virtualized network functions* (VNFs), such as IP Routing, firewall, intrusion detection etc. [7][8]. VNFs are part of modern *network function virtualization* infrastructures (NFV) where VMs run on top of hardware network infrastructure and take responsible for providing network services [9].In addition to the above topics, virtual machines are also used in distributed execution environments, such as Hadoop MapReduce [11]. This framework is widely used in applications that require intensive data computations [12]. Virtual machines have become an attractive entity for hosting MapReduce workloads, which require fast communication between parallel tasks. For example, cloud-based services, such as Amazon's EC2, rely on VMs to process large amount of data by spawning tasks on different VMs.

Thus, recent virtualization techniques have given rise to a major set of new capabilities, but also to a number of limitations that researchers try to overcome. The field of improving virtualized computing environments is of great interest and refers to a large number of topics, from hypervisor scheduler optimization [13] to virtual machine reconfiguration [14][15].

In addition to these aspects, one important limitation that arise in both HPC applications but also in Cloud Computing applications is the communication cost between VMs. Virtual machines can reside in the same physical node or in different nodes. Proper placement or migration of VMs is a basic factor for providing low-latency and high-bandwidth communication for the reason that VMs hosting HPC or cloud applications can exploit their physical locality to increase performance. For instance, VNFs running in co-located VMs (such as routing, load balancing, firewall) may intensively exchange traffic, hence, taking advantage of proper VM placement and optimizing intra-node communication can offer significant overall performance gain. We focus on Xen [10] hypervisor and explore communication mechanisms in VMs located in the same physical node to achieve improvement in both latency costs and bandwidth rates.

We introduce YASMIN, a generic socket-compliant, efficient intra-node communication framework for co-located VMs in the Xen hypervisor. Although our implementation is build on Xen mechanisms, the basic concept can be applied to other hypervisors as well. YASMIN exploits the Xen's *grant table* and *event channel* mechanisms and provides page sharing between co-located VMs to simplify the data path in the network stack without sacrificing transparency. We achieve this by creating a communication channel between VMs that are aware of their location, bypassing the TCP/IP stack. We evaluate YASMIN using generic micro-benchmarks and compare it to conventional communication paths and bare-metal memory bandwidth (Sect. 4). We can observe that our framework outperforms the conventional methods both in terms of throughput as well as latency.

2 Background

2.1 Overview of Xen Architecture

Xen is a bare-metal hypervisor (Virtual Machine Monitor - VMM) which enables virtualization in paravirtualized mode. This means that the kernel of the guest VMs (domains) is modified in order to allow them to communicate with the privileged guest VM (Dom0). Basic operations for paravirtualized guests (disk, networking, GPU, etc.) are serviced through requests to the control domain which is responsible for communication with the hardware. Xen also exposes a set of hypercalls to guests. Hypercalls are privileged requests to the hypervisor which include granting page access to foreign domain, transferring and copying pages between domains and setting up an interrupt mechanism between domains.

2.2 Xen Default Networking

An overview of Xen's default network data path is shown in Fig.1. Networking is based upon the split-driver model; control domain is responsible for the coordination between the two communication ends. One end (domainX) forwards packets through the network stack (*TCP/IP*) to a virtual ethernet driver (*netfront*). The driver then copies the requests to a memory area mapped to the control domain. The driver in the control domain (*netback*) reads the requests from a ring buffer and copies the data in a proper kernel structure of the other end's *netfront* memory and delivers a signal. The other end (domainY) can now accept the new packet and forward it to the network stack (TCP/IP). The main limitation of this method is that all networking has to pass through the control domain which is a huge bottleneck for scaling to either a large number of processes between the same pair of VMs or a large number of processes between different pairs of VMs in the same physical node.

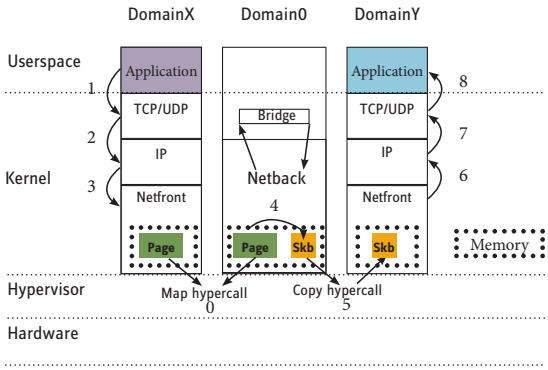

Fig. 1. The default inter-domain communication path in the Xen hypervisor. Numbers correspond to steps involved in the data path. Page is mapped before any exchange of data (step 0).

3 Design and Implementation

3.1 Design Overview

We decide to implement YASMIN design on top of the Xen hypervisor 4.4 using Linux 3.16 as guest OS. We build on Vchan [24], a Xen library which invokes system calls (`open()`, `ioctl()`, `mmap()`) to Xen's exported devices (`xen_gntdev`, `xen_gntalloc`) in order to initialize a channel between co-located domains and exchange data. We take this idea further and implement a transport layer for *vSockets* [25], i.e a generic sockets API similar to the POSIX interface which supports fast and efficient communication between guest virtual machines. vSockets API introduces a new address family (`AF_VSOCK`) and refers to the common socket-layer calls (`socket()`, `bind()`, `connect()`, etc.). A socket connection between two guest VMs can be established by using their domain ID numbers and a remote port. YASMIN consists of a loadable kernel module and a shared library to intercept system calls. The shared library intercepts IPv4 socket calls and translate them to vSockets socket calls, by using a 1-1 mapping between intra-node IP addresses and local domain IDs. An overview of YASMIN design is shown in Fig. 2.

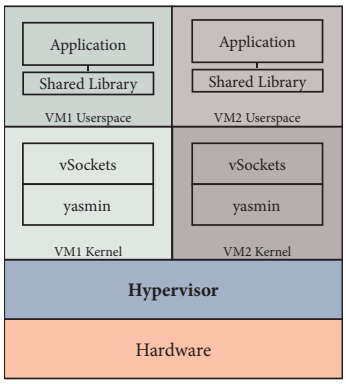

Fig. 2. YASMIN design overview

3.2 Implementation details

YASMIN implementation is based on Xen's primitive hypercalls, i.e granting page access to foreign VMs through grant-table mechanism, mapping pages using grant tables index number and invoking interrupts through the event channel mechanism. We exploit the producer-consumer shared ring technique, which does not require any locking mechanisms between the reader and the writer. Contrary to common approaches which do not take transparency into account, thus resulting in efficient but not binary-compatible code, we decide to design not only an efficient but also a transparent framework. In order to achieve this, we bypass the TCP/IP protocol

stack which introduces an extra overhead to each packet transmission, and we take utilize the vSockets socket protocol layer. It is a part of Linux kernel release and currently designed to support VMCI [26] as well as VIRTIO [27] transport layers. We extend this work and build a new transport layer for vSockets by adapting to Xen mechanisms. In this way, not only we avoid building a new network protocol from scratch but we also provide users with the capability of choosing the transport layer on the fly. To provide an architectural overview, we briefly describe how the operations are realized in each layer, from top to bottom:

Application layer: One of the most important aspects of our design is the API compatibility with the generic 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, reimplement or even recompile them. We implement a shared library which intercepts all system calls, filters out socket-API system calls (`bind()`, `listen()`, `accept()` etc.) and replaces them as follows: Our library queries a file which consists of entries of *domain_id-to-IP-addresses* mappings of all running guest domains in the same physical node. If the socket-call's target IPv4 address is matched, then the respective structures are initialized and the system call is forwarded to the kernel as a vSockets socket call (i.e `AF_VSOCK`). Otherwise, the remote application is not located in the same node and the default data path is followed.

Transport layer - Link layer: Each socket call invoked by userspace that corresponds to `AF_VSOCK` is serviced by vSockets protocol. This protocol is responsible for data fragmentation and packet delivery to the transport layer. The transport layer is the core of our implementation and is capable of creating a communication channel between co-located VMs, deliver messages and notify the remote domain for new packets. It is implemented as a kernel module and dynamically inserted to the kernelspace of each guest VM. Link layer is embedded in this module as a producer - consumer ring buffer in memory mapped between communicating VMs.

As mentioned earlier, Xen provides *grant table* mechanism which enables page sharing between VMs; one domain (granter) allocates a new page, grants access to the foreign domain by invoking a hypercall and refers to that page using its index in the grant-entry table. The other-end domain (grantee) allocates a new page and maps this page to the granter's page (also by invoking a hypercall) using the same grant-entry table index. The shared producer-consumer ring is part of the communication channel and is implemented as a set of pages shared by the two ends using the previous mechanism. Xen also introduces a simple signal passing mechanism between VMs, the *event-channel*, so as to inform the other end for packet delivery. The first domain (*allocator*) creates a new connection with the remote domain (*binder*) by invoking a hypercall which returns a local channel port number. *Allocator* then registers a new interrupt handler to this port. *Binder* can now "bind" to the port by invoking a hypercall, which in turn returns a local channel port number. *Binder* then registers a new interrupt handler to its local port. Each end can then invoke a hypercall and raise a virtual interrupt to notify the other end that data are available in the ring buffer and the respective interrupt handler will be invoked. We can now describe the path for a successful client-server message transmission between two co-located VMs. The overview of our design is presented in Fig. 3.

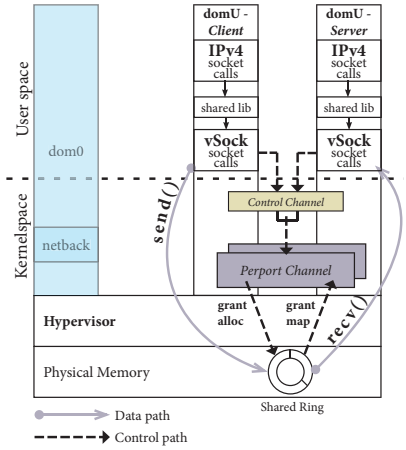

Fig. 3. YASMIN implementation overview. Each new socket connection is established through the *control channel* (control path arrows). For every connected pair of sockets, a new *perport channel* with its own shared ring is created, where data are exchanged (data path arrows)

– The inserted module, exports a XenStore path which will be monitored for incoming connection requests. When the client invokes a `connect()` socket call, our transport checks if previous requests to the specified remote domain have been made. If not, it creates a new intra-node communication channel between the domains and caches channels parameters for future communication requests. Channels parameters[1] are transmitted to remote domain via its XenStore path. This channel (*control channel*) is used only for transmission of control messages between domains (e.g *new socket connection request, socket release*). After the establishment of the control channel between a pair of guest VMs, a single-page queue is realized, which is used for sending new socket connections requests. These packets consist of the packet header, the grant-table's indices and event channel ports which will be used by the remote domain for mapping and registering respectively.
Next, it creates a new *persocket channel* and sends a new connection request to the remote application using the *control channel*. The new connection request specifies a remote domain ID and a remote port to connect to, similarly to IPv4 requests to a remote IP and a remote port. This channel is used for packet transmission between connected sockets. When the remote domain successfully registers the *persocket channel*, a reference to this channel is stored and vSockets `connect()` returns successfully. Each connection request to a new socket between the communicating VMs will create a new *persocket channel* but the *control channel* is unique for each pair of communicating VMs and will be teared-down only if guests shutdown or migrate.

[1] grant-entry index and event-channel port number

– Server-side applications can call `socket()`, `bind()`, and `listen()` to wait for incoming connections similarly to corresponding IPv4 socket calls. When the new connection request is made by the client-side application through the *control channel*, a virtual interrupt is triggered and the server-side's interrupt handler is invoked causing proper packet processing and enqueuing in listener's accept queue. This packet contains the grant references and event-channel ports of the *persocket channel*. The server-side will map the shared pages and bind the event-channel.
– A call to `accept()` by the server-side will dequeue the new connection request and send a *Connection OK* message to the client.
– `send()` socket call will cause a memory copy from userspace to the shared ring located in kernelspace and the update of the producer index.
– Similarly, `recv()` will cause a memory copy from the shared ring to userspace and the update of the consumer index.

Finally, to retain compatibility and transparency with `AF_INET` applications, we wrap around socket calls a library that re-issues all IPv4 calls with `AF_VSOCK` family.

4 Performance Evaluation

We setup a host machine with 2x Xeon E5335, 8GB RAM and single core guest VMs in order to evaluate the performance of our implementation in comparison to the default *netback/netfront* data path. We perform two microbenchmark experiments to test throughput and latency as well as scaling. We compare our results with the performance of bare-metal Unix Sockets and also with the system's bare-metal memory bandwidth. We use NetPIPE [28] to test latency and scaling, Iperf [29] to test throughput, netperf [30] to measure Unix Sockets throughput and STREAM benchmark [31] to compare to bare-metal memory bandwidth.

4.1 Microbenchmark Evaluation

As shown in Fig. 4 and Fig. 5, YASMIN outperforms the *netback/netfront* model in comparison to latency as well as throughput.

However, ring size is an important variable of performance. In Fig. 4 we can observe the effect of ring size on latency and in Fig. 5 the effect on throughput. For low message sizes (up to 1Kb), latency is not affected by the increase in ring size. We also observe that bandwidth is increasing for message sizes up to 2MB. However, there is a decrease in performance for messages up to 4MB, as depicted in Fig. 5. We are certain that this is caused due to increased contention on the memory bus. We plan to perform a detailed break-down analysis to validate our assumption.

In addition, throughput increases proportionally to the increase in ring size, as shown in Fig. 6. Throughput performance for ring size of 2MB reaches 76% of throughput performance of Unix Sockets on the bare-metal system, as shown in Fig. 6.

Nonetheless, we choose to implement a ring size of 512kB (128 pages) trading-off throughput and lower kernel memory consumption. For this ring size, latency is

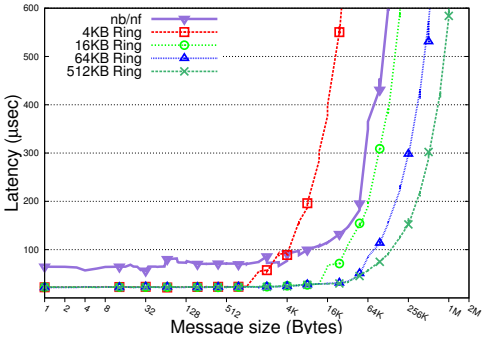

Fig. 4. Latency–to–message–size performance plot. "nb/nf" refers to the default netback/netfront data path.

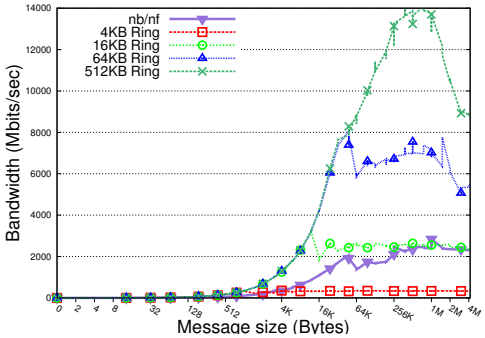

Fig. 5. Throughput performance for different ring sizes in comparison to the *netback/netfront* (nb/nf line).

reduced by 65%[2] compared to *netback/netfront* and average throughput is increased by a factor of 4.4, as measured by *Iperf*. Compared to the system's bare-metal memory bandwidth, YASMIN can perform at 16 Gbps (2048 MBytes/sec) while memory bus performance is measured at 2813 MBytes/sec for 1 executing thread and Unix Socket performance at 3250MB/s. The difference between memory bus and Unix Sockets performance is observed due to cache-miss penalties. Indeed, as we can measure using system's Performance Counters (PC), L2 cache misses were 32.6% of L2 cache hits while testing memory bandwidth but L2 cache misses were only 0.02% of L2 cache hits while testing Unix Sockets.

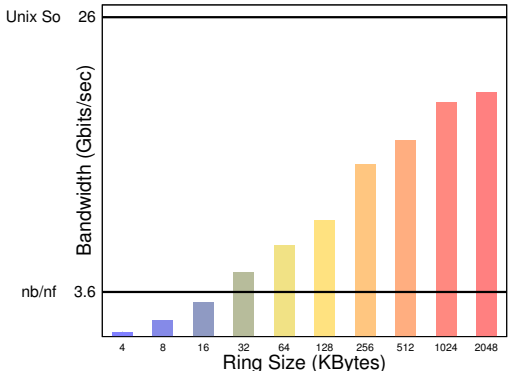

Fig. 6. Ring size effect on throughput. The line labeled "nb/nf" refers to throughput performance of the netback/netfront model and the line labeled "Unix So" to bare-metal Unix Sockets throughput.

[2] This value refers to a 1 Byte message

4.2 Scaling evaluation

Finally, in order to test YASMIN scaling performance, we setup in parallel up to 8 single core VMs which exchange messages in pairs (VM1 to VM2, VM3 to VM4, and so on...). Each VM is pinned to a CPU core and communicating VMs share a 4MB L2 cache memory. For example, when VM1 and VM2 are exchanging data, VM1 is pinned to CPU0 and VM2 to CPU1, where CPU0 is sharing a L2 cache with CPU1. The results of this experiment are depicted in Fig. 7. We can observe that the aggregate throughput increases proportionally to the number of communicating VMs. For instance, two VMs are exchanging a 512 KB message at 13.2 Gbps, while 8 VMs achieve 4x aggregate throughput for the same message size (53 Gbps or 6625MBytes/s). In comparison to the above result, we point out that bare-metal memory bus throughput for 8 threads of execution is measured at 3784Mbytes/s.

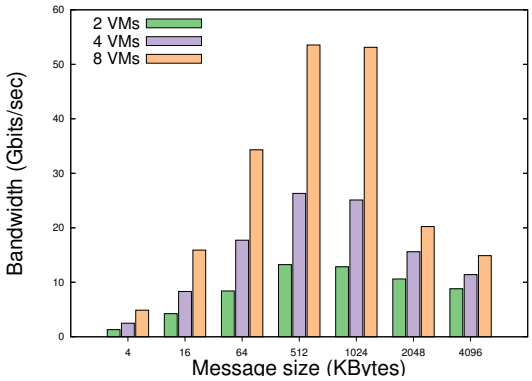

Fig. 7. Scaling performance

5 Related Work

Due to the significance of optimizing intra-node communication, literature in this field include a large number of proposals. A common proposed concept involves shared memory buffers between communicating VMs. Diakhate et al. [17] use shared memory techniques on the KVM hypervisor [17] by modifying QEMU [18] instances. IVC [19] proposes creating a one-way channel by using shared pages techniques and a new userspace API. XenSocket [20] also uses shared pages through Xen grant table hypercalls in order to create an one-way channel, by modifying BSD Sockets API and presenting a new address family. In addition to this work, an new address family implemented from scratch is introduced in XenSock [21]. Although these techniques can achieve better latency and bandwidth performance compared to the default model, transparency and compatibility are sacrificed. Applications need to be aware of running in co-located VMs and source code needs to be refactored and recompiled. XenLoop [22] creates a full-duplex channel between co-located VMs without sacrificing

transparency by intercepting outgoing packets from the network layer and establishing a fast communication channel between these VMs. A kernel module is responsible for analysing the packet destination MAC address and forwarding it in the established channel. A software bridge is responsible for keeping records of co-located MAC addresses. Although this technique can perform better in terms of throughput, there is not significant reduction in latency. Another approach in intra-domain communication is proposed by V4VSockets [23], a generic socket-applicant framework which performs better in terms of bandwidth as well as latency. The key idea of this implementation is based on copies made by the hypervisor to the receiver via the V4V mechanism, which resides in the Xen hypervisor. However, in this approach, the hypervisor is modified and the data path consists of three copies between the sender and the receiver.

In our implementation we combine the best parts of each of these techniques by bypassing both the control domain and the TCP/IP network stack. We also provide transparency and avoid the need of binary recompilation as well as hypervisor-intrusive techniques.

6 Conclusion and Future Work

YASMIN is a complete framework for intra-node communication which optimizes both throughput and latency compared to the default *netback/netfront* model. The data path includes only two copies, the first from sender's userspace to the kernelspace shared ring and the second from the shared ring to the receiver's userspace. Moreover, our implementation can also successfully respond to scaling challenges, as shown in Fig. 7. In addition to these, YASMIN optimizes communication between co-located VMs without the need to recompile binaries.

We conclude that in a large field of applications where communication is a critical factor of performance, placement of VMs in the same physical node is crucial for performance due to the fact that optimization techniques can be exploited. For these reasons, YASMIN can provide benefits to applications running in virtualized context.

We plan to improve YASMIN by upgrading the *hosts* file query process. Currently, the hosts file, which is queried by guest VMs to determine if a remote IP is co-located in the same node, is maintained by the node administrator. Therefore, in order to resolve VM migration issues, we plan to build a control domain backend driver or a guest VM daemon which will be responsible for monitoring any changes due to migration of virtual machines.

Finally, we plan to test our framework in NFV environments, where different VNFs can run on top of YASMIN transport layer. Suitable for testing are network functions such as routers, firewalls, load balancers, because they require fast packet processing and low latency response time.

YASMIN is an open-source framework and can be found at `https://github.com/mrozis/YASMIN.git`

References

1. Wei Huang, Jiuxing Liu, Bulent Abali, Dhabaleswar K. Panda, A case for high performance computing with virtual machines, ICS'06 June 2830, Cairns, Queensland, Australia.
2. Albert Reuther, Peter Michaleas, Andrew Prout, Jeremy Kepner, HPC-VMs: Virtual Machines in High Performance Computing Systems, IEEE High Performance Extreme Computing (HPEC) Conference, Waltham, MA, September 10-12, 2012.
3. Anand Tikotekar, Hong Ong, Sadaf Alam, Geoffroy Vallee, Thomas Naughton, Christian Engelmann, Stephen L. Scott, Performance Comparison of Two Virtual Machine Scenarios Using an HPC Application A Case study Using Molecular Dynamics Simulations, HPCVirt '09 Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing ,Nuremburg, Germany – March 31 - 31, 2009
4. Yang CT., Tseng CH., Chou KY., Tsaur SC., Hsu CH., Chen SC. (2010) A Xen-Based Paravirtualization System toward Efficient High Performance Computing Environments. In: Hsu CH., Malyshkin V. (eds) Methods and Tools of Parallel Programming Multicomputers. MTPP 2010. Lecture Notes in Computer Science, vol 6083. Springer, Berlin, Heidelberg.
5. Shehabi, A., Smith, S.J., Horner, N., Azevedo, I., Brown, R., Koomey, J., Masanet, E., Sartor, D., Herrlin, M., Lintner, W. 2016. United States Data Center Energy Usage Report. Lawrence Berkeley National Laboratory, Berkeley, California. LBNL-1005775.
6. Peter Strazdins, Richard Alexander, and David Barr. Performance Enhancement of SMP Clusters with Multiple Network Interfaces Using Virtualization, in G. Min, B. Di Martino, L.T. Yang, M. Guo and G. Runger (Eds), Frontier on High Performance Computing and Networking, LNCS 4331, Springer-Verlag, pp. 452–463, 2006.
7. Network Functions Virtualisation, An Introduction, Benefits, Enablers, Challenges & Call for Action, `https://portal.etsi.org/nfv/nfv_white_paper.pdf`
8. John DiGiglio, Davide Ricci, High Performance, Open Standard Virtualization with NFV and SDN, A Joint Hardware and Software Platform for Next-Generation NFV and SDN Deployments.
9. Network Function Virtualisation (FV); Use Cases, ETSI GS NFV 001, `http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf`
10. Xen Project Hypervisor, `https://wiki.xen.org/wiki/Xen_Project_Software_Overview`.
11. Apache Hadoop, `https://wiki.apache.org/hadoop`.
12. List of institutions that are using Apache Hadoop for educational or production uses, `https://wiki.apache.org/hadoop/PoweredBy`, `https://wiki.apache.org/hadoop/Distributions%20and%20Commercial%20Support`.
13. Hui Kang, Yao Chen, Jennifer Wong, Radu Sion, Jason Wu. Enhancement of Xen's Scheduler for MapReduce Workloads, HPDC'11, June 8–11, 2011, San Jose, California, USA.
14. Jongse Park, Daewoo Lee, Bokyeong Kim, Jaehyuk Huh, Seungryoul Maeng. Locality-Aware Dynamic VM Reconfiguration on MapReduce Clouds, HPDC'12, June 18–22, 2012, Delft, The Netherlands.
15. Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, Song Wu. Adaptive Disk I/O Scheduling for MapReduce in Virtualized Environment, 2011 International Conference on Parallel Processing.
16. François Diakhaté, Marc Pérache, Raymond Namyst, Herv e Jourdren. Efficient shared memory message passing for inter-VM communications. 2008.
17. Kernel Virtual Machine, `https://www.linux-kvm.org/page/Main_Page`.

18. QEMU, http://www.qemu.org/.
19. 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.
20. 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.
21. XenSock protocol design, https://patchwork.kernel.org/project/xen-devel/list/.
22. 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.
23. Anastassios Nanos, Stefanos Gerangelos, Ioanna Alifieraki and Nectarios Koziris. V4VSockets: low-overhead intra-node communication in Xen. In CloudDP15, April 21-24, 2015, Bordeaux, France.
24. Vchan Xen Library, https://github.com/mirage/xen/tree/master/tools/libvchan
25. VMware vSockets, https://pubs.vmware.com/vsphere-65/index.jsp#com.vmware.vmci.pg.doc/vsockAbout.3.2.html#1023121.
26. Virtual Machine Communication Interface, https://pubs.vmware.com/vmci-sdk/.
27. Virtio - IO Virtualization in KVM, http://www.linux-kvm.org/page/Virtio
28. NetPIPE – Network Protocol Independent Performance Evaluator, https://linux.die.net/man/1/netpipe.
29. iPerf Benchmark, https://iperf.fr/.
30. netperf - a network performance benchmark, https://linux.die.net/man/1/netperf.
31. McCalpin, John D., 1995: "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995