# Linux-based virtualization for HPC clusters

Lucas Nussbaum, Fabienne Anhalt, Olivier Mornard, Jean-Patrick Gelas

# Linux-based virtualization for HPC clusters

Lucas Nussbaum
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
lucas.nussbaum@ens-lyon.fr

Fabienne Anhalt
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
fabienne.anhalt@ens-lyon.fr

Olivier Mornard
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
olivier.mornard@ens-lyon.fr

Jean-Patrick Gelas
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
jpgelas@ens-lyon.fr

## Abstract

There has been an increasing interest in virtualization in the HPC community, as it would allow to easily and efficiently share computing resources between users, and provide a simple solution for checkpointing. However, virtualization raises a number of interesting questions, on performance and overhead, of course, but also on the fairness of the sharing. In this work, we evaluate the suitability of KVM virtual machines in this context, by comparing them with solutions based on Xen. We also outline areas where improvements are needed, to provide directions for future works.

## 1 Introduction and motivations

Operating System Virtualization, and all its variants, already largely proved their usefulness in the context of traditional servers. However, in the area of High Performance Computing (HPC), for computing clusters or, on a larger scale, grid or cloud computing, virtualization still has to convince most end users and system administrators of its benefits. The use of virtualization in the context of HPC offers several immediate advantages.

First, any computing center or large scale computing infrastructure under-uses a non-negligible number of physical resources. This is for example due to the fact that all the computing applications are not perfectly embarassingly-parallel. Using virtualization would allow to dynamically allocate resources to jobs, allowing to match their exact performance needs.

Next, many processing jobs do not take full advantage of the multicore architecture available on processing nodes. Deploying several Virtual Machines (VM) per node (e.g. 1 VM per core) would provide an easy way to share physical resources among several jobs.

On most computing grids, a user books a number of resources for a given period of time (also called *lease*). This period of time is generally a rough estimation made by the user of the time required for his application to complete. When the lease expires, results will be lost if the job did not have enough time to finish, and if no checkpointing mechanism is implemented. Good checkpointing mechanisms are difficult to implement, and virtualization provides an easy way to implement it, by freezing and migrating virtual machines.

Finally, the software configuration of computing platforms is generally static, which might be a problem for users with specific needs. Virtualization could allow to deploy customized user environments on the computing nodes, thus allowing users with specific software needs to customize the operating system on which their application will be executed.

Given the arguments listed above, virtualization seems to be an attractive solution for the HPC community. However, using virtualization in this context also has drawbacks.

Indeed, the overhead caused by the additional layers is not well known and controlled, mainly due to a lack of understanding of the underlying virtualization infrastructure.

Another issue is the physical resource sharing. Virtual machines need to access concurrently the physical devices, and it is possible that this sharing mechanism impacts the performance. This raises also the question of the scalability of the number of VMs it is possible to host on a physical machine.

The following section proposes a reminder about the common virtualization solutions currently available. It may help readers to set up the vocabulary in this domain and get familiar with Xen [3] and KVM [8]. Section 3 details our experimental testbed. This section is followed by an evaluation with several micro benchmark (CPU, disk, network) (section 4) and then with classic HPC benchmarks (section 5). Finally, before concluding (section 7) we propose a brief state of the art (section 6).

## 2 Virtualization

In this section, we describe the different virtualization techniques, and then introduce more specifically Xen [3] and KVM [8].

### 2.1 Virtualization approaches

The goal of virtualization is to partition one physical node (or system) into several independent virtual machines. Common applications of virtualization are server consolidation, and testing and development environments.

One common technique is OS-level virtualization where a single kernel is shared by containers which represent the VMs (*e.g* VServer). An other approach would be to allow several OS with distinct kernels to run on a single physical machine inside the VMs to give the user a maxium reconfiguration facility.

To manage several of these reconfigurable VM running on a physical node and sharing *de facto* the same hardware, we need a layer acting as a supervisor (a sort of arbiter to access hardware ressources). However, as VM include already a supervisor, we call this layer a hypervisor (*i.e* a supervisor of supervisors) also called VMM (Virtual Machine Monitor).

The hypervisor manages the requests of VMs and their access to the resources (i.e IRQ routing, time keeping and message passing between VMs).

Hypervisor virtualization can be divided in two types, Full Virtualization (FV) and Paravirtualization (PV), which can be both combined with hardware-assisted virtualization.

#### 2.1.1 Full virtualization

Full virtualization (FV) allows the execution of unmodified guest operating systems by emulating the real system's resources. This is especially useful to run proprietary systems. One pionneer was VMware providing a full virtualization solution. However, providing the guest system with a complete real system interface has an important cost. This cost can be mitigated by using *Hardware-assisted virtualization* discussed in section 2.1.3. KVM takes advantage of this evolution. Currently, in the x86 architecture, the hardware assistance is available in the CPU only, not in the other parts of the computer (like network or video adapters). The gap is then filled by emulation, having an impact on the performance. An alternative solution called hybrid [11] approach consists in using specific paravirtualized drivers which is more efficient than emulation (in terms of CPU consumption) and reaches better performances.

#### 2.1.2 Paravirtualization

Paravirtualization (PV) is also based on a hypervisor, but the devices are not emulated. Instead, devices are accessed through lightweight virtual drivers offering better performance.

The drawback is that guest kernels must be upgraded to provide new system calls for the new services. At the lowest level the syscalls are interrupts (0x80) with a function number, which allows to switch from the user mode to the privileged mode in former Linux system call. The newest Linux system uses now a faster method with the syscall/sysenter opcodes (in x86 architecture). In the same way in Xen [3], the OS executes hypercalls with the interrupt 0x82. Like in the Linux system, the use of interrupts is deprecated and replaced by the use of hypercall pages [18], a similar mechanism in Linux called vDSO used to optimize the system call interface[1]. vDSO chooses between int 0x80, sysenter or syscall opcodes (the choice is made by the kernel at boot time).

#### 2.1.3 Adding hardware virtualization support

Virtualization software techniques consisting in doing binary translation to trap and virtualize the execu-

[1] `http://www.trilithium.com/johan/2005/08/linux-gate/`

tion of some instructions are very cost inefficient (ex: VMware). Running a VM on a common architecture (ex: IA32 PC) for which it has not been designed is difficult. The original x86 architecture does not comply with the base conditions for being virtualized (equivalence, resource control (safety), efficiency) [14]. In particular, there are some unpriviledged instructions changing the state of the processor that can not be trapped.

In 2007, Intel and AMD designed (independently) some virtualization extensions for the x86 architecture [12](VMX for Intel, Virtual Machine eXtension; and AMD-V/Pacifica for AMD). Each one allows the execution of a hypervisor in order to run an unmodified operating system while minimizing the overhead due to emulation operations.

The kernels can run in priviledged mode on the processor, which means on ring 0. Ring 0 is the most privileged level. On a standard system (i.e not virtualized) this is where the operating system is running. The rings strictly over 0 run instructions in a processor mode called unprotected. Without specific hardware virtualization support, the hypervisor is running in ring 0, but the VM's operating system can not reach this level of priviledge (they access ring 1, at best). Thus, in full-virtualization, priviledged instructions are emulated, and in paravirtualization the kernel is modified in order to allow those instructions to access ring 0. The hardware assisted virtualization not only proposes new instructions, but also a new priviledged access level, called "ring -1", where the hypervisor can run. Thus, guest virtual machines can run in ring 0.

Despite these advantages, using an untouched/unmodified operating system means a lot of VM traps and then a high CPU consumption used by the emulation of hardware (network manager, video adapter, ...). An alternative solution, called "hybrid" [11], consists in using paravirtualized drivers in combination with the hardware-assisted virtualization.

## 2.2 Introducing Xen and KVM

In this article, we limit our study to free and open source virtualization solutions. Thus, we chose to study and evaluate exclusively the lastest releases of Xen and KVM [8] at the time of writing, that are Xen 3.3.1 and KVM 84.

### 2.2.1 Xen

Xen started as a research project by Ian Pratt at Cambridge University. The very first public release of Xen was delivered in October 2003. Then, Ian Pratt created the XenSource company, which develops the project in an open source fashion and distributes customized Xen versions (Xen Enterprise). Major releases 2.0 and 3.0 were delivered respectively in 2004 and 2005. The latest current release available is 3.3.1 (2009). Xen is compatible with x86 processors (Intel or AMD), x86_64 since 3.0, SMP architectures, HyperThreading technology, IA64, PPC. ARM support should also be available soon.

Xen is a hypervisor and has the ability to run guest operating systems, called domains. There are two types of domains. Unpriviledged domains (called *DomU*) are the guest systems, while the priviledged domain (called *Dom0*) is a special guest with extended capabilities, that contains the applications to control the other guests. *Dom0* is running above the Xen hypervisor when the physical machine starts. It runs a modified Linux kernel, and is generally the only domain able to interact directly with the hardware through the linux kernel drivers. It also allows DomUs to communicate with hardware devices using their virtual drivers.

When a virtual machine hosted in a domU previously described wants to use hardware devices, e. g. the network interface or the block device, the data has to go to dom0 which is than in charge of transmitting it to the physical device. Several mechanisms are invoked to make the transfert between domU and dom0 and to minimize overhead. However, the data path is longer than without virtualization, as shown in Figure 1.

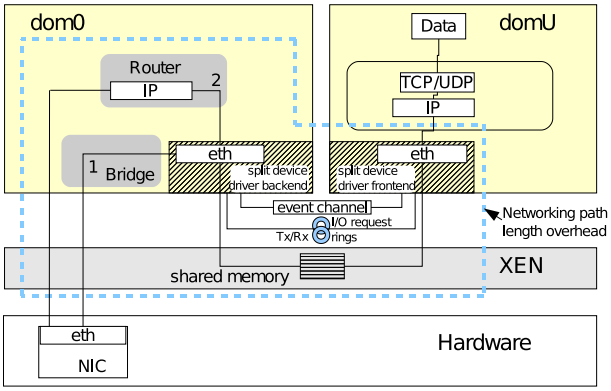


Figure 1: Network path in Xen.

In this example where domU uses the physical network interface to send packets to a remote physical station, packets go through the domU TCP/IP stack, then are transferred to dom0. To make this transfert, dom0 invokes a grant to domU's memory page to fetch the data by page flipping. The other way around, during packet reception on domU, dom0 copies the data into a shared memory segment so that domU can get it [5]. This copy and page flipping mechanisms offer security but are heavy for the performance.

From a technical point of view, on the x86 architecture (with no hardware support), the Xen hypervisor is running in ring 0, kernels in ring 1 and finally applications in ring 3. On the x86_64 architecture, the hypervisor is running in ring 0, and guest domains (*Dom0* and *domUs*) and applications run in ring 3 (i.e rings 1 and 2 have been removed).

Moreover, the x86 Intel architecture proposes two levels of memory protection. One is based on a segmentation mechanism and the other on page management. Each of these protections may be used to isolate virtualized systems. (NB: Two modes are currently available on x86 architectures: 32 bit mode and 64 bit mode. Only x86 64 bit system architectures (a.k.a IA32e or AMD64) may use both modes). In the 32 bit mode, segment management is fully operational and is used for the memory protection.

In 64 bit mode, segment management almost disappears (for example there is no more segment base and limit management), memory protection through segments is not possible anymore, thus protection through memory page management is used (via the MMU unit)[6]. However, with this mechanism, there are only two levels of protection called normal mode and supervisor mode. Xen must then manage protection through the page level which is the most CPU intensive. Without forgetting that context switching introduced by virtulization is time consuming and so impacts the guest systems performances.

Finally, the inclusion of different parts of Xen in the Linux kernel has been the subject of animated discussions. DomU support is already included, but the inclusion of Dom0 support faced a lot of opposition. In addition to that, Linux distributions use the XenSource-provided patch for 2.6.18, and forward-port this patch to the kernel releases they want to ship in their stable release. The process of forward-porting those patches is difficult, and not supported by the upstream author, leading some distributions to choose to stop supporting Xen recently.

### 2.2.2 KVM

KVM (Kernel based Virtual Machine) is an open source Linux kernel virtualization infrastructure[2] which relies on the hardware virtualization technologies, fully integrated in the Linux kernel. Its first version was introduced in the 2.6.20 Linux kernel tree (released in february 2007). KVM developers are primarily funded by a technology startup called Qumranet, now owned by RedHat. Developers had an original approach. Instead of creating major portions of an operating system kernel themselves, they choose to use the Linux kernel itself as a basis for a hypervisor. Thus, KVM is currently implemented as loadable kernel modules. `kvm.ko`, that provides the core virtualization infrastructure and a processor specific module, `kvm-intel.ko` or `kvm-amd.ko`. The code is relatively small (about 10,000 lines) and simple. This original approach has several benefits. The virtualized environment takes advantage of all the ongoing work made on the Linux kernel itself.

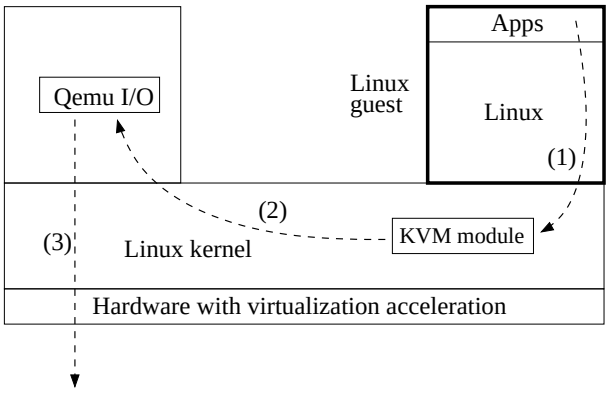


Figure 2: Path of I/O requests in KVM

KVM makes use of hardware virtualization to virtualize processor states (an Intel processor with VT (virtualization technology) extensions, or an AMD processor with SVM extensions (also called AMD-V)). With KVM, each virtual machine is a regular Linux process, scheduled by a standard Linux scheduler. Memory management of the VM is handled from within the kernel but I/O in the current version is handled in user space

[2] `http://www.linux-kvm.org`

through a userspace component also used to instantiate the virtual machines. This component is actually a modified version of QEMU handling I/O hardware emulation: as shown in figure 2, when a process in a guest system issues an I/O request, the request is trapped by KVM, then forwarded to the QEMU instance in the host system's userspace, that issues the real I/O request. KVM emulates virtual devices, such as network interfaces or hard disks. In order to improve performance, recent KVM versions propose a hybrid approach called *virtio* [16]. Virtio is a kernel API that improves the performance of communications between guest systems and the host system by providing a simpler and faster interface than the emulated devices from QEMU. Virtio-based devices exist both for network interfaces and hard disks.

#### 2.2.3 Conclusion

Hardware Assisted Full Virtualization (FV) is often believed to be the best virtualization solution, performance-wise. However, this is not true: paravirtualization approaches may be much better in terms of performance, especially in the context of the IO. In the following sections, we perform several benchmarks outlining the performance differences of the different virtualization techniques and explain why there are such differences.

## 3 Evaluation

In the following experiments, we compare four different virtualization solutions:

**Xen FV** : Xen using full hardware-assisted virtualization (also called Xen HVM for Hardware Virtual Machine)

**Xen PV** : Xen using paravirtualization

**KVM FV** : standard KVM, using the I/O devices emulated by QEMU

**KVM PV** : KVM using the virtio I/O devices

All the experiments were performed on a cluster of Dell PowerEdges 1950 with two dual-core Intel Xeon 5148 LV processors with 8 GB of memory, 300 GB Raid0 / SATA disks and interconnected by 1 Gb/s network links. We used Xen 3.3.1 and KVM 84 for all tests, except when specified otherwise.

We first evaluate all solutions with a set of micro-benchmarks, to evaluate the CPU, the disk accesses and the network separately, then use the HPC Challenge benchmarks, a set of HPC-specific benchmarks.

## 4 Evaluation with micro-benchmarks

In this section, we evaluate the different virtualization solutions with a set of micro-benchmarks.

### 4.1 CPU

In our first experiment, we focused on CPU-intensive applications. We evaluated the overhead caused by using a virtualization solution for such applications, which are obviously crucial for HPC.

Both Xen and KVM support SMP guests, that is, giving the ability to a guest system to use several of the host's processors. We executed a simple application scaling linearly on hosts with four CPUs, then inside guests to which four CPUs had been allocated.

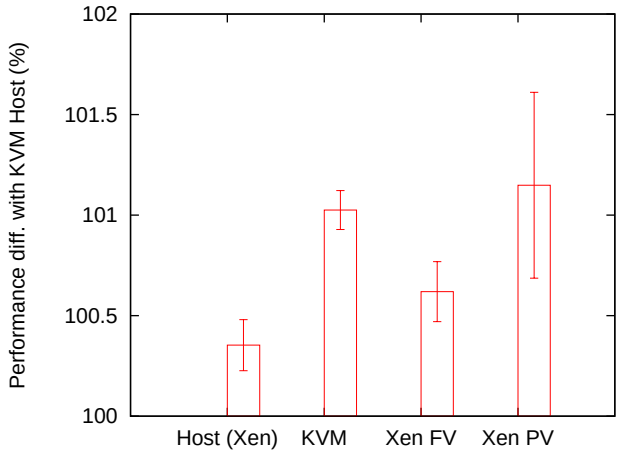


Figure 3: Influence of virtualization on CPU-intensive applications: the same application is executed on a 4-CPU system, then on guests allocated with 4 CPUs.

Figure 3 shows the difference between our host system running Linux 2.6.29, used for KVM, and several other configurations. In all cases, the overhead was minimal (lower than 2%). However, it is worth noting that running the application in the Xen dom0 is slightly slower

than on Linux 2.6.29 (maybe because of improvements in Linux since the 2.6.18 kernel used as Dom0), and that both KVM and Xen guests suffer from a small slowdown.

### 4.2 Disk

While disks can now be considered slow devices (compared to high-speed NICs, for example), it can still be difficult to fully exploit their performance for virtualization solutions.

In this experiment, we compare the different solutions by writing large files using `dd`, using different block sizes. This allows to measure both the influence of per-I/O overhead (for small block sizes) and available I/O bandwidth. We also confirmed our results using `bonnie++`.

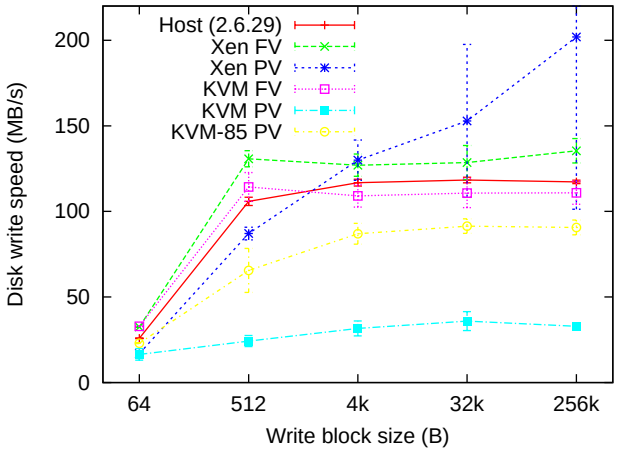


Figure 4: Disk write performance

Results are presented in Figure 4. Each test was run 10 times, and the vertical bars indicate the 95% confidence interval. We used file-backed virtual machines for all tests. Our test systems allowed a maximum write speed of about 120 MB/s on the host system, thanks to the RAID-0 disks setup.

In our tests, Xen PV reported write speeds much higher than the write speed that we obtained from the host system, with a very high variability. While we were not able to confirm that, it seems that Xen reports write completions before they are actually completely committed to disk.

While KVM provided good performance (close to the host system) in full virtualization mode, *virtio* provided more disappointing results. In fact, we identified with `blktrace` that a lot of additional data was written to disk with *virtio*: writing a 1 GB-file resulted in about 1 GB of data written to disk without *virtio*, versus 1.7 GB of data written with *virtio*. This is very likely to be a bug. Since our tests were originally performed with KVM 84, we also re-ran the tests with a version from the KVM git tree, very close to the KVM 85 release date. This more recent version provided better performance, but still far from the one obtained with the other configurations.

It is also worth noting that, while the size of block sizes clearly affects the resulting performance (because of the influence of latency on the performance), it affects all solutions in a similar way.

### 4.3 Network

In this section, the network performance in virtual machines with KVM is compared to Xen network performance using either hardware virtualization or paravirtualization techniques.

To measure throughput, the `iperf` [17] benchmark is used sending TCP flows of 16 kByte messages on the virtual machines. The corresponding CPU cost is measured with the Linux `sar` utility on KVMs and with `xentop` on Xen. Each result is the average of 10 runs of 60 seconds of each test. To each virtual machine, one of the 4 physical CPUs is attributed. The different domains are scheduled in Xen to use the CPUs with default credit-scheduler [19]. In KVM, virtual machines use the emulated e1000 driver for hardware virtualization and virtio for paravirtualization. The virtual machines communicate using virtual tap interfaces and a software bridge interconnecting all virtual machines and the physical network interface. Xen virtual machines under paravirtualization use the virtual split device driver and Xen HVMs use the emulated Realtek 8139 driver. They communicate also using a software bridge in host domain 0.

#### 4.3.1 Inter virtual machine communication

In this first experiment, network performance between virtual machines hosted on the same physical machine is evaluated not invoking the use of the physical network

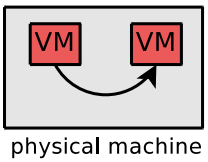


Figure 5: Test setup with two communicating virtual machines hosted on the same physical machine.

| | FV | PV |
|---|---|---|
| KVM | 648.3 | 813.2 |
| Xen | 96.05 | 4451 |

Table 1: Average TCP throughput in Mbit/s between two virtual machines hosted on a single physical machine under full virtualization (FV) or paravirtualization (PV).

interface and allowing to evaluate the network speed allowed by the CPU under the virtualization mechanisms. This setup is represented on Figure 5.

The two virtual machines communicate using different mechanisms according to the virtualization technique. In Xen, packets go to the virtual or emulated driver to reach dom0, than to dom0's backend driver to reach the destination virtual machine. On KVM, the packets use also an emulated or virtual interface and are than handled by the kvm module to be sent to the destination virtual machine.

The results in terms of TCP throughput in the four configurations are represented in Table 1.

The best throughput is obtained on Xen paravirtualized guests which can communicate in a very lightweight way achieving nearly native Linux loopback throughput (4530 Mb/s) to the cost of an overall system CPU use of around 180% while native Linux CPU cost is about 120%. However with hardware assisted virtualization, Xen has very poor throughput with even more CPU overhead (about 250% of CPU use) due to the network driver emulation. KVM achieves a throughput between 14 and 18% of native Linux loopback throughput generating a CPU cost between about 150 and 200%.

#### 4.3.2 Communcation with a remote host

For communications between virtual machines hosted by distinct physical servers, the packets need to use the

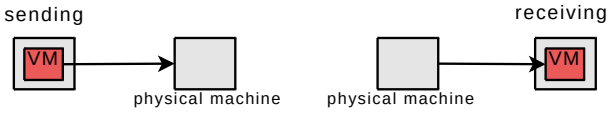


Figure 6: Test setup with a virtual machine communicating with a remote physical host.

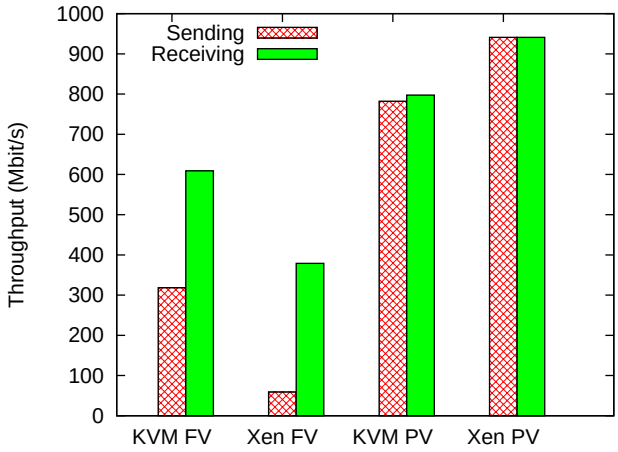


Figure 7: TCP throughput on a virtual machine sending to or receiving from a remote host.

physical network interfaces of the hosts. This experiment evalutes the resulting performance. As sending and receiving do not invoke the same mechanisms, sending throughput is evaluated separately from receiving throughput as represent the two configurations on Figure 6.

Figure 7 shows the throughput obtained on the different types of virtual machines.

Para-virtualization shows much better results in terms of throughput than hardware virtualization like in the previous experiment, with KVM and Xen. While with Xen para-virtualization, the theoretical TCP throughput of 941 Mb/s is reached, with KVM and the paravirtualized driver, throughput reaches only about 80% of native Linux throughput. In Xen, the network interface is the bottleneck as loopback throughput reaches 4451 Mb/s. In KVM paravirtualization, the virtualization mechanism is the bottleneck, as the same throughput is obtained, whether the network interface card is used or not. With hardware-virtualization, the network performance is very poor, especially with Xen HVM and in the case of sending. This shows that the sending mechanism from the Xen HVM is obviously the bottleneck also in the previous experiment. KVM FV uses about

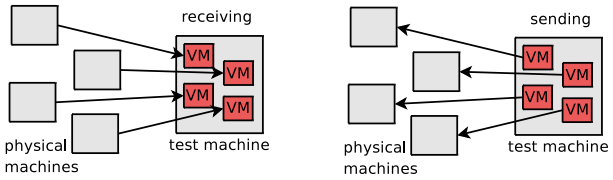


Figure 8: Test setup with 4 virtual machines communicating with 4 remote physical host.

100% of the CPU, which is assigned to it and can not achieve better throughput, needing more CPU capacity. In the case of paravirtualization with virtio, KVM needs less CPU, about 76% for sending and 60% for receiving while the physical host system is performing a part of the work to acces the physical NIC. The overall system CPU usage is still about 100% of one CPU, but the resulting bandwidth more than doubles in the case of sending.

#### 4.3.3 Scalability

This experiment evaluates the impact on throughput while scaling up to either 2, 4 or 8 virtual machines hosted on a sigle physical host. Figure 8 shows an example with 4 VMs.

As in the previous experiment, sending an receiving throughput is evaluated separately.

The aggregated TCP throughput obtained on the virtual machines for sending and receiving is represented respectively on Figures 9 and 10 in each configuration (KVM and Xen, with para- or full-virtualization).

In both configurations, KVM and Xen, paravirtualization achieves better throughput, like before. Observing the evolution of the aggregate throughput with an increasing number of virtual machines, it can be seen that a bigger number of VMs achieve a better overall throughput than a single virtual machine. In the case of KVM, this might be related to an important CPU overhead necessary for networking. With a single virtual machine sending a single TCP flow, KVM consumes the capacity of an entire CPU whether it uses the emulated e1000 driver or virtio. Using two virtual machines sending two flows, they can each one use one CPU which actually happens for KVM full virtualization where throughput still not reaches the maximum value allowed by the NIC. Paravirtualization needs less

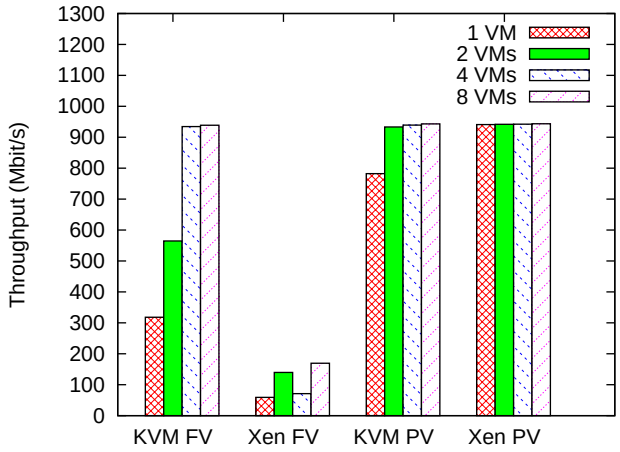


Figure 9: TCP sending throughput on a set of up to 8 VMs.

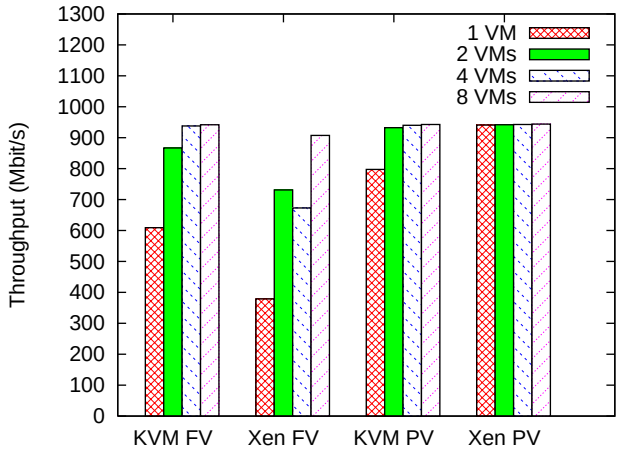


Figure 10: TCP receiving throughput on a set of up to 8 VMs.

instructions making KVM use only between 130 and 165% of the 4 CPU cores and achieving nearly maximum TCP throughput.

Xen HVM has the most important CPU overhead, especially with 4 or 8 virtual machines, and achieves the poorest throughput. Only dom0 uses almost 250% of the CPUs to forward the traffic of 4 virtual machines. This means that it uses at least 3 CPUs simultanously and need to share them with the domUs. This sharing needs more context switches. With Xen paravirtualization the overall system CPU utilization does not exceed 160% for dom0 and 8 domUs, allowing to achieve maximum TCP throughput.

In each experiment, the different virtual machines achieve almost the same individual throughput. For this reason, only the aggregated throughput is represented. Per virtual machine throughput corresponds to the aggregated throughput to the number of VMs. This means that the resource sharing is fair between the different virtual machines.

For inter-VM communications on a same physical host and also for communications using the physical interface, despite it's virtualization overhead, Xen paravirtualization achieved the best network throughput having the lowest CPU overhead compared to hardware virtualization and KVM. However, KVM with the virtio API and the paravirtualized drivers can achieve similar throughput if it has enough CPU capacity. This solution could be a good tradeoff between performance and isolation of virtual machines.

## 5 Evaluation with classical HPC benchmarks

In this section, we report on the evaluation of the various virtualization solutions using the HPC Challenge benchmarks [9]. Those benchmarks consist in 7 different tools evaluating the computation speed, the communication performance, or both, like the famous LINPACK/HPL benchmark used to rank the computers for the Top500 [3].

The following results were obtained with HPCC 1.3.1 on a cluster of 32 identical Dell PowerEdge 1950 nodes, with two dual-core Intel Xeon 5148 LV CPUs and 8 GB of RAM. The nodes are connected together using a Gigabit ethernet network. The benchmarks were run on several configurations:

[3] http://www.top500.org

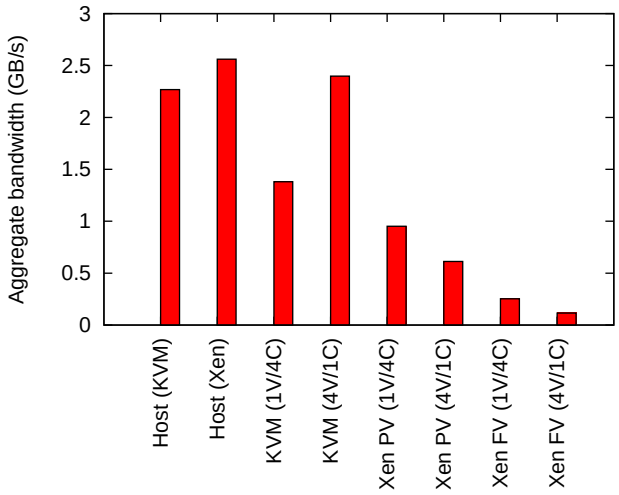


Figure 11: PTRANS benchmark: aggregate network bandwidth

- the host system used to run KVM virtual machines (using Linux 2.6.29) ;
- the host system used to run Xen virtual machines (Xen dom0, using Linux 2.6.18) ;
- 32 KVM virtual machines allocated to 4 CPUs each, using *virtio* for network and the classic emulated driver for disk;
- 128 KVM virtual machines allocated to 1 CPU each, using *virtio* for network and the classic emulated driver for disk ;
- 32 Xen paravirtualized virtual machines allocated to 4 CPUs each ;
- 128 Xen paravirtualized virtual machines allocated to 1 CPU each ;
- 32 Xen virtual machines using full virtualization, allocated to 4 CPUs each ;
- 128 Xen virtual machines using full virtualization, allocated to 1 CPU each.

### 5.1 PTRANS benchmark

PTRANS (parallel matrix transpose) excercises the communication network by exchanging large messages between pairs of processors. It is a useful test of the total communications capacity of the interconnect. Results shown in Figure 11 indicate that :

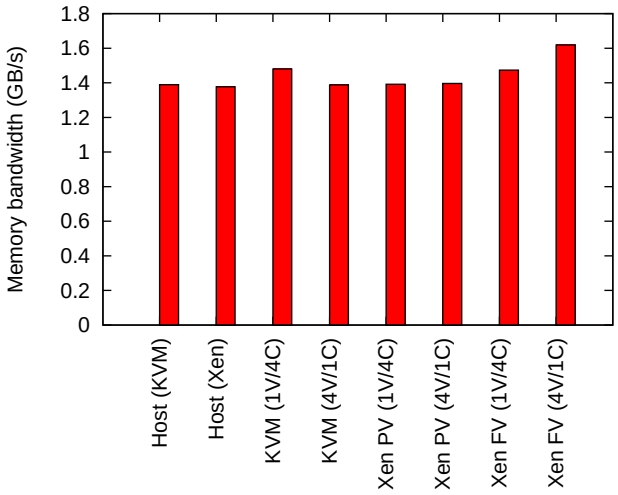


Figure 12: STREAM benchmark: per-node memory bandwidth

- The setup using four KVM VMs per node performs better than the one using 1 VM per node, and provides performance that is close to native. This might be explained by the fact that having several virtual machines allows the load to be better spread across CPUs ;
- Xen setups perform very poorly in that benchmark.

### 5.2 STREAM benchmark

STREAM is a simple benchmark that measures the per-node sustainable memory bandwidth. As shown in figure 12, all configurations perform in the same way (differences are likely to be caused by measurement artifacts).

### 5.3 Latency and Bandwidth Benchmark

The latency and bandwidth benchmark organizes processes in a randomly-ordered ring. Then, each process receives a message from its predecessor node, then send a message to its successor, in a ping-pong manner. 8-byte and 2-MB long messages are used.

Figure 13 presents the latency results. Results for Xen with full virtualization are not included, as the average is 11 ms (1 VM with 4 CPU case) or 8 ms (4 VM with 1 CPU), probably because of the much lower available

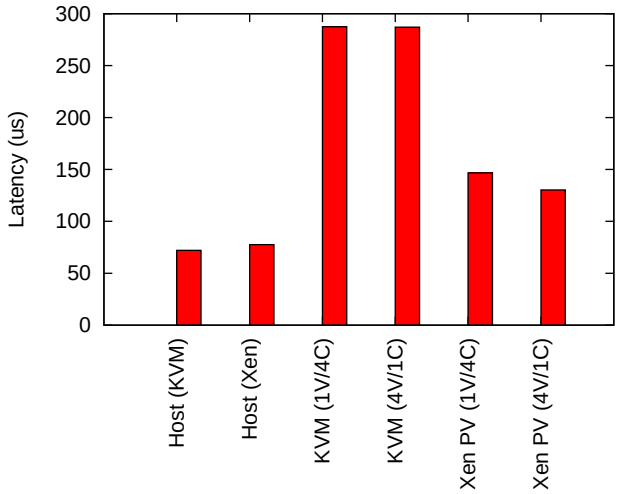


Figure 13: Average node-to-node latency

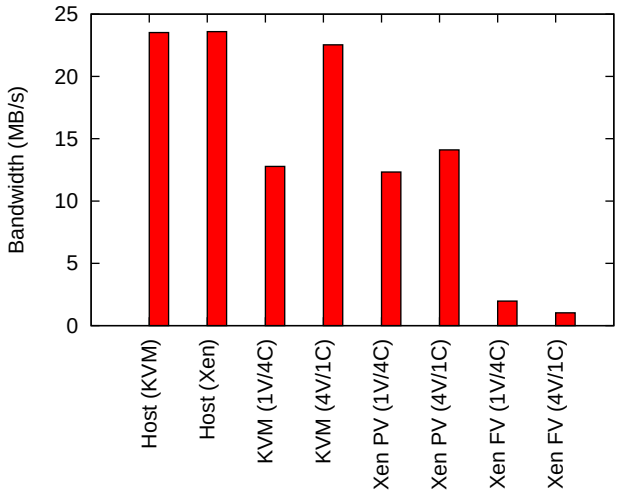


Figure 14: Average node-to-node bandwidth

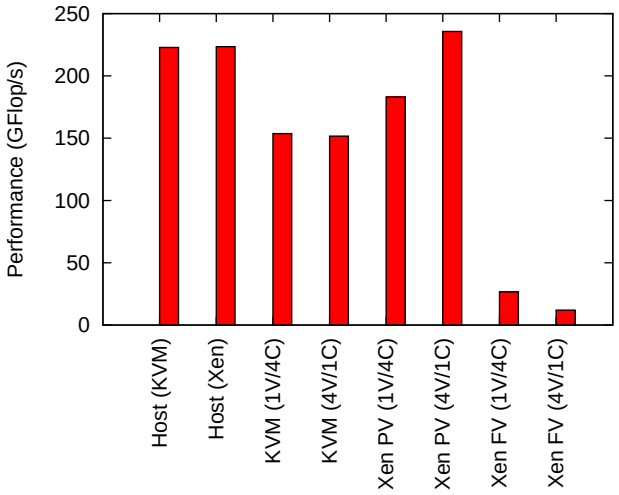


Figure 15: LINPACK benchmark: overall performance

bandwidth. Xen with paravirtualization performs much better than KVM (146 or 130 $\mu$svs 286 $\mu$s).

Figure 14 presents the bandwidth results, which are similar to those of the PTRANS benchmark.

### 5.4 LINPACK/HPL Benchmark

The LINPACK benchmark combines computation and communications. It is used to determine the annual Top500 ranking of supercomputers. Figure 15 shows the LINPACK results (in GFlop/s) for our configurations.

The best configuration is the Xen setup with 4 virtual machines per node, which reaches 235 GFlops, compared to 222 for the host system. This might be caused by the fact that splitting a physical host into four virtual machines allows for better scalability, compared to when the same kernel is used for the four processes. Also, as we showed in section 4.3.1, the inter-VM bandwidth is 4.4 Gbps, leading to no performance degradation compared to the host system case.

KVM results show more overhead, likely to be caused by the important communication latency between virtual machines (section 5.3), which is not compensated by the inter-VM bandwidth. Contrary to what happens with Xen, both KVM configurations (4 VMs with 1 CPU, and 1 VM with 4 CPU per node) give similar performance.

## 6 Related work

I/O performance is a critical point in virtual machines, and it depends on the kind of virtualization. Previous evaluations compared para-virtualization and full-virtualization like Xen and VMware to OS-level virtualization like VServer[4] and UML[5]. This showed best network and disk access performance for Xen and VServer [15].

Between these two solutions, oftenly VServer performing only control plane virtualization is prefered as in VINI [4] in order to maximise performance. However this offers less isolation, security and reconfigurability, virtualizing at the OS level and so sharing a single OS, while our goal is to have completely isolated systems for more flexibility and security. This lead to concentrate on full- or para-virtualization solutions rather than container based ones.

Xen is probably the most evaluated virtualization solution. Since its appearance in 2003, Xen I/O and especially network virtualization has been constantly improved achieving growing network performance with its successive versions [2] to reach today native Linux throughput on para-virtual machines. Offloading features have been added to virtual NICs in Xen 2.0 and page flipping has been changed to copying to leightweight the operations [10]. Unfairness problems have been corrected in the Credit-Scheduler and the event channel management [13].

Detailed Xen I/O performance has been examined [7] rejecting the Xen data-plane paravirtualization for its performance overhead but proposing Xen virtualization as a viable solution on commodity servers when using direct hardware mapped virtual machines. However, this would not offer the same flexibility requiering dedicated hardware for each virtual machine. Having this isolation and flexibility goal in mind, this paper shows that Xen data-plane virtualization achieves better performance compared to other techniques. In fact, it seems that KVM did not yet reach the same maturity than Xen in I/O management with paravirtualization.

Studies on Xen para-virtualization in the HPC context showed that Xen performs well for HPC in terms of memory acces, and disk I/O [21] and communication

[4] http://linux-vserver.org

[5] http://user-mode-linux.sourceforge.net

and computation [20]. To know if it was due to the paravirtualized driver or to specific Xen implementations, we also compared Xen performance ton KVM performance which is a very recent KVM solutions offering the same isolation features (data-plane virtualization and full isolation) offering also full and paravirtualization.

## 7 Conclusion and future work

In this work, we evaluated different aspects of KVM and Xen, focusing on their adequacy for High Performance Computing. KVM and Xen provide different performance characteristics, and each of them outperforms the other solution in some areas. The only virtualization solution that consistently provided bad performance is Xen with full virtualization. But both Xen with paravirtualization, and the KVM approach (with paravirtualization for the most important devices) clearly have their merits.

We encountered problems when setting up both solutions. Our choice to use Xen 3.3 implied that we had to use the XenSource-provided 2.6.18 kernel, and couldn't rely on an easier-to-use and up-to-date distribution kernel. This brought the usual issues that one encounters when compiling one's own kernel and building software from source. KVM proved to still be a relatively young project (especially its *virtio* support) and also brought some issues, like the unsolved problem with *virtio_disk*.

Finally, while we aimed at providing an overview of Xen and KVM performance, we voluntarily ignored some aspects. The first one is Xen's and KVM's support for exporting PCI devices to virtual machines (*PCI passthrough*). This is important in the context of HPC to give virtual machines access to high-performance networks (Infiniband, Myrinet), but also raises questions on how those devices will be shared by several virtual machines. Another aspect that can be useful in the field of HPC is VM migration, to be able to change the mapping between tasks and compute nodes. In our performance results, we ignored the problem of fairness between several virtual machines: the execution of a task in one VM could have consequences on the other VM of the same physical machine.

Finally, it would be interesting to explore the new virtualization possibilities known as Linux Containers [1]. By providing a more lightweight approach, they could provide a valuable alternative to Xen and KVM.

## Acknowledgements

We would like to thank Pascale Vicat-Blanc Primet for her support and leadership.

## References

[1] Linux containers. http://lxc.sourceforge.net/.

[2] Fabienne Anhalt and Pascale Vicat-Blanc Primet. Analysis and experimental evaluation of data plane virtualization with Xen. In *ICNS 09 : International Conference on Networking and Services*, Valencia, Spain, April 2009.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI veritas: realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14. ACM, 2006.

[5] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

[6] Intel Corporation, editor. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2008.

[7] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Fairness issues in software virtual routers. In *PRESTO '08*, pages 33–38. ACM, 2008.

[8] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Linux Symposium*, 2007.

[9] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, New York, NY, USA, 2006. ACM.

[10] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2. USENIX Association, 2006.

[11] Jun Nakajima and Asit K. Mallick. Hybrid - virtualization–enhanced virtualization for linux. *Ottawa Linux Symposium*, June 2007.

[12] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. Technical report, Intel Technology Journal, 2006.

[13] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling I/O in virtual machine monitors. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[14] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. In *Communications of the ACM 17 (7)*, pages 412–421, July 1974.

[15] Benjamin Quetier, Vincent Neri, and Franck Cappello. Scalability comparison of four host virtualization tools. *Journal of Grid Computing*, 2006.

[16] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[17] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. iperf : testing the limits of your network. http://dast.nlanr.net/Projects/Iperf.

[18] Willam von Hagen. *Professional Xen Virtualization*. Wiley Publishing, Inc., 2008.

[19] Yaron. Creditscheduler - Xen wiki. *http://wiki.xensource.com/xenwiki/CreditScheduler*, 2007.

[20] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Evaluating the performance impact of Xen on MPI and process execution for hpc systems. In *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, 2006.

[21] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for HPC systems. In *In Proc. Workshop on Xen in High-Performance Cluster and Grid Computing*, pages 474–486. Springer, 2006.