

# CHALMERS



## Graphics processing on HPC virtual applications Graphics performance of Windows applications running on Unix systems

Master of Science Thesis  
Computer Systems and Networks

Roi Costas Fiel

Department of Computer Science and Engineering  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Gothenburg, Sweden, September 2014

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Graphics processing on HPC virtual applications  
Graphics performance of Windows applications running on Unix systems

Roi Costas Fiel

Examiner: Marina Papatriantafylou

Department of Computer Science and Engineering  
Chalmers University of Technology  
SE4412 96 Göteborg  
Sweden  
Telephone + 46 (0)314772 1000

## **Abstract**

Simulation, graphic design and other applications with high graphic processing needs have been taking advantage of high performance computing systems in order to deal with complex computations and massive volumes of data. These systems are usually built on top of a single operating system and rely on virtualization in order to run applications compiled for different ones. However graphics processing on virtual applications has performance and capability problems that are accentuated when these applications are run remotely. Therefore combination of virtualization and remote execution may produce important yield loss in graphics processing which is inappropriate for high performance computing. Furthermore this overhead can also produce big delays in screen updates which cannot be accepted in interactive applications. System designers need to choose the most appropriate architecture in order to gain the desired behavior and performance for their applications. Thus, this thesis performs an analysis on graphics processing performance with different operating systems, virtualization and remote desktop technologies that reveals their bottlenecks and limitations when working together. Moreover it analyses which technologies are more suitable for different scenarios based in applications needs and architecture constraints.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Short problem statement . . . . .	3
1.3	Goal . . . . .	3
1.4	Limitations . . . . .	4
1.5	Description of remaining sections . . . . .	4
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Virtualization . . . . .	5
2.2	GPU Virtualization . . . . .	7
2.2.1	GPU virtualization problem . . . . .	8
2.2.2	GPU virtualization analysis . . . . .	8
2.3	Remote 3D rendering . . . . .	10
2.4	State of the art . . . . .	14
2.4.1	3D graphics on Virtualization . . . . .	14
2.4.2	3D graphics on Remote Desktop Protocols . . . . .	17
2.5	Previous and related work . . . . .	18
<b>3</b>	<b>Method</b>	<b>20</b>
3.1	Problem Analysis . . . . .	20
3.2	Technology selection . . . . .	21
3.2.1	Virtualization technologies and virtual GPUs . . . . .	21
3.2.2	Remote display protocols . . . . .	22
3.3	Evaluation criteria . . . . .	24
3.4	Test methodology . . . . .	28
3.5	Benchmarking . . . . .	29
<b>4</b>	<b>Case studies</b>	<b>32</b>
4.1	Introduction . . . . .	32
4.1.1	Test systems . . . . .	32

4.2	Baseline . . . . .	33
4.2.1	Linux . . . . .	33
4.2.2	Windows . . . . .	34
4.2.3	Comparison between Windows and Linux results . . . . .	35
4.3	WINE . . . . .	37
4.4	Hosted hypervisors on Linux . . . . .	39
4.5	XEN . . . . .	41
4.5.1	Quadro 600 tests . . . . .	41
4.5.2	Quadro 4000 tests . . . . .	43
4.6	Kernel based Virtual Machine . . . . .	45
4.6.1	Quadro 600 tests . . . . .	46
4.6.2	Quadro 4000 tests . . . . .	46
4.7	Virtual machine deployment tool . . . . .	47
4.8	VMware ESXi . . . . .	49
4.8.1	Software graphics GPU and API remotng GPU . . . . .	49
4.8.2	PCI pass-through and virtual Dedicated Graphics Acceleration . .	50
<b>5</b>	<b>Conclusion</b>	<b>53</b>
5.1	Future work . . . . .	56
	<b>Appendices</b>	<b>57</b>
<b>A</b>	<b>VM deployment tool pseudo-code</b>	<b>58</b>
A.1	run vm . . . . .	58
A.2	start vm . . . . .	58
A.3	destroy vm . . . . .	63
<b>B</b>	<b>Test Results</b>	<b>65</b>
B.1	Linux . . . . .	65
B.2	Windows . . . . .	66
B.3	WINE . . . . .	67
B.4	Virtual machine monitors . . . . .	68
B.5	XEN . . . . .	69
B.6	KVM . . . . .	70
B.7	VMware ESX . . . . .	71
	<b>Bibliography</b>	<b>77</b>

# 1

## Introduction

### 1.1 Motivation

Over the past decade, organizations and companies have abruptly increased their computation requirements either to process massive volumes of data generated by their applications or to resolve complex calculations and simulations. This phenomena, also known as the “Big Data” problem, requires powerful and expensive equipment in order to process all data in reasonable time. To overcome this issue, High Performance Computing (HPC) providers offer on-demand supercomputing power which saves huge costs of capital and equipment maintenance to their customers.

HPC systems are composed of hundreds or thousands of computation nodes usually built on top of a single operating system (OS) in order to save maintenance costs. However there are multiple applications that are only available (or vendor supported) for a single operating system which may be different than the one deployed in the HPC cluster. A simple solution to support an application from a different OS is to install and maintain also the new required OS. However it is not desirable at all to maintain a new operating system alongside the existing infrastructure. Besides the extra maintenance costs (extra equipment, services, licences, specialist workers ...) and the desire of maintaining an uniform cluster (security, available resources, maintainability ...), the new OS may lack important features or be unable to take advantage of the main OS resources. Moreover the number of non supported applications may be minimum till some exceptions.

Within this context, open source Linux-based operating systems have become popular to host HPC systems due to its zero prize, large ecosystem, good hardware support, good performance and reliability. Moreover Linux provides support for parallel file systems and computing infrastructures that are common in HPC but may not be supported

by other operating systems like Windows, Solaris or other Unix like OSes. On the other hand, Windows is the most commonly used operating system for desktop environments and there are some applications that need to take advantage of HPC that only work in this operating system. For simplicity sake, from now on Linux is taken as reference of host OS for HPC systems whereas Windows applications are the ones which need to be introduced in the HPC ecosystem. However the same reasoning is valid for other OS combinations.

Windows applications cannot run directly on Linux. These OSes provide different system libraries, APIs, system calls etc. which are not compatible between each other. However these applications may work in Linux adding an abstraction layer between Linux OS and Windows applications that mimics Windows runtime environment. This layer is provided by a technology called *virtualization* that emulates in software the behavior of different computer resources like CPUs or system libraries. This technology is available for multiple OSes, in fact Linux applications may also run in Windows in the same fashion. The main problem with virtualization is that it may introduce an important overhead and lack some features of the original OS.

Traditionally HPC services consisted in the scheduling of jobs that ran in the background in the underlying HPC infrastructure. Nonetheless graphical applications also started to take advantage of this service allowing users to interact with their applications remotely. Nowadays typical HPC users are scientists and engineers in fields such as bio-sciences, energy exploration and mechanical design [41] who use CAD/CAM/CAE (Computer Aided Design/Manufacturing/Engineering) applications as part of their daily work-flow. These applications are meant to create three dimensional, i.e. *3D*, models and simulations. 3D models are composed of simpler ones in a recursive manner till be reduced to basic 3D objects or surfaces like triangles or squares. In the end a 3D model is composed of millions of simple 3D objects that during simulations suffer 3D operations (operations performed over a 3D object in a 3D space) like translations (move an object from one position to another) or rotations (rotate an object a certain angle within a reference edge). In order to process all these objects in real time, there exists a special hardware device called *graphics processing unit* (GPU), composed of millions of parallel CPU cores able to perform 3D operations. This way a GPU can process millions of 3D operations simultaneously. CAD applications rely on heavily 3D computing in order to perform simulations over 3D models and they only achieve acceptable performance with *hardware-accelerated 3D graphics* i.e. if 3D operations are processed in hardware by a 3D graphics accelerator or GPU.

CAD applications like AutoCAD, Solid Edge, Femap and Hexagon among others only have a Windows version. Thus in order to run these applications in a HPC system built on Linux, the following challenges must be addressed: (a) running Windows applications on Linux, (b) provide hardware accelerated 3D graphics to virtual applications and (c) interact with these applications remotely in real time. Virtualization, 3D virtual-

ization aware drivers and remote desktop protocols respectively resolve these problems. However the integration of such heterogeneous technologies in the same system may severely degrade its performance and bring important drawbacks. Here virtualization plays a major role because all other technologies in some way rely on it and it is also the one that brings more constraints to the system. On the other hand support of 3D graphics on virtual applications is still experimental and may crash or cause damage in the underlying system in some cases. Moreover, remote desktop protocols heavily rely on the OS to provide hardware accelerated 3D graphics and may produce big overheads, accentuated due to virtualization, in applications that produce multiple screen updates per second. Thus, organizations that want to take advantage of HPC in such environment have been constrained by their inability to run remote 3D accelerated applications on virtual environments.

## 1.2 Short problem statement

Running virtual applications with 3D graphics and remotely is a complex task that involves the integration of multiple heterogeneous technologies. From the application that issues a 3D call to the final user that sees the rendered 3D frame, the control flow goes through the virtualization layer, one or more 3D drivers, the operating system and the remote visualization application, which may also require extra technologies to process 3D calls. Each one of these technologies adds new constraints and limitations making the resulting system difficult to study and analyze. Moreover HPC systems require maximum performance due to small overheads have a multiplier effect with enormous amounts of data or operations. Further, users require instant feedback from interactive applications. However both remote desktop applications [9] and virtualization [17] are known to produce performance overheads due to the extra computations required to overcome their tasks. Further, this overhead is accentuated with 3D graphics because extra layers are needed in order to mediate accesses to the GPU between virtual applications and remote desktop applications.

This thesis deals with the problems of running remote virtual applications with 3D graphics and performs an in-depth analysis of the technologies required for providing this service. Further the problems of running 3D graphics both in virtualization and in remote desktop applications are investigated in order to find their bottlenecks and limitations.

## 1.3 Goal

This thesis investigates how to build a system able to run virtual applications with 3D graphics remotely. In order to accomplish this task, it is necessary to study what technologies can be used, what are their constraints and limitations and how they can be combined with each other. Other questions which should be answered are how much



overhead is introduced by each technology and what are the consequences of maintaining such setup.

To support the analysis of such system, an experimental study on Windows applications running on Unix-like OSes (Linux and VMware) is carried out. This study is composed of several case studies which test different technology combinations which are analysed and compared with each other. It is focused on HPC applications so the main evaluation criteria in case studies is graphics performance, i.e. the speed graphics operations are processed, although other factors are also taken into account. This work also aims to be a guideline for system engineers and developers to deploy and research respectively on providing an efficient and high quality infrastructure in this area. Moreover this thesis explores the boundaries and limitations of the actual technology and tries to address new research lines in this field.

## 1.4 Limitations

There are some constraints which bound this work. The most relevant ones are due to hardware resources and time limits. Time constraints the number of technologies that can be studied and the precision of the conclusions obtained from them. Thus this thesis presents more detailed analysis on the most representative and evolved technologies whereas others may be briefly mentioned and left for future research. On the other hand, hardware constraints what architectures, devices and technology can be included in case studies.

This thesis tests several technologies that are in an unstable state or beta version so failures or software bugs are expected to appear. Upon such events occur, this thesis makes an effort to work around these problems within time limits. Thus for known issues, existing or experimental workarounds are applied whereas unknown ones are reported to their developers in order to fix them.

Technologies that require licensing and do not provide a free or trial period are not included in case studies.

## 1.5 Description of remaining sections

Section 2 introduces and analyses technologies needed to setup a system able to run remote virtual applications with 3D graphics and studies their state of the art. Section 3 selects relevant implementations of technologies introduced on section 2, explains how they can be evaluated and introduces how case studies are performed. Section 4 presents case studies and discusses their results. Finally section 5 summarizes this thesis and presents final conclusions.

# 2

## Background

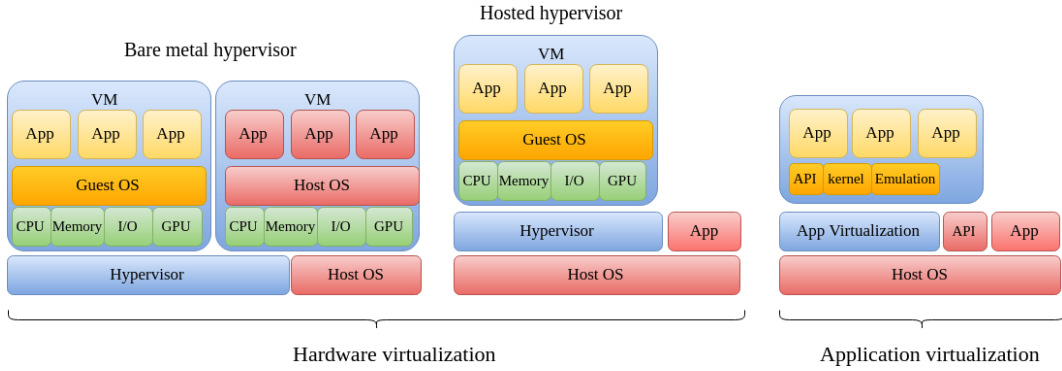
This section investigates how a system able to run virtual applications remotely with 3D graphics can be built. It starts analysing how applications compiled for one operating system can be run on another operating system and take advantage of the GPU in order to process 3D graphics. After that, section 2.3 studies ways of performing 3D calls when running applications remotely. For simplicity these sections refer to Linux as host OS and Windows as target application's OS however this analysis can be applied to other OSes. Finally the state of the art of virtualization and remote desktop applications is reviewed in section 2.4 where all introduced technologies are bound together. At the end of the chapter, related work in this matter is presented and discussed.

### 2.1 Virtualization

Regular applications cannot usually run in different operating systems if they weren't specifically designed for it. This is due to operating systems have specific application programming interfaces (APIs) and different system calls, systems libraries, protocols, file paths etc. Moreover libraries with the same API may have divergent behaviors even among versions of the same OS. Using cross-platform programming environments like Java and avoiding any OS specific feature allows applications to be compiled for different OSes. However sometimes this may not be possible or too expensive. In these cases, virtualization is needed in order to run a compiled application in a different OS. However this technology may introduce some limitations and make the application lack features of the original one. This section introduces some virtualization basics needed to follow this thesis.

In order to run a Windows application on Linux, an abstraction layer (the virtualization layer) which mimics Windows execution environment is introduced between the application and Linux OS. There are different kinds of virtualization depending on where

this virtualization layer is introduced and what computer resources are emulated. For example, Java Virtual Machine operates at the application level creating an application run-time environment independent from the underlying OS while a hardware virtual machine emulates hardware resources that can be controlled by another operating system. This thesis focuses on *hardware virtualization* and *application level virtualization* because they are the only virtualization types that support running unmodified applications among different OSes. Figure 2.1 shows hardware and application virtualization architecture.



**Figure 2.1:** Hardware and application virtualization architecture

With hardware virtualization, hardware resources like a CPU, memory or a GPU are emulated and an unmodified instance of an operating system or *guest OS* runs on top of them. Guest accesses to real hardware resources are mediated by a special application called *virtual machine monitor* (VMM) or *hypervisor*. If the hypervisor runs on directly on hardware or in the host OS kernel it is called *bare metal hypervisor* however if it runs as a regular application on the host OS then it is known as *hosted hypervisor*. Bare metal hypervisors provide their own drivers and control CPU and memory directly thus providing good performance and advanced features to virtual machines. Further usually no application runs directly on the host thus providing better security and high availability. On the other side, hosted hypervisors are better integrated with other host applications as they run as one. However they rely on the host to access and control hardware limiting features and performance for virtual machines.

There are two main classes of hardware virtualization: *paravirtualization* and *hardware assisted virtualization*. In paravirtualization, the guest OS is adapted for virtualization and no special hardware is needed. Here, the guest communicates with the host through an special API avoiding the need of wrapping accesses to real resources. Due to guest OS needs to be modified, not all OSes can take advantage of paravirtualization, for example there is no paravirtual version of Windows whereas Linux has several ones. One of the most extended paravirtual hypervisors is XEN [5] and it is currently

supported for Linux and FreeBSD as host OSes. On the other hand, hardware assisted virtualization relies in hardware extensions to perform virtualization and guest OSes do not have to be modified. This technology relies in hardware to intercept guest's privileged calls and send them to the hypervisor that mediates access to resources, generating some overhead of system calls. Some of this overhead is reduced with paravirtual drivers (virtualization aware drivers) installed in the guest but, in general, paravirtual OSes provide slightly better performance because they are customized for virtualization. The most well known open source solutions are KVM (Kernel based Virtual Machine) [38] which is only available for Linux and XEN which can also take advantage of hardware extensions. Other hardware virtualization solutions are HyperV that works on Windows or VMware ESX which is itself an OS.

Hardware virtualization is known to produce some performance overhead [17] [40] [42] compared with a system running directly on hardware so performance of guest OS should be upper limited by the performance of the native implementation i.e the OS runs directly on hardware. However under certain circumstances, an application may perform better in a virtual environment than in a native one. This phenomena appears when for example guest drivers are much slower than host drivers (for example, Windows drivers are way slower than Linux drivers). In this case, the guest OS instead of using its slow driver, relies on a fast virtual device to communicate with the host OS which is the one that access the real device through its fast driver.

Application virtualization relies on emulation and on alternative implementations of system libraries to execute applications from different OSes. This approach requires less resources because it does not need to run another complete operating system. However it is more complex to implement because while hardware virtualization emulates resources with easier and well known APIs like a CPU or a memory controller, application level virtualization has to emulate the behavior and implement the libraries of a complete OS. Thus if it is not perfectly done applications may fail or produce unexpected behaviors. Further, extra layers added to wrap or emulate system libraries may produce important performance leaks. The most well known Windows application virtualization solution is WINE [7], and it works on Linux, FreeBSD, MAC and Solaris.

## 2.2 GPU Virtualization

Section 2.1 introduced hardware virtualization and application level virtualization and some of their advantages and disadvantages. It also mentioned that, in general, it is easier to emulate simple hardware resources than system libraries however regarding 3D graphics this is not true. While application virtualization only needs to redirect (or pass-through) *OpenGL* calls (standard API for 3D graphics) or emulate *DirectX* calls (specific Windows API for 3D graphics), hardware virtualization needs to provide to the guest OS a virtual GPU, which is way more complex. This section explains the problems of virtualizing the GPU and analyses different technologies that deal with them.

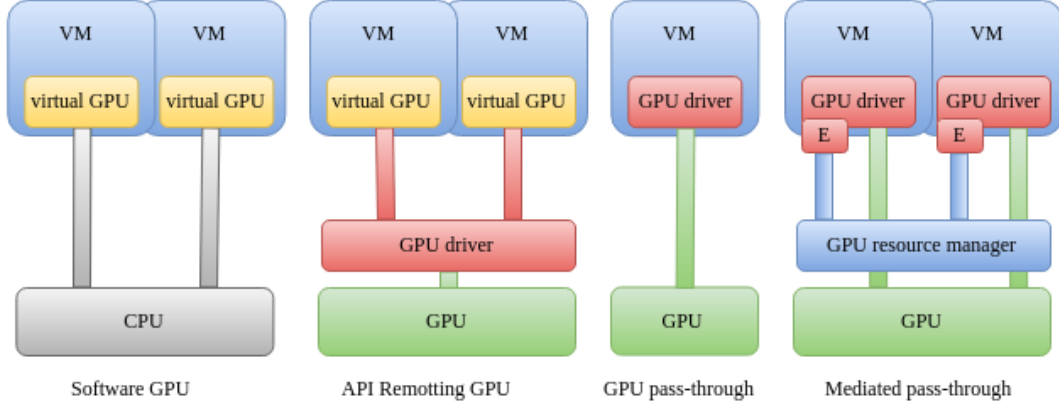
### 2.2.1 GPU virtualization problem

I/O virtualization has always been a big challenge for virtual environments because guest I/O drivers have to be modified or need special hardware support in order to isolate accesses among virtual machines. Therefore these drivers may have to run with several limitations (for example remove privileged functions or work with limited bandwidth) and could create security problems [36]. Moreover the display system (the GPU) compared to other I/O subsystems is really hard to multiplex in a way that is both efficient and safe. This is evidenced by the fact that all major operating systems provide ‘direct’ avenues of programming the graphics card i.e. *direct rendering*, largely without operating system involvement, but at the danger of being able to crash the graphics adapter or lock up the entire machine [15]. Furthermore GPUs present a challenging mixture of broad complexity, high performance, rapid change and limited documentation for a number of reasons: (a) GPU hardware interface is proprietary and non public, (b) there is no standard GPU interface like IDE or SCSI in disk drives and (c) GPU drivers usually come in a closed-source device driver and it is unusable for more than one operating system.

### 2.2.2 GPU virtualization analysis

Regarding performance, while the CPU overhead generated by the virtualization layer was minimized until close to native performance [1], I/O virtualization is still a hot topic between researchers. Specifically, GPU virtualization is quite complex and brings several performance and feature set constraints although, virtual GPUs may also provide some virtualization related features. Both limitations and added features depend mainly on whether the GPU driver runs on the host OS i.e. *front end virtualization* or on the guest OS i.e. *back end virtualization*. Figure 2.2 shows the different types of virtual GPUs where the GPU driver is colored in red.

In front end virtualization, access to the physical GPU is entirely mediated through GPU vendor provided APIs and drivers on the host while the guest only interacts with software. This solution’s performance relies highly in guest’s driver implementation and there are different techniques on a continuum between two extremes : API remoting and device emulation. API remoting GPUs blindly forward 3D API calls from the guest to the external graphics stack via remote procedure call. However with device emulation a virtual GPU is emulated in software (software GPU) and the emulation processes graphics operations in response to actions by the guest device drivers. It is clear that emulated GPUs will not perform well because 3D calls are processed by the CPU instead of the GPU. Thus it is possible that some applications do not work at all with these virtual GPUs. On the other hand, API Remoting may achieve good performance because 3D operations are processed by the GPU. However the extra processing made by the virtualization layer and the exchange of 3D data and commands between both operating systems (guest and host) increase the CPU load and create delays in the application.



**Figure 2.2:** virtual GPU types

Thus the overhead produced by this technology may be bigger than the one created by application virtualization but still way better than software GPUs. There are several implementations of both approaches like VMware SVGA software GPU which works mainly in VMware products or VMGL (API remoting GPU) that works in XEN and KVM. Both of them are introduced in section 2.4.

On the other hand, the GPU driver can run in the guest OS instead of the host OS. This way, since the VM interacts directly with hardware resources, its execution state is bound to the specific GPU vendor (possibly the exact GPU model in use). This technique is called *PCI pass-through* and consists in the permanent association of a virtual machine with full exclusive access to a physical PCI (Peripheral Component Interconnect) device i.e. a device like a GPU attached to the virtual machine. Thus PCI pass-through performance is really high and close to native OS one (performance in the guest OS running directly on hardware) due to the native driver is the one that controls the device. Thus graphic applications performance may be limited by other factors like CPU, memory or disk I/O which introduce certain virtualization overhead rather than PCI pass-through itself. However this technology has a big inconvenience and it is that only one virtual machine can take advantage of a GPU at a time. Another factor should be taken into account is that the GPU driver may have a different versions for guest and host OS. Thus GPU performance, features or behavior may differ between guest and host OS. Therefore it is possible that some applications work better in the virtual OS than in the host one. This way it is expected that this technique performs close to the native guest OS and differs in some way with the host OS.

There is one extension of PCI pass-through called *mediated pass-through*. GPUs support multiple application independent contexts and mediated pass-through proposes dedicating just a context, or set of contexts, to a virtual machine rather than an entire GPU. In this approach, high-bandwidth operations (command buffer submission, ver-

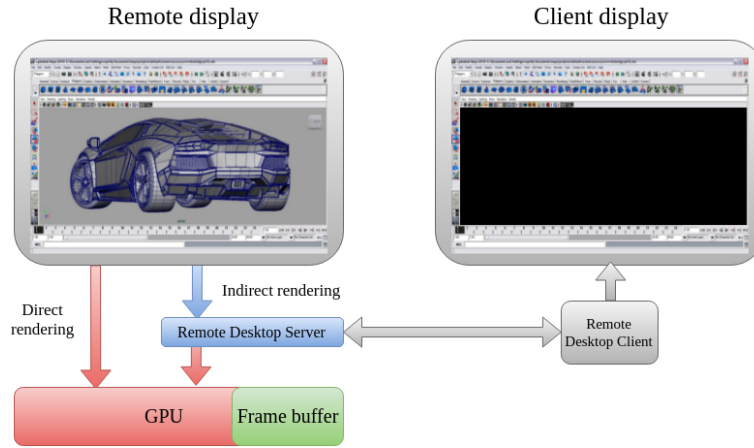
tex and texture direct memory access (DMA)) are performed using memory resources mapped directly to the physical GPU, thus these operations perform at a similar speed as simple pass-through. On the other hand, low-bandwidth operations (resource allocation, legacy features) are implemented using software virtualization with almost no performance penalty. This allows that multiple virtual machines can work directly with the same physical GPU, but incurs additional costs: (a) the GPU hardware must implement multiple isolated contexts in a way that they can be mapped to different virtual machines efficiently and securely, (b) the host/hypervisor driver must allocate and manage GPU resources such as memory and (c) contexts and logical GPUs which appear in each VM may not have the same hardware interface which would be exposed by an equivalent physical GPU. This means that mediated pass-through may require changes into guest device drivers. There are some examples of GPUs that support mediated pass-through like Nvidia Grid GPU family where different Nvidia drivers run in the guest and in the host. There are also some Intel integrated cards that can be passed to multiple XEN VMs.

It is clear that the best solution for performance sake is simple PCI pass-through but at the cost of dedicating one GPU per virtual machine. Mediated pass-through solves this problem by sharing a GPU with multiple virtual machines. However its complexity makes that only some expensive GPUs can support this feature and they may carry several limitations like low memory limits per virtual machine. On the other hand, API remotoring techniques solve the problem of sharing the GPU between virtual machines without requiring special hardware but with some performance overhead due to extra software layers and computations. Finally, pure software GPUs will probably fail to achieve enough 3D graphics performance for most 3D applications. Besides performance and *multiplexing* (the number of users/instances that can take advantage of a GPU) there are other virtual GPU features that may be important to take into account. Here an important feature is the OpenGL version supported by the GPU because multiple applications require specific OpenGL versions to work. This problem appears in front end virtualization where GPU features are implemented in software whereas with back end virtualization few features may be lost as the driver that controls the GPU is the original one or a modified version. Other interesting features which may be important are the ones added by the virtualization layer. The most important feature regarding GPU virtualization is the support of virtual machine live migrations (move a VM from one host to another without shutting the VM down) that is only supported by software GPUs due to pass-through ones are bound to the physical assigned GPU.

## 2.3 Remote 3D rendering

HPC systems are hosted in computer clusters so in order to interact with them graphically it is necessary to use *remote desktop software* which allows an user to control a computer remotely. Most well known general purpose remote desktop protocols are *remote desktop protocol* (RDP) and *virtual network computing* (VNC). Both have imple-

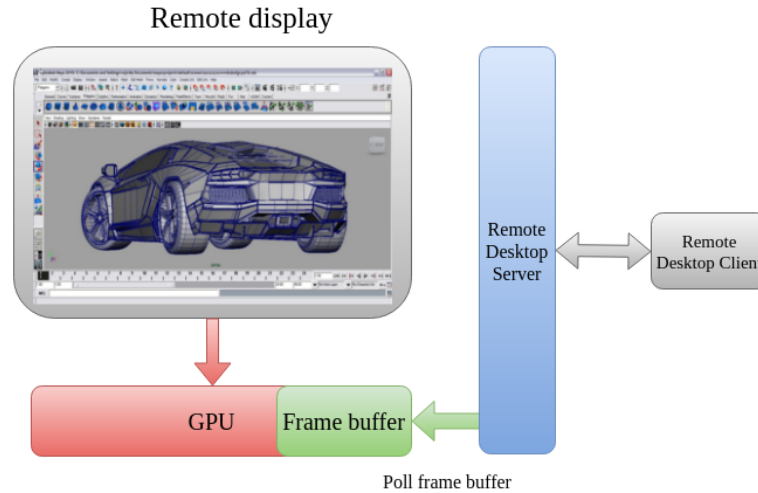
mentations, also known as *remote desktop applications* for multiple OSes. However not all of them provide 3D graphics to applications running through them. This happens because the only display (or session) that supports 3D graphics is the display directly connected to the GPU, called *local display* or local session. Thus virtual displays created by RDP and VNC cannot issue 3D calls to the GPU. The following paragraphs study which implementations support 3D graphics and techniques for providing 3D graphics for the others if possible.



**Figure 2.3:** 2D remote desktop application

Remote desktop applications rely on the OS *windowing system* in order to get the screen image that should be sent to the client. These applications monitor the virtual display for events (such as window expose events) that might cause the pixels to change. As such events occur, they read back affected regions of the display, compress them and send compressed images to all connected clients. This is the approach used by most 2D remote display packages like Windows RDP implementation and multiple Unix VNC ones. Due to only the local display can take advantage of 3D graphics, a simple solution is to monitor this display instead of creating a virtual one. However this is not enough for providing 3D graphics. The reason is that 3D applications use direct rendering to send OpenGL commands directly to the 3D hardware bypassing the windowing system. OpenGL rendered pixels go straight to the *frame buffer*, the buffer with the display pixel information, so neither the windowing system nor the remote desktop application knows when a 3D application has finished rendering a frame. This can be seen in figure 2.3 where a remote desktop client cannot see the rendered car image due to it has been rendered directly into the frame buffer. This problem can be solved following two different approaches, (a) polling the frame buffer to check for screen updates (*screen scraping*) and (b) wrapping 3D APIs and redirecting 3D calls to the main display (*API interception*).

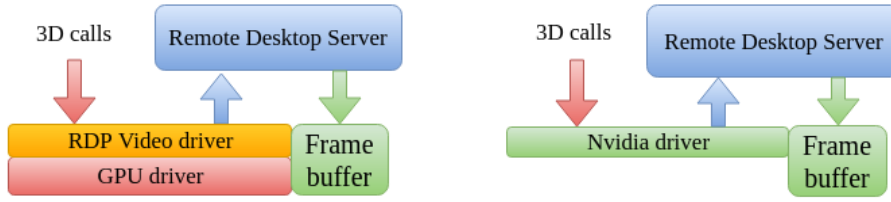


**Figure 2.4:** Screen scraping

Windows VNC implementation and x11VNC in Linux follow the *screen scraping* approach shown in figure 2.4. They asynchronously read back the entire framebuffer on a periodic basis, compare the current screen snapshot against the last, and send differences to all connected clients. This configuration has several drawbacks:

- It consumes a lot of CPU polling the framebuffer in order to get constant screen updates which is required in interactive applications
- It does not get instant updates and it is likely to produce lags in user interaction. Once a 3D image is rendered, it may take some time for the polling processes to query the frame buffer. Further in a contention scenario a polling process is likely to consume its CPU time fast and be constantly enqueued. This produces lags in user interaction.
- Only one user can interact with a work station at a time

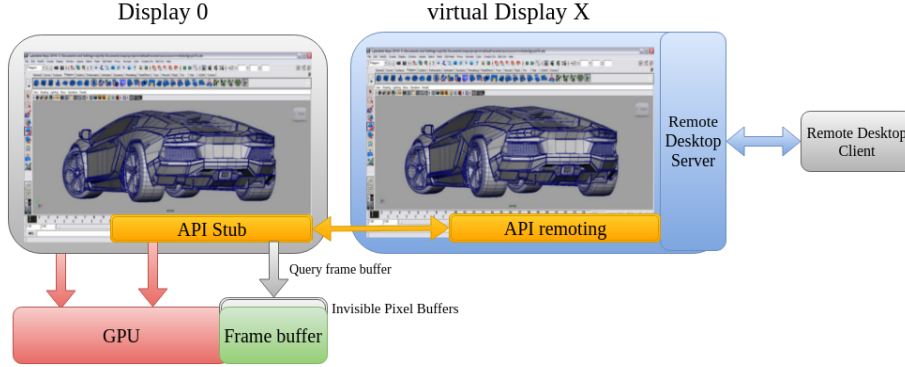
An improvement to this solution that avoids polling constantly the framebuffer is to monitor the application for certain 3D function calls that indicate that the application has finished drawing a frame. Then a read back trigger is sent, along with coordinates of the region to be read back, to the screen scraper. There are several Windows VNC implementations with a special driver which provides this functionality and considerably improve user experience with more updates per second and less CPU consumption. However this is also at the expense of some graphic performance due to extra required layers for wrapping rendering calls. A further improvement for this technique is getting video streaming updates directly from the GPU that notifies the screen scraper. An example of this is VMware PCoIP (PC over IP) that uses Nvidia proprietary APIs to retrieve the video signal. The architecture of both improvements is shown in figure 2.5.



**Figure 2.5:** Screen scraping using a GPU driver

On the other hand, implementations that follow *API interception* approach intercept 3D calls, redirect them to 3D hardware and compose the final image that is sent to the client. Besides OpenGL, windowing systems usually provide special APIs for integrating OpenGL calls with windowing system calls like *GLX* (OpenGL extension to the X windows system) , *WGL* (OpenGL extensions for Windows windowing system) or *CGL* (core OpenGL in OS X). Thus 3D wrappers should intercept these APIs too. Using windowing system APIs instead of OpenGL directly is called *indirect rendering* as 3D calls go through the windowing system before going to the 3D hardware. API interception has a similar architecture than the XVNC (3D VNC implementation for Linux) server in which each user has an independent virtual display but in this case, XVNC supports also GLX extensions. In this case XVNC wraps GLX calls issued by applications and re-routes them inside GLX commands to Linux windowing system which then executes OpenGL commands directly in the graphic card. At the end of every frame, the X proxy reads back rendered images directly from the 3D hardware and composites them into the appropriate window. The main problem of this solution besides the overhead of wrapping all 3D calls is that the XVNC has to handle OpenGL commands and extensions (probably also GPU vendor specific extensions). Thus it has to be updated when OpenGL API changes, that in fact usually happens more quickly than other 3D APIs like GLX. This may be one of the reasons why there is no free version of RDP or VNC on Windows that use this technology and there is no good support for 3D XVNC.

There is an independent technology called VirtualGL [10] that follows a similar approach than API interception for providing 3D graphics for remote desktop applications. VirtualGL consists of a wrapper that intercepts GLX commands that create OpenGL contexts and ensures that these contexts are allocated in the 3D server. These OpenGL contexts are rendered in *PBuffers* (invisible Pixel buffers) instead of windows so a mapping between PBuffers and windows is needed. However this is more simple than maintaining a complete OpenGL implementation. Once an OpenGL context is established, the application is free to issue OpenGL commands directly to the graphic card. Thus after a frame is rendered (detected after certain calls as `glXSwapBuffers()` or `glFinish()`) the GLX wrapper reads back the rendered pixels and displays them into the appropriate window using standard image drawing commands. Combining this technique with a remote desktop application, multiple client sessions can enjoy 3D graphics simultaneously. However this technology only works with GLX extensions so it only



**Figure 2.6:** 3D graphics with API remoting

works with the X windowing system, present by default in Unix-like OSes. There are also implementations of the X windowing system for other OSes like Windows but applications have to be modified to work with X windowing system instead of Windows native one.

## 2.4 State of the art

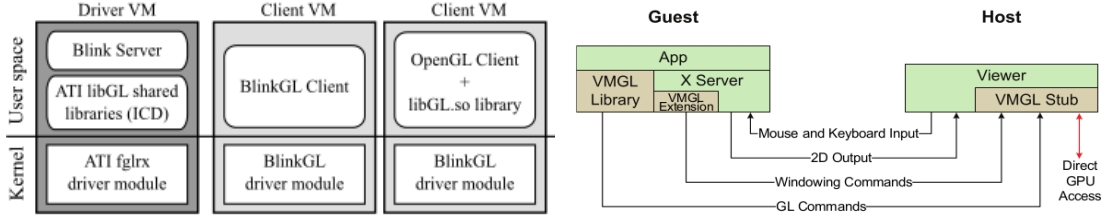
Sections 2.2 and 2.3 have introduced issues encountered when processing 3D graphics in virtualization and remote desktop protocols respectively as well as available techniques which deal with those problems. This section takes a look at the state of the art to find actual technology implementations which use these techniques.

### 2.4.1 3D graphics on Virtualization

Virtualization and GPU virtualization technologies need to be bound together in order to compose systems able to provide 3D graphics to virtual applications. Moreover windowing systems are also taken into account because some GPU virtualization techniques rely on them to function. This limits which operating systems are compatible too because an OS usually only supports a single windowing system.

There have been several research projects that have worked on providing 3D graphics for virtual machines. Blink [20] and VMGL [27] use a Chromium-like [24] approach to redirect OpenGL calls to the GPU in Linux and other UNIX-like guests. Blink implements an OpenGL wrapper for each VM which communicates via shared memory to a server running on top of XEN that mediates accesses to the proprietary GPU driver. On the other hand VMGL similarly achieves 3D graphics but sending OpenGL commands via loop-back network to a VMGL daemon running in an independent process per VM. This way mediation and address space conflicts are handled by the GPU driver instead of VMGL itself. Loop-back communications are slower than shared memory ones due to

TCP/IP stack overhead so Blink is expected to perform better. However this architecture design makes VMGL cross-platform (it works in XEN, VMware or KVM) whereas Blink is only available for XEN. Unfortunately, both projects work only with X11 servers (server for X windowing system) so Windows OS for example is not supported.



**Figure 2.7:** Blink (left) vs VMGL (right) approach

While these solutions use API interception to redirect OpenGL calls, Dowty et al. [14] from VMware propose an emulated virtual GPU (software 3D) named SVGA with support for basic 3D graphics (3D Windowing System and basic 3D applications). This virtual GPU is also based in Gallium3D [32] architecture, showed in figure 2.8, and therefore it works with any virtualization platform with a Gallium3D back-end architecture like the ones supported in Linux (XEN or KVM). The main advantage of SVGA is that it can achieve basic 3D graphics without hardware support, being useful in the absence of a graphic card. However its graphic performance is quite low. More recently in 2012, VMware released virtual Shared Graphic Acceleration (vSGA) [6] vGPU using API interception for both Direct3D and OpenGL. This virtual GPU is based on a special driver developed by Nvidia for VMware hypervisor that mediates accesses between virtual machines. Thus this implementation requires VMware proprietary software and Nvidia cards.

On the counterpart, VirtualBox is a free open source virtual machine monitor with implementations for multiple OSes. It provides 3D graphics both via API interception and PCI pass-through. In the API interception approach, the remote 3D application may run both inside the guest OS or in the host OS because guest OS graphics are projected in the host OS through VirtualBox. In this case 3D graphics are processed using direct rendering. On the other hand support for PCI pass-through requires running the remote desktop application in the guest because the host has no control at all of OpenGL calls issued to the pass-through GPU. However this technology is quite experimental and it is in an early development state that may only work with basic PCI devices such as USBs or network cards.

There are other proposals that aim to provide 3D acceleration for KVM virtual machines. Virgil3D [3], still under development, implements a Gallium3D-like driver to support OpenGL hardware acceleration and eventually Direct3D on KVM virtual machines. Also spice project [21] is working in a new WinQXL [13] driver but is still far

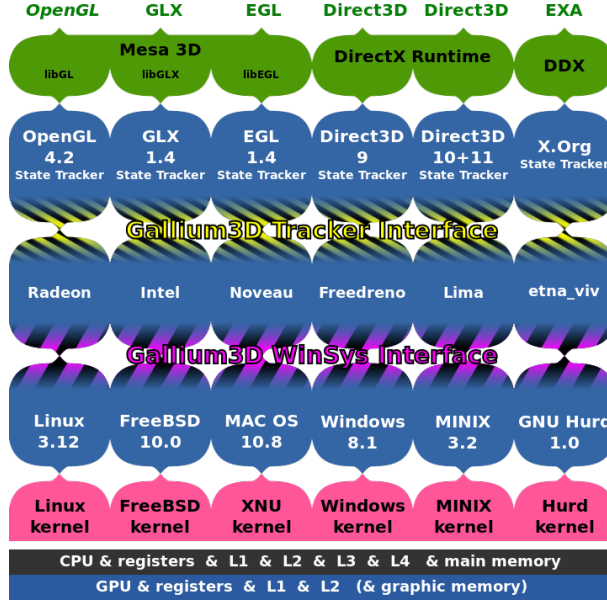


Figure 2.8: Gallium3D architecture

from getting a working version that supports 3D graphics.

All previous solutions use front-end virtualization however there are also some back-end virtualization approaches that can provide direct access to the GPU with some hardware support. VMware, KVM and XEN support GPU pass-through. In KVM there are two drivers that provide this functionality, the original driver for PCI pass-through named *pci-stub* and a more advanced feature called *VFIO* [43] that takes advantage of new MMIO (Memory Mapped IO) hardware functions. However this last driver is still experimental for GPUs and only works with some graphic cards and with new versions of the Linux kernel (>3.9). Meanwhile in XEN, VGA pass-through has been available since XEN 4.0 release in 2010 so it is more mature than in KVM but only high end graphic cards are supported. Furthermore XEN recently has released a experimental technology called XenGT [19] capable for mediated pass-through with at most 4 virtual machines accessing a single Intel integrated GPU. XenGT allows all virtual machines native pass-through to the graphic card trapping and emulating only privileged operations. However this technology only works with Intel integrated graphic cards that usually provide poor 3D graphics performance (compared to dedicated graphics cards).

A mediated pass-through implementation has been proposed by Nvidia [22] which introduces the GRID technology. It consists on a hardware virtual GPU where the GPU itself implements a Memory Management Unit (MMU) which maps guest's virtual addresses to physical ones and isolates each host on its own address space. This allows

each virtual machine to work as it had an assigned hardware GPU. Further GPU resources mediation is handled completely in hardware by the GPU itself and not by the hypervisor. Unfortunately this technology is currently only supported for proprietary solutions like VMware or Windows Hyper-V (Windows hypervisor).

Another approach for running virtual applications is using a program that translates OS specific calls into another OS ones. Regarding graphic APIs, there are only two main ones: OpenGL which is standard for almost all OSes and DirectX which is Windows specific. DirectX library is required to run any Windows application that works with this API like many games. WINE provides DirectX support translating DirectX to OpenGL.

Finally, there are some software renders like SwiftShader [39] especially designed for computers with no graphic hardware that with enough CPU power may provide basic 3D performance.

### 2.4.2 3D graphics on Remote Desktop Protocols

As it was explained in section 2.3, remote desktop protocols also rely on the OS windowing system for providing 3D graphics. There are two main windowing systems which have implementations from almost all remote desktop protocols which are the X Windowing System available in almost all Unix like OSes and the Desktop Window Manager used in Windows OS. Many different remote desktop protocols support these windowing systems so here only the ones most widely used and related to the technologies used in this project are mentioned.

- X forwarding: this protocol is only available for the X Windowing System and consists in forwarding X applications windows across the network. One of the implementations of this protocol is through a SSH (Secure Shell) remote connection which encapsulates the X forwarding session. This protocol creates a virtual session and uses API remoting through VirtualGL in order to process 3D graphics.
- VNC (Virtual Network computing): this is a open source and multi-platform remote desktop protocol that uses the Remote Frame Buffer protocol (RFB) to remotely control another computer. It has many implementations for both windowing systems like turboVNC in X windows, ultraVNC in Windows or tightVNC which works in both of them. All Windows implementations use screen scraping technique while Linux has implementations using both techniques like turboVNC with API remoting (virtualGL) and X11VNC with screen scraping.
- RDP (Remote Desktop Protocol): it is a proprietary protocol developed by Microsoft, which provides a Windows virtual desktop independent of the local user session. However there are also open source implementations and servers for both Windows and Linux. This protocol is more advanced than VNC and provides more features like file transfers and sound. Windows provides an official RDP version for Windows desktops however it does not support 3D graphics. There are other free

implementations like XRDP which work with X windows and provide 3D graphics through VirtualGL.

- NX: This technology optimizes and encapsulates in a SSH session a remote desktop connection in different operating systems. In Unix it handles remote X windows connections while in Windows it encapsulates Windows terminal sessions (RDP protocol). Thus its support for 3D graphics relies on the protocols used underneath.
- Virtualization specific protocols: several virtualization platforms provide a specific remote desktop protocols like ICA protocol in Citrix-XEN, PCoIP in VMware or SPICE (Simple Protocol for Independent Computing Environments) in KVM and XEN. These protocols are meant for connecting to Virtual Desktops (Virtual Machines with a Desktop OS installed) and have implementations for both Windows and Linux. However they only work in their respective virtual platforms and do not work in a non virtual desktop. By now SPICE does not provide yet 3D graphics however ICA and PCoIP do.

## 2.5 Previous and related work

This section introduces the most relevant works related to this thesis and also similar studies on this matter.

Performance overhead of virtualization has been examined for various I/O devices in different open source virtualization platforms. Jiuxing Liu [30] studies the virtualization overhead of network cards whereas Jim Salter [33] focuses on disk I/O performance. These studies analyse different virtualization I/O approaches and highlight I/O virtualization bottlenecks for their respective devices. This contrasts with this thesis which focuses on graphic cards besides other technologies like virtualization and remote desktop protocols. Moreover all these I/O devices have quite different bottlenecks, performance metrics and problems: a) NICs suffer for IRQ (interruption) affinity and balancing across CPUs to process packets, b) disks rely on file systems, cache modes and storage formats to increase reads and writes throughput and c) GPUs suffer from memory bandwidth and CPU performance to process more 3D operations per second.

Other papers focus on concrete virtualization I/O problems. Specifically A. Gordon et al. [18] analyze the virtualization overhead of interruption calls in I/O operations while the bottleneck for pass-through devices created by the I/O memory management unit (IOMMU) TLB is closely analyzed in [4]. These works cover concrete I/O virtualization problems but do not focus on any concrete I/O device. This thesis does not study such low level problems of I/O virtualization but relies on them and other factors to explain GPU performance overheads on virtualization.

Closer to this thesis topic, Ryan Shea et al. [34] perform an experimental study on GPU pass-through performance for both KVM and XEN virtualization platforms. This

study is focused on GPU pass-through techniques on Linux for cloud gaming, leaving out the analysis of other important factors in 3D graphics performance like the remote desktop application required for playing remotely. Further it does not consider other alternative virtualization types like application virtualization which is an interesting alternative to hardware virtualization because it does not require to install and support a different operating system. Moreover it uses Linux for both host and guest OS which is an OS that is quite coupled with both XEN and KVM and may not suffer the same overheads and problems than other OSES like Windows. On the other hand, this thesis studies different GPU virtualization techniques besides GPU pass-through like API-remoting and pure software GPUs. Moreover it covers a wider range of virtualization platforms (VMware and VirtualBox), virtualization types (application virtualization) and operating systems (Windows).

Jarschel et al. [26] perform QoE (quality of experience) study of a gaming service using graphic performance and response time as quality metrics. In contrast to [34], this work does take into account remote desktop applications in their study. However they perform a user side analysis, without paying any attention to the underlying gaming system neither their architecture and involved technologies which are actually the main topic of this thesis.

GPU virtualization performance and utilization of determined GPU APIs is also discussed in several papers. More specifically, over the last years there have been multiple works focused on supporting GPGPU APIs (usually CUDA) [35] [44] on virtual machines. These papers cover specific GPGPU operations whereas this thesis is focused on OpenGL API (API for 3D graphics). GPGPU applications are quite different from OpenGL ones due to GPGPU applications are not interactive so they do not have response time requirements neither the problems of remote graphic rendering. Therefore some of these papers propose redirecting CUDA calls to be processed in other hosts, which would be unfeasible for OpenGL. Thus their solutions and conclusions greatly differ with this thesis although both address similar problems.

On the other hand, WINE performance is studied in [28], which compares native Windows and WINE performance. This work uses games to benchmark WINE and native Windows 7 (on a local machine) but it does not compare its results with any other virtualization solution neither performs further studies on its performance overheads and limitations.



# 3

## Method

### 3.1 Problem Analysis

The problem introduced in section 1.2 is complex and should be divided into subproblems to be properly analysed. Section 2 introduced different technologies that address distinct aspects of this problem and allow to divide it in several parts. Thus the following sub-problems can be identified alongside the technologies that cover them: (a) running applications of one OS in another one (virtualization), (b) providing 3D graphics to virtual applications (virtual GPU) and (c) providing remote interaction with the application (remote desktop protocol).

Virtualization and virtual GPUs are quite related because each virtualization platform implements its own set of virtual GPUs if any, which usually are incompatible with others. However section 2.2 showed that in virtualization there are huge differences between different types of virtual GPUs which condition important aspects of the GPU like performance or GPU features. These aspects are really important for 3D applications and condition other architectural decisions like the selected virtualization platform itself. Further there may be big differences between implementations of the same virtual GPU between different virtualization platforms. Thus virtual GPUs are addressed separately for each virtualization platform.

On the other hand, remote desktop protocols do not rely directly on the virtual GPU nor in the virtualization technology for processing 3D operations but on the OS where applications run. This is due to remote desktop protocols rely on the OS display system to retrieve the image that is sent to the client. Thus there is a different implementation of almost all remote desktop protocols for each OS family. In fact, the OS where the remote desktop application runs may be another entity to be considered. However its analysis is done alongside the remote desktop application because this one is already

integrated with the OS display system which is the main component that affects 3D graphics besides the GPU driver. Further almost all remote desktop protocols have more than one implementation for the same OS. Regarding this variability, this thesis avoids comparisons between implementations of the same protocol in the same OS due to it is out of the scope of this thesis. Moreover there are already studies that compare them [8].

## 3.2 Technology selection

For each sub-problem introduced in section 3.1 there are several technology implementations or solutions that cover it and a representative set of them should be selected for further analysis. Here it is important to include differentiated technology implementations which provide a diverse set of features to choose from. Thus not only enterprise wise solutions but also research projects which may bring new features and contributions are considered.

Motivation section (section 1.1) encourages the case of running Windows applications on Unix-like systems for HPC. This case is quite representative for this thesis problem because more than 80% of desktop market share are Windows desktops whereas more than 95% of mainframe market share are Unix-like OSes [12]. Other interesting configuration is the opposite, running Unix applications on Windows. However this is not a common combination in HPC due to higher Windows Server licensing costs. Testing both scenarios would be too time consuming so this thesis focuses on virtualizing Windows applications on Unix-like OSes and leaves other possible scenarios for future work.

### 3.2.1 Virtualization technologies and virtual GPUs

Virtualization technologies hardly rely in the host OS because they tightly need to control system resources and in some cases, they even consist of their own OS (e.g. VMware ESX). This way host OSes are determined by the virtualization technologies that support them although it is also important to narrow the number of host OSes included in case studies. This reduces the variability of case studies and eases the comparison and study of the technologies running on them. Further it requires less time to study and deploy case studies. On the other hand each virtualization solution should also provide at least one 3D capable virtual GPU. Section 2.2 introduced different types of virtual GPUs which are interesting to cover in order to provide a wider range of alternatives for virtual applications and system architects.

Section 2.4 introduced multiple virtualization solutions and this thesis have considered the following: XEN, KVM, VMware ESX, WINE and two virtual machine monitors, VMware Workstation and VirtualBox. XEN and KVM are both open source, wide used and both provide at least one technology (PCI pass-through) for 3D support with

Windows virtual machines. Both solutions offer also software virtual GPUs for virtual machines like VMGL but they do not work on Windows because this OS does not support the X windowing system required by VMGL and other likewise virtual GPUs. It is interesting to differentiate between XEN and KVM because they have different virtual GPUs for PCI pass-through and also use different kernels to control the CPU and handle PCI interrupts. XEN is supported for different Unix-like OSes like BSD or Linux and it also has an enterprise distribution called 'XENSever'. On the other hand KVM is only available in Linux and it has also an enterprise version promoted by Red Hat. This way Linux is at this point a good candidate for host OS because it supports both XEN and KVM.

Although VMware ESX is a proprietary solution, it was by the time of writing the most used technology for server virtualization. Further, it supports three different alternative technologies for 3D graphics on virtual machines and it provides a trial licence which is enough for performing a case study. Besides these hardware virtualization alternatives, WINE is an interesting choice for running Windows applications on Unix like systems without the need of executing a complete instance of the Windows OS. However the main problem of WINE is that some applications may not work correctly because its alternative implementation of Windows dlls (Windows system libraries) may fail to behave exactly as expected. Moreover the behavior of Unix OSes is different than the Windows one so there could be unexpected behaviors in the application which may lead to application failures. Taking this into account and that bare metal hypervisors (XEN, KVM and VMware ESX) imply a big infrastructure change (architecture, hardware, complexity etc), VirtualBox and VMware workstation become interesting solutions to avoid these problems. On the one hand they do not require an OS or infrastructure change as they can be installed in for example a Linux desktop and on the other they do not have WINE problem for supporting Windows applications. However these two solutions may introduce big performance leaks and feature loss than the other alternatives because they run as an application on top of other OS. Due to Linux is the only OS than supports all proposed virtualization technologies but VMware that is itself its own OS, it is selected as host OS.

### 3.2.2 Remote display protocols

There are big differences between Windows and Linux windowing systems which drive the implementation of the different remote desktop protocols and they support for 3D graphics. Linux X windowing system is implemented using a client server architecture that makes it really easy to adapt for remote control. Further it supports an API remoting technique, VirtualGL which provides 3D graphics to all protocols which create a virtual session. Thus many protocols support 3D graphics in Linux like SSH with X forwarding, VNC, RDP or NX although it also supports a screen scraping implementation with VNC. All this protocols are quite interesting because they have quite different implementations and features.

X forwarding consists in forwarding desktop application windows (the rendered application display) across the network where the client sends display commands to the server and this one replies with screen updates. SSH just encapsulates the X forwarding session in a ssh one providing a direct connection with the X server with almost no further computation besides encryption. This protocol used alongside VirtualGL differs 2D rendering to the ssh client alleviating the server for some computation which may lead to some performance gain. Specifically it uses a VirtualGL feature called VGL transport where the server responses with special VirtualGL commands to the VGL client which renders the final image. Another difference between X forwarding and any other remote desktop protocol is that with X forwarding a single application is forwarded whereas remote desktops control and render an entire desktop session so it is a little bit more lightweight.

VNC is a quite simple protocol that just reads back either the local display frame buffer (screen scraping) or the virtual display one and streams it to the client. Although different VNC implementations may provide extra features like encryption, different compression algorithms ... VNC is the only protocol which provides 3D graphics for both Windows and Linux so it is interesting to compare the overhead generated by it in both OSes. There are two different types of VNC implementations on Linux, one grabs the local display like in Windows and the other creates a virtual desktop. There are many VNC implementations for Linux which create a virtual desktop however there is one called turboVNC, developed by the same people than VirtualGL, which is optimized for working with 3D graphics. Thus it seems to be the most appropriate choice for this use case. On the other side the only VNC screen scraping implementation in X11VNC.

Other protocols that create a virtual desktop are RDP and NX. RDP is quite more complex than VNC because it deals with windowing commands to detect changes, transmits sound, adapts the compression algorithm depending on the available bandwidth etc. This benefits user experience in low bandwidth/ high latency scenarios but it requires extra CPU processing which may degrade application performance with some extra latency in Windowing Operations and in situations with high CPU contention. There is a free implementation for Linux called XRDP. The same reasoning applies to the NX protocol which has an open source implementation called FreeNX.

In contrast to Linux, Windows display model has fewer places to insert hooks to monitor display updates. Furthermore it has a not clearly defined model for multi-user operation that complicates implementing multi-user sessions. Thus there are not many protocols which provide 3D graphics. Further there is no free API remoting library for Windows so virtual desktop cannot issue 3D commands to the GPU. Windows uses RDP protocol by default for providing remote access to its desktops however RDP creates a virtual session with no 3D support. Moreover other protocols like Windows NX version are based in RDP, thus they bear the same problem. On the other hand

VNC implementations do provide 3D graphics through screen scraping thus these solutions are not 'multi-user'. VNC for Windows has many different implementations like tightVNC, ultraVNC, realVNC ... It is not the purpose of these thesis to compare between different VNC implementations so only tightVNC is considered for case studies. There is no VNC version specially designed for 3D applications so tightVNC was chosen because it is source of almost all other VNC applications and all its code is open source. TightVNC provides a video driver to improve response times so both versions are included for tests. There are other third party commercial protocols which provides Windows desktops with 3D graphics with multi-user support like ThinAnywhere 3D, Desktop Cloud Virtualization (DCV) [2] or HP Remote Graphics Software [23]. However these protocols are no free to use or are only designed for specific software platforms.

In all case studies Windows OS is vitalized so virtualization specific protocols can also be included. VMware provides PCoIP (PC over IP) which is an interesting implementation because it relies on the graphic card proprietary API to retrieve the video signal. This may greatly improve performance of applications avoiding polling the frame buffer and adding a further video driver. However this protocol works only with VMware virtualization. On the other hand KVM provides SPICE protocol but it still does not support 3D graphics in Windows so it is not eligible.

### 3.3 Evaluation criteria

This section introduces the evaluation criteria that governs case studies. The problem solved with proposed technologies is to provide graphic processing to virtual applications. Thus we aim to evaluate graphic processing quality of each case study but also the consequences of deploying all involved technologies. By graphic processing quality we mean a set of characteristics directly provided by the graphic card like graphic performance or graphic card features. On the other hand, it is also important to consider in any technology deployment the consequences associated with their setup. Although this study comprehends many heterogeneous technologies which have different characteristics and provide wide variability of features, only attributes that differ from a system with no virtualization are considered. This is done to avoid comparisons between virtualization technologies, which are already well known, and to emphasise the side effects of using virtualization as a solution for this concrete problem. Figure 3.1 shows the evaluation criteria used in this thesis which is introduced in the following paragraphs.

The main reason for using virtualization is due to the maintenance cost of running a new operating system directly in hardware so it is important to evaluate the consequences of maintaining all technologies included in each different setup. Besides the maintenance of the added technologies like the virtualization technology itself it is also important to consider the virtual resources (virtual hardware and emulated libraries), extra applications (virtualGL or extra remote desktop applications), if an architecture

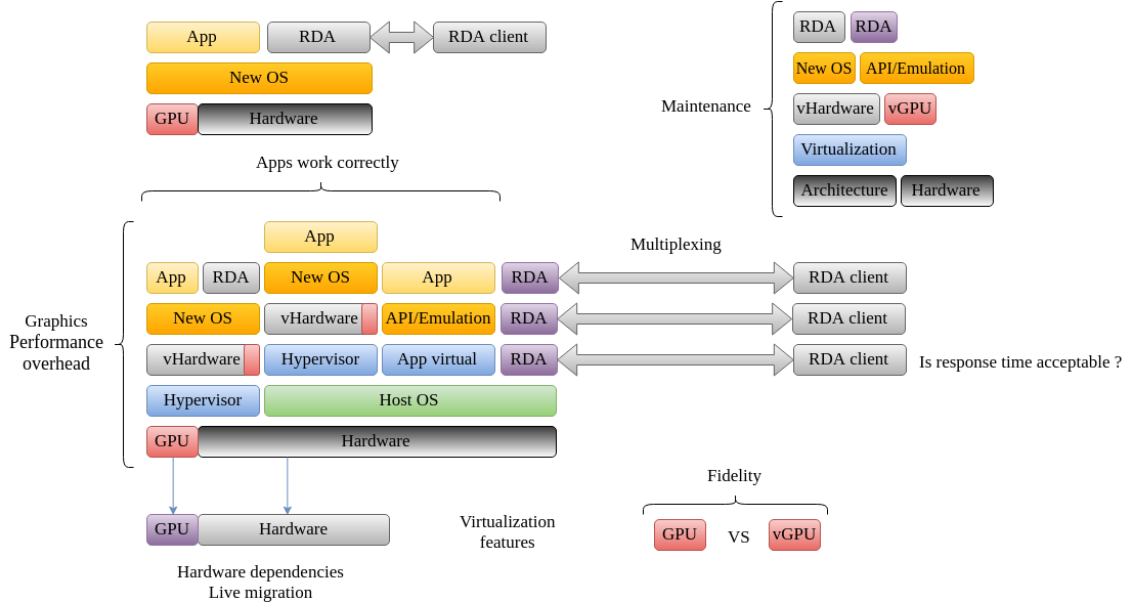


Figure 3.1: Evaluation criteria

change is required, integration with the underlying infrastructure etc.

The main evaluation criteria on case studies is *graphic performance* which means speed of operation of the graphic card. Graphic performance may greatly differ between a virtual and a not virtual system and it is affected by all technologies included in this thesis. Further, 3D demanding applications require a minimum performance to function so it is important to bound the expected performance loss for each of the proposed technologies regarding a non virtual system. Graphic performance can be measured in operations per second although it is usually measured in *frames per second* (FPS). Operations per second is mainly used in micro-benchmarks which use a single operation for stressing the GPU. For example *LuxMark* benchmark measures the number of rays per second generated by a ray tracing technique. On the other side, when a graphic application is used for benchmarking, graphic performance is measured in the number of frames that the graphic card renders per second (FPS). Micro-benchmarks are used to tests only a concrete component of the system, here the GPU, however their results are usually not representative for real applications. This is due to applications have different workloads, are limited also by other resources like CPU or memory and use a wider range of operations. Further, besides the virtual GPU, other components of the system also consume resources and produce performance overhead that may be negligible in micro-benchmarks which only use the GPU but not on applications. Taking into account that this thesis is motivated by 3D demanding applications like CADs, benchmarks which use these kind of applications seem to be more suitable for our purposes.

Using applications for testing is also interesting for validating that virtualization is working correctly. Some virtualization technologies like WINE provide an alternative implementation of Windows libraries and may lack some features of the original library or behave differently. On the other hand, hardware virtualization technologies may also fail to run applications due to problems on emulating hardware components or just because performance is too low. These factors can make an application crash or behave unexpectedly. It is not possible to test all applications neither to prove that given some conditions all applications of some kind will run with no problems. However it is possible to study how likely is that a kind of application works and also study the limitations of some technologies to see which applications may not work correctly. Here a special case arises when an application is unable to behave correctly because of performance limitations. This is likely to happen with software emulated GPUs or poor virtual GPU implementations. These exceptions are penalised in their evaluation because they are not suitable for target applications of this thesis but still taken into account because they could be useful in other less demanding scenarios.

Besides applications work correctly, other important requirement is a good *user experience* when interacting with remote applications, i.e. small lags in screen updates. Zeldovich et al. [31] proposed that response time is the most accurate way to measure user experience for remote desktops. Thus in order to evaluate user experience, the response time of an application running remotely in the target system is used. However analysing response time of applications is not in the scope of this thesis, it is just a mean to evaluate if a setup is good enough to be used interactively by a user. Thus this work uses a threshold response time from which a solution is considered not usable. ETSI [16] specifies that for an instantaneous experience, response time should be less than 100 milliseconds while for an uninterrupted experience it should be less than one second. Moreover ITU-T [25] (Telecommunication standardization sector of ITU) recommends that the one-way-delay (response time unit) for video transmissions should be below 150 ms and fixes an upper bound in 400 ms. This metric does not take into account the time a user takes to process a new computer interaction but it just specifies the maximum delay for instantaneous user interaction. Thus it is feasible to use it for delimiting system's delay. Combining both metrics and taking into account that this thesis tests are performed on a LAN environment it is reasonable to set 100 ms as an upper limit for response time.

Although graphic performance is important for applications which are performance limited by the GPU, other applications do not require the exclusive use of the GPU and it can be shared with others. This way although graphics performance of an application was quite worse in a virtual GPU, two applications running in two different virtual GPUs (but the same physical GPU) may provide similar performance than two applications in the non virtual system. However, taking into account that for a single user only an application is displayed at a time because it is the only one rendered (the GPU only

renders visible surfaces to avoid extra computations), it is better in this case to measure the number of users that can take advantage of a single GPU. This differentiation is important because an user can run concurrently many applications in any of the proposed solutions however the number of users that can access the GPU is limited in several virtual GPUs and also in some remote desktop protocols. In this thesis we refer to this characteristic as *GPU multiplexing* and it is categorized in three categories depending on how many users can take advantage of the same physical GPU at a time: *single user*, *limited multi-user* (limited number of users per device) and *unlimited multi-user* (user limitations are not fixed or predefined).

Virtual GPUs may also restrict features of the real GPU to the application. Dowty et al. [14] introduce this phenomena as *fidelity* which represents how close in features is a virtual GPU to the real one. While performance gives quantitative values to compare between different proposals, fidelity cannot be measured in that way. There are many different features that may be valuable or not in different scenario. Thus the presence or not of a feature may be critical for some cases (CUDA for example for GPGPU processing) or not important at all (any 3D graphics application that does not need CUDA). Furthermore there could be a lot of different features and they may variate between vendors and GPUs. This way in order to be able to compare between virtual GPUs, three different categories for describing the fidelity of a virtual GPU are created. These categories are based in classifications made by virtualization vendors like VMware or Citrix of their virtual GPUs and their feature set. *Native feature set* category, implies that the virtual GPU provides the same features as the native GPU driver. Here there is a special case for pass-through implementations where the OS and consequently the GPU driver changes between the host and the guest (Linux and Windows for example) and their features may change too. In this particular case it is considered that all features are present because it is a native driver (guest OS) the one that controls the GPU. *Basic feature set* defines a set of features that at least provide hardware 3D acceleration and support OpenGL but lack other features that the native driver supports like for example cooling or rendering quality control. This class describes a set of virtual GPUs that support some compute intensive 3D applications but do not require special GPU features to work. Finally *Incomplete feature set* includes virtual devices that do not provide essential features for 3D acceleration like OpenGL support or provide it via software acceleration. This last feature set will generally fail to run any application with minimum 3D requirements.

On the other hand, virtualization provides also extra features that the non virtual system lacks. Focusing only in features related to virtual GPUs, it is important to mention *live migration* (moving a virtual machine/application to other host without power it off/close it) and *machine/hardware independence*. This characteristics should be taken into account because some technologies bound the application to a specific machine which may be a big drawback in diverse scenarios.



### 3.4 Test methodology

This section explains how all technologies introduced above are analysed and tested. As it was explained in section 3.3, analysis is focus on graphic performance which is actually the only evaluation criteria which is not known beforehand besides response time. Furthermore this factor is quite affected by the aggregation of many technologies working together and also between different technology implementations. Given the three variable components of the architecture introduced before: virtualization platform, virtual GPUs and remote desktop protocols, it is easy to see that the virtualization platform is the core of the architecture which conditions all the other components. This way it defines which host OS is required, what virtual GPUs are available and in which OS the remote desktop protocol runs (in the host OS too or in a guest OS). Therefore there is a case study for each selected virtualization solution that includes tests for all supported virtual GPUs. Regarding remote desktop protocols, there could be many available protocols (like in Linux) so in order to save time, one protocol is selected to perform tests with all virtual GPUs. Then the others are tested with one of the virtual GPUs and compared against the base display protocol.

Further a good analysis should include a reference system to compare with. In this case, due to target applications are Windows ones, it is reasonable to compare results of virtual applications with native Windows ones. However technologies that run directly on top of Linux are limited by the maximum performance of this OS which may be quite different from Windows one. Thus it is interesting to include both OSes as reference systems.

Remote desktop protocols allow remote access to applications but at the expense of some performance overhead. This overhead comes from the actual application that is interposed between the user and the computer system and also from additional features provided by the display application like compression or encryption. Thus in order to identify which overhead comes from virtualization and which comes from remote display protocols, tests with and without remote protocols are performed. This also allows to see which virtual GPU is most affected by the display protocol and to select it for further analysis with the other available display protocols.

Case studies of this thesis follow this work flow:

1. Technology analysis: Initial study of target technologies. Here it is interesting to check the state of its last release (it is not required to be stable), expected results, differences with other alternatives, compatibility with the underlying system etc. At this early state this thesis avoids discarding any interesting technology. However if expectations are not good and it would involve a big effort to test, a technology may be discarded.
2. Installation and configuration: This thesis follows general configurations and recommendations from official guidelines without too much tuning in order to make

fair tests between all technologies and do not expend too much time in reading long advance administration and configuration guides. At this point and taking into account that some of the tested technologies are still unsupported and under active development, some effort is made in order to report bugs and get a working solution for the testing system.

3. Profiling and testing in local machine: tests are performed in the local machine, i.e. without a remote application. This way it is expected to get maximum performance of the virtual system. At this point, the virtual system is compared to the non virtual one and its bottlenecks are analysed for all virtual GPUs. Further comparisons with other virtualization platforms are included.
4. Remote desktop protocol tests: remote desktop protocols are evaluated. First one desktop protocol is evaluated with all virtual GPUs. Then one virtual GPU is selected for being tested with all the other remote desktop protocols. This virtual GPU is selected based on the biggest variability between results with and without a remote desktop protocol. Here it is interesting to test the overhead added by the remote protocol but also the overhead created by the combination of virtualization and remote desktop applications. Further differences between desktop protocols are also interesting because there could be big differences between them, for example in terms of CPU consumption, which may affect the entire system.
5. Results analysis and conclusions: Each case study concludes with an analysis and a little summary of the obtained results. Previous results of other case studies are used for comparison.

### 3.5 Benchmarking

Benchmarking remote graphic processing applications is a complex procedure due to the big amount of factors involved in providing this service and, in this case, the huge differences between the included technologies. Hardware, operating system, hypervisor, virtual GPUs or remote desktop protocol can play a major role in graphic performance. Furthermore benchmarking tools should be carefully selected in order to provide a stable performance for each setup avoiding as much as possible any transient phenomenon that may occur. Further they also should mimic real user workloads for them to be representative.

#### SPEC viewperf 11 benchmark

The selected benchmark suite is SPEC viewperf 11 [11], from now on viewperf, a graphics performance evaluation software that does not use games but scientific, 3D design and CAD applications to stress the GPU. This benchmark is interesting because it uses exactly the kind of applications targeted by this thesis. Benchmarks based on games may be equally good however games have quite different workloads and requirements than scientific applications and they also address different ranges of graphic cards. On the other

side, micro-benchmarks are useful to compare graphic processing power between GPUs however they are not representative for applications. This is due to they are based on the repetition of basic operations which do not represent applications workloads and are usually not affected by any other factors like CPU or memory. Thus using target applications for evaluating performance is the most representative solution. However for more detailed conclusions, *Furmark*, *OpenCL memory bandwidth benchmark* and *3dmark03* benchmarks are also used. The reason for using these additional benchmarks is for further analysis and for testing technologies that are not able to run the main benchmark but could also provide reasonable graphic performance for less demanding applications. Furmark is a GPU micro-benchmark which uses fur rendering algorithms to measure the performance of the graphics card. It is used for virtual GPUs that are unable to run SPEC viewperf benchmark due to memory or GPU feature set incompatibilities. On the other side, OpenCL memory bandwidth benchmark is used to detect memory bandwidth bottlenecks between GPU and CPU. 3dmark03 is a game based benchmark with low GPU requirements, only 128MB of video memory and DirectX9, an old version of DirectX library. It is used as complement of Furmark because it is an application benchmark and there was no other benchmark found which used scientific applications with such low requirements.

SPEC viewperf is composed of 8 *viewsets* of different scientific and 3D design applications. Each viewset is composed of several independent tests which run for a predefined period (between 12 and 30 seconds) and record the number of frames per second processed during that time. Once all tests of a viewset have finished, the weighted geometric mean of all tests is performed which is the final score of this viewset and the application it represents. There is a predefined weight for each viewperf test. SPEC viewperf has a version for Windows and another for Linux which is really interesting because it allows to use both Linux and Windows as base systems. The screen resolution for the tests is set to 1600x1200 pixels, resolution of the available monitor. It is important that resolution fits the monitor because if the resolution is bigger, all non displayed areas are not rendered so the final result is unfair. Here a short explanation of how tests are for each application of SPEC viewperf is given:

- catia-03: uses traces of graphics workloads generated by the CATIA V5 and CATIA V6 applications. This viewset is composed of eight tests of 12 seconds each that mainly simulate car and submarine 3D models, ranging in size from 6.3 to 25 million vertices which use a variety of common CATIA graphics modes.
- ensight-04: represents engineering and scientific visualization workloads created from traces of CEI's EnSight 8.2 application. Five models ranging from 36 to 45 million vertices are included in this viewset using display list paths through OpenGL. Each test last 30 seconds.
- lightwave-01: created from traces of graphics workloads generated by the SPECapc for Lightwave 9.6 benchmark. Models for this viewset range in size from 2.5 to

6 million vertices, with heavy use of vertex buffer objects (VBOs) mixed with immediate mode. It is composed of 5 tests of 30 seconds each.

- maya-03: 11 tests of 12 seconds that go from detailed 3D models of objects or characters till complete scenarios that are replicated multiple times and rotated or translated. The models used in tests range in size from 6 to 66 million vertices, and are tested with and without vertex and fragment shaders.
- proe-05: composed of 6 simulations of cars and engines that are rotated and translated. Models are created from traces of the graphics workload generated by the Pro/ENGINEER Wildfire<sup>TM</sup> 5.0 application from PTC. Model sizes range from 7 to 13 million vertices and run for 12 seconds each.
- sw-02: 5 test with a GTX car and 5 with an Suzuki engine that perform different 3D operations and render them with different detail using both solid surfaces and 3D models. This viewset is created from traces of the graphics workload generated by the Solidworks 2004 application. Each test last 12 seconds and renders a maximum of 3.13 million vertices.
- tcvis-02: based on traces of the Siemens Teamcenter Visualization Mockup application (also known as VisMockup) used for visual simulation. Models range from 10 to 22 million vertices and incorporate vertex arrays and fixed-function lighting. It is composed of 5 tests of 12 seconds in which two models of the same car are rotated from different perspectives and with different levels of detail from solid surfaces till components and engine structure.
- snx-01: based on traces of the Siemens NX 7 application. Traces represent very large models containing between 11 and 62 million vertices, which are rendered in different modes available in Siemens NX 7. Composed of 13 tests of 30 seconds each with 3D models of car and engine designs. Instead of solid surfaces, all models show structure and components of the car and its engine with different detail and textures.

# 4

## Case studies

### 4.1 Introduction

This section presents case studies of Windows applications running on Unix-like systems (VMware and Linux) for all technologies selected in section 3.2. Following the evaluation criteria introduced in section 3.3, tests start with the analysis of both Windows and Linux OSes with no virtualization in order to get a baseline to compare with. It is expected that Linux and Windows behave differently so a little comparison between them is done. There are no native VMware tests due to this operating system is made only for virtualization and does not provide graphic libraries for running user applications. Then it comes WINE, the only application virtualization technology in the test set. After two hosted hypervisors, VirtualBox and VMware Workstation come into picture and are compared against WINE. These technologies are the least intrusive ones and also the ones that are expected to produce the worst performance results. Tests continue with XEN and KVM and conclude with VMware ESX. Finally results and conclusions of case studies are summarized in section 5.

#### 4.1.1 Test systems

Two different machines are used for case studies. One machine is an IBM System x3550 M3 equipped with two Intel Xeon X5675 (12MB cache, 6 cores, 3.06 GHz, max turbo 3.46 GHz) and 72.5 GB of RAM at 800 MHz. The graphic processor is a Nvidia Quadro 600 with 1GB DDR3 RAM. The other machine is an IBM System x3650 M4 with two Intel Xeon E5-2667 v2 processors (25MB cache, 8 cores, 3.3 GHz, max turbo 4 GHz) and 256 GB of RAM at 1600 MHz. The GPU this time is a Nvidia Quadro 4000 with 2GB DDR5 RAM.

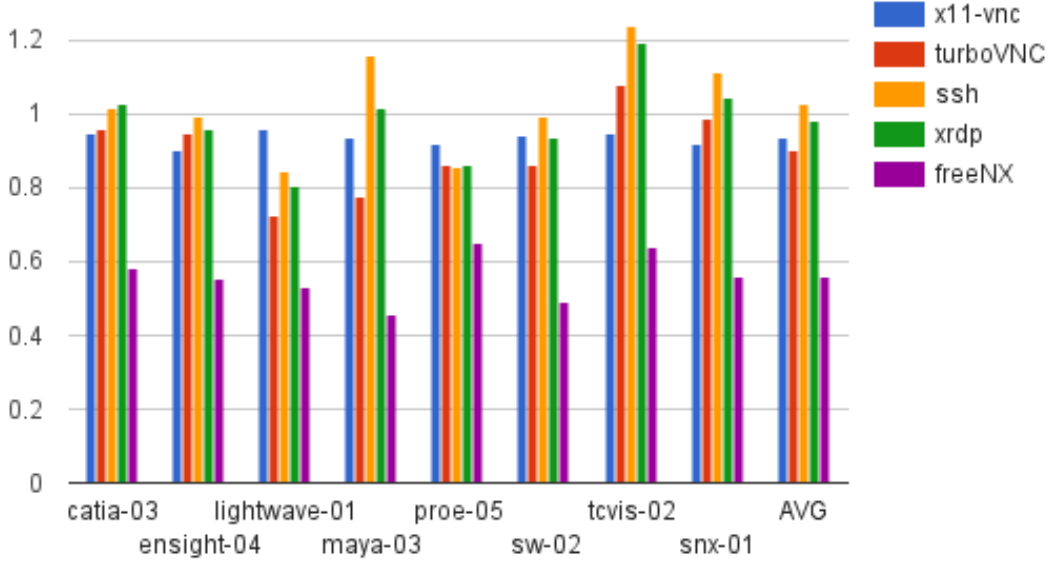
## 4.2 Baseline

This section describes tests which are run in bare metal systems i.e. the operating system runs directly on the machine with no virtualization. Thus results obtained from these tests are the reference for all the following case studies. Both Linux and Windows operating systems are tested because applications run on top of one of these operating systems. Due to these OSes are non-virtual only one GPU driver type is available i.e the GPU is assigned to the OS driver which is equivalent to PCI pass-through in hardware virtualization. There may be different implementations/versions of drivers of the same GPU for both OSes however it is out of the scope of this thesis to compare different native drivers. Thus the latest version of one of them is selected for both bare metal and virtual systems (PCI pass-through case).

### 4.2.1 Linux

Linux is the base system for all case studies where the GPU driver runs in the host OS. Therefore WINE, VirtualBox (API interception case) and VMware workstation should be limited by Linux maximum performance. This is due to these virtualization technologies run like any other Linux application and an application running directly on Linux shouldn't be outperformed by its Windows virtual version. However this is not necessarily true due to Windows and Linux versions of the same application may be completely different, and consequently their performance may greatly differ between them. This is not a common scenario because applications with versions for both OSes are usually based in the same source code and use simple directives or cross-platform tools to adapt the code to the OS APIs if they differ. Regarding this thesis, due to OpenGL API is the same for both OSes and the target API of SPEC viewperf benchmark, it is reasonable to say that Windows and Linux versions are similar and that the main factor which drives graphic performance is the GPU driver. Thus the reference system for case studies is the one that controls the GPU driver which in Linux is the proprietary Nvidia driver. This driver is well known for achieving the best performance over open source implementations [29] and it is beyond this project to compare drivers for Nvidia GPUs. Figure 4.1 shows the speedup of all previously selected remote display protocols compared to Linux running in a local session (no remote desktop application).

Although the version with no VNC was supposed to achieve the highest frame rates due to it is free of the extra computations needed to handle remote connections, figure 4.1 shows that this is not always true. X forwarding and RDP protocols achieve in several benchmarks better results than the bare metal implementation. Furthermore X forwarding has in average better performance than Linux. As it was explained before, X forwarding uses VGL transport in order to exchange rendered 3D images with the client. This images are combined at the client side with 2D elements of the application's GUI that are sent over the network using standard remote X windows protocol. This way the 2D (X11) rendering is performed in the client side releasing the server for this duty and saving CPU time. Moreover X forwarding doesn't create a virtual desktop,



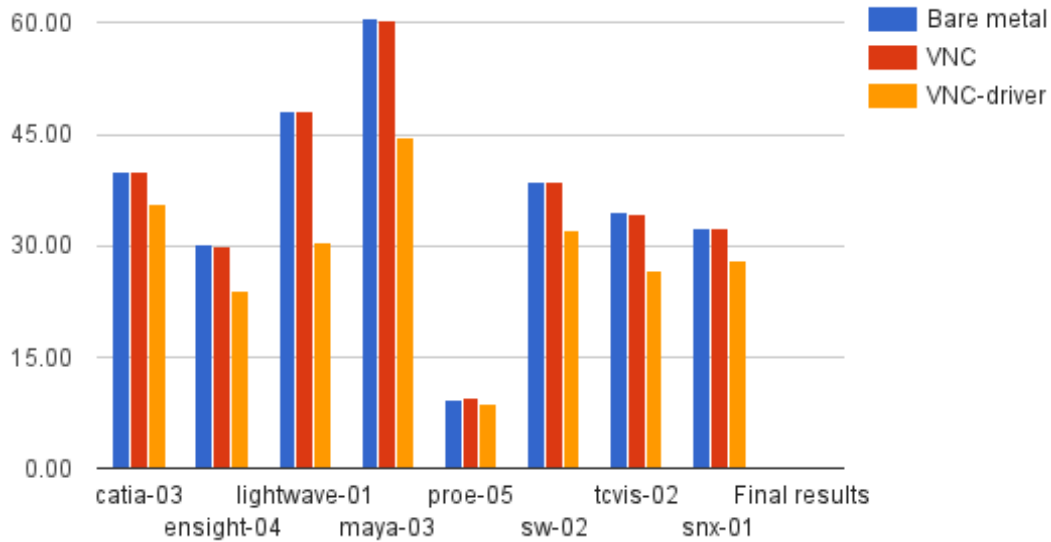
**Figure 4.1:** Remote desktop application speedup on native Linux

using seamless windows instead, thus avoiding to render a entire desktop and further, it does not perform any compression. On the other side XRDP has also shown that it can achieve almost as good marks as Linux. This protocol does create a virtual desktop but like 'VGL transport' it can leverage the 2D rendering to the RDP client. It also avoids compressing the data if there is enough bandwidth, like in this environment (LAN). Moving to VNC implementations, it was expected that both implementations suffer some performance overhead because everything (2D rendering and compression) is done in the server. Between x11VNC and turboVNC also was expected that x11VNC achieved higher scores because it doesn't have to deal with a virtual desktop neither uses VirtualGL. However results have shown that this is not always true. X11VNC in fact is always below Linux however TurboVNC is able to score higher rates in some cases like tcvis. Finally FreeNX results are not good, barely a 55% of bare metal Linux in average. It may be due to it is based in a legacy version of the original NoMachine NX server version 3 which is quite old and may be quite inefficient.

#### 4.2.2 Windows

Windows is the base system for all bare metal hypervisors like XEN, KVM and VMware with PCI pass-through due to it is the Windows Nvidia driver the one that controls the GPU. Figure 4.2 shows SPEC viewperf results with VNC and VNC driver compared to bare metal Windows.

Whereas VNC basic implementation achieves almost the same frame rates than the bare metal one, the VNC driver shows important performance leaks in all tests. This is because whereas the basic VNC just queries repeatedly the frame buffer in order to



**Figure 4.2:** VNC performance on Windows

get the display, the driver wraps the video signal and notifies VNC server with screen changes saving CPU time and intensive screen blitting<sup>1</sup>. However the extra layer added by the driver in the display stack delays the execution of rendering calls so graphics performance is expected to be lower than in the basic version. It is easy to see that the VNC driver is always below the other implementations. Performance leaks due to this driver go up to a 35% in lightwave benchmark.

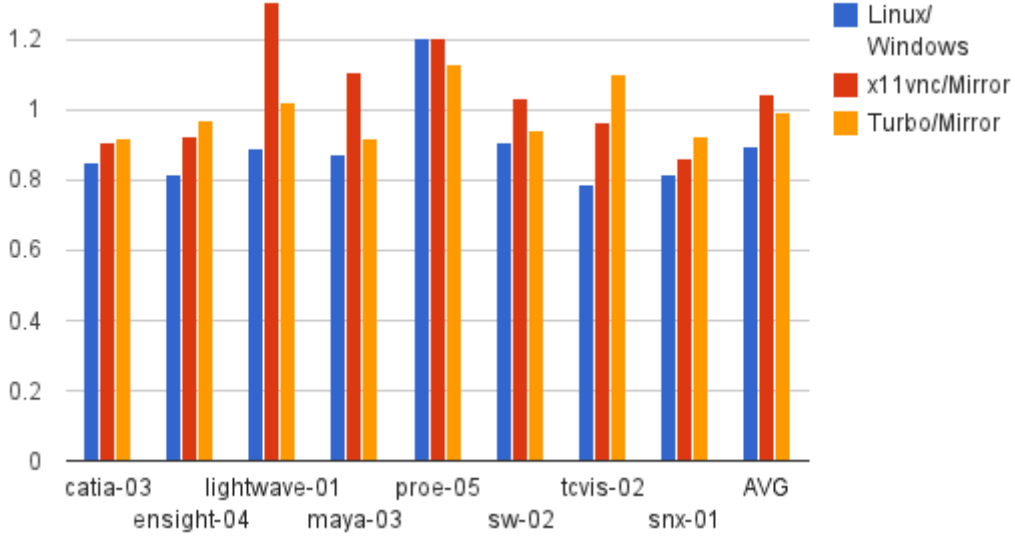
On the other side, VNC without mirror version produces big lags in user interaction. Tests have shown that the basic VNC version does not provide good enough response times for an interactive session so it cannot be used for this workload. However this driver can be activated and deactivated dynamically allowing the user to choose whether to use it or not. Furthermore if the user starts a program and closes the VNC connections, the VNC server stops queueing the framebuffer because it doesn't receive any update request. Thus performance only suffers VNC overhead in interactive workloads, that in general are less intensive than the batch ones.

### 4.2.3 Comparison between Windows and Linux results

It is not always interesting or fair to compare Windows and Linux results however it is in this case because we aim to evaluate and compare technologies that run applications on top on of these OSes. This way some technologies like WINE issue 3D graphics through

<sup>1</sup>Movement and modification of data within a computer's memory, usually movement of a bitmap, such as windows and fonts in a graphical user interface or sprites and backgrounds on a 2D computer game.





**Figure 4.3:** Linux vs Windows performance with and without VNC

Linux graphic stack and others like virtual machines with PCI pass-through work with Windows display model. Figure 4.3 shows Linux speedup compared to Windows. It is easy to see that Linux bare metal performance is worse than Windows one in almost all workloads, being Linux speedup close to 90% of Windows performance in average. Further, analysing profiling results of different spec viewperf applications, it can be seen that the ones with more GPU usage are Catia, Ensign, Tcvis and Snx that also are the ones that achieve the worst results in Linux. On the other side, Proe is one of the applications with less GPU usage and best results on Linux.

In order to get more detailed results a memory bandwidth test was performed in both systems. It consisted on benchmarking the memory bandwidth between the GPU and the host using OpenCL (Open Computing Language) memory transfers. Here host-to-device, device-to-host and device-to-device memory tests are performed and results show that in average memory transfers between the host and the GPU are 5% worse in Linux than in Windows. These results are far from being the only reason for Linux performance loss but actually the application with less memory consumption i.e. proe (also the one with less GPU consumption) is the one that works better on Linux. On the other side, comparing VNC results, Linux is the one that gets higher scores. In fact this is not a surprise because as it was explained in section 2.3 VNC performs better in Linux due to the implementation of the X server, designed for easing remote connections. This can be seen also in the speedup of VNC in both OSes which in Windows is about 80% in average while is close to 90% in Linux. Thus for the remote user, Linux may perform better than Windows even though in local session it doesn't.

## 4.3 WINE

WINE is a compatibility layer capable of running Windows applications on several POSIX-compliant operating systems, such as Linux, Mac OSX and BSD. Instead of simulating internal Windows logic like a virtual machine or emulator, WINE translates Windows API calls into POSIX (Portable operating system Interface, implemented in Linux) calls on-the-fly. This eliminates performance and memory penalties of other methods and allows to cleanly integrate Windows applications into Unix-like operating systems. API translation is done by providing alternative implementations of Windows DLLs (Windows system libraries), and a process to substitute Windows kernel. This way interpretation and recompilation of software is avoided. Therefore Windows applications running on top of WINE in theory should run close to Windows speed. However this is not always true and WINE's performance highly depends on the application. Further, widely used applications (without Linux version) tend to work better because extra testing and effort is done to improve them. However, WINE is in active development and many applications that do not work correctly are still being reported. Moreover there are big differences between WINE versions, where usually the higher WINE version, the higher performance and number of applications that work without problems. Regarding 3D graphics, WINE as a Linux application uses the Gallium3D driver model. While Gallium3D implementations of modern versions of Direct3D like DirectX10 or DirectX11 may have performance issues and present some problems, OpenGL should work without several performance issues (compared to Linux) because it uses Linux libraries directly. DirectX is a concern regarding game compatibility but since most scientific applications are written using OpenGL this thesis does not concern on DirectX performance.

SPEC viewperf11 executes 8 different applications so it is possible that not all applications can successfully run all tests. Actually the first tests made with WINE 1.6 version, the stable version at that time, failed to run some programs. However WINE 1.7 development version worked without problems. WINE as a Linux application is limited in performance by the Linux graphic stack (X windowing system and Nvidia driver) so Linux is its reference system. The first two columns in figure 4.4 reflect Windows version of viewperf11 performance running in WINE compared with native Linux implementation (with and without TurboVNC). The followings ones show VNC speedup for both Linux ([Linux + VNC] vs Linux) and WINE ([WINE + VNC] vs WINE) to compare performance loss caused by VNC.

As it was mentioned before, WINE performance has a high dependency on the running application. This way some applications keep scores close to their native version like Ensight or Snx but others like Tcvis or Proe experience important performance loss, up to the 70%. OpenGL functions are processed directly by the Linux driver so the only things WINE has to care about regarding 3D graphics is to emulate Windows windowing system (WGL), translating it to GLX, and to handle calls to OpenGL functions (the call itself, not the OpenGL function; concretely stdcall from Windows to cdecl in

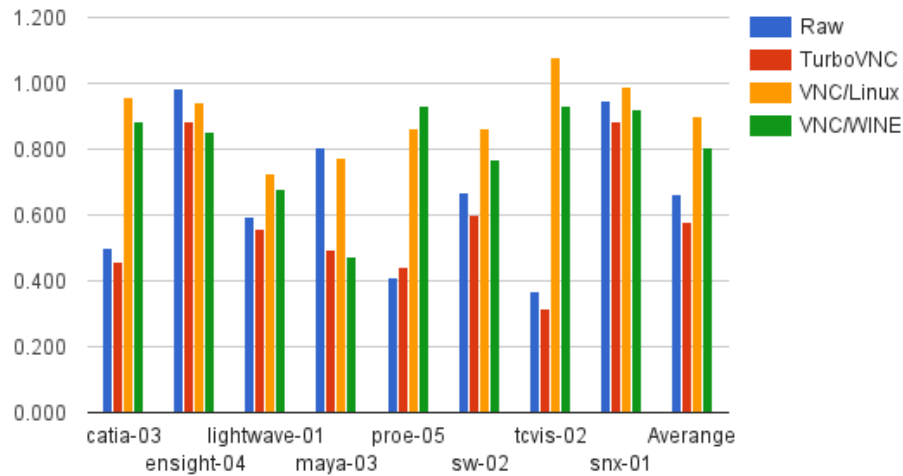


Figure 4.4: WINE speedups

Linux). Thus WINE OpenGL implementation is not likely to be the reason of its performance drops but the implementation of other Windows libraries. Actually, performing an OpenCL bandwidth test as in section 4.2 showed that memory transfers with WINE have no performance leaks.

Another interesting fact that appears in figure 4.4 is that for Maya, WINE speedup drops from 0.807 without VNC to 0.495 with VNC. This is about a 30% difference between the speedup of WINE with and without VNC whereas in the other applications is less than a 10%. This fact can be due different factors taking into account that after WINE layer comes VirtualGL wrapper and TurboVNC. It is uncertain why Maya has such big performance drops because CPU, GPU and memory stats do not show significant changes. This behavior was shown in other study [9] where VirtualGL was identified as the limitation factor in Maya tests.

One of the advantages of WINE besides it is easily integrated in a Linux desktop is that several WINE sessions can be run concurrently by different users. Figure 4.5 shows how viewperf11 scales from two concurrent sessions to eight sessions. It is easy to see that both Ensight and Snx that got the best speedup with one session are the ones that scale the less while Proe, that was one of the worst, is one of the programs that scales better. Ensight and Snx have almost a 95% of GPU usage with one process so with more than one session multiple processes are disputing the GPU time. Thus it is not a surprise that performance drops more than half with the double of concurrent sessions. In the case of Proe, the GPU usage is about a 60% in average so multiple sessions can interleave GPU operations with other ones. Thus it scales better.

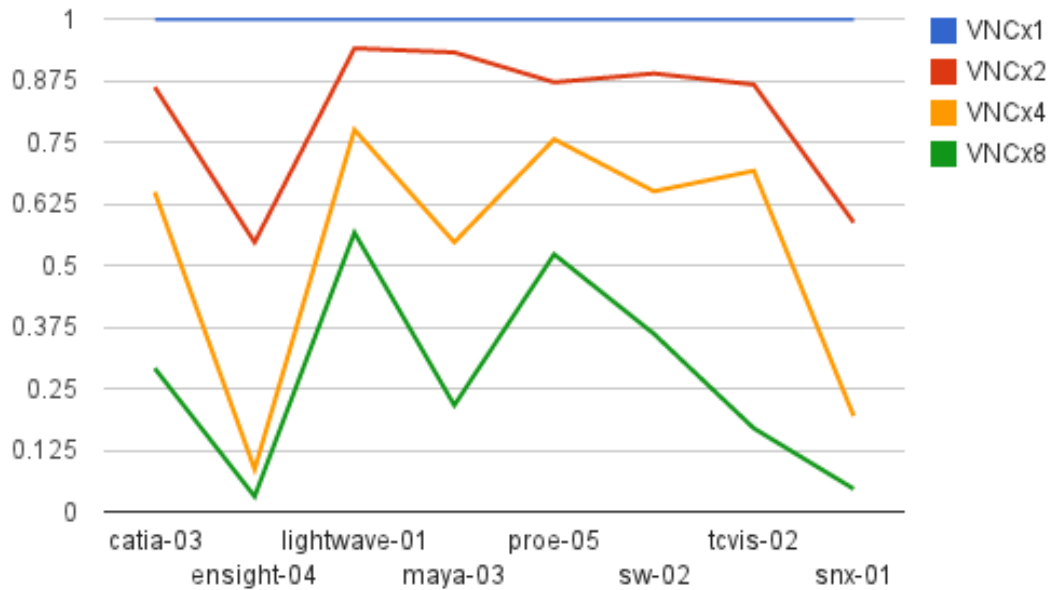
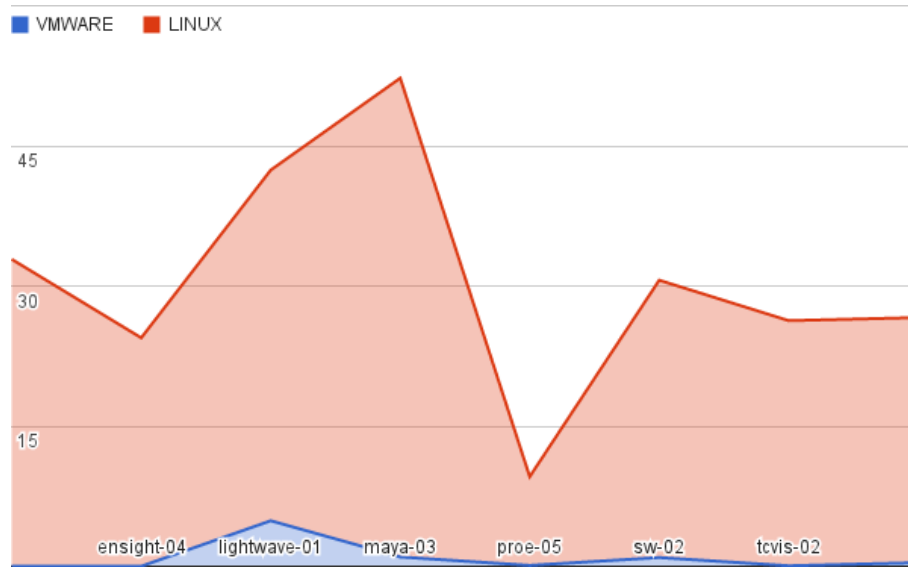


Figure 4.5: WINE multi-session speedup

## 4.4 Hosted hypervisors on Linux

In this section a Windows OS is installed in a virtual machine managed by a hosted hypervisor running in a Linux desktop. This kind of hypervisors (also known as hypervisors of type 2) run like a common program and leverage hardware and resource management on the host operating system. In order to run it remotely and get 3D graphics it is necessary to launch the virtual machines using VirtualGL as any other application. There is also the possibility of running the remote desktop application inside Windows OS however it has shown that remote desktop protocols work better on Linux and that virtual applications suffer some virtualization overhead. Thus remote desktop protocols are only run on Linux. Two hosted hypervisors are tried: VMware Workstation and VirtualBox.

VMware workstation for Linux only provides a virtual graphic card with support for DirectX (translated to OpenGL) and OpenGL that performs 3D operations redirecting them to the host GPU driver. On the other side, VirtualBox provides several modules for 3D graphics. It has a virtual graphic card that performs API interception only for OpenGL calls and redirects them to the host GPU providing this way 3D graphics. It also has two more experimental drivers for both DirectX and OpenGL. Further, it supports an experimental module for PCI pass-through which has been proven to work with basic PCI devices like USBs or network cards which may also work for passing graphic adapters.



**Figure 4.6:** VMware workstation performance on Linux

Figure 4.6 shows only VMware workstation performance compared to Linux because VirtualBox ones were unsuccessful. VMware Workstation achieved really bad performance and Catia and Ensign tests didn't work because both applications ran out of memory. Furthermore VirtualBox with the OpenGL driver was unable to run any SPEC viewperf11 application. VirtualBox uses an implementation of the chromium driver in order to redirect OpenGL calls. This driver supports up to OpenGL 2.0 whereas viewperf11 requires OpenGL 1.5, so they should have been compatible. However application logs showed that the chromium driver does not implement some NVIDIA extensions that are needed for running viewperf applications. Concretely log files report some non implemented functions and other OpenGL errors like invalid operations. On the other side VirtualBox crashes when trying to pass-through the graphic card. PCI pass-through worked successfully with USBs and the sound device of the graphic card however it crashes before loading the operating system when it tries to map the registers of the graphic device. Both DirectX modules failed to work too.

The main advantage of VirtualBox and VMware workstation is that an user has an entire dedicated Windows desktop allowing him to run any Windows application out of the box while he works normally in his Linux desktop. Other advantage is that multiple users and virtual machines can take advantage of one single GPU. However tests have shown that these technologies are far from providing decent 3D graphics.

## 4.5 XEN

XEN is an open source paravirtual hypervisor that consists on a XEN micro-kernel running in a privileged CPU state. This kernel controls among other things CPU scheduling and memory management. There is a special virtual machine called dom0 (domain 0), where the host OS runs, which has direct access to hardware and controls the hypervisor (XEN micro-kernel) and the other unprivileged virtual machines, also called ‘domUs’. The only available method in XEN for providing 3D graphics to virtual machines is PCI pass-through which is done through xen-pciback driver which exposes the PCI device in user space allowing it to be controlled directly by a virtual machine.

### 4.5.1 Quadro 600 tests

As it was explained in section 1, graphic cards are more complex than other PCI devices and it is more difficult to assign them to virtual machines. There are some high end graphic cards with special hardware for PCI pass-through, for example, in Nvidia cards this feature is called multi-OS. Unfortunately the first GPU available for testing, Nvidia Quadro 600, doesn’t have this feature. However several experiments with other Nvidia cards and new patches in Linux PCI drivers have shown that with the appropriate patches and customizations any graphic card may be used for pass-through.

Depending on the GPU, GPU architecture and GPU driver, OS kernel, XEN version etc different problems may arise including problems in interrupt handling, in PCI configuration and in the device memory mappings. Although making this concrete GPU work with PCI pass-through is not the main objective of this thesis, it allows to analyse the limitations of PCI pass-through and also the treats which this technology may bring to the system. Thus several tests with different XEN patches and kernels were performed in order to make Quadro 600 work in a PCI pass-through configuration. However, despite in some of then the card was recognised by the guest operating system and Nvidia drivers, the GPU was never able to function.

One of the reasons graphic cards are not able to function with PCI pass-through is XEN inability to correctly configure GPU memory mappings in the virtual machine when passing through the GPU. In order to work around this problem, several authors have proposed to manually assign GPU memory mappings in XEN code. Specifically, this thesis follows some fixes for XEN recollected and maintained by David Techer [37]. The proposed solution consists in modifying the code of XEN HVM (hardware virtual machine) loader and hard-code memory-mappings and I/O port addresses of the graphics’s card configuration registers i.e. GPU *base address registers* (BARs). When a device is initialized, its driver requests to the operating system the size of memory needed. Then the operating system maps the device memory to the system memory and writes these memory mappings to the device’s BARs. These address mappings remain valid till the system is turned off and are the ones passed to the virtual machine loader so it knows

exactly where the pass-through device is mapped. Listing 4.1 shows Quadro 600 memory BARs where the first three memory ranges are highlighted in bold. These are the ones David Techer proposes to hard-code in XEN HVM configuration.

```
[0.685470] PCI 0000:1f:00.0: reg 10: [mem 0x97000000-0x97ffff]
[0.685479] PCI 0000:1f:00.0: reg 14: [mem 0xf0000000-0xf7ffff]
[0.685488] PCI 0000:1f:00.0: reg 1c: [mem 0xf8000000-0xf9ffff]
[0.685494] PCI 0000:1f:00.0: reg 24: [io 0x2000-0x207f]
[0.685500] PCI 0000:1f:00.0: reg 30: [mem 0xfff80000-0xffffffff]
```

**Listing 4.1:** Quadro 600 MMIO BARs

These ranges were hard coded in the DSDT<sup>2</sup> of the ACPI<sup>3</sup> implementation of XEN's HVM loader. Listing 4.2 shows the one to one memory mappings between the virtual machine loader and the graphic card. In each memory range (here DWordMemory) can be seen highlighted in bold the minimum, maximum and the size of each range respectively.

```
DWordMemory(
    ResourceProducer, PosDecode, MinFixed, MaxFixed,
    Cacheable, ReadWrite,
    0x00000000,
    0x97000000,
    0x97ffff,
    0x00000000,
    0x01000000)

DWordMemory(
    ResourceProducer, PosDecode, MinFixed, MaxFixed,
    NonCacheable, ReadWrite,
    0x00000000,
    0xf0000000,
    0xf7ffff,
    0x00000000,
    0x08000000)

DWordMemory(
    ResourceProducer, PosDecode, MinFixed, MaxFixed,
    Cacheable, ReadWrite,
    0x00000000,
    0xf8000000,
    0xf9ffff,
    0x00000000,
    0x02000000)
```

**Listing 4.2:** Reserve MMIO BARs of q600 for 1:1 mapping

<sup>2</sup>The Differentiated System Description Table (DSDT) is the main table in the ACPI implementation of a computer's BIOS.

<sup>3</sup>The Advanced Configuration and Power Interface (ACPI) defines the interface between an ACPI-compliant operating system and the system firmware.

The original patch only specifies three memory ranges but more ranges can be assigned. Up to five memory ranges with different XEN versions have been tried but all of them were unable to make the graphic card work although the GPU was actually recognised by the Nvidia driver. It is uncertain why the driver was unable to function but due to its complexity and time limits this thesis has finally left this case for further research.

#### 4.5.2 Quadro 4000 tests

Nvidia Quadro 4000 card worked out of the box without any patch or source modification. Figure 4.7 shows the speedup of a XEN Windows virtual machine which is in average almost 90% of the native Windows performance. It is easy to see that the speedup oscillates among applications being almost 100% in applications like Ensight or Tevis and close to 75% in other applications like Maya, Lightwave and Proe. Despite CPU utilization for these applications can be quite high is the GPU utilization the one that makes the difference between the other applications. Here applications with more GPU time are the ones which perform better. This is because it is the native Nvidia driver the one that controls the GPU so GPU operations are performed with no overhead whereas CPU operations and memory transfers do. Therefore OpenCL tests were performed in order to check if memory transfers are affected by the virtualization layer. Results have shown that memory transfers between the GPU and the CPU are a 25% slower in XEN than in native Windows. Thus the overhead of memory transfers is one of the main factors that limits GPU performance on XEN Windows virtual machines.

Moving to VNC tests, basic VNC achieves similar results than the raw (local display) version however the driver one generally produces lower scores. Last column shows that while basic VNC achieved about a 90% of Windows performance, the driver version only gets an 80%. This is produced by the extra CPU time needed to wrap the video signal (about a 5% of CPU time) that is accentuated in the virtual machine. However performance loss compared to basic VNC version is quite big (a 10%) so in order to corroborate that performance leaks were due to the video driver wrapper a different version of VNC have been tested along tightVNC: UltraVNC. However results showed that there is no big difference between both VNC versions.

XEN as a paravirtual hypervisor is able to reduce virtualization overhead for paravirtual Oses so it is interesting to see if a paravirtual OS is able to perform better than Windows. Linux has a paravirtual version for XEN but it does not support PCI pass-through directly. However in its non-paravirtual version is also able to run extra paravirtual drivers for handling interrupts, timers and extended page tables. Thus it is also interesting to see if a Linux virtual machine with paravirtual extensions (also called PVHVM, paravirtual on hardware virtual machine) is able to get better results than its Windows counterpart. Figure 4.8 shows that in fact Linux achieves better scores achieving a 90% of bare metal results with both x11VNC and turboVNC against 80%



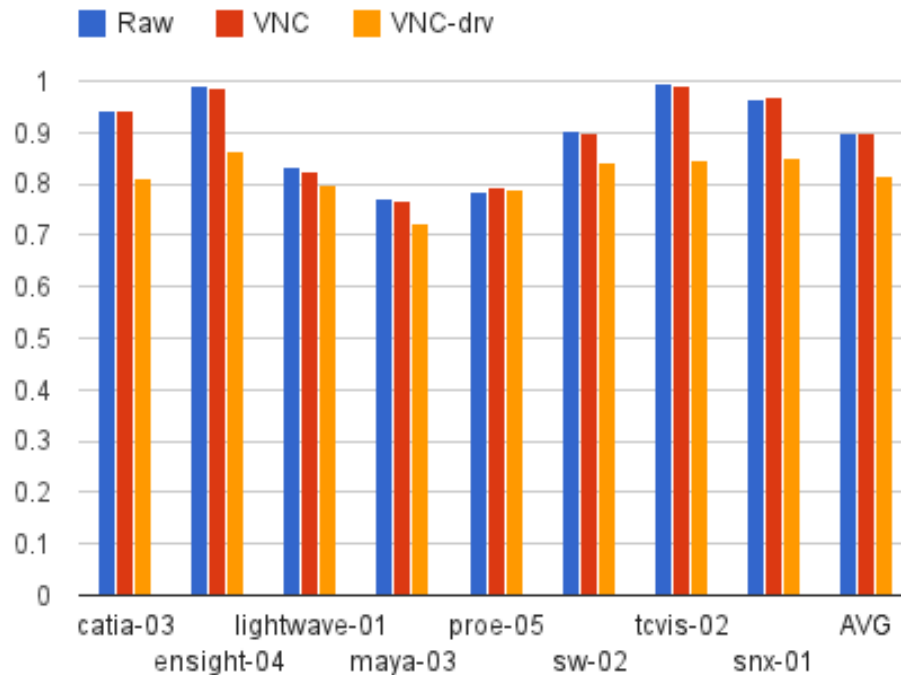


Figure 4.7: Windows-XEN speedup

of VNC driver in Windows. Here Ensignt and Tcvis are still the applications that perform better while Lightwave, Maya and Proe continue having the worst results compared with bare metal Linux. Looking at GPU performance, again these applications have less GPU intensive workloads and are more dependent on memory transfers and I/O performance than the other applications. One interesting thing regarding Linux viewperf implementation is that in general all applications require more RAM in the GPU than their Windows versions. These can imply less GPU performance due to more memory transfers between CPU and GPU have shown to be slower than in bare metal.

Finally it is important to mention some problems occurred during PCI pass-through tests. PCI devices were unable to function after they were used by a virtual machine. This happens because xen-pciback driver is unable to re-initialize the device after it was used by a virtual machine so the device became useless. A temporal solution to this problem is to eject the device before the virtual machine was shutdown but an abrupt shutdown still makes the device unable to function. The second and biggest problem was that PCI pass-through produced system crashes in the host (not only in the virtual machine) in several scenarios which is a big concern in virtualization. These crashes happened mainly due to (a) a bad initialization of the device because it was previously used by other virtual machine or the driver was not well installed and (b) it also crashed sometimes during system shutdown while installing drivers. However once the device is functional in a virtual machine, no other problems have appeared.

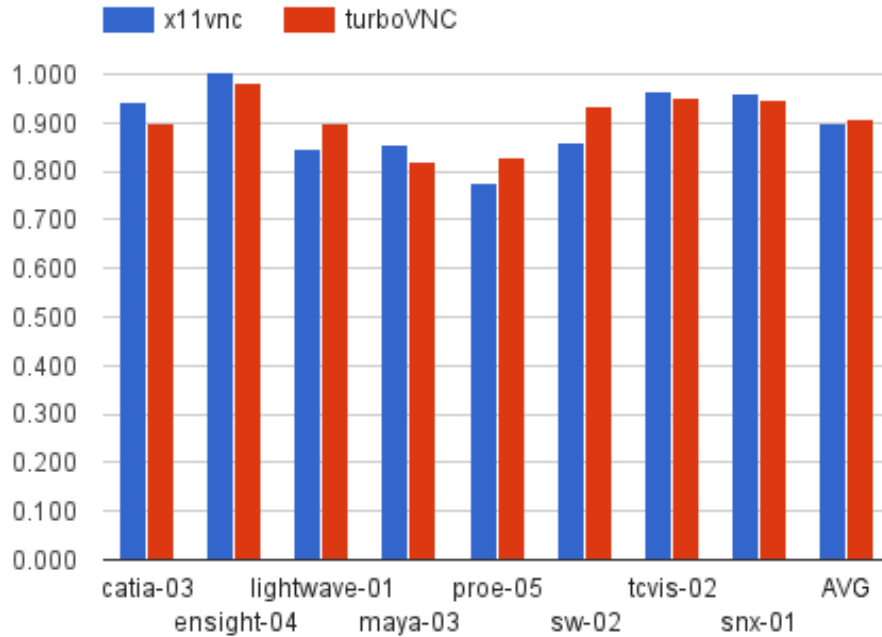


Figure 4.8: Linux XEN speedup

Although it does not mean that an application able to crash the GPU driver in the virtual machine couldn't break also the entire host. XEN project security team explains in [36] security problems regarding direct assignment of PCI devices. Here they mention that non standard methods of PCI configuration space accesses may produce system DoS (Denial of Service) and crashes. This explains system crashes after a device was used by other virtual machine or while installing drivers. However it is difficult to conclude if the crash is due to a bad device configuration, due to other device back-doors (non documented behaviors) or due to a xen-pciback bug. At this point we leave further research PCI pass-through security concerns for future work.

## 4.6 Kernel based Virtual Machine

KVM (kernel-based Virtual Machine) is a full virtualization solution for x86 hardware with virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module (kvm.ko) for Linux kernel that allows user space programs to take advantage of hardware virtualization features. However it requires a modified version of QEMU emulator for running virtual machines. KVM is part of mainline Linux kernel since version 2.6.20.

Like in XEN, the only solution for providing 3D graphics in KVM to virtual machines is to pass through the graphic card. KVM supports two different drivers for this purpose, pci-stub and VFIO. Pci-stub is the legacy KVM pci driver and the one used by Shea and

Liu [34] in their tests. Although this driver is being replaced by VFIO, it is still functional and has been improved since it was tested by them thus results and conclusions may be different. However due to both systems, implementations and benchmarks are different, it is not possible to make a direct comparison with their results and the ones obtained in this case study.

#### 4.6.1 Quadro 600 tests

As with XEN (section 4.5.1) several attempts were performed in order to get Quadro 600 working with PCI pass-through. Unlike XEN that only works with high end Nvidia graphic cards, KVM is making efforts in supporting all kind of PCI devices. Several attempts were made with different kernels and packages from different distributions but all failed to work. Further, different patched versions of the Linux kernel, seabios (bios emulator) and QEMU emulator proposed by different authors were unsuccessfully tried. All tests were performed using both pci-stub and VFIO drivers.

#### 4.6.2 Quadro 4000 tests

Quadro 4000 worked out of the box with KVM. Figure 4.9 shows results of different KVM configurations compared with their bare metal counterparts. Columns one to four of each viewperf application show Windows speedups with KVM while the last column shows the results of running SPEC viewperf on a Linux KVM virtual machine. Looking into KVM Windows results, the first two columns show the ones of the new VFIO driver while the following two show pci-stub driver ones. Although the main objective of VFIO wasn't performance but security and feature support, results show that it performs also better than pci-stub, about a 5% better in average. Comparing KVM Windows results with XEN (Section 4.5), in general, they are slightly better in KVM, about a 1% higher in average. Further, there is no difference in OpenCL tests between both hypervisors. Thus it is safe to say that both technologies (VFIO and xen-pciback) are equivalent in performance terms.

Looking at Linux results, KVM also outperforms XEN by about a 5% in average where KVM gets a 96% of native Linux scores in average. This improvement does not seem to be due to PCI driver implementation because both KVM and XEN Linux virtual machines use the same driver, i.e. Linux Nvidia driver, whereas the host driver is the same one that runs with Windows virtual machines. However disk and network drivers, interruption and memory controllers etc change between both technologies.

Regarding VNC, KVM achieves similar speedups than the non-VNC version compared to native Windows. This means that there is almost no performance overhead in VNC due to virtualization. This contrasts with XEN, which in VNC suffers a 10% extra loss. VNC adds extra computations to process mainly display changes and packet transfers which increase the CPU time required to draw a frame. This means that KVM

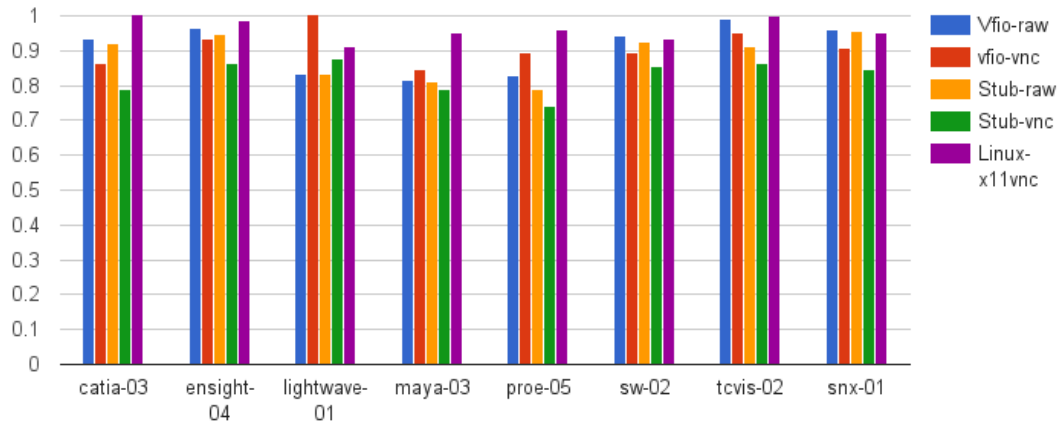


Figure 4.9: KVM speedup

CPU processing causes less overhead than in XEN which can also be seen in non-VNC results where KVM always outperforms XEN.

Like in XEN, KVM also produced system failures with both drivers. However VFIO is able to restart the device after the virtual machine is shutdown but it crashes when resetting GPU's audio device. Stack traces suggest that this problem may be due to a bad assignment of device interrupts. However it is beyond the scope of this thesis to study the concrete reasons why this crashes occur.

## 4.7 Virtual machine deployment tool

PCI pass-through is still experimental for graphic cards and in some platforms they are not easy to configure in a flexible way. This is due to a custom virtual machine configuration has to be done for each machine and graphic card. Further, in order to access remotely, it is necessary to customize the network configuration. Thus either the virtual machine is configured for remote access (with a public IP) or host ports have to be redirected to virtual machine ones. Moreover failures and system crashes are common during testing and deployment phase so all this configurations have to be done repeatedly. Thus in order to automate these steps and ease PCI pass-through testing and deployment a tool has been developed. This tool is focused on XEN and KVM due to the inability of VirtualBox and VMware Workstation to work with this feature. On the other side VMware ESX already has tools for doing this and WINE does not require it.

The objective of this tool is to setup a Windows (or other OS) virtual machine from a virtual machine template with some configuration customizations through the command line. In order to be flexible, instead of providing physical port of PCI pass-through

devices, it is enough to provide any keyword that identifies the required device, e.g. Nvidia or quadro 4000, and the tool selects and configures (if not already in use) the device that matches those keywords. This way the same virtual machine works in any host with GPUs with compatible drivers. Further not only graphic cards but any PCI device works with this feature. Besides this, other important feature for graphic card pass-through is to access the virtual machine remotely. In order to do that, virtual machine and host network is configured to accept connections to the remote desktop server port from the outside. In order to perform all this customizations, this tool follows these phases:

1. Configuration and customization: here parameters are parsed and virtual machine is customized. Only a small set of arguments are considered to fulfill a configuration request. Here are included a virtual machine template (configuration file and image path or the name of a registered virtual machine), GPUs to pass-through (and optionally other PCI devices), network configuration, virtual CPUs and memory. For GPU configuration, in order to be 'independent' of the PCI port or physical GPU, this tool allows to specify GPU characteristics (driver version, GPU model or brand) instead of PCI port. This way the only requirement for this tool to work in different machines or physical GPUs is for the GPUs to work with the same driver. Furthermore it only looks for free GPUs (not used for other virtual machines or the host) and fails if all are in use. At this point, differences between hypervisors like network options and devices configuration are handled.
2. Host configuration: the system is configured to run and connect to the virtual machine. Here PCI devices (included graphic cards) are assigned to pass-through drivers, network ports from the virtual machine are allocated in the host firewall for remote connections, network interfaces are configured and the virtual machine template is soft-cloned. Assigning PCI devices to pass-through drivers has shown to be quite tricky because it could lead to system crashes. However, what has shown to fail is to release the PCI device after execution or to reuse an unreleased PCI device. In order to avoid system crashes this tool follows the next considerations:
  - Bind all devices with the same *IOMMU group* to the pass-through driver: a physical device may have more than one functionality or virtual functions and some PCI pass-through drivers require that all of them were assigned to the pass-through driver. Some GPUs consist of the GPU itself and an audio device.
  - Pass-through only the GPU: some system crashes were experimented with some drivers when both devices (GPU and audio device) were pass-through to the virtual machine when working with the audio device.
  - Try to reset the GPU: after the guest is shutdown the GPU has to be reset. Some drivers fail to do this and the GPU remains in an unstable state and cannot be used again. In order to avoid this the following solution also worked.

- Eject or disable the GPU before shutdown and enable it at start up. The GPUs after being used by a guest host are in a unstable state. In order to force it to be reset, this task has to be performed on the guest host with the help of some scripts.
3. Virtual machine image customization for different purposes like setting network configuration (hostname, IPs ...) on the guest or setting custom VNC configurations (port and password). In order to accomplish this it is necessary to mount the guest file system in the host and then perform the customization. In Windows case it is also necessary to manage Windows registry to change VNC passwords for example.
  4. Run the virtual machine: the last part of the process is to power on the created virtual machine, wait till it is on and connect to it. When the virtual machine is powered off all system changes performed before are undone.

Regarding development design, the selected programming language is python. The main reason is that it is a scripting language with bindings for *libvirt* (the administration tool for KVM and XEN) and it also allows to issue bash commands and parse its outputs in a really straightforward way. The tool pseudo-code can be found in appendix A

## 4.8 VMware ESXi

VMware ESXi is a bare metal hypervisor developed by VMware for deploying and serving virtual machines. It provides its own kernel called *vmkernel* which handles CPU and memory among other hardware resources.

VMware provides several virtual GPUs depending on user needs. Between these virtual GPUs are included a basic software 3D GPU and a direct assigned GPU (PCI pass-through) like in XEN and KVM. However it also offers API remoting virtual GPU able to provide 3D graphics for multiple virtual machines. PCI pass-through and software 3D technologies do not require any license at all and can be used with the free version of this hypervisor but the API remoting GPU does require a license. However it comes with a free trial period. Here both free and non-free technologies with trial licenses are analysed. The main motivation for including technologies with licence is to encompass more types of virtual GPUs not available in free technologies.

### 4.8.1 Software graphics GPU and API remoting GPU

VMware provides a software virtual GPU with support for basic 3D graphics (no OpenGL redirection to the GPU) designed for 3D desktop effects and basic video rendering. It's

well known that this virtual graphic card provides poor results for 3D demanding applications but it is interesting to compare it to the API remoting GPU called *virtual Shared Graphics Acceleration* GPU (vSGA). This card uses the same driver as the SVGA in the guest OS, supporting only DirectX 9.0c and OpenGL 2.1, but it provides hardware accelerated 3D graphics. This is done redirecting OpenGL and DirectX calls to the GPU which is controlled by a special Nvidia driver for VMware ESXi.

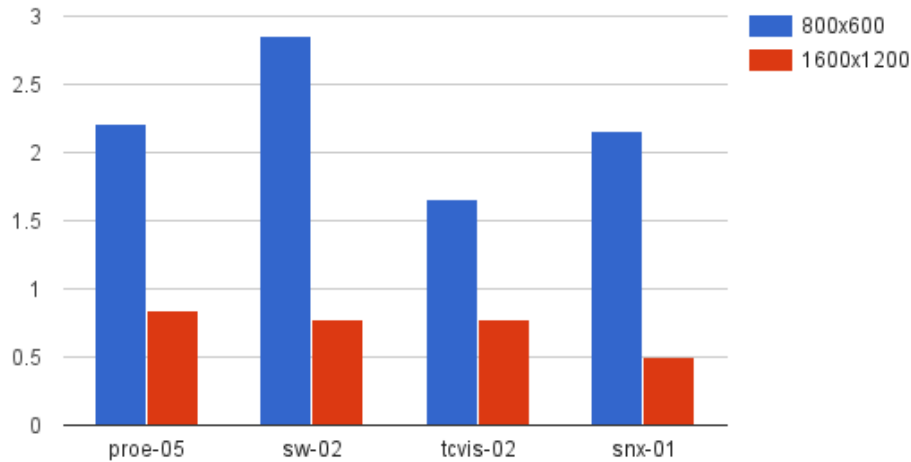
Tests performed with both cards have revealed that both are too limited to run 3D demanding applications. This is not surprising for the software GPU cause this graphic card is not made for dealing with complex textures or calculations but it was not expected for vSGA GPU. Actually, the software GPU successfully ran all tests although results were not good at all. In fact, it scored 0 frames per second in several viewperf test cases. However vSGA failed to run several viewperf applications due to they ran out of GPU memory. VMware only designates a maximum of 512 MB of RAM of the virtual machine to this graphic card, exactly the minimum amount of memory recommended for running viewperf11. However VMware specifies that from those 512 MB, half are reserved in the GPU and the other half from the virtual machine's RAM. This explains why some applications couldn't work properly with only 256 MB of GPU memory.

In order to properly evaluate vSGA, further tests with viewperf were performed with a smaller screen resolution (800x600) which requires less GPU memory. Figure 4.10 shows that with this resolution vSGA outperforms up to 2.86 times the software vGPU. On the other hand, with the original resolution, i.e. 1600x1200, this behavior changes in favor to the software render that outperforms vSGA results up to twice its performance. The reason of this results is due to the overhead of memory transfers between CPU and GPU due to bigger textures and pictures. Performance drops between both resolutions, taking into account that one has the double of pixels than the other, with the software card are quite small (20% in average) compared to the vSGA (close to 70%).

Even though vSGA results with a smaller resolution already revealed the memory limitations in the original tests, they didn't showed the real difference in terms of computation power between both virtual GPUs. Thus two more benchmarks with less memory requirements were performed: 3Dmark3D, an application benchmark for OpenGL and DirectX with a maximum of 128 MB of memory and Furmark, an OpenGL micro-benchmark. Table 4.1 shows that without memory limitations the hardware renderer greatly outperforms the software one.

#### 4.8.2 PCI pass-through and virtual Dedicated Graphics Acceleration

PCI pass-through does not require any licence in VMware ESX but as with KVM and XEN to access the graphic card output, a display protocol like VNC is needed. However VMware provides another solution using its proprietary protocol PCoIP (PC over IP). This solution, also known as *virtual Dedicated Graphics Acceleration* (vDGA), gets graphic card's output through a special API provided by Nvidia (thus it only works with

**Figure 4.10:** Speedup of vSGA over Software 3D GPU

Benchmark	Soft 3D	vSGA	Speedup
3Dmark2003 1024x768	544	36267	66
Furmark points 1600x1200	11	1019	92
Furmark FPS 1600x1200	0	19	-

**Table 4.1:** 3DMark and Furmark with soft3D and vSGA

a limited number of Nvidia cards) and uses PCoIP to interact with the virtual machine. Nvidia API allows to query directly the display driver avoiding continuously checking the framebuffer or adding an extra layer to wrap the video signal. Table 4.11 shows the results after executing SPEC viewperf11 with both technologies.

vDGA solution outperforms PCI pass-through with VNC in all applications but in tcvis-02 and especially in proe-05 where results are worse. Proe-05 is characterized by a high CPU consumption during all test set whereas VNC CPU time is close to 0. This low CPU consumption by VNC is due to proe generates low frame rates which provoke small screen changes. Workloads in both programs are mainly objects that are moving in a uniform background. This favors a simple protocol like VNC that just works with the entire image while PCoIP deeply analyses the display looking for text, buttons, images, videos etc in order to save bandwidth or choose the best compression algorithm. In average, PCoIP achieves an 8% better performance than VNC. Furthermore, PCoIP



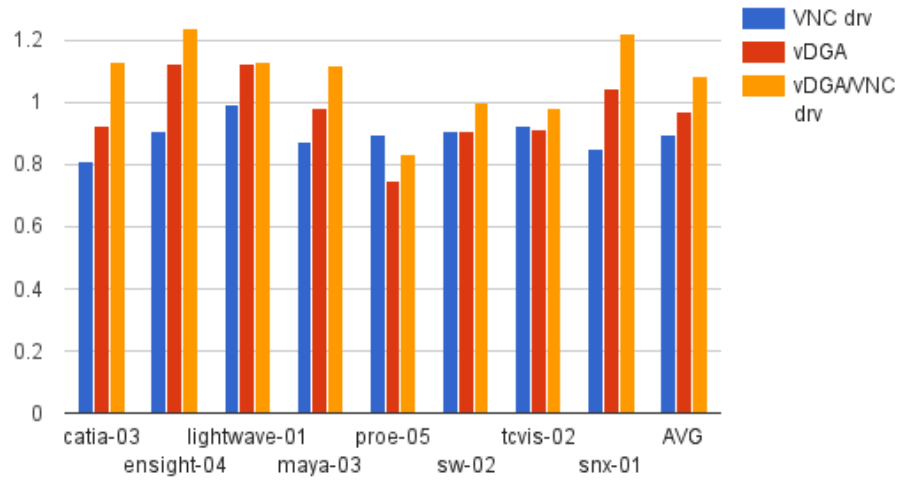


Figure 4.11: VMware Esx speedup over Windows

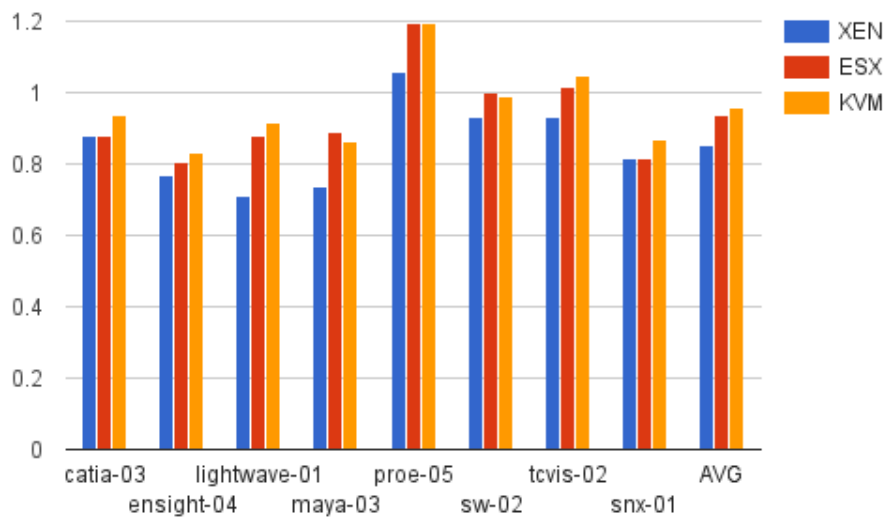


Figure 4.12: PCoIP vs VNC

generally outperforms KVM and XEN with VNC. Figure 4.12 shows a comparison between these three technologies with VNC over PCoIP in ESX. Here KVM is the closest to PCoIP which achieves a 96% of its performance, improving VMware results with VNC.

# 5

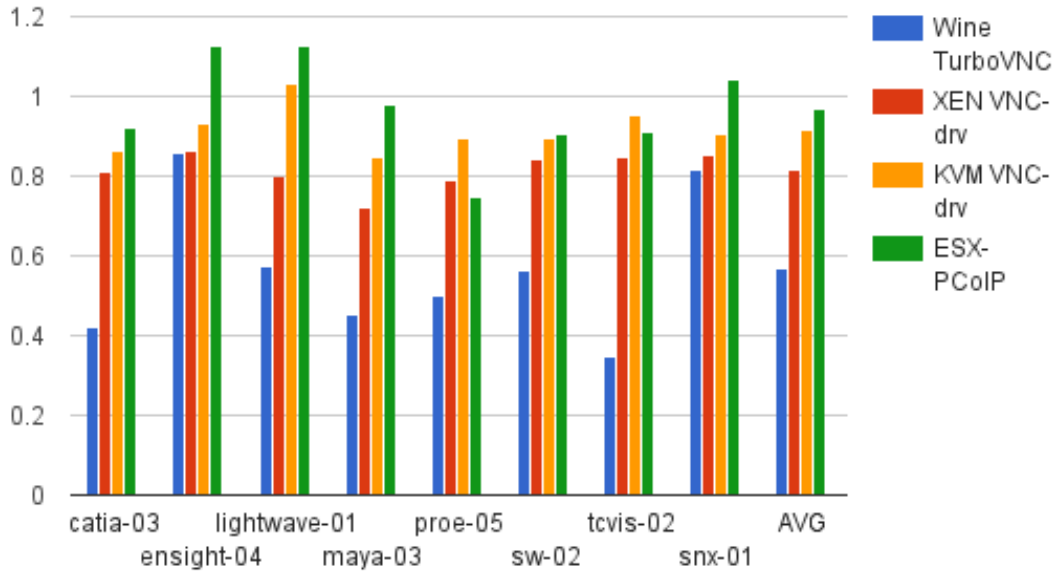
## Conclusion

This thesis has investigated 3D graphics performance problems on remote virtual applications and studied the constraints and limitations produced by virtualization and remote desktop technologies when working together. This study is conducted through several case studies performed on Windows applications running on Unix-like systems which covered a wide range of the latest technology implementations in this field. The following paragraphs summarize these findings and aim to serve as a guideline for system engineers to build a high quality infrastructure for HPC users. Here some boundaries and limitations of the actual technology are highlighted in order to address new research lines in this area. This work also included the implementation of a tool for on-demand virtual machine setup with automatic handling of PCI devices for XEN and KVM. It allows to create fast dynamic configurations of virtual machines with PCI pass-through devices.

Case studies results have shown that there is no universal solution for providing 3D graphics to remote virtual applications where the application itself is the main factor which drives design and technology selection for the whole infrastructure. Thus, supporting any application or making a bad estimation of application requirements leads to an inefficient or nonperforming system which highly increases service costs. This is due to the performance overhead and the problems of sharing a physical graphic card between multiple users when running remote virtual applications. Both graphic performance and GPU sharing limitations are provoked by many components but the virtual GPU technology and the remote desktop technology have shown to play a mayor role on them.

Results have shown that virtual GPUs are the ones which drive graphics performance and GPU sharing capabilities of virtual instances. This way XEN, KVM and VMware ESX scored similar results with PCI pass-through, the only virtual GPU for Windows supported by KVM and open source XEN, which showed that the virtualization technology itself is not an important factor in graphic performance. This type of

virtual GPU achieved the best performance results, a 90% overall of native performance but at the expense of maintaining another operating system and dedicating one graphic card per virtual machine. Individual application scores highlighted that applications with more GPU consumption were the ones that achieved better grades which showed that the main bottleneck was not in the virtual GPU itself. However OpenCL memory bandwidth test determined that PCI pass-through suffers a 25% overhead in CPU-GPU memory transfers compared with a native OS which reveals one of the main bottlenecks of this technology. Again all three hypervisors scored similar values in memory transfers. However results of these hypervisors considerably change when applications are run remotely. Figure 5.1 shows the speedup of all solutions which were able to run spec viewperf benchmark including XEN and KVM with VNC+driver and ESX with PCoIP. Results differences have shown to be both due to the virtualization technology and remote desktop protocol. Whereas XEN with VNC performs an 10% worse than KVM and ESX which suffer almost no overhead due to the remote desktop protocol with the same VNC implementation than XEN. On the other hand ESX with PCoIP outperforms VNC version by a 7% achieving a 97% of native Windows VNC performance which is quite impressive.



**Figure 5.1:** Speedup over native Windows with VNC driver

Any other form of virtual GPU in virtual machines showed to be impractical for HPC, either limited by memory (vSGA) or 3D performance (software 3D). However VMware vSGA was able to take advantage of hardware acceleration but it did not perform well due to graphic memory limitations (256 MB). Thus an improvement in terms of memory can make it a better alternative than PCI pass-through for several reasons: (a) it allows

sharing the GPU between virtual machines, (b) it does not bind a virtual machine to a physical GPU so the GPU may be changed for another one, (c) it allows the virtual machine to be migrated and (d) it does not have PCI pass-through security problems. Other hardware accelerated virtual GPUs that allow to share the GPU between virtual machines and haven't been evaluated like VMGL could also have been a good solution for OSes with X display system. However their study has been left for future work because Windows does not support them. On the other hand, hosted hypervisors like VirtualBox and VMware Workstation which are less intrusive than bare metal hypervisors, lack of PCI pass-through (due to malfunction or absence) and any other 3D graphics alternative.

Although PCI pass-through achieved the best results, it came with several maintenance drawbacks. It is a quite intrusive technology where no application usually runs with the hypervisor and everything is virtualized, even the host OS applications which run on a virtual instance of the host OS. Further it is necessary to maintain the virtual instances of the guest OSes. However it is way easier to maintain a virtual OS than a bare metal one because in a virtual OS almost all important tasks carried out by an OS like hardware maintenance, drivers, users, permissions, file systems etc are handled by the host OS or the hypervisor. Another big drawback was the security implications of PCI pass-through technology that in XEN and KVM lead to system crashes when booting or powering-off the virtual machine. Regarding this thesis, it considered that this issue, although important, should not be a major technology discriminant but a factor to take into account when building and managing PCI pass-through instances.

In contrast to hardware virtualization, application virtualization showed to be a good alternative in several scenarios. WINE, the only application virtualization technology tested, showed to perform well with several applications but it did quite worse with others. OpenCL tests showed that there is no memory overhead compared to native Linux and OpenGL calls go directly to the graphic card so WINE overhead seems to be due to its implementation of Windows libraries. This made WINE too much application dependant and to be infeasible for any scenario where target applications cannot be tested beforehand. Moreover it may not even work with some applications. Thus a further virtualization solution may have to be maintained besides WINE in order to support any application, which can be a big drawback. In this scenario, KVM or XEN can be good choices because they use Linux as host OS like WINE. However WINE seems to be the best choice for any application with which it performs well enough because it allows to share the graphic card between multiple users increasing GPU usage and saving hardware costs. Further it is the least intrusive solution, integrating Windows applications seamlessly with Linux ones.

In a similar way than virtual GPUs, remote desktop applications also constraint graphics performance and GPU sharing between users although their implementation is highly dependant on the operating system they run. In Windows, these protocols fixed to one the number of users that could take advantage of a virtual machine and in

Linux they required a further processing layer in order to share the GPU, this time an OpenGL wrapper (VirtualGL). Moreover, remote desktop applications introduced important yield losses that have shown to be application and technology dependant where the biggest overhead was shown in WINE, up to 50% in maya application with VNC. This overhead was bigger in Windows than in Linux due to Linux client-server display model avoided the need to poll the framebuffer but the display server directly. Some VNC Windows overhead have been mitigated by VMware PCoIP taking advantage of Nvidia API instead of wrapping the video signal. Although it didn't improve performance in all applications, it has shown to perform a 5% better in average. On the other hand SSH in Linux achieved even better results remotely than in local session due to it is able to lever 2D rendering to the SSH client.

## 5.1 Future work

WINE's performance have shown to be too application dependant. Thus a better characterization of WINE libraries may show which library implementations produce more overhead. Further, OpenGL library calls may perform differently so a custom implementation of some OpenGL functions for WINE may greatly improve graphics performance.

Another bottleneck found in case studies was the overhead of memory transfers between the CPU and the GPU in virtual machines. There are different techniques like shared memory, zero-copy or other technologies like RDMA in network cards which avoid unnecessary memory copies and avoid user-space to kernel-space copies which are really expensive. Studying how to apply these concepts on graphic cards or virtual GPUs may considerably improve pass-through performance.

# Appendices

# A

## VM deployment tool pseudo-code

This tool consist of three scripts. The main script is *run-vm* which connects to a remote host, starts the virtual machine with the script *start-vm*, connects to it through VNC and after the VNC session is closed, it destroys the virtual machine with the script *destroy-vm*.

### A.1 run vm

```
# Start virtual machine
args = getArgs()
remoteHost = args[1]
connect(remoteHost)
vm_name, ip, port = call("startVM", args[2..N])
disconnect()

# Connect to virtual machine
call("vncviewer", concat(ip, ":", port))

# Free virtual machine resources
connect(remoteHost)
call("destoryVM", vm_name, ip, port)
disconnect()
```

**Listing A.1:** run virtual machine

### A.2 start vm

```
# Create parser and add options
parser = createOptionParser()
addOption("--master", "Master/parent virtual machine")
addOption("--conf-file", "Configuration file")
addOption("--vm-image", "Virtual machine image")
```

```
addOption("--redir", "Port redirection")
addOption("--gpu", "GPU name, model or characteristic")
addOption("--pci-device", "Comma separated list of PCI devices")
addOption("--vcpus", "Number of virtual GPUs")
addOption("--memory", "Memory in MBs")
addOption("--net", "Network type: nat | bridge=<bridge-name> | net=<network-name>")

# Parse options through OptionParser
options = parseOptions(parser)

# Get configuration file
if getOption(options, "master") then
    master = getOption(options, "master")
    conf_file = getVMConfig(master)
    xml_tree = openXML(conf_file)
else
    conf_file = getOption(options, "conf-file")
    xml_tree = openXML(conf_file)
    master = xmlGetXpath(xml_tree, '/domain/name')
endif

# Domain extra configuration options
config = ""

# Get virtual machine image
if not getOption(options, "vm-image") then
    # Get image form configuration file
    vm_image = xmlGetXpath(xml_tree, "/domain/devices/disk[@device='disk']/source")
endif

# Get virtualization technology i.e virtualization driver
driver = xmlGetXpath(xml_tree, "/domain/type")

# Get network configuration
if getOptions(options, "net") == then
    vnc_net = getOptions
else
    vnc_net = xmlGetXpath(xml_tree, "/domain/devices/interface")
endif

# Get PCI devices to pass-through
devs = getOption(options, "pci-device")

# Get PCI devices in use
vms = getVMs()
for vm in vms do
    devices = getDevices(vm)
    add(used_devices, devices)
endfor

# Check PCI devices not in use
for dev in devs do
    if dev in used_devices then
        print("error device in use")
    end
end
```



```
        exit()
    endif
endfor

# Find GPUs
found = false
if getOption(options, gpu) in gpu
    gpus = filter(getAllPciDevices(), "VGA")
    for gpu in gpus:
        if gpu not in used_devices then
            iommu_devices = filter(getAllPciDevices(), getIommuGroup(gpu))
            add(devs, gpu)
            add(iommu_devs, iommu_devices)
            found = true
            break
        endif
    endfor
endif

if getOption(options, gpu) AND not found then
    print "Error, free GPU not found"
    exit()
endif

# Add PCI devices, GPUs and GPU iommu devices to pci driver
for dev in join(devs, iommu_devs) do
    detachDeviceDriver(dev)
    if driver == KVM then
        bind-vfio(dev)
    else
        bind-xen-pciback(dev)
    endif
endfor

# Add devices to configuration
if devs then
    config = concat(config, "--pci-device", arrayToString(devs))
endif

# Configure networking and port redirection

vm_mac = getRandomMAC()
net = getOption(options, net)
if not net then
    net = getXpath(xml_tree, "/domain/devices/interface/type")
endif

if net == "nat" OR startsWith(net, "net") then
    if getOption(options, redir_port) then
        vnc_port = getOption(options, redir_port)
    else
        vnc_port = getFreeHostPort()
    endif
endif
```

```
# No --redir option for XEN
if net == "nat" AND driver == "KVM" then
    config = concat(config, "--redir ", vnc_port)
else
    vnc_network = getIndex(split(net, "="), 2)
    if not vnc_network then
        vnc_network = "default"
        net="net=default"
    endif
    # Get the system bridge name from the virtual network name
    vnc_bridge = getBridgeFromNetwork(vnc_network)
endif
config = concat(config, "--net ", net)
else
    vnc_bridge = split(net, "=")[2]
    config = concat(config, "--net bridge=", "vnc_bridge")
endif

# Configure other port redirections
if getOption(option, redir) then
    config = concat(config, "--redir", getOption(option, redir))
endif

# Generate vm name
vm_name = concat(master, "-", vm_mac)

# Clone hard disk
if getOption(options, "vm-image") then
    new_image = getOption(options, "vm-image")
else
    new_image = concat(vm_image, "-", vm_mac)
endif

if driver == KVM then
    clone_kvm(vm_image, new_image)
else
    if format(vm_image) == "vhd" then
        clone_xen(vm_image, new_image)
    else
        copy(vm_image, new_image)
    endif
endif

# Mount windows image with guestmount
guestmount(new_image, "mount_folder")

# Customize tight vnc password in windows registry
vnc_file=open(".vnc/passwd")
hex_passwd=readLine(vnc_file)
regfile=open("vnc-pass.reg")
writeLine(regfile, '[HKEY_LOCAL_MACHINE\SOFTWARE\TightVNC\Server]')
writeLine(regfile, concat('Password=hex(3):', hex_passwd))
```

```
# Merge windows registry with hivexregedit
hivexregeditMerge("mount_folder", regfile)

# Customize machine hostname in windows registry (more complex)
changeWindowsHostname("mount_folder", vm_name)sh

# Umount windows image
guestunmount("mount_folder")

# Generate new configuration file
if getOption(options, "vcpu") then
    config = concat(config, "--vgpu", getOption(options, "vcpu"))
endif

if getOption(options, "memory") then
    config = concat(config, "--memory", getOption(options, "memory"))
endif

new_conf_file = concat(conf_file, "-", vm_mac)
modifyDomain(conf_file, new_conf_file, config)

# Start and connect to the virtual machine
startVM(new_conf_file)

# Wait for machine to start to get IP
max_count = 15
if startsWith(net, "bridge") OR startsWith(net, "net") then
    bridge_ip = getBridgeIP(vnc_bridge)
    for count=0 to max_count do
        # Get all active IPs in bridge network
        ips = getAllIPs(bridge_ip)
        for ip in ips do
            if getMac(ip) == vm_mac then
                vm_ip = ip

                if startsWith(net, "bridge") then
                    # VM ip is public
                    conn_port = 5000
                    conn_ip = vm_ip
                else
                    # Configure firewall to redirect port to private network
                    conn_port = vnc_port
                    conn_ip = getHostname()
                    vm_port = 5000
                    if firewall == firewalld then
                        firewalld-add-forward_port(conn_port, vm_ip, vm_port)
                    else
                        yast_firewall_masqredirect(conn_port, vm_ip, vm_port)
                    endif
                endif
            endif
        endfor
        break
    endfor
endif
```

```

        endfor
        sleep(1)
    endfor
else
    for count=0 to max_count do
        if isPortOpen(vnc_port) then
            break
        else
            sleep(1)
        endif
    endfor
    conn_port = vnc_port
    conn_ip = getHostname()
endif

# Print vm configuration
printf("%s %s %s\n", vm_name, conn_ip, conn_port)

```

**Listing A.2:** start virtual machine

### A.3 destroy vm

```

# Parse args
vm_name=args[1]
ip=args[2]
port=args[3]

config = getVMConfig(vm)

# Remove firewall rules
net = getConfig(config, "network")
if startsWith(net, "net") then
    vm_ip = getVMIP()
    if firewall == firewalld then
        firewalld_remove_forward_port(port, vm_ip, 5000)
    else
        yast_firewall_remove_masqredirect(port, vm_ip, 5000)
    endif
endif

# Shutdown virtual machine
status=getStatus(vm_name)
if status == "running" then
    shutdown(vm_name)
    timeout = 15
    for t = 0 to timeout do
        status = getStatus(vm_name)
        if status == shutdown then
            break
        endif
    endfor
    destroy(vm_name)
endif

```

### A.3. DESTROY VM APPENDIX A. VM DEPLOYMENT TOOL PSEUDO-CODE

```
# Detach PCI devices
for dev in getConfig(config, "pci-devices") do
    detachDeviceDriver(dev)
endfor

# Remove virtual machine image and
vm_image = getConfig(config, "disk")
remove_file(vm_image)
removeVM(vm_name)
```

**Listing A.3:** destroy virtual machine

# B

## Test Results

This appendix shows the actual results for all graphs in this thesis. These results are displayed in tables that are organized in sections. There is one section for each evaluated virtualization platform and there are also sections for native Windows and native Linux. Moreover, tables may display more information than the one showed in the graphs due to graphs are meant to display relevant information and should be easy to understand whereas these tables show all results for further reference.

### B.1 Linux

Table B.1 shows the speedups of remote desktop protocols compared to native Linux.

	x11vnc	turboVNC	ssh	xrdp	nx
catia-03	0.943	0.959	1.013	1.023	0.583
ensight-04	0.899	0.943	0.9947	0.959	0.550
lightwave-01	0.958	0.726	0.842	0.805	0.532
maya-03	0.934	0.775	1.156	1.015	0.457
proe-05	0.919	0.861	0.856	0.863	0.649
sw-02	0.943	0.861	0.992	0.936	0.490
tcvis-02	0.949	1.079	1.237	1.191	0.641
snx-01	0.920	0.987	1.113	1.041	0.558
AVG	0.933	0.899	1.025	0.979	0.558

**Table B.1:** Remote desktop application performance on Linux (Figure 4.1)

## B.2 Windows

Table B.2 shows raw (without VNC), VNC and VNC with VNC driver results for Windows. Speedups of VNC and VNC-driver are compared to raw results. Average results are only displayed for the speedups because frame-rates averages for different applications do not provide valuable information. On the other side, table B.3 compares Linux (L) and Windows (W) results. Each speedup (SP) column is calculated dividing the two previous columns (Linux column/ Windows column) but the last one which is calculated dividing TurboVNC (Linux) between W-VNC.

Windows	Raw	VNC	Speedup	VNC-driver	Speedup
catia-03	40.04	40.01	0.999	35.54	0.887
ensight-04	30.07	30.01	0.998	23.93	0.796
lightwave-01	48.11	48.10	1.000	30.42	0.632
maya-03	60.55	60.49	0.999	44.47	0.735
proe-05	9.38	9.46	1.008	8.63	0.920
sw-02	38.68	38.67	1.000	32.15	0.831
tcvis-02	34.49	34.38	0.997	26.67	0.773
snx-01	32.35	32.26	0.997	28.12	0.869
Average			1.000		0.806

**Table B.2:** Windows performance (Figure 4.2)

	W	L	SP	W-VNC	L-VNC	SP	TurboVNC	SP
catia-03	40.04	34.07	0.85	35.54	32.15	0.90	32.68	0.92
ensight-04	30.07	24.59	0.82	23.93	22.11	0.92	23.21	0.97
lightwave-01	48.11	42.74	0.89	30.42	40.98	1.35	31.04	1.02
maya-03	60.55	52.66	0.87	44.47	49.23	1.11	40.84	0.92
proe-05	9.38	11.29	1.20	8.63	10.39	1.20	9.73	1.13
sw-02	38.68	35.17	0.91	32.15	33.19	1.03	30.31	0.94
tcvis-02	34.49	27.12	0.79	26.67	25.74	0.96	29.29	1.10
snx-01	32.35	26.29	0.81	28.12	24.21	0.86	25.96	0.92
AVG			0.89			1.04		0.99

**Table B.3:** Linux vs Windows performance (Figure 4.3)

## B.3 WINE

Table B.4 compares Linux (L) and WINE (W) results where SP is the speedup between last WINE result and the last Linux result. Last two columns represent Linux VNC speedup (L/L-VNC) and WINE VNC speedup (W/W-VNC) respectively. On the other hand table B.5 shows WINE results without VNC (Raw) and with multiple turboVNC sessions from 1 to 8. Further table B.6 shows different WINE speedup results for multi-session tests where the ‘X’ that goes with ‘vncX’ is the number of concurrent turboVNC sessions.

	Linux	WINE	Sp	L-VNC	W-VNC	Sp	L-VNC SP	W-VNC SP
catia-03	34.070	17.045	0.500	32.675	15.040	0.460	0.959	0.882
ensight-04	24.587	24.195	0.984	23.205	20.570	0.886	0.944	0.850
lightwave-01	42.743	25.530	0.597	31.035	17.385	0.560	0.726	0.681
maya-03	52.660	42.490	0.807	40.835	20.195	0.495	0.775	0.475
proe-05	11.290	4.635	0.411	9.730	4.310	0.443	0.862	0.930
sw-02	35.173	23.575	0.670	30.305	18.130	0.598	0.862	0.769
tcvis-02	27.117	10.010	0.369	29.285	9.300	0.318	1.080	0.929
snx-01	26.287	24.930	0.948	25.960	23.005	0.886	0.988	0.923
Average			0.661			0.581	0.899	0.805

**Table B.4:** WINE performance (Figure 4.4)

	Raw	Turbo x1	Turbo x2	Turbo x4	Turbo x8
catia-03	17.045	15.040	12.955	9.755	4.376
ensight-04	24.195	20.570	11.245	1.773	0.638
lightwave-01	25.530	17.385	16.345	13.473	9.841
maya-03	42.490	20.195	18.820	11.040	4.363
proe-05	4.635	4.310	3.755	3.258	2.251
sw-02	23.575	18.130	16.120	11.785	6.541
tcvis-02	10.010	9.300	8.060	6.433	1.574
snx-01	24.930	23.005	13.505	4.470	1.068

**Table B.5:** WINE multi-session performance



	vnc1	vnc2/vnc1	vnc4/vnc1	vnc8/vnc1	vnc4/vnc2	vnc8/vnc4
catia-03	1	0.861	0.649	0.291	0.753	0.449
ensight-04	1	0.547	0.086	0.031	0.158	0.360
lightwave-01	1	0.940	0.775	0.566	0.824	0.730
maya-03	1	0.932	0.547	0.216	0.587	0.395
proe-05	1	0.871	0.756	0.522	0.868	0.691
sw-02	1	0.889	0.650	0.361	0.731	0.555
tcvis-02	1	0.867	0.692	0.169	0.798	0.245
snx-01	1	0.587	0.194	0.046	0.331	0.239

**Table B.6:** WINE multi-session speedup (Figure 4.5)

## B.4 Virtual machine monitors

Table B.7 shows results of VMware workstation, VirtualBox and Linux.

	VMWARE	VirtualBox	LINUX
catia-03	0.29	0	32.86
ensight-04	0.21	0	24.45
lightwave-01	4.87	0	37.54
maya-03	0.99	0	51.26
proe-05	0.09	0	9.5
sw-02	0.93	0	29.69
tcvis-02	0.05	0	26.24
snx-01	0.38	0	26.24

**Table B.7:** VMware workstation and Virtual Box performance on Linux (Figure 4.6)

## B.5 XEN

Table B.8 shows speedups of Windows virtual machines running in XEN compared to native Windows while table B.9 displays Linux speedups of XEN virtual machines compared with native Linux.

	No VNC SP	VNC SP	VNC-driver SP
catia-03	0.945	0.944	0.812
ensight-04	0.990	0.987	0.866
lightwave-01	0.833	0.826	0.798
maya-03	0.774	0.765	0.722
proe-05	0.785	0.795	0.791
sw-02	0.903	0.900	0.844
tcvis-02	0.996	0.992	0.847
snx-01	0.967	0.968	0.851
AVG	0.899	0.897	0.816

**Table B.8:** Xen performance (Figure 4.7)

	x11vnc	turboVNC
catia-03	0.941	0.899
ensight-04	1.004	0.981
lightwave-01	0.847	0.901
maya-03	0.856	0.818
proe-05	0.776	0.828
sw-02	0.859	0.933
tcvis-02	0.963	0.952
snx-01	0.961	0.949
AVG	0.901	0.908

**Table B.9:** Linux XEN speedup ( Figure 4.8)

## B.6 KVM

Table B.10 shows speedups of vfio and pcistub drivers in KVM with and without VNC compared with its native Windows counterpart. Further, last column displays vfio Linux X11vnc speedup compared to native Linux.

	vfio-raw	vfio-vnc	pcistub-raw	pcistub-vnc	Linux-x11vnc
catia-03	0.933	0.863	0.922	0.788	1.003
ensight-04	0.966	0.933	0.947	0.865	0.986
lightwave-01	0.832	1.033	0.832	0.879	0.913
maya-03	0.816	0.847	0.813	0.791	0.952
proe-05	0.829	0.896	0.788	0.739	0.963
sw-02	0.944	0.896	0.925	0.857	0.933
tcvis-02	0.992	0.954	0.912	0.864	0.999
snx-01	0.961	0.907	0.955	0.845	0.950
AVG	0.909	0.916	0.887	0.828	0.962

**Table B.10:** KVM speedup (Figure 4.9)

## B.7 VMware ESX

Table B.11 shows results of software 3D virtual GPU in VMware ESX compared to vSGA virtual GPU. The speedup is calculated dividing vSGA results by Software 3D results. On the other hand, table B.12 compares VMware ESX results with vDGA virtual GPU results. Speedup is calculated dividing vDGA results by pass-through ones. Finally table B.13 displays speedups of XEN, VMware ESX and KVM with VNC compared to VMware ESX with PCoIP.

Resolution	800x600			1600x1200		
vGPU	Soft 3D	vSGA	SP	Soft 3D	vSGA	SP
proe-05	0.095	2.21	2.21	0.095	0.84	0.84
sw-02	0.36	2.86	2.86	0.19	0.78	0.78
tcvis-02	0.09	1.66	1.66	0.09	0.77	0.77
snx-01	0.06	2.16	2.16	0.04	0.5	0.5

**Table B.11:** Software 3D vs vSGA (Figure 4.10)

	Pass-through	vDGA	Speedup
catia-03	28.86	32.8	1.13
ensight-04	21.68	26.905	1.24
lightwave-01	30.135	34.22	1.13
maya-03	38.695	43.58	1.12
proe-05	7.71	6.465	0.83
sw-02	29.14	29.17	1
tcvis-02	24.685	24.315	0.98
snx-01	23.955	29.355	1.22

**Table B.12:** Esx speedup (4.11)

	XEN	ESX	KVM
catia-03	0.880	0.880	0.934
ensight-04	0.770	0.806	0.830
lightwave-01	0.709	0.881	0.918
maya-03	0.737	0.888	0.864
proe-05	1.056	1.193	1.196
sw-02	0.930	0.999	0.988
tcvis-02	0.929	1.015	1.046
snx-01	0.815	0.816	0.869
AVG	0.853	0.935	0.956

**Table B.13:** PColP vs VNC (Figure 4.12)

# Bibliography

- [1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGARCH Comput. Archit. News*, 34(5):2–13, October 2006. ISSN 0163-5964. doi: 10.1145/1168919.1168860. URL <http://doi.acm.org/10.1145/1168919.1168860>.
- [2] Christopher G. Willard Addison Snell. IBM deep computing visualization. Technical report, IDC, January 2005. URL [http://www-06.ibm.com/systems/jp/deepcomputing/pdf/idc\\_white\\_paper.pdf](http://www-06.ibm.com/systems/jp/deepcomputing/pdf/idc_white_paper.pdf).
- [3] David Airlie. Virgil3d - a virtio based 3D GPU. Technical report, Red Hat, 2013.
- [4] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA 10, pages 256–274, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-24321-9. doi: 10.1007/978-3-642-24322-6\_22.
- [5] 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. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945462. URL <http://doi.acm.org/10.1145/1165389.945462>.
- [6] Aaron Blasius, Pat Lee, Warren Ponder, Joel Lindberg, Rasmus Jensen, Josh Spencer, Tommy Walker, and Vincent Wu. Virtual machine graphic acceleration deployment guide. Technical report, VMware, 2013. URL <https://www.vmware.com/files/pdf/techpaper/vmware-horizon-view-graphics-acceleration-deployment.pdf>.
- [7] Yu-Ling Chang, Denis Yuen, and kah kuen. Wine software architecture. Master’s thesis, University of Waterloo, 2009.
- [8] Darrell Commander. From tight to turbo and back again: Designing a better encoding method for turboVNC, 2014. URL <http://www.turbovnc.org/pmwiki/uploads/About/tighttoturbo.pdf>.

- [9] Darrell Commander. A study of the performance of virtualGL 2.1 and turboVNC 0.4, 2014. URL <http://www.virtualgl.org/pmwiki/uploads/About/vglperf21.pdf>.
- [10] Darrell Commander. Virtualgl in-depth background, 2014. URL <http://www.virtualgl.org/About/Background>.
- [11] Standard Performance Evaluation Corporation. Spec viewperf11, 2010. URL <https://www.spec.org/gwpg/gpc.static/vp11info.html>.
- [12] Stats counter. Desktop operating systems share, 2015. URL <http://gs.statcounter.com/#desktop-os-ww-monthly-201411-201510-bar>.
- [13] Erlon R. Cruz, Sandro Rigo, Fabiano Fidêncio, and Breno Leitao. Virtioqxl: a virtio video device for KVM guests, 2014.
- [14] Micah Dowty and Jeremy Sugerman. GPU virtualization on VMware’s hosted I/O architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009. ISSN 0163-5980. doi: 10.1145/1618525.1618534. URL <http://doi.acm.org/10.1145/1618525.1618534>.
- [15] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters, 2003.
- [16] ETSI. Telecommunications and internet converged services and protocols for advanced networking (TISPAN); review of available material on QoS requirements of multimedia services. Technical report, ETSI, February 2006. URL [http://www.etsi.org/deliver/etsi\\_tr/102400\\_102499/102479/01.01.01\\_60/tr\\_102479v010101p.pdf](http://www.etsi.org/deliver/etsi_tr/102400_102499/102479/01.01.01_60/tr_102479v010101p.pdf).
- [17] Matheus Santos Virgilio Almeida Jussara Almeida Fabricio Benevenuto, Cesar Fernandes. A quantitative analysis of the XEN virtualization overhead. Technical report, Federal University of Minas Gerais, Brazil, 2010.
- [18] Abel Gordon, Nadav Amit, Nadav Har’El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafir. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, pages 411–422, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151020. URL <http://doi.acm.org/10.1145/2150976.2151020>.
- [19] Eddie Dong Haitao Shan, Kevin Tian. XenGT: a software based Intel graphics virtualization solution. Technical report, Intel, October 2013.
- [20] Jacob Gorm Hansen. Blink: Advanced display multiplexing for virtualized applications, 2007.

- [21] Red Hat. Spice remote computing protocol definition. Technical report, Red Hat, 2009.
- [22] Alex Herrera. NVIDIA GRID: graphics accelerated vdi with the visual performance of a workstation. Technical report, nvidia, 12 2013. URL <http://www.nvidia.com/content/grid/vdi-whitepaper.pdf>.
- [23] HP. Advantages and implementation of HP remote graphics software. Technical report, HP, May 2007. URL [http://h20331.www2.hp.com/Hpsub/downloads/hp\\_remotegraphics.pdf](http://h20331.www2.hp.com/Hpsub/downloads/hp_remotegraphics.pdf).
- [24] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 693–702, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi: 10.1145/566570.566639. URL <http://doi.acm.org/10.1145/566570.566639>.
- [25] ITU-T. Series g: Transmission systems and media, digital systems and networks; international telephone connections and circuits – general recommendations on the transmission quality for an entire international telephone connection. Technical report, ITU-T, May 2003.
- [26] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hossfeld. An evaluation of QoE in cloud gaming based on subjective tests. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pages 330–335, June 2011. doi: 10.1109/IMIS.2011.92.
- [27] H. Andrés Lagar-cavilla and M. Satyanarayanan. Vmm-independent graphics acceleration. In *In Proceedings of VEE 2007*. ACM Press, 2007.
- [28] Michael Larabel. Linux gaming: Native vs. WINE vs. Windows 7 performance. Technical report, Phoronix, Dec 2010. URL [http://www.phoronix.com/scan.php?page=article&item=wine\\_win7\\_2010&num=1](http://www.phoronix.com/scan.php?page=article&item=wine_win7_2010&num=1).
- [29] Michael Larabel. NVIDIA vs. Nouveau drivers with Linux 3.18 + Mesa 10.4-devel. Technical report, Phoronix, November 2014.
- [30] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010. doi: 10.1109/IPDPS.2010.5470365.
- [31] Ramesh Chandra Nickolai Zeldovich. Interactive performance measurement with VNCplay. In *USENIX 2005 Annual Technical Conference*. Computer Science Department, Stanford University, April 2005. URL <http://suif.stanford.edu/vncplay/freenix05-html/>.



- [32] Zack Rusin. Gallium3d - graphics done right. Technical report, Tungsten Graphics, 2008.
- [33] Jim Salter. Benchmarking Windows guests on KVM:I/O performance. Technical report, JRS Systems, May 2013. URL <http://jrs-s.net/2013/05/17/kvm-io-benchmarking>.
- [34] R. Shea and Jiangchuan Liu. On GPU pass-through performance for cloud gaming: Experiments and analysis. In *Network and Systems Support for Games (NetGames), 2013 12th Annual Workshop on*, pages 1–6, Dec 2013. doi: 10.1109/NetGames.2013.6820614.
- [35] Claudio Tanci. GPU computing on virtual machines a feasibility study, April 2011.
- [36] Xen Project Security Team. Non-standard PCI device functionality may render pass-through insecure. Technical report, XEN, 2015. URL <http://xenbits.xen.org/xsa/advisory-124.html>.
- [37] Jean David techer. XEN 4.2.unstable: Patches/notes for VGA pass through and NVIDIA, 2011. URL <http://www.davidgis.fr/blog/index.php?2011/12/07/860-xen-%2042unstable-patches-for-vga-pass-through>.
- [38] Hirt Timo. KVM - the kernel-based virtual machine, 2010.
- [39] TRANSGAMING. Swiftshader: Why the future of 3D graphics is in software. Technical report, TRANSGAMING, January 2013.
- [40] Peter Senna Tschudin. Performance overhead and comparative performance of 4 virtualization solutions, 2012.
- [41] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. High-performance cloud computing: A view of scientific applications. In *Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, ISPAN '09*, pages 4–16, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3908-9. doi: 10.1109/I-SPAN.2009.150. URL <http://dx.doi.org/10.1109/I-SPAN.2009.150>.
- [42] Ram Rajamony Juan Rubio Wes Felter, Alexandre Ferreira. An updated performance comparison of virtual machines and linux containers. Technical report, IBM Research, Austin, TX, July 2014.
- [43] Alex Williamson. VGA assignnet using VFIO. Technical report, Red Hat, October 2013. URL <http://www.linux-kvm.org/wiki/images/e/ed/Kvm-forum-2013-VFIO-VGA.pdf>.
- [44] Chao-Tung Yang, Hsien-Yi Wang, and Yu-Tso Liu. Using PCI pass-through for GPU virtualization with CUDA. In JamesJ. Park, Albert Zomaya, Sang-Soo Yeo, and Sartaj Sahni, editors, *Network and Parallel Computing*, volume 7513 of *Lecture*

*Notes in Computer Science*, pages 445–452. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35605-6. doi: 10.1007/978-3-642-35606-3\_53. URL [http://dx.doi.org/10.1007/978-3-642-35606-3\\_53](http://dx.doi.org/10.1007/978-3-642-35606-3_53).