VMM-Independent Graphics Acceleration [*]

H. Andrés Lagar-Cavilla
University of Toronto
andreslc@cs.toronto.edu

Niraj Tolia
Carnegie Mellon University
ntolia@cmu.edu

M. Satyanarayanan
Carnegie Mellon University
satya@cs.cmu.edu

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

Abstract

This paper describes VMGL, a cross-platform OpenGL virtualization solution that is both VMM and GPU independent. VMGL allows applications executing within virtual machines (VMs) to leverage hardware rendering acceleration, thus solving a problem that has limited virtualization of a growing class of graphics-intensive applications. VMGL also provides applications running within VMs with suspend and resume capabilities across GPUs from different vendors. Our experimental results from a number of graphics-intensive applications show that VMGL provides excellent rendering performance, coming within 14% or better of native graphics hardware acceleration. Further, VMGL's performance is two orders of magnitude better than that of software rendering, the commonly available alternative today for graphics-intensive applications running in virtualized environments. Our results confirm VMGL's portability across VMware Workstation and Xen (on VT and non-VT hardware), and across Linux (with and without paravirtualization), FreeBSD, and Solaris. Finally, the resource demands of VMGL align well with the emerging trend of multi-core processors.

Categories and Subject Descriptors I.3.4 [*Computer Graphics*]: Graphics Utilities,Virtual Device Interfaces; C.5.m [*Computer Systems Implementation*]: Miscellaneous

General Terms Design, Performance, Experimentation

Keywords Virtualization, Graphics, Hardware Acceleration, Portability, VMM-independence

[*] This research was supported by the National Science and Engineering Research Council (NSERC) of Canada under grant number 261545-3 and a Canada Graduate Scholarship, by the Canadian Foundation for Innovation (CFI) and the Ontario Innovation Trust (OIT) under grant number 7739, by the National Science Foundation (NSF) under grant number CNS-0509004, and by Google under the Summer of Code 2006 program.

1. Introduction

Virtual machine monitor (VMM) technology has been put to many innovative uses, including mobile computing [16, 28, 40], system management [17, 39], intrusion detection [18], and grid computing [20]. However, the difficulty of virtualizing graphical processing units (GPUs) has so far limited the use of virtual machines (VMs) for running interactive applications. The performance acceleration provided by GPUs is critical to high-quality visualization in many applications, such as computer games, movie production software, computer-aided design tools for engineering and architecture, computer-aided medical diagnosis, and scientific applications such as protein modeling for drug synthesis. For this class of applications, software rendering is the prevalent option for virtualized execution, and it is unacceptably slow.

Virtualizing GPUs is difficult for a number of reasons. First, the hardware interface to a GPU is proprietary, and many technical details are closely held as trade secrets. Hence, it is often difficult to obtain the technical specifications necessary to virtualize a GPU. Second, because the hardware interface is not public, GPU vendors make significant changes in the interface as their product lines evolve. Trying to virtualize across such a wide range of interfaces can result in a weak lowest common denominator. Third, the software needed to integrate a GPU into an operating system is typically included with the hardware as a closed-source device driver. In a virtualized environment, the driver is unusable for other guest operating systems. For reasons mentioned earlier, the technical details necessary to create a new driver for other guests are typically not available. In summary, virtualization of a hardware component presumes the existence of a standard interface such as the x86 instruction set, or the IDE and SCSI interfaces to disks; GPUs lack such a standard.

This paper proposes a solution that is strongly influenced by how applications actually use GPUs. Many of the virtualization challenges discussed in the previous paragraph would also complicate the authoring of applications. For example, the absence of a stable GPU interface would require frequent application changes to track hardware. The large diversity of technical specifications across GPUs and the difficulty of obtaining them publicly would severely restrict the market size of a specific application implementation. The graphics community avoids these problems through the use of higher-level APIs that abstract away the specifics of GPUs. Practically all applications that use GPUs today are written to one or both of two major APIs: *OpenGL* [5] and *Direct3D* [33]. Of these two, OpenGL is the only cross-platform API supported on all major operating systems. For each supported operating system, a GPU vendor distributes a closed-source driver and OpenGL library. The job of tracking frequent interface changes to GPUs is thus delegated to the GPU vendors, who are best positioned to perform this task. Although OpenGL is a software interface, it has become a *de facto* GPU interface. We therefore make it the virtualization interface.

We describe *VMGL*, a virtualized OpenGL implementation that offers hardware accelerated rendering capabilities to applications running inside a VM. VMGL runs the vendor-supplied GPU driver and OpenGL library in the VMM host: the administrative VM for a hypervisor like Xen, or the hosting OS for a VMM like VMware Workstation. The host runs a vendor-supported operating system and has direct access to the GPU. Using a GL network transport, VMGL exports the OpenGL library in the host to applications running in other VMs. When those applications issue OpenGL commands, the commands are transported to the GPU-enabled host and executed there. VMGL thus preserves complete application transparency; no source code modification or binary rewriting is necessary. VMGL also supports suspend and resume of VMs running graphics accelerated applications, thus supporting several novel applications of VM technology [16, 28, 39]. Further, VMGL allows suspended VMs to be migrated to hosts with different underlying GPU hardware. VMGL is not critically dependent on a specific VMM or guest operating system, and is easily ported across them.

We evaluate VMGL for diverse VMMs and guests. The VMMs include Xen on VT and non-VT hardware, and VMware Workstation. The guests include Linux with and without paravirtualization, FreeBSD and Solaris. In experiments with four graphics-intensive applications, including one that is closed source, the observed graphics performance of VMGL comes within 14% or better of native performance, and outperforms software rendering by two orders of magnitude. Although this approach incurs the performance overhead of cross-VM communication, our experimental evaluation demonstrates that this overhead is modest. Moreover, our results also show that multi-core hardware, which is increasingly common, can help in reducing the performance overhead.

The rest of this paper describes the design, implementation and experimental validation of our VMGL prototype for OpenGL virtualization on X11-based systems. We begin in Section 2 with an overview of GPUs and OpenGL. Section 3 then describes the detailed design and implementation of VMGL. Section 4 presents the experimental validation of VMGL. Section 5 discusses related work and Section 6 concludes the paper and presents our plans for future work.

2. Background

In this section, we provide an introduction to graphics hardware acceleration and OpenGL, the most commonly used cross-platform 3D API. We also describe how X11-based applications leverage hardware acceleration capabilities. Readers familiar with these topics can skip ahead to Section 3.

2.1 Hardware Acceleration

Almost all modern computers today include a Graphics Processing Unit (GPU), a dedicated processor used for graphics rendering. GPUs have become increasingly popular as general purpose CPUs have been unable to keep up with the demands of the mathematically intensive algorithms used for transforming on-screen 3D objects, or applying visual effects such as shading, textures, and lighting. GPUs are composed of a large number of graphics pipelines (16–112 for modern GPUs) operating in parallel. For floating point operations, GPUs can deliver an order of magnitude better performance that modern x86 CPUs [22].

Modern GPUs range from dedicated graphics cards to integrated chipsets. As individual hardware implementations might provide different functionality, 3D graphics APIs have arisen to isolate the programmer from the hardware. The most popular APIs are OpenGL, an open and cross-platform specification, and Direct3D, a closed specification from Microsoft specific to their Windows platform. We describe OpenGL below in more detail.

2.2 OpenGL Primer

OpenGL is a standard specification that defines a platform-independent API for 3D graphics. The OpenGL API supports application portability by isolating developers from having to program for different hardware implementations.

Vendors implement the OpenGL API in the form of a dynamically loadable library that can exploit the acceleration features of their graphics hardware. All OpenGL implementations must provide the full functionality specified by the standard. If hardware support is unavailable for certain functions, it must be implemented in software. This isolates the programmer from having to determine available features at runtime. However, the OpenGL specification does allow for vendor-specific extensions; applications can only determine the availability of these extensions at runtime.

The OpenGL calls issued by an application modify the *OpenGL state machine*, a graphics pipeline that converts drawing primitives such as points, lines, and polygons into pixels. An *OpenGL context* encapsulates the current state of the OpenGL state machine. While an application may have multiple OpenGL contexts, only one context may be rendered on a window at a given time. OpenGL is strictly a rendering API and does not contain support for user input or windowing commands. To allow OpenGL contexts to interact with the window manager, applications use *glue* layers such as GLX for X11-based systems, WGL for Microsoft Windows, and AGL for the Macintosh.

Today, OpenGL is the only pervasive cross-platform API for 3D applications. The competing proprietary API, Microsoft's Direct3D, only supports the Windows operating systems. OpenGL implementations are available for Linux, Windows, Unix-based systems, and even embedded systems. Bindings exist for a large number of programming languages including C, C++, C#, Java, Perl, and Python.

2.3 X11 Hardware Acceleration

GLX, the OpenGL extension to the X Window System, provides an API that allows X11-based applications to send OpenGL commands to the X server. Depending on hardware availability, these commands will either be sent to a hardware-accelerated GPU or rendered in software using the Mesa OpenGL implementation [36]. As GLX serializes OpenGL commands over the X11 wire protocol, it is able to support both local and remote clients. Remote clients can only perform non-accelerated rendering.

Using GLX can lead to significant overhead as all data has to be routed through the X server. In response, the Direct Rendering Infrastructure (DRI) was created to allow for safe direct access to the GPU from an application's address space, while still relying on Mesa for a software fallback. The Direct Rendering Manager (DRM), a kernel module, controls the GPU hardware resources and mediates concurrent access by different applications (including the X server). While the DRI provides a direct path for OpenGL commands, GLX must still be used for interactions with the X window server.

3. VMGL

VMGL offers hardware accelerated rendering capabilities to applications running inside a VM. VMGL virtualizes the OpenGL API v1.5, providing access to most modern features exposed by 3D graphics hardware, including vertex and pixel shaders. VMGL also provides VM suspend and resume capabilities. The current VMGL implementation supports Xen and VMware VMMs, ATI, Nvidia, and Intel GPUs, and X11-based guest operating systems like Linux, FreeBSD, and OpenSolaris. VMGL is implemented in userspace to maintain VMM and guest OS agnosticism, and its design is organized around two main architectural features:

- **Virtualizing the OpenGL API** removes any need for application modifications or relinking, guarantees portability to different guest operating systems, and guarantees compatibility with graphics hardware from different manufacturers.
- **Use of a Network Transport** guarantees applicability across VMMs and even for different types of VMs supported by the same VMM.

VMGL is open-source software publicly available on the author's website [29]. In the rest of this paper, the term *host* refers to the administrative VM in Xen, or the underlying OS for hosted VMMs like VMware Workstation. The term *guest* refers to a VM or domain.

3.1 VMGL Architecture

Figure 1 shows the VMGL architecture, which consists of three user-space modules: the VMGL library, the VMGL stub, and the VMGL X server extension. The figure also shows an application running on the guest VM and a viewer which runs on the host and handles user input and the guest's visual output.

Applications inside guests use the VMGL library as a replacement for standard or vendor-specific OpenGL implementations. Upon application startup, the VMGL library creates a VMGL stub on the host to act as a sink for OpenGL commands. The VMGL stub links against the OpenGL library available on the host to obtain direct rendering capabilities on behalf of the virtualized application. When the application inside the guest issues GL commands, the VMGL library forwards those commands to the VMGL stub using a network transport over a loopback connection. Each application uses a different VMGL stub, and each stub executes as a separate process in the host, thus leveraging the address space protection guarantees offered by the vendor OpenGL implementation.

Figure 1 illustrates the use of viewer software, typically based on VNC [38], that displays the 2D output generated by a guest, and captures user input and relays it to the guest. The guest 2D output is generated by an X server drawing to a virtual 2D framebuffer. In the absence of a virtual framebuffer, 2D output is generated by a VNC X server. We modified a VNC viewer to interact with VMGL stubs; the VNC viewer modifications are minimal as we offload most of the functionality onto the stubs. This allowed us to easily add support for the alternative viewers used by Xen and other VMMs like QEMU [12].

To compose the viewer 2D output and the VMGL stub's 3D GL output, we augment the guest's X server with an extension. The VMGL library uses this extension to register windows bound to OpenGL contexts. The extension monitors changes to the size, position and visibility of OpenGL-enabled windows, and forwards those changes to the VMGL stub. The stub applies this information on its GL graphical output by clipping it to remove sectors that are not currently visible in the guest's desktop, resizing it, and finally superimposing it on the viewer window at the appropriate relative coordinates. The VMGL X extension is a loadable module that can be added to an existing X server configuration. The extension operates at the common device independent layer shared by all variants of X servers, ensuring support for the X11 and VNC servers used inside guests.

The rest of this section first describes WireGL, the OpenGL network transport used by VMGL. We then describe VMGL's suspend and resume implementation, and discuss driver changes necessary to obtain direct rendering capabilities in a Xen configuration. Finally, we discuss limitations of the current implementation.

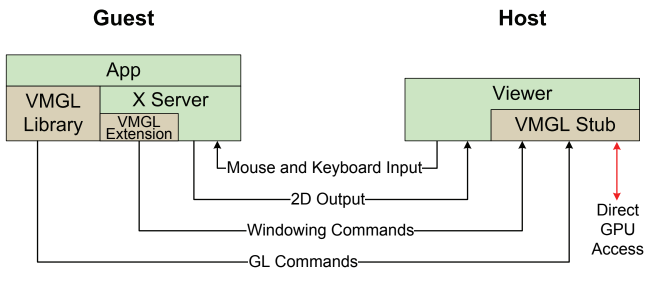


Figure 1: VMGL Architecture

3.2 OpenGL Transport

The standard OpenGL transport for remote rendering is GLX, previously described in Section 2.3. When used over network connections, GLX has two important disadvantages. First, it cannot provide a direct rendering path from the application to the graphics card. This is solved in VMGL by interposing a GL stub that channels GL commands into a direct rendering context. Second, GLX involves costly network round-trips for each and every OpenGL command being invoked. VMGL avoids this cost by leveraging the WireGL protocol [15, 26]. WireGL optimizes the forwarding of OpenGL commands by only transmitting changes to screen-visible state, and by aggregating multiple OpenGL commands in a single network transmission.

WireGL applies the changes to OpenGL state requested by the application to a local cache. Dirty cache contents are flushed lazily as needed. This enables smart discarding or postponing of ineffectual state changing commands. For example, if *glTexSubImage* is used to modify a texture that is currently not visible, no network packets will be sent until the modified area becomes visible.

WireGL further optimizes network utilization by reordering and buffering commands until a commit point arrives. Geometry commands are buffered in queues. Whenever possible, commands are merged in these queues. For example, consecutive *glRotate* and *glTranslate* calls are collapsed into a single matrix modification command. When the application issues state changing or flushing commands (like *glFlush* or *glXSwapBuffers*), the buffered block of geometry modifications is sent, along with outstanding state changes associated to that geometry.

3.3 Suspend and Resume Functionality

VMGL provides support for VM suspend and resume, enabling user sessions to be interrupted or moved between computers [16, 28, 39]. Upon VM resume, VMGL presents the same graphic state that the user observed before suspending while retaining hardware acceleration capabilities.

VMGL uses a *shadow driver* [43] approach to support guest suspend and resume. While the guest is running, VMGL snoops on the GL commands it forwards to keep track of the entire OpenGL state of an application. Upon resume, VMGL instantiates a new stub on the host, and the stub is initialized by synchronizing it with the application OpenGL state stored by VMGL. While the upper bound on the size of the OpenGL state kept by VMGL is in principle determined by the GPU RAM size, our experiments in section 4.6 demonstrate it is much smaller in practice.

VMGL keeps state for all the OpenGL contexts managed by the application, and all the windows currently bound to those contexts. For each window we track the visual properties and the bindings to the VMGL X extension. For each context, we store state belonging to three categories:

- **Global Context State:** Including the current matrix stack, clip planes, light sources, fog settings, visual properties, etc.
- **Texture State:** Including pixel data and parameters such as border color or wrap coordinates. This information is kept for each texture associated to a context.
- **Display Lists:** A display list contains a series of OpenGL calls that are stored in GPU memory in a compact format, to optimize their execution as a single atomic operation at later times. For each display list associated to a context we keep a verbatim "unrolling" of its sequence of OpenGL calls.

Like the rest of VMGL, the code implementing OpenGL state restore resides in userspace, thus retaining OS and VMM independence. Furthermore, OpenGL state is independent of its representation in GPU memory by a particular vendor. Therefore, VMGL can suspend and resume applications across physical hosts equipped with GPUs from different vendors. The only prerequisite is for the graphics card of the target computer to provide a superset of the extensions supported by the card of the source machine, or that vendor-specific OpenGL extensions are altogether disabled.

3.4 Porting GPU Drivers For Xen

VMMs such as VMware Workstation run an unmodified OS as the host, thus enabling the VMGL stubs to readily take advantage of hardware-specific drivers for direct rendering. However, this is not always the case for Xen. Its administrative VM, also known as domain0, is itself a VM running a paravirtualized kernel, and incompatibilities with closed-source drivers arise.

Xen's architecture prevents virtual machines from modifying the memory page tables through direct MMU manipulation. Paravirtualized kernels in Xen need to invoke the `mmu_update` hypercall to have Xen perform a batch of page table modifications on its behalf. Before manipulating the hardware MMU, Xen will sanity-check the requested changes to prevent unauthorized access to the memory of another virtual machine. Transferring MMU-manipulation responsibilities to the hypervisor has introduced another level of indirection in memory addressing: physical frame numbers in a domain kernel are mapped by Xen into *machine frame numbers*, the actual memory frame numbers handled by the MMU.

To enable direct rendering functionality, OpenGL implementations need to communicate with the graphics card. This is typically achieved by memory mapping a character device, which results in the kernel remapping GPU DMA areas into the GL library's address space. In the absence of IOMMU hardware support for virtualized DMA addressing [13], Xen needs to interpose on these operations, translate them to machine frame numbers, and sanitize them. Drivers included in the Linux kernel distribution and using the Direct Rendering Manager described in Section 2 (e.g. Intel's), use functions that Xen paravirtualizes to provide the proper DMA addressing. Unfortunately, this is not the case with the proprietary closed-source drivers of Nvidia and ATI cards.

Luckily, these drivers are wrapped by an open-source component that is recompiled to match the specifics of the current kernel. As long as all DMA mapping functions are contained in the open-source component, the proprietary driver can be adjusted to run in domain0. We have ported the fglrx driver version 8.29.6 for an ATI Radeon X600 PCI-Express card. By changing the DMA mapping macro to use a paravirtualization-aware function, we were able to use the driver in domain0. Similar modifications to the proprietary Nvidia driver version 1.0-8756 also provide direct rendering functionality for domain0 [4].

3.5 VMGL Limitations

VMGL currently supports 59 OpenGL v1.5 extensions, including vertex programs, fragment programs, and 13 vendor-specific extensions. We are constantly working to extend VMGL support to more GL extensions. For instance, the Unreal Tournament 2004 benchmark used in the next section demanded the implementation of a number of extensions including `GL_EXT_bgra`. Vendor-specific extensions could represent a source of incompatibility if a VMGL-enabled guest is resumed on a new physical host with a different GPU from the one available where it was last suspended. If the GPU at the resume site does not support some of the vendor-specific extensions in use by an application, we will have to temporarily map their functionality to supported variants, possibly suffering a performance hit. An alternative solution is to altogether disable vendor-specific extensions, at the expense of sacrificing functionality in some cases.

VMGL currently does not support Windows or MacOS guests. We have not yet developed the necessary hooks into the windowing systems to provide functionality similar to that of our X server extension, although we do not anticipate any major obstacles in this area. Finally, we conjecture that the Direct3D API used by some Windows applications can be supported through Direct3D to OpenGL translation layers, such as WineD3D [2].

Application	Release Date
Quake 3	Dec, 1999
Unreal Tournament 2004	Mar, 2004
Wolfenstein: Enemy Territory	May, 2003
Mplayer	Jun, 2006

Table 1: Application Benchmarks

4. Evaluation

Our evaluation of VMGL addresses the following questions:

Performance How does VMGL compare to software rendering alternatives, such as the Mesa OpenGL library [36]? How close does it come to providing the performance observed with unvirtualized graphics acceleration?

Portability Can VMGL be used with different VMMs? Can VMGL be used with different VM types supported by the same VMM? Can VMGL be used with different guest operating systems?

Suspend and Resume What is the latency for resuming a suspended OpenGL application? What is the size of an application's OpenGL suspended state? Can we migrate suspended OpenGL applications across GPUs from different vendors?

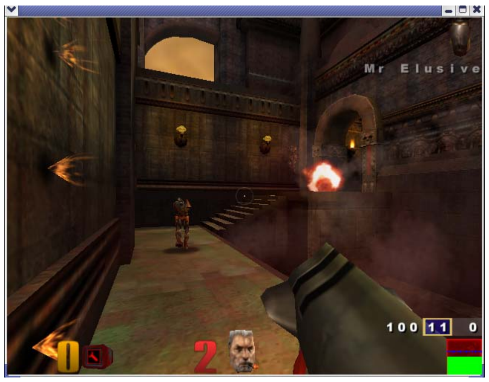
Sensitivity to Resolution What is the effect of rendering resolution on VMGL performance?

Sensitivity to Multiple Processors How sensitive is VMGL to processing power? Can it take advantage of multi-core CPUs?

Scalability How well does VMGL scale to support multiple VMs performing 3D drawing concurrently? A proposed use for VMs is the deployment of virtual appliances [47]. It is expected that users will run multiple virtual appliances in a single physical platform, with perhaps several appliances doing 3D rendering simultaneously.

4.1 Benchmarks

Table 1 summarizes the four benchmarks we use in the evaluation of VMGL. We focus our evaluation on computer games and entertainment as these classes of applications have effectively become the driving force in the development of consumer graphics applications and hardware [37]:


(a) Quake 3 Arena

(b) Enemy Territory

(c) Unreal Tournament 2004

(d) Mplayer

Figure 2: Benchmark screenshots

- **Quake 3**: Quake III Arena [27] (Figure 2 (a)), was first released in December, 1999. Quake 3 employs an extensive array of OpenGL drawing techniques [49], including shader scripts; volumetric textures, fog and lighting; vertex animation; Gouraud shading; spline-based curved-surfaces, and others. This set of features has enabled Quake 3, despite its relative age, to remain a popular application for benchmarking 3D performance [34, 44]. Quake 3 was open-sourced in 2005.
- **Enemy**: Wolfenstein Enemy Territory [41] (Figure 2 (b)) was released in May of 2003. The game is a third-generation successor to the Quake 3 engine, including enhancements such as skeletal animation and substantially increased texture and scenic detail. Enemy's logic was open-sourced in 2004.
- **Unreal**: Unreal Tournament 2004 [19] (Figure 2 (c)) has a modern graphics engine [45] that exploits a variety of features such as vertex lighting, projective texturing, sprite or mesh particle systems, distance fog, texture animation and modulation, portal effects, and vertex or static meshes. Like Quake 3, Unreal is also heavily favored by the industry as a *de facto* benchmark for 3D graphics performance [9, 44]. Unlike Quake 3 and Enemy, this application is closed source.
- **Mplayer**: Mplayer [3] (Figure 2 (d)) is a popular open source media player available for all major operating systems. It supports a number of different video codecs, and a number of output drivers, including texture-driven OpenGL output.

For the first three benchmarks, we replayed publicly available demos for 68 seconds (Quake 3 [27]), 145 seconds (Enemy [8]), and 121 seconds (Unreal [6]). For the Mplayer benchmark we replayed the first 121 seconds of a video clip encoded at two different resolutions.

4.2 Experimental Setup

We tested VMGL with two virtual machine monitors, Xen 3.0.3 [10] and VMware Workstation 5.5.3 [7]. All experiments were run on a 2.4 GHz Intel Core2 machine with two single-threaded cores, VT hardware virtualization extensions, and 2 GB of RAM. For most experiments, we employed a Radeon X600 PCI-Express ATI graphics card; for a set of suspend and resume experiments we used an Intel 945G PCI-Express Graphics card. The machine ran the Fedora Core 5 Linux distribution with the 2.6.16.29 kernel in 32 bit mode, and X.Org version 7.0 with the fglrx proprietary ATI driver version 8.29.6, or the DRI driver based on Mesa version 6.4.2 for the Intel Card. All virtual machines were configured with the same kernel (modulo para-virtualization extensions for Xen), same distribution, 512 MB of RAM, and no swap. We ran each benchmark in three different configurations:

- **Native**: an unvirtualized environment with direct access to hardware and native OpenGL drivers. VMGL was not used. This represents the upper bound on achievable performance for our experimental setup.

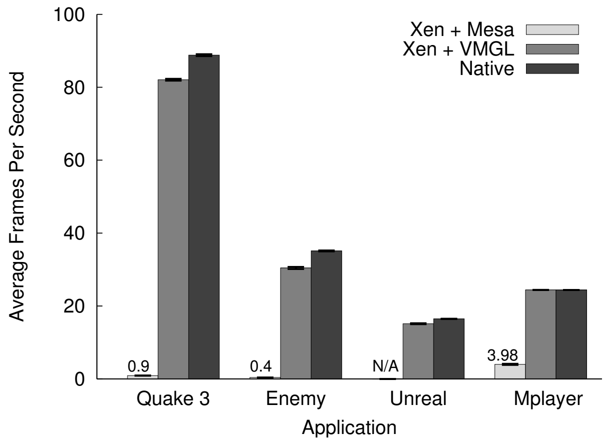


Figure 3: VMGL performance – Average FPS, high resolution.

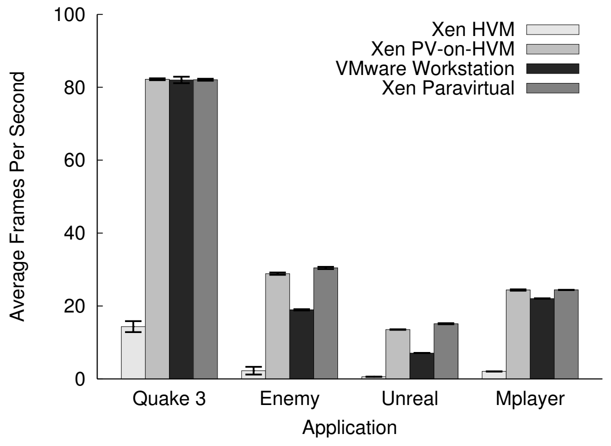


Figure 5: VMM portability – High resolution.

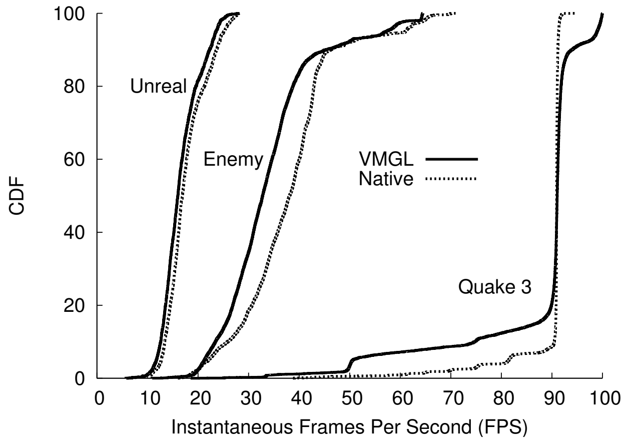


Figure 4: VMGL performance – FPS variability, high resolution.

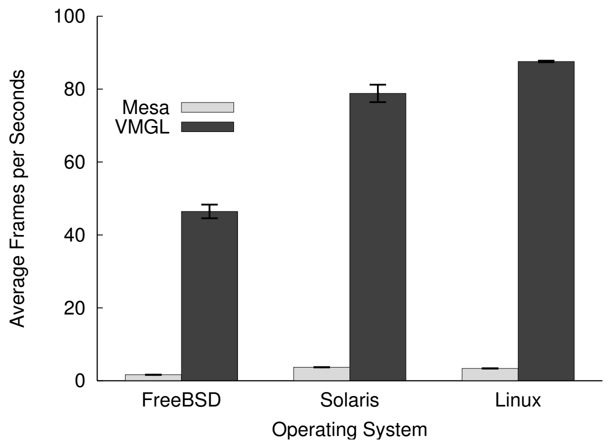


Figure 6: Guest OS portability – Quake 3, VMware Workstation guests.

- **Guest + Mesa Software Rendering**: a virtualized guest using software rendering provided by the Mesa OpenGL library. No hardware rendering facilities were used. This is the commonly available configuration for current users of 3D applications in virtualized environments. The Unreal benchmark refuses to run in the absence of hardware acceleration.
- **Guest + VMGL**: a virtualized guest using VMGL to provide 3D hardware acceleration. This configuration is depicted by Figure 1.

In each of these configurations, all benchmarks were executed at two different resolutions:

- **High Resolution**: The resolution was set to 1280x1024 except for Mplayer, which had a resolution of 1280x720 (the closest NTSC aspect ratio).
- **Low Resolution**: The resolution was set to 640x480 except for Mplayer, which had a resolution of 640x352 (the closest NTSC aspect ratio).

We quantify graphics rendering performance in terms of framerate or Frames per Second (FPS), a standard metric used for the evaluation of 3D graphics [9, 34, 44]. We also measure VMGL's resource utilization in terms of CPU load and network usage. All data points reported throughout the rest of this section are the average of five runs. All bar charts have standard deviation error bars.

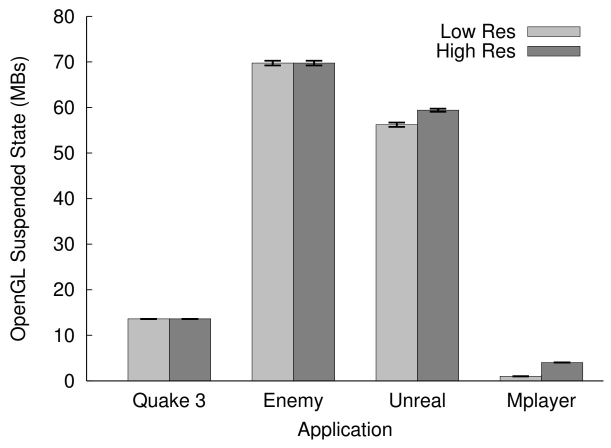
4.3 Performance

Figure 3 shows the results from running the benchmarks under three configurations, native, Xen paravirtualized guest with VMGL, and Xen paravirtualized guest with Mesa software rendering. All benchmarks in the figure are run in high resolution mode.

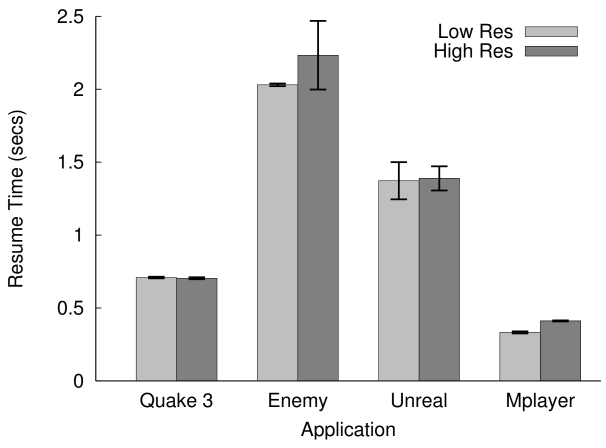
First, we observe that VMGL's performance is two orders of magnitude better than software rendering. The number of FPS delivered by Mesa ranges from 0.4 to 4. From the user's perspective, this low framerate renders the applications unusable.

Next, we observe that VMGL's performance approximates that of the native configuration, with the performance drop ranging from 14% for the Enemy benchmark to virtually no loss for the Mplayer and Unreal benchmarks. In our subjective experience, the user experience delivered by VMGL is indistinguishable from that of the native configuration for all of the benchmarks.

Figure 3 reports a global average metric. To further understand VMGL's performance we need to compare its variations in framerate and peak FPS values against those observed under native execution. Crisp interaction with highly detailed graphics applications not only demands a high framerate, but also a uniform experience without jitter [21]. Figure 4 plots a cumulative distribution func-



(a) Size of suspended OpenGL state



(b) Resume time

Figure 7: Suspend and resume – Xen Paravirtual guest.

tion for the instantaneous FPS across all five trials on each benchmark. Plots to the right indicate better performance than plots to the left; the more vertical a plot is, the smaller variability in framerate. We exclude Mesa results given their very low quality; we also exclude the Mplayer benchmark as it presents a constant framerate of 25 FPS across the remaining configurations. VMGL results closely follow the behavior of their native execution counterparts. The variability in frame rates is consistent with that observed under native execution. Differences in the framerate distribution and peak FPS values with respect to native execution are minimal.

4.4 VMM Portability

Figure 5 shows VMGL's performance for one VMware Workstation and three Xen configurations: *Xen HVM* leverages Intel's VT extensions to run an unmodified Linux kernel as a guest, and emulates network I/O using code derived from the QEMU emulator [12]; *Xen PV-on-HVM* is similar to Xen HVM, but a loadable kernel module provides the guest with Xen-aware paravirtualized network functionality; *VMware Workstation* runs an unmodified Linux kernel as the guest OS and uses VMware Tools for proprietary network virtualization; Finally, *Xen Paravirtual* is the same Xen paravirtualized guest configuration as the *Xen + VMGL* bars of Figure 3.

As expected, Figure 5 shows that the quality of network virtualization is a fundamental factor affecting VMGL's performance. Without paravirtualized extensions, a Xen HVM presents very low FPS ratings. The PV-on-HVM configuration provides almost identical performance to that of Xen paravirtualized guests. VMware Workstation's similar use of virtualization-aware drivers on an otherwise unmodified OS also yields an order of magnitude better performance than a pure Xen HVM. We expect VMGL performance under a VMware hypervisor product like VMware ESX server to be closer to that provided by Xen paravirtualization.

4.5 Portability Across Guest Operating System

VMGL userspace design and its implementation in standard programming languages makes it easy to port across operating systems. In particular, we have ported VMGL to FreeBSD release 6.1 and OpenSolaris 10 release 06/06. The source code logic remained unmodified. All necessary changes had to do with accounting for differences in the OS development environment, such as header inclusion, library linking, and tools used in the build process.

To test our VMGL port for these two operating systems, we configured them as VMware Workstations guests running the open-source Quake 3 port ioquake3 (Quake 3's authors did not port the application to OpenSolaris or FreeBSD). Figure 6 compares the performance of Mesa software rendering and VMGL accelerated rendering for each OS, including Linux. While FreeBSD did not perform in general as well as OpenSolaris, in both cases VMGL conserves its notable performance advantage over software rendering. Configuring our experimental machine to natively run FreeBSD or OpenSolaris was beyond our time availability. We are confident VMGL will show a trend similar to that with Linux and maintain performance on par with an unvirtualized configuration.

4.6 Suspend and Resume

To measure the performance of VMGL's suspend and resume code, we suspended a guest running the benchmarks at five different and arbitrary points in time. We then resumed the guest and verified successful resumption of the OpenGL application. We measured the size of the OpenGL state necessary to synchronize the GL stub to the current application state, and the time it took to perform the entire resume operation. We did not observe any noticeable effect of the suspend and resume code on the application's framerate performance. The results of these experiments are displayed in Figure 7. This Figure displays results for both application resolutions obtained with Xen paravirtualized guests; similar results were obtained with VMware Workstation guests.

The resume time (Figure 7 (b)) is strongly dependent on the size of the suspended OpenGL state (Figure 7 (a)), which can be as large as 70 MB for the Enemy benchmark. Nevertheless, the latency for reinstating the suspended OpenGL state on a new VMGL stub never exceeded 2.5 seconds. Regardless of the suspend point, the size of Mplayer's state is always the same, as this state is almost exclusively composed of the texture corresponding to the current frame. Since the frame is twice as big on each dimension, the state is four times larger in high resolution mode than in low resolution mode. Finally, we were surprised to note that the size of Quake 3's OpenGL state is also invariant with respect to the suspend point. We conjecture that Quake 3 preallocates the entire OpenGL state for a new environment before allowing interaction.

We performed a second set of experiments in which we suspended and resumed a guest across two different hosts: our experimental machine and a similar physical host using an Intel 945G GPU. The tests completed successfully with similar latency and

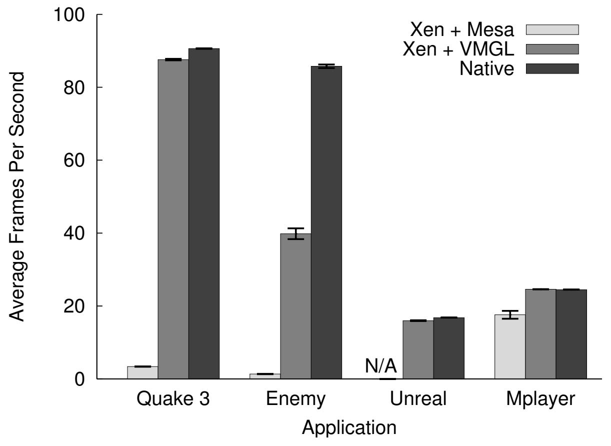


Figure 8: Benchmarks in low resolution mode.

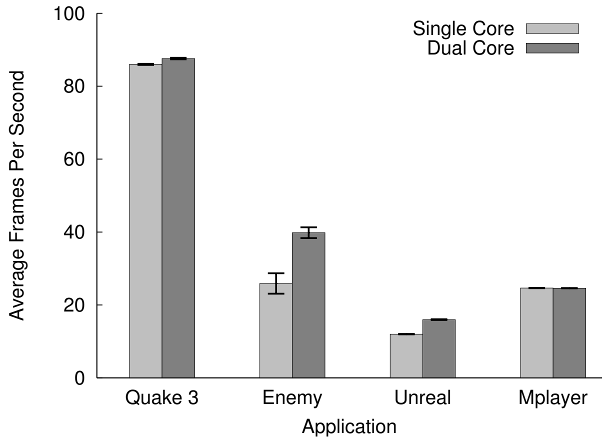


Figure 9: CPU sensitivity – Xen paravirtual + VMGL, low resolution.

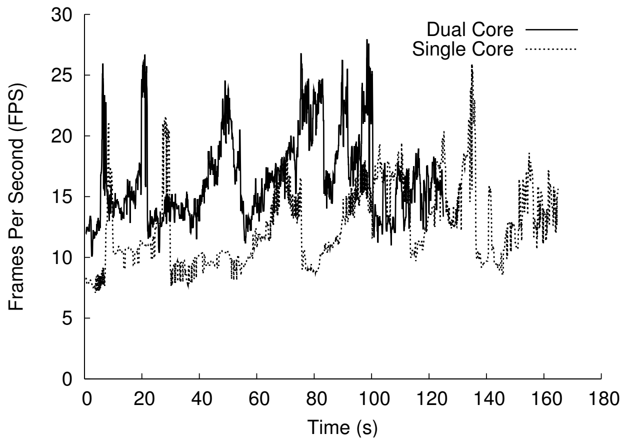
state size results. We had to disable in VMGL five extensions provided by the ATI card but not by Intel's (including `GL_ARB_occlusion_query`, for example), and four extensions available in the Intel card but not in ATI's (including `GL_NV_texture_rectangle`).

4.7 Sensitivity to Screen Resolution

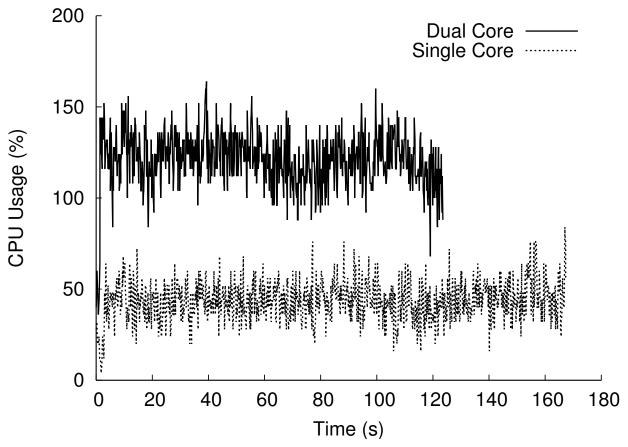
OpenGL drawing primitives use a normalized coordinate system, and rely on the hardware capabilities of the graphics card to scale the geometry to match the current screen resolution. This implies that the higher the resolution, the busier the GPU and therefore the less noticeable the VMGL command marshaling overhead becomes. The slightly counter-intuitive consequence is that it is preferable to run applications under VMGL at higher resolutions, something which is desirable anyway.

Figure 8 shows the results from running the benchmarks at low resolution (640x480, Mplayer runs at 640x352) for three configurations: Xen with Mesa, Xen with VMGL, and native. The first three benchmarks generate the same stream of OpenGL commands as in the high resolution experiments (Figure 3), and rely on automatic scaling. Mplayer is different, as each frame is generated by synthesizing an appropriately sized texture from the input video data, and therefore it does not involve any hardware scaling.

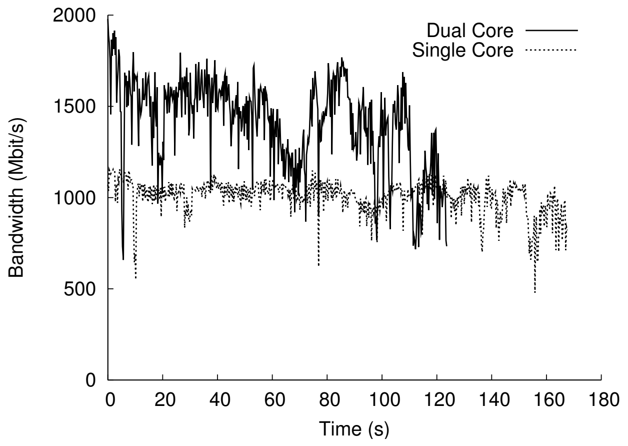
The increased pressure on the VMGL transport is evident for the Enemy benchmark, presenting a performance drop with respect to



(a) Instantaneous frames per second



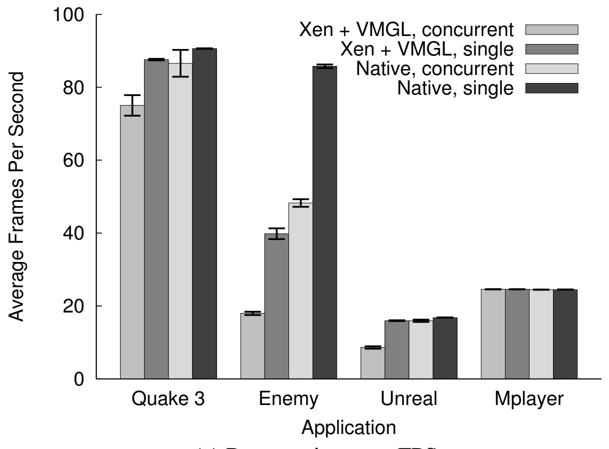
(b) CPU usage



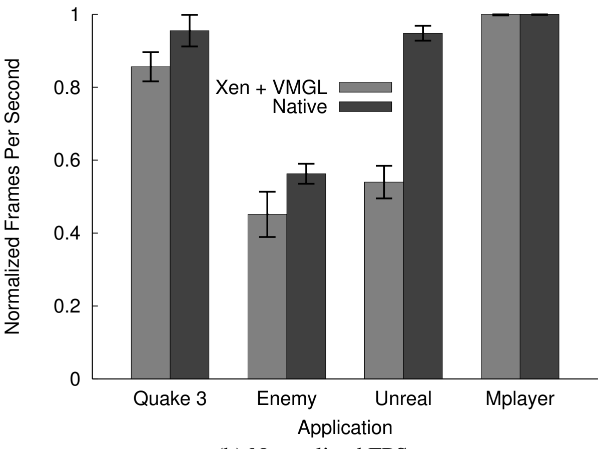
(c) Network usage

Figure 10: Unreal instantaneous FPS, and CPU and network usage on dual- vs. single-core configurations, low resolution. CPU utilization includes all components depicted in Figure 1. With a single-core, the benchmark takes longer to complete due to the reduced framerate.

the unvirtualized baseline to approximately half the rate of frames per second. However, for the remaining three benchmarks the performance of Xen+VMGL closely matches that of the native configuration. Software rendering is still unable to provide reasonable



(a) Raw per-instance FPS



(b) Normalized FPS

Figure 11: Concurrent guests – FPS for running two simultaneous instances of each benchmark at low resolution. Xen paravirtual.

performance, perhaps with the exception of the Mplayer benchmark achieving 17.6 average FPS due to the smaller sized frames.

For the remainder of this section, we concentrate on low-resolution experiments as they bias the results *against* VMGL.

4.8 Sensitivity to Multi-Core Processing

To determine the benefits that VMGL derives from multi-core processing, we also ran all of our application benchmarks after disabling one of the two cores in our experimental machine. These results, presented in Figure 9, show a performance drop for Enemy and Unreal, the more modern applications. There is no significant difference for the older applications.

We analyze the benefits arising from a multi-core setup using Unreal as an example. Figure 10 shows the differences in resource usage for the single and multi-core cases. The increased CPU utilization possible with dual-core parallelism (Figure 10 (b)) results in a higher rate of OpenGL commands pushed per second through the VMGL transport (Figure 10 (c)). The consequence is a higher framerate in the dual-core case (Figure 10 (a)). Unreal's behavior seems to be a work-conserving: rather than dropping frames at a low framerate, it takes longer to complete the demo.

The presence of two cores leads to increased resource utilization for a number of reasons. First, multiple cores allow concurrent execution for the two networking stacks: in the guest where the application executes and in the host where the viewer resides. It also allows for parallelizing the marshaling and unmarshaling cycles of OpenGL commands by VMGL. The availability of two cores also ameliorates the VMM's overhead of constantly needing to context switch between the two VMs, and to switch to the hypervisor to handle the interrupts generated by the bridged networking setup, a previously documented overhead [31, 32].

4.9 Concurrent Guests

To examine VMGL's ability to support concurrent guests, we compare the performance of two instances of an application executing concurrently in an unvirtualized configuration, to the performance of two instances executing in two separate Xen paravirtual guests.

Figure 11 (a) presents the average per-instance FPS results for the concurrent execution of two instances, compared to the average FPS results for a single instance (taken from Figure 8). Figure 11 (b) normalizes the concurrent execution results against the single-instance results (i.e. *Xen + VMGL, concurrent* divided by *Xen + VMGL, single*). The purpose of normalization is to observe the "natural" scalability inherent to the application: simultaneous instances may compete for CPU and memory resources. The additional drop in normalized FPS for the VMGL configurations reflects the overheads of GL marshaling and context-switching VMs.

The Mplayer benchmark, more representative of a multiple virtual appliance scenario, presents excellent scalability results. We observe decreasing VMGL scalability as the application becomes more heavyweight and places a larger demand on the GL transport: 10% additional overhead for Quake 3, 20% for Enemy, and 43% for Unreal. Figure 10 (c) indicates that the bandwidth demands of a single instance of Unreal can peak at almost 2 Gbit/s. Extreme configurations with multiple high-end applications rendering concurrently may impose an aggregate bandwidth demand on VMGL of several Gbit/s. A VMM-specific shared memory transport may be preferable under those circumstances.

5. Related Work

A number of solutions have been proposed to provide 3D acceleration to Virtual Machines. The most straightforward solution is to grant a *driver VM* [30] direct access to the GPU hardware. The driver VM would thus occupy the same role as the host in our architecture, with the advantage of isolating potential driver malfunctions on a separate protection domain. However, in the absence of an IOMMU [13], granting hardware access rights to a VM will weaken the safety and isolation properties of VM technology. A rogue VM with direct hardware access would be able to initiate DMA to and from memory owned by other VMs running on the same machine. Further, a VM with direct hardware access cannot be safely suspended or migrated to a different machine without driver support.

Other proposed solutions to graphics virtualization have focused on properties other than providing VMM or guest OS independence. The Blink [25] system for the Xen hypervisor multiplexes graphical content onto a virtual GPU, with an emphasis on safety in the face of multiple untrusted clients. While no details are available regarding its implementations, VMware [48] provides a solution that virtualizes the Direct3D API for Windows applications. Both systems are VMM-specific as they use shared memory and do not support suspending or migrating a VM.

Accelerated Indirect GLX (AIGLX) [1], has been developed to provide accelerated GLX rendering for remote clients. While originally designed to enable OpenGL-accelerated compositing window managers, it could be used as an alternative transport for VMGL.

Since AIGLX lacks the transport optimizations used by WireGL, we believe it would severely constrain applicability with its greater bandwidth utilization.

A number of projects for remote visualization of scientific data have tried to optimize remote OpenGL rendering. Some, like Visapult [14], Cactus [23], and SciRun [35], require their applications to be written to a particular interface and are therefore useful only when application source code is available. Other systems [42, 46] render data using remote GPUs and ship the resulting images using slow or lossy thin client protocols such as X11 or VNC.

6. Future Work and Conclusion

GPUs are critical to high-quality visualization in many application domains. Running such applications in VMM environments is difficult for a number of reasons, all relating to the fact that the GPU hardware interface is proprietary rather than standardized. This paper describes the design, implementation, and evaluation of VMGL, a VMM-independent, GPU-independent, cross-platform solution to this problem. VMGL virtualizes the OpenGL software interface, recognizing its widespread use in graphics-intensive applications. By virtualizing at the API level, VMGL is able to support multiple guest OSs and to provide suspend and resume capabilities across GPUs from different vendors. Our experiments confirm excellent rendering performance with VMGL, coming within 14% or better of native hardware accelerated performance measured in frames per second. This is two orders of magnitude better than software rendering, which is the commonly available alternative today for graphics-intensive applications in virtualized environments.

Our results also show that the resource demands of VMGL align well with the emerging trend of multi-core processors. In other words, there is natural and easy-to-exploit parallelism in the VMGL architecture. Our work thus reveals an opportunity for three emerging trends (virtualization, growing use of GPUs by applications, and multi-core processing) to evolve in a mutually supportive way.

Our work so far has focused on portability across VMMs and guest operating systems. We have therefore avoided all performance optimizations that might compromise portability. By carefully relaxing this constraint, we anticipate being able to bring VMGL performance closer to native performance for very demanding applications at high levels of concurrency. Under such workloads the total bandwidth between application VMs and the OpenGL stubs becomes the performance bottleneck. A shared-memory rather than network transport implementation could relieve this bottleneck. By implementing this optimization in a way that preserves the external interfaces of VMGL, we could enable VMM-specific and guest-specific code to be introduced with minimal negative impact on portability. The network transport would always remain a fallback for environments without support for shared-memory transport.

While its main target is graphical applications, VMGL can provide access to the computing power of GPUs to an emerging class of GPU-based scientific applications [11, 22]. The highly parallel and efficient architecture of GPUs has proved tremendously useful in producing high-performance solutions to several scientific problems. Algorithms that solve these problems using GPU processing are written mainly in OpenGL [24]. We believe that scientific application running in virtualized environments, like those proposed for the Grid [20], will be able to leverage VMGL for improved performance. We intend to test this hypothesis in future work.

Finally, while VMGL's current implementation supports X11-based guest operating systems, we anticipate no major obstacles when porting VMGL to Windows and Apple's MacOS. We also believe that the Direct3D API can be virtualized in a similar way to what we have presented here for OpenGL.

Acknowledgments

We thank the anonymous reviewers and fellow University of Toronto syslab students for their helpful suggestions. We also thank VMware for granting permission to publish benchmarking results obtained with their VMware Workstation product. We thank Anil Madhavapeddy and Dave Scott for mentoring during the early stages of this project, and Jacob Gorm-Hansen for the enriching discussions on our common interests.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSERC, CFI, OIT, NSF, Google, the University of Toronto, or Carnegie Mellon University. All unidentified trademarks mentioned in the paper are properties of their respective owners.

References

[1] Rendering project/AIGLX. `http://fedoraproject.org/wiki/RenderingProject/aiglx`.

[2] WineD3D. `http://wiki.winehq.org/WineD3D`.

[3] Mplayer Headquarters. `http://www.mplayerhq.hu/`.

[4] Personal communications with J. Gorm-Hansen and Y. Liu.

[5] OpenGL – The Industry Standard for High Performance Graphics. `http://www.opengl.org`.

[6] Unreal Torunament 2004 Demo: Assault. `http://www.3dcenter.org/downloads/ut2004demo-assault.php`.

[7] VMware Workstation. `http://www.vmware.com/products/ws/`.

[8] Enemy Territory Demo: Radar. `http://www.3dcenter.org/downloads/enemy-territory-radar.php`.

[9] AnandTech. Next-Generation Game Performance with the Unreal Engine: 15-way GPU Shootout. `http://www.anandtech.com/video/showdoc.html?i=1580&p=1`.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[11] G. Baron, C. Sarris, and E. Fiume. Fast and accurate time-domain simulations with commodity graphics hardware. In *Proc.Antennas and Propagation Society International Symposium*.

[12] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proc. FREENIX track, Usenix Technical Conference*, pages 47–60, Anaheim, CA, Apr. 2005.

[13] M. Ben-Yehuda, J. Mason, O. Krieger, and J. Xenidis. Xen/IOMMU, Breaking IO in New and Interesting Ways. `http://www.xensource.com/files/xs0106_xen_iommu.pdf`.

[14] W. Bethel. Visapult: A prototype remote and distributed visualization application and framework. In *Proc. SIGGRAPH Annual Conference*, New Orleans, LA, July 2000.

[15] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 87–95, Interlaken, Switzerland, Aug. 2000.

[16] R. Caceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating PCs with Portable SoulPads. In *Proc. 3rd International Conference on Mobile Systems Applications and Services (MobiSys)*, Seattle, WA, June 2005.

[17] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Archi-

tecture. In *Proc. 2nd Symposium on Networked Systems Design & Implementation (NSDI)*, Boston, MA, 2005.

[18] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[19] Epic Games. Unreal Tournament. `http://www.unrealtournament.com/`.

[20] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes. A case for grid computing on virtual machines. In *Proc. 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, page 550, 2003.

[21] T. Funkhouser and C. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. In *Proc. 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 247–254, Anaheim, CA, Aug. 1993.

[22] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *Proc. Supercomputing, International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WA, Nov. 2005.

[23] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing - VECPAR '2002, 5th International Conference*, Porto, Portugal, June 2003. Springer.

[24] GPGPU – General Purpose Programming on GPUs. What programming API's exist for GPGPU. `http://www.gpgpu.org/w/index.php/FAQ#What_programming_APIs_exist_for_GPGPU.3F`.

[25] J. G. Hansen. Blink: 3d display multiplexing for virtualized applications. Technical Report 06/06, DIKU – University of Copenhagen, Jan. 2006. `http://www.diku.dk/~jacobg/pubs/blink-techreport.pdf`.

[26] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proc. 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 693–702, New York, NY, USA, 2002.

[27] ID Software. Quake III Arena. `http://www.idsoftware.com/games/quake/quake3-arena/`.

[28] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proc. Fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, New York, June 2002.

[29] H. A. Lagar-Cavilla. VMGL Site. `http://www.cs.toronto.edu/~andreslc/vmgl`.

[30] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[31] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *VEE '05: Proc. 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 13–23, Chicago, IL, June 2005.

[32] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proc. USENIX Annual Technical Conference (USENIX 2006)*, pages 15–28, May 2006.

[33] Microsoft. DirectX Home Page. `http://www.microsoft.com/windows/directx/default.mspx`.

[34] Motherboards.org. How to benchmark a videocard. `http://www.motherboards.org/articles/guides/1278_7.html/`.

[35] S. Parker and C. Johnson. Scirun: A scientific programming environment for computational steering. In *Proc. ACM/IEEE conference on Supercomputing*, San Diego, CA, Dec. 1995.

[36] B. Paul. Mesa 3d library. `http://www.mesa3d.org/`.

[37] T. M. Rhyne. Computer games' influence on scientific and information visualization. *IEEE Computer*, 33(12):154–159, Dec. 2000.

[38] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. Virtual Network Computing. *IEEE Internet Computing*, 2 (1):33–38, Jan/Feb 1998.

[39] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[40] M. Satyanarayanan, M. A. Kozuch, C. J. Helfrich, and D. R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Pervasive and Mobile Computing*, 1(2):157–189, 2005.

[41] Splash Damage. Enemy Territory Press Release. `http://www.splashdamage.com/?page_id=7`.

[42] S. Stegmaier, M. Magallón, and T. Ertl. A generic solution for hardware-accelerated remote visualization. In *VISSYM '02: Proc. Symposium on Data Visualisation 2002*, pages 87–, 2002.

[43] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[44] Tom's Hardware. 3D Benchmarking – Understanding Frame Rate Scores. `http://www.tomshardware.com/2000/07/04/3d_benchmarking_/index.html`.

[45] UT2K4 Engine Technology. Unreal engine 2. `http://www.unrealtechnology.com/html/technology/ue2.shtml/`.

[46] VirtualGL. `http://virtualgl.sourceforge.net/`.

[47] VMware. Virtual appliances and application virtualization. `http://www.vmware.com/appliances/`.

[48] VMware. Experimental Support for Direct3D. `http://www.vmware.com/support/ws5/doc/ws_vidsound_d3d.html`.

[49] Wikipedia. Quake3 engine technology. `http://en.wikipedia.org/wiki/Quake_III_Arena#Technology/`.