Datalogisk Institut
KØBENHAVNS UNIVERSITET

Blink: 3D Display Multiplexing for Virtualized Applications

Jacob Gorm Hansen

Technical Report no. 06/06
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

Blink: 3D Display Multiplexing for Virtualized Applications

Jacob Gorm Hansen
Department of Computer Science
University of Copenhagen, Denmark

Abstract

For management and security reasons, Virtual Machines are increasingly being deployed on Desktop PC's, but because existing VMM technologies are mostly targeted at servers, graphical performance is a stumbling block to many modern applications such as games, simulation, and video-conferencing. Existing high-performance display systems sacrifice safety and provide applications with direct hardware access, but the aim of our work is to build a display system with Virtual Machine quality safety and isolation, while retaining performance comparable to less safe "direct" approaches. We are developing Blink, a Display System for Virtual Machines, the core component of which is a JIT-compiler for extended OpenGL programs which execute safely inside the address space of the display server. Our results show that this model can lead to fewer context switches than traditional client/server approaches, can eliminate redundant copying and clearing of graphics buffers, and reduce the visible effects of scheduler timing variance.

1 Introduction

After a number of years in relative obscurity, Virtual Machines (VM's) have been gaining in popularity on the Desktop. Modern PCs are more than capable of hosting multiple operating systems, and with the relatively weak security models of current desktop operating system becoming increasingly problematic, many see VM's as the solution, because they allow a commodity PC to be divided into multiple sandboxed security compartments, and are compatible with a large body of existing software. For example, a user may choose to run the web browser in its own VM, so that worms and spyware are less likely to leak into other parts of the system.

A recent development is the occurrence of *Virtualized Applications*—desktop applications being deployed inside VM's. VMWare has recently started shipping a Linux VM containing an instance of the Firefox web browser, for secure web access, and the Stanford Collective project ships

Figure 1: Blink in action

prepackaged application VM's to users who are considered incapable of administering their own desktops. The current generation of Virtual Machine Monitors (VMM's) are mostly optimized for data center use, and as a result initial desktop VM deployments have not been suited for graphics-heavy applications such as video conferencing, games, and simulation. Current systems either run over wire protocols such as X11 or VNC that do not take advantage of shared memory, or at best have access to a primitive, two-dimensional framebuffer abstraction, that takes little advantage of the hardware acceleration features found in most modern graphics cards. Our work attempts to fill this gap by providing Virtual Machines with safe access to the powerful accelerated drawing features of modern display hardware.

The contribution of this paper is that we design and implement a system that addresses the following problems:

- How to provide an isolated VM with native-speed access to graphics hardware without jeopardizing system integrity and isolation,
- How to prevent excessive CPU context switching, when multiple VM's with short refresh intervals (such

as Video Players or Games) all need to be consulted during the preparation of the next video frame, or in response to user input.

- And finally, how to reduce the visible effects of scheduling "jitter" on applications with soft-realtime demands for steady screen refresh rates, when many applications are sharing a display.

Our new display system, called Blink, multiplexes OpenGL [1] content coming from multiple untrusted Virtual Machines onto a single Graphics Processing Unit (GPU). Blink does not allow clients to program the graphics card directly, but instead provides a Virtual Processor (VP) abstraction to which they can program. VP programs execute within the context of Blink, and in turn program the GPU on behalf of the client. Blink employs JIT-compilation and simple static analysis of VP programs, with increased flexibility and a reduction in processing and context switching overheads as results. Visible jitter is reduced because client code executes independently of the client VM time-slice. Blink is targeted at paravirtualized VM's—it supplies virtualized applications with an OpenGL-like abstraction that they can program to, but does not aim to be a transparent adaptation layer for existing windowing systems such as X11. Instead, Blink supplies the mechanisms on which such a layer can be constructed if needed.

2 Design

Compared to other I/O subsystems in an OS, the display system is harder to multiplex in a way that is both efficient and safe, especially for demanding applications such as 3D games or realtime video. This is evidenced by the fact that the major operating systems all provide "direct" avenues of contacting the GPU, largely without operating system involvement. However, some level of cooperation between applications is necessary, so that multiple applications are able to share a single screen without exceeding their screen space, and without causing the GPU to crash by unsafely intermingling hardware command streams. In Linux for example, applications are required to respect a lock provided by the kernel, and on MacOSX applications draw their content to off-screen buffers which are later copied to the shared screen buffer by the *Quartz Compositor* process.

The problem with direct GPU access is that malicious applications may be able to bypass window system labeling mechanisms, and may be able to affect overall stability. While the above mentioned systems restrict direct hardware access to a trusted set of users, today it is often the case that untrusted code (such as a game or animation downloaded from the Internet) is run by a trusted user, so in reality existing systems trade safety for performance.

Virtual Machine systems cannot afford to make this trade-off. The ability to run untrusted code sandboxed within a VM is the main motivation for desktop deployment, so the choice is between either limiting the VM display model to a dumb but safe 2D framebuffer model, or a model which provides safe access to accelerated graphics hardware. Another limitation is that VMM's are expected to provide an abstract model of the actual hardware in the machine, and this conflicts with the way systems such as OpenGL are typically implemented, with part of their implementation being loaded into the address spaces of client programs as shared libraries. Both for safety and for compatibility reasons, a layer of abstraction between client and hardware will be necessary. One way to implement such a layer is by letting clients program in high-level OpenGL (rather than programming hardware registers directly), and have the display system interpret or translate the OpenGL commands into native programming of the GPU, after verifying their harmlessness.

2.1 BlinkGL Stored Procedures

In the GLX extension to X11, OpenGL commands are serialized over the X11 wire protocol, and an interpreter runs in the display system and translates serialized commands into actual programming of the GPU. Translation costs may be amortized to some extent by the use of *display lists*, sequences of OpenGL commands stored in video memory. However, display lists are static and only useful in the parts of the GL program that need not adapt to frequently changing conditions. Blink is similar to GLX in that it serializes OpenGL commands in the client, and de-serializes and runs them in the server. Blink clients program to "BlinkGL", in reality a set of C macros and inline functions that turn GL commands like

```
glBegin(n);
```

into store operations such as:

```
op->code=GL_Begin;
op->args[0]=n;
```

- which the Blink display server then translates into native GPU programming. In the hope of amortizing the cost of translating BlinkGL over several display updates, Blink uses *Blink Stored Procedures* (SP's). In contrast to display lists which run on the GPU, a Blink stored procedure is a BlinkGL program that runs on the CPU—inside the display server—and in addition to GL calls can perform operations such as register arithmetic and conditional jumps. Stored procedures are sequences of serialized OpenGL commands, with each command consisting of an opcode and a set of parameters. A part of the opcode space is reserved for special operations for virtual register copying, arithmetic, or conditional forward-jumps. External state, such as mouse coordinates or window dimensions,

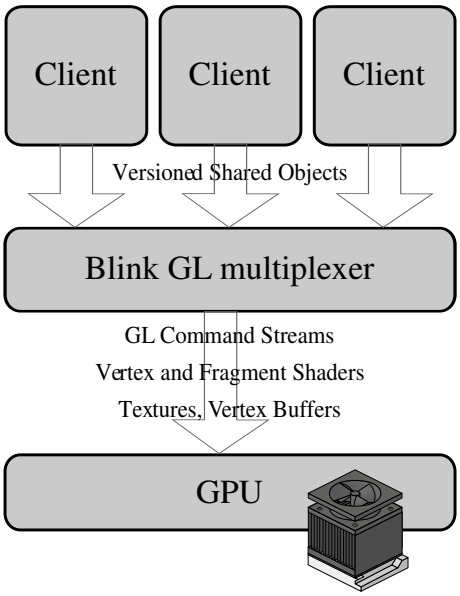

Figure 2: The flow of Information in Blink.

can be read into registers with special BlinkGL calls, processed, and the results given as arguments to other BlinkGL calls that take register arguments. Each SP has a register file residing in memory shared with the client VM, so that registers may be used for communicating simple values between the SP and the VM. The Blink server contains a Just-In-Time (JIT) compiler which converts BlinkGL into native CPU machine code that is invoked as *callbacks* during screen redraw or in response to user input. Because of the simplicity of BlinkGL, JIT compilation is fast, and for OpenGL calls the generated code is of similar quality to the output of `gcc`. Table 1 lists the most common SP's that a Blink client will provide.

2.2 Versioned Shared Objects

Blink differs from GLX and X11 in that it assumes the presence of shared memory between Blink clients and the Blink server. Because there is no virtual memory overlap between VM's, shared objects are referred via opaque integer handles rather than pointers. Figure 2 shows the flow of information in Blink. The communication protocol has been optimized for shared memory, and for allowing reuse of already processed (e.g. JIT-compiled) content. Client VM's communicate with Blink through an array of Versioned Shared Objects (VSO's). A VSO is a in-memory data record containing an object identifier (OID), an object type identifier, a version number, and a list of memory pages containing object data. When Blink receives an update notification from a client VM, it scans through the client's VSO array, looking for updated objects. When a changed or new object is encountered, Blink performs type-specific processing of object contents, such as JIT compilation of stored procedures whose version numbers have changed. Figure 3 shows the first two objects in a VSO array containing a stored procedure and a texture object.

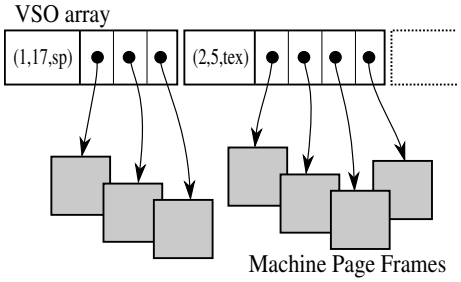

Figure 3: Versioned Shared Objects with OID's 1 and 2 respectively, pointing to physical machine page frames. The first object contains a BlinkGL stored procedure, and the second a texture image.

Callback Name	Executed
`init()`	At first display
`update()`	On client VM request
`reshape()`	On window move or resize
`redraw()`	For each display
`input()`	On user input

Table 1: List of client Stored Procedures callbacks and their times of invocation.

2.3 Static Verification

One problem with using OpenGL for a shared display is that, per the OpenGL specification, the entire display has to be redrawn for every update. In a shared system, either each client must be consulted for each redraw, or client areas must be cached in off-screen buffers, so the screen can be composed by drawing all client buffers to the screen in back-to-front order. The problem with the first solution is that some clients may take very long to redraw, and with the latter that it may be costly in terms of video memory for off-screen buffers and in terms of redundant copy operations. We leave the decision of whether to use off-screen buffers to the client. A video player for example will often decode the current video frame to a texture with the CPU, then use the GPU to display the texture mapped onto a rectangular surface. If the video player is forced to go via an off-screen buffer, the frame will have to be copied twice by the GPU, first when it is drawn to the off-screen buffer, and second when the display is composed by the display system. In such cases, the application is likely to bypass the use of off-screen buffers.

In order to do this safely our JIT-compiler performs simple *static verification* of SP behavior during compilation.

If Blink wishes to confine a client to a certain region of the screen, this may be achieved with the `glScissor()` command which restricts drawing to a subset of the screen, but the client must then be prevented from extending the Scissor rectangle outside of its allowance, and from disabling it altogether. The static verifier allows filtering or adjustment of unwanted client actions, but also enables global optimizations that exploit advance knowledge of client behavior. If the client is known not to use the Z-buffer for depth-ordering of drawing operations, this buffer does not need to be cleared before invoking the client's redraw code, and if the client is not enabling transparency, content covered by the client's rectangle does not need to be redrawn. The constraints placed on a SP during compilation depend on how the SP is used, as not all callbacks are allowed to perform operations that draw to or clear the screen.

The static checker is simple and conservative; it does not attempt to predict the outcomes of conditional jumps. Instead it errs on the side of safety and only assumes that a given command will execute if it is outside of any conditional scope.

2.4 Frame Rate Independence

Often, application content must be adjusted to fit the system screen refresh rate. For instance, a sequence of video frames encoded at 50fps can be adjusted to a 70fps display by repeating some frames two times, but this uneven display rate will make the video appear choppy. An alternative is to interpolate content, for instance by blending subsequent video frames into artificial intermediate frames. Blink SP's can perform simple interpolation and blending, by rendering multiple frames over another with variable transparency. Another advantage of Blink SP's here is that multiple frames may be batch-decoded during the VM's time slice, and then later rendered to the display by the SP. This will lead to fewer context switches and improved system throughput. For 3D content, such *frame rate independence* may be achieved simply by specifying 3D object positions as functions of time instead of as constants hard-coded in GL command arguments. Simple animations such as the classic spinning gears in the GLXGears example application may run mostly independent of the client VM, with gear angles specified as a function of time.

3 Implementation

Blink runs on top of the Xen [2] Virtual Machine Monitor. One VM is given full control of the graphics hardware, and runs as a graphics server for a number of client VM's. By virtue of running on Xen, the graphics server is an ordinary OpenGL application running under Linux,

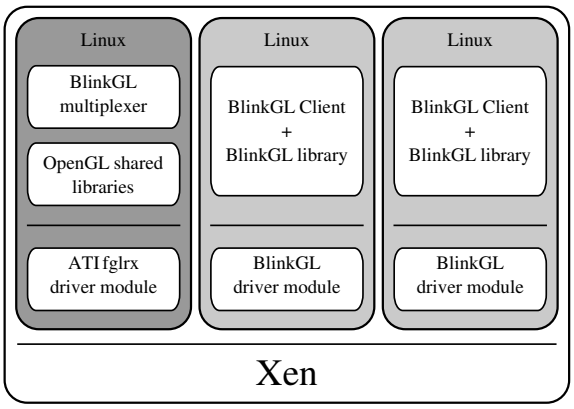

Figure 4: Components of a Blink system. Clients execute as user mode processes in unprivileged VM's. They communicate with the Blink OpenGL Multiplexer through a special kernel module. The Multiplexer runs as a user mode process in a privileged VM which has direct access to graphics hardware.

using native OpenGL drivers.[1] Client VM's place stored procedures and textures in shared memory, structured as Versioned Shared Objects, and signal the server VM. During the next screen update, Blink scans for updated VSO's, performs necessary JIT compilation, and incorporates any changes in the next display update. Currently, most client VM's run Linux, but dedicated application VM's without a operating system are also in development. In Linux VM's, applications access the display through a device that clients can `mmap` to create shared memory segments for sharing VSO's with the Blink GL Multiplexer residing in the display VM.

4 Evaluation

We evaluated Blink on a low-end desktop PC, a Dell Optiplex GX260 PC with a 2GHz single-threaded Intel Pentium4 CPU, with 512kB cache and 1024MB SDRAM. The machine was equipped with an ATI Radeon 9600SE 4xAGP card with 128MB DDR RAM, at the time of writing (January 2006) retailing for less than $70.

We first evaluated JIT compiler performance. We instrumented the compiler to read the Time Stamp Counter

[1] A practical problem faced by any experimental system is how to obtain device drivers. Only a few years back, the task of outfitting a prototype operating system with the most important drivers, e.g. for a network adapter and a disk controller was merely a question of porting the relevant components from an open code base such as FreeBSD or Linux. But with increasing complexity of graphics adapters, and without access to hardware specifications and driver source code for major brand hardware, manual porting of drivers is not a feasible solution. Fortunately, Xen allows most drivers to run inside a VM with no or few changes, and with only minor tweaks we were able to convince the ATI accelerated `fglrx` OpenGL driver to run in a Linux VM under Xen.

Type of input	#Instr.	Compile	Execute
OpenGL-mix	8,034	102 (41) cpi	41 cpi
Arith-mix	8,192	99 (55) cpi	50 cpi

Table 2: Performance of the JIT compiler, cpi = cycles-per-instruction. Numbers in parentheses are from warm-cache runs.

Scenario	Execute
OpenGL-mix Native	552 cpi
OpenGL-mix Blink	554 cpi
OpenGL-mix Blink + JIT	656 cpi

Table 3: Cycles per GL command, natively in the display server, and from JIT-compiled code.

(TSC) before and after compilation, and report average number of CPU cycles spent per input BlinkGL instruction. Because we did not measure OpenGL performance in this test, all call-instructions emitted by the compiler point to a dummy function which simply returns. We measured instructions spent per executed virtual instruction, again with the TSC, and report per virtual instruction averages.

As input we created two programs; the first (OpenGL-mix) is the setup phase of the GLXGears application, repeated 6 times. This program performs various setup operations, followed by upload of a large amount of vertexes for the gear objects with the `glVertex()` command. The second (Arith-mix) is mix of 8 arithmetic operations over two virtual registers, repeated for every combination of the 32 virtual registers. Both programs consist of roughly 8K instructions, performance figures are in table 2. We noticed that subsequent invocations (numbers in parentheses) of the compiler were almost twice as fast as the first one, most likely because of the warmer caches on the second run. We expect larger programs and multiple SP's compiled in a row to gain a similar benefit. Arithmetic operations on virtual floating point registers are costlier than GL calls, as we make little attempt to optimize them because we expect them to be infrequent compared to GL calls.

To validate our claim that for OpenGL-calls the JIT-compiler produces similar-quality to `gcc` (version 3.3.6 run with `-O3`), we also ran the OpenGL-mix code in two scenarios—statically compiled into the display server, and in the JIT-compiled version. Performance of the two programs is nearly identical, as can be seen in table 3. When adding the 102 cpi cost of compilation, for cases where this will not be amortized, we see a modest 18% overhead compared to the native case.

Secondly we measured overall system throughput. For this we ported two applications to display via Blink. This first was the "Gears" OpenGL demo, consisting of three flat-shaded spinning gears, and the second the popular open source "MPlayer" media player. Figure 1 shows both ap-

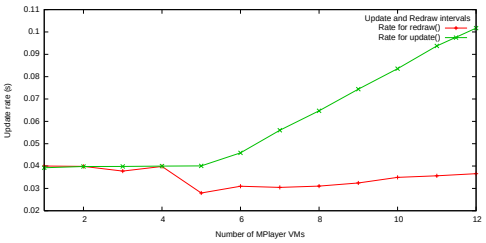

Figure 5: MPlayer performance on Blink.

plications running within Blink. We ran multiple instances of each application in separate VM's, and measured the average time delta between client updates.

MPlayer decodes video frames from a 512x304 resolution DivX file to a memory buffer, which is copied to a texture by the `update` SP, and composed to the screen by the `redraw` SP. We measured time deltas in both SP's, because the former is only run on request by the client, whereas the latter runs every time the shared screen is redrawn. Times for `update` are a measure of the system level of service as seen by each client, where times for `redraw` are a measure of display server responsiveness. MPlayer shows a video running at 25fps, so `update` and `redraw` should be serviced at 40ms intervals. Figure 4 shows time deltas for `update` and `redraw` as functions of the number of client VM's executing. For up to 5 concurrent MPlayer instances, the system is able to provide adequate service to client VM's. However, for up to 12 VM's (at which point the machine ran out of memory), the `redraw` SP is still serviced at more than the required rate. We expect this to be partly due to scheduler interaction between MPlayer's timing code, Blink, and Xen, and partly due to bottlenecks in other parts of the system.

The Gears demo uses SP register arithmetic to transform the gear objects without contacting the client. Gears implements no `update` SP, so we only measure redraws. To gauge the improvement obtained by not having to contact the client for each display update, we run two versions of Gears: GearsSP which uses stored procedure arithmetic and avoids context switching, and GearsSwitch which is updated by the client for each screen redraw. Figure 4 shows time-deltas as a function of the number of VM's. We see that GearsSP is able to maintain our target frame rate (50fps for this test) for more than twice the amount of VM's than GearsSwitch, due to its avoidance of context switches.

5 Related Work

Safe-language kernel extensions have been proposed before, e.g. in the SPIN [3] and Exokernel [4] operating sys-

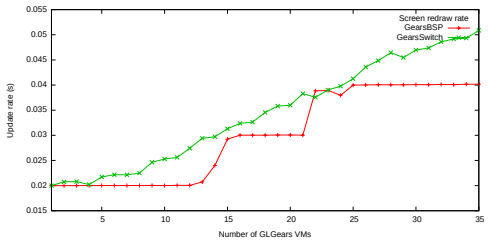

Figure 6: GLGears performance on Blink.

tems. In Exokernel, applications would download code such as packet filters into the kernel, where it would be compiled into native code and check for misbehavior before letting it execute. Our work is an extension of these ideas in that we let applications and their in-server extensions (SP's) exist in a producer-consumer relationship, as in the case of a video player decoding multiple frames during a single time-slice for subsequent display, and that we reason about SP behavior not only for safety, but also for performing global optimizations based on this knowledge.

Our work builds on OpenGL, a display system derived from IRIS GL, developed by Silicon Graphics Inc. GLX is an extension to the X11 protocol for integrating OpenGL applications into the X Window System, and for remote OpenGL display. We have not compared the performance of our system to GLX, because the version available (XFree86/X.org) on Linux currently cannot utilize accelerated hardware for remote connections.

Specialized secure 2D display systems for microkernels have been described by Feske et.al. [5] for L4, and by Shapiro et.al. [6] for EROS. Both systems make use of shared memory graphics buffers between client and server, and both describe mechanisms for secure labeling of window contents. Our work addresses the growing need for 3D acceleration but, other than mandating a yellow border around client areas, does not currently address labeling in detail.

6 Future Work

While we expect our work to be useful for Xen systems in general, the scope of our future work is broader. Our long-term aim is to build a complete desktop operating system around Xen Virtual Machines, in which Blink will be the framework for user interaction. Our current prototype is limited in a number of respects; first of all we do not account resources such as video memory used by clients, and we make no attempt at preempting very long SP's. In future work we will implement features for cooperative multitasking, in the form of a yield instruction periodically inserted by the compiler, and we will implement resource accounting. Another problem is that we currently lack support for multiplexing of clients *within* a VM, as each VM can currently only use a single OpenGL hardware context. We are planning to extend our interface by allowing each VM access to multiple hardware contexts.

7 Conclusion

With this work we have shown that strong isolation does not have to entail poor graphical performance, and that virtualized applications do not have to be limited to "secretarial" tasks such as word processing. With the help of a simple and fast JIT-compiler and a shared-memory protocol, we have managed to reduce client/server communication overheads and to amortize translation costs over multiple display updates. Lastly, we have shown ways of composing multiple applications to the same screen without a need for off-screen buffers, and of and performing other types of optimizations by exploiting advance knowledge of client behavior obtained during compilation.

References

[1] Mark J. Kilgard. Realizing opengl: two implementations of one architecture. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 45–55, New York, NY, USA, 1997. ACM Press.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, New York, NY, USA, 1995. ACM Press.

[4] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 52–65, New York, NY, USA, 1997. ACM Press.

[5] N. Feske and C. Helmuth. A Nitpickers guide to a minimal-complexity secure GUI. In *In Proceedings of the 21st Annual IEEE Computer Security Applications Conference*, pages 85–94, December 2005.

[6] Jonathan S. Shaprio, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS trusted window system. In *Proceedings of the Thirteenth USENIX Security Symposium*, San Diego, CA, August 2004.