

# Blink: Advanced Display Multiplexing for Virtualized Applications

Jacob Gorm Hansen

*Department of Computer Science, University of Copenhagen, Denmark*

## Abstract

*Providing untrusted applications with shared and dependable access to modern display hardware is of increasing importance. Our new display system, called Blink, safely multiplexes complex graphical content from multiple untrusted virtual machines onto a single Graphics Processing Unit (GPU). Blink does not allow clients to program the GPU directly, but instead provides a virtual processor abstraction to which they can program. Blink executes virtual processor programs and controls the GPU on behalf of the client, in a manner that reduces processing and context switching overheads. To achieve performance and safety, Blink employs just-in-time compilation and simple static analysis.*

## 1 Introduction

In addition to their popularity in data centers, Virtual Machines (VMs) are increasingly deployed on client machines, e.g. to allow for compartmentalization of untrusted software downloaded from the Internet [1], or for ease of management [2]. In addition to fast and dependable access to I/O subsystems such as the disk and the network, graphical performance—for games, video conferencing, or simulation—is of critical importance in such *Virtualized Application* scenarios.

Compared to other I/O subsystems, 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 full-screen video. This is evidenced by the fact that the major operating systems all provide “direct” avenues of contacting the graphics card, largely without operating system involvement. However, some level of cooperation between applications is necessary, so that multiple applications are able to share access to screen buffers and the Graphics Processing Unit (GPU). In Linux for example, applications are required to respect a lock provided by the kernel when accessing GPU hardware registers, as otherwise the GPU may crash [3].

Untrusted software running inside VMs cannot be expected to behave cooperatively, and so VMs cannot be trusted with direct hardware access, and a layer of abstraction between client and hardware is necessary. One way to implement such a layer is by letting clients program using a high-level API (rather than programming hardware registers directly), and have the display system interpret or translate API commands into native programming of the GPU, after having verified their safety.



Figure 1: Blink running GLGears and MPlayer.

Untrusted virtualized applications should be expected to try to trick the user into entering confidential information such as passwords into spoofed dialog boxes, to attempt to crash the GPU or entire machine, or to consume all system-resources in a Denial-of-Service (DoS) attack. To counter these types of threats, a safe display system needs the following properties:

1. Application drawing operations must be confined to clearly labeled areas of the screen, to prevent interface spoofing attacks.
2. Application GPU access must be restricted to a safe subset of commands, and commands that may have unsafe side-effects should be dropped completely or modified by the system in a way that makes them safe. Similarly, applications should be prevented from exploiting bugs in graphics hardware or driver APIs, whether for privilege-elevation or DoS.
3. All application resource-use should be checked against a global or per-application policy, to prevent one application from commanding all resources (such as texture or video memory), at the expense of others.

## 2 Design and Implementation

This paper describes Blink, a safe and efficient display system for virtual machines. Blink runs on top of the Xen [4] virtual machine monitor. One VM is given full control of the graphics hardware, and runs as a graphics server for a number of client VMs, as shown in figure 2.

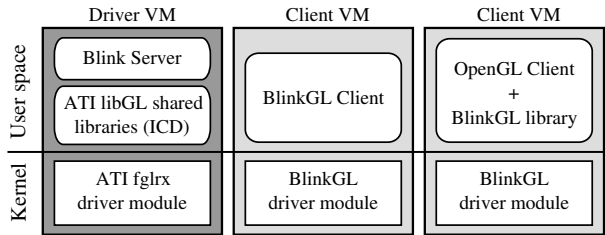


Figure 2: Blink client VMs communicate with the server VM using shared memory. Dedicated BlinkGL clients access the BlinkGL primitives directly, and unmodified OpenGL clients go through a wrapper library.

The Blink server provides client applications with mediated access to the features of the GPU, while making sure that the client is unable to violate any of the safety goals listed above.

## 2.1 GPU Programming

Most modern GPU programming is done using standardized APIs such as OpenGL [5] or Direct3D [6]. Our work focuses on OpenGL, though our results may generalize to Direct3D as well. An OpenGL (GL) program is a sequence of API-calls, of the form  $glName(args)$ , where  $Name$  is the name of the called GL command. GL commands may modify state such as a transformation applied to all drawing operations via matrix multiplication, lighting and texturing parameters. Some of the commands result in drawing to the screen, and are enclosed in  $glBegin()$  and  $glEnd()$  pairs, e.g., the following GL program draws a triangle:

```
glBegin(GL_TRIANGLES);
glVertex(0,0);
glVertex(1,0);
glVertex(1,1);
glEnd();
```

An OpenGL API implementation is often split into kernel and user space parts. The kernel space part provides primitives for accessing graphics card DMA buffers and hardware registers, and the user space part—a GPU-specific shared library known as the Installable Client Driver (ICD)—accesses these primitives to translate GL API calls into programming of the GPU. This split responsibilities model will not work if trying to provide a VM with a Virtual OpenGL abstraction, because VMs cannot be expected to load a hardware-specific ICD, as doing so would break the abstraction of having a constant interface across different types of hardware.

## 2.2 BlinkGL Stored Procedures

In the GLX extension to X11, GL 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 GL 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 GL commands in the client, and deserializes and runs them in the server. Blink clients program to “BlinkGL”, a superset of the OpenGL API which serializes its input into memory shared with the display server. BlinkGL is a superset of OpenGL because it also contains extended functions for communicating with the Blink windowing environment, and opcodes for virtual register arithmetic.

In the hope of amortizing the cost of translating BlinkGL over several display updates, Blink uses *Blink Stored Procedures* (SPs). 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 GL 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, can be read into registers with special BlinkGL calls, processed, and the results given as arguments to other BlinkGL calls that take register arguments. The Blink server contains a Just-In-Time (JIT) compiler which converts BlinkGL into native CPU machine code that is invoked during screen redraw or in response to user input. Because of the simplicity of BlinkGL, JIT compilation is fast, and for GL calls the generated code is of similar quality to the output of `gcc`. Apart from amortizing translation costs, the use of SPs also has two additional benefits: CPU context switching is greatly reduced because each client does not have to be consulted upon every display update, and in many cases the use of large *off-screen buffers* to hold client areas can be avoided by drawing client areas on the fly by invoking the relevant SPs. Table 1 lists the most common SPs that a Blink client will provide.

During compilation, the safety of each SP command is checked, so that out-of-bounds parameters or illegal command sequences may be detected before the SP is allowed to run. Out-of-bounds parameters are modified to be safe, and unsafe commands are skipped altogether. This ensures that once compiled, SPs are safe to execute within the context of the Blink server. This static analysis is simple and conservative; it does not attempt to predict the outcomes of conditionals. Instead it errs on the side of safety and only assumes that a given mandatory command will execute if it is outside of any conditional scope. The reverse is true for commands that may have

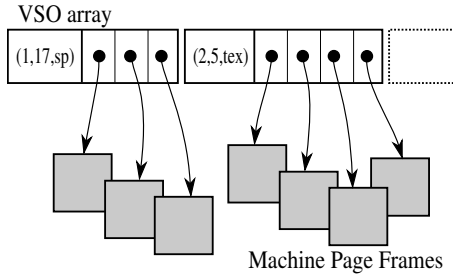


Figure 3: Versioned Shared Objects with OID’s 1 and 2 respectively, pointing to physical machine page frames. The first object contains a BlinkGL stored procedure, and the second a texture image.

dangerous side-effects—they are assumed to always execute regardless of any enclosing conditional scopes.

### 2.3 Versioned Shared Objects

Client VMs communicate with Blink through an array of Versioned Shared Objects (VSO’s). A VSO is an 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, and incorporates any changes in the next display update. Each VM may maintain several VSO arrays, to accommodate the use of multiple OpenGL hardware contexts for different clients within the VM, and care is taken to avoid scanning unmodified arrays. The scan is linear in the number of VSO’s in the array, but more elaborate data structures can be envisioned if the scalability of the current approach becomes an issue. Figure 3 shows the first two objects in a VSO array containing a stored procedure and a texture object. Most GL commands are called directly by the JIT’ed machine code, but commands taking pointer arguments are treated specially. For example, the *glTexImage2D()* command uploads a texture from a main memory address to the graphics card. Blink’s version of the command instead takes a VSO OID, which is then resolved into a memory address during compilation.

### 2.4 Virtualizing Standard OpenGL

Rather than expecting all existing OpenGL software to be rewritten as BlinkGL, Blink also contains a compatibility mode which allows unmodified OpenGL software to display via Blink. This mode is implemented as a special BlinkGL command, called *glEval()*. This command invokes an interpreter which understands serialized standard OpenGL and executes it at once, and by combining client-side driver code with the `update()`

SP in a producer-consumer pair, it is possible to transparently host unmodified OpenGL<sup>1</sup> software. Like other display systems that compose multiple OpenGL clients to a shared display, the compatibility-mode will often need off-screen buffers to avoid flicker or artifacts. However, this functionality does not have to be supported by the Blink display server. Instead, client SPs handle this by executing *glEval()* within the context of the off-screen buffer, and drawing the off-screen buffer onto a texture during `redraw()`.

Because the interpreted code only runs once, the overhead of full JIT compilation is not justifiable. Instead, Blink implements a specialized interpreter for serialized OpenGL streams. When writing an interpreter, the choice is basically between implementing a “giant switch” with a case for each opcode number, or the “threaded code” approach, with a jump-table with an entry for each opcode number, both of which may be implemented as portable C code. On modern architectures, both approaches suffer from poor branch prediction accuracy, as low as 2%-50% according to [7], due to a large number of indirect branches. Furthermore, the ratio of arguments to opcodes is high for serialized OpenGL, so a traditional interpreter has to spend much of its time copying arguments from the input program to the parameter stack of the called GL API function. For Blink, we implemented an x86-specific hybrid JIT compiler and interpreter designed specifically for the characteristics of OpenGL. When using a purely stack-based calling convention (as is the common case when calling shared libraries on x86), parameter copying can be avoided altogether by pointing the stack pointer directly to opcode parameters before each API call. Branch prediction can be improved by using a minimal code-generator which converts input programs into machine code sequences of API calls interleaved with increments of the stack pointer register, so that arguments are consumed directly from the input program without copying.<sup>2</sup> This approach is platform-specific, but does offer better performance than platform neutral alternatives such as a giant-switch interpreter, which may be provided as a fall-back on other platforms.

## 3 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

<sup>1</sup>The Blink client driver also supports displaying the Linux kernel framebuffer and X11 on an OpenGL texture, left out for brevity.

<sup>2</sup>The reader will notice that using the stack in this manner destroys the input program, and that space must be available at the head of the program, as otherwise the stack will overflow. The first is not a problem, because programs need only be interpreted once. The latter we deal with by proper use of virtual memory mappings.

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

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

4xAGP card with 128MB DDR RAM, using the proprietary ATI `fglrx` OpenGL display driver.

We first evaluated JIT compiler performance. We instrumented the compiler to read the CPU time stamp counter 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, and report per virtual instruction averages.

As input we created two programs; the first (OpenGL-mix) is the setup phase of the GLGears application, repeated 6 times. This program performs various setup operations, followed by upload of a large amount of vertices 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 SPs 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, as they are expected to appear infrequently compared to GL calls.

To validate our claim that for GL-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 ported the classic GLGears demo to display using Blink Stored Procedures. GLGears displays three spinning gears 3D rendered into a 512x512 window. We ran multiple GLGears instances, each in a separate VM, and measured the average time deltas between client updates.

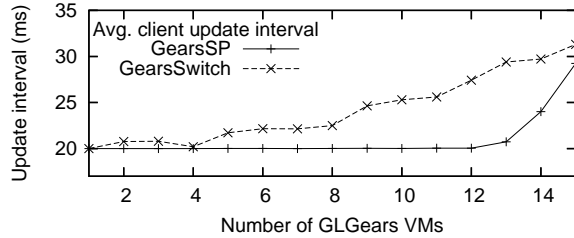


Figure 4: Averaged delay between frame updates of the OpenGL Gears demo. GearsSP uses stored procedures; GearsSwitch context switches between VMs.

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

Table 2: Cycles-per-instruction for the JIT compiled SPs. Numbers in parentheses with warm cache.

The Blink version of GLGears 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 3 shows time-deltas as a function of the number of VMs. We see that GearsSP is able to maintain our target frame rate of 50 frames per second for more than three times the amount of VMs than GearsSwitch, due to its avoidance of CPU context switches.

Blink achieves the safety properties listed in section 1. Goal 1 is addressed by enclosing each VM inside a clearly labeled, yellow border, and by adjusting client drawing operations to be clipped against the client area rectangle. Goal 2 is addressed by the JIT compiler and interpreter, in that unsafe API calls are either skipped or adjusted to make them safe. Blink can meet goal 3 since resources are tracked per client, although the current implementation does not enforce a policy on those resources.

## 4 Related Work

GLX is an extension to the X11 protocol for integrating OpenGL applications into the X Window System, and for remote OpenGL display. However, the version available (XFree86/X.org) on Linux currently cannot utilize accelerated hardware for remote connections. Recently, the WireGL/Chromium [8] project has provided remote OpenGL acceleration for rendering clusters. While the aim is slightly different, i.e. to combine the rendering power of multiple GPU's, there is also a large over-

Scenario	Execute
Native ATI driver call	552 cpi
Blink Stored Procedure	554 cpi
Blink Stored Procedure with compilation	656 cpi

Table 3: Cycles per OpenGL command, when executed using the native driver, or through JIT compiled Blink code.

lap in that both Blink and WireGL/Chromium serialize and later interpret GL command streams. In addition, Blink employs JIT compilation and static verification of stored procedures, and saves the overhead of passing all data through a sender-receiver pair of network protocol stacks by optimizing for the common case of client and server being on the same physical machine.

Specialized secure 2D display systems for microkernels have been described for L4 [9] and EROS [10]. 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 currently our labeling mechanism is rather crude. VMWare Inc. has recently announced experimental support for Direct3D acceleration in their VMWare Workstation product. For legal reasons, we have chosen not to benchmark Blink against VMWare Workstation at this time.

## 5 Future Work

The precursor to Blink was the Tahoma [1] Display System (TDS). TDS was completely stateless, with the display acting as a cache of client content. Among other benefits, this allowed for checkpointing and migration [11] across hosts. BlinkGL clients can be implemented to be stateless, e.g. the server just calls the `init()` SP to recreate any missing state, but the transparently virtualized OpenGL programs cannot. In the future, we are planning to make the transparent virtualization stateless by tracking GL state as is done in [12] in the client-side Blink driver.

We currently do not address per-pixel or per-vertex GPU programmability, though we believe this should be relatively straight-forward by exploiting a serialized form of the OpenGL Shader Language (GLSL) or similar.

## 6 Conclusion

Blink demonstrates that today’s advanced display devices can be multiplexed in a safe manner without poor performance. Blink emphasizes achieving safety with high performance, since enforcement overhead is often the main obstacle to the adoption of security mechanisms. In particular, a less efficient enforcement mechanism might not scale with the currently rapid growth in

GPU capabilities.

Blink achieves safety by using a simple, fast JIT compiler and a shared-memory protocol that also helps reduce the cost of client/server communication. Blink further reduces overhead by amortizing JIT translation costs over multiple display updates and composing multiple applications to the same screen without a need for off-screen buffers. As a result, Blink allows for safe, practical display sharing even between soft-realtime graphics applications and legacy code.

## 7 Acknowledgments

The author wishes to thank Úlfar Erlingsson for his help in preparing this paper, and Jørgen Sværke Hansen and Eske Christiansen for their comments on earlier drafts.

## References

- [1] Richard S. Cox, Jacob G. Hansen, Steven D. Gribble, and Henry M. Levy. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.
- [2] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M.S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation*, pages 259–272, May 2005.
- [3] Rickard E. Faith and Kevin E. Martin. A security analysis of the direct rendering infrastructure, 1999. <http://precisioninsight.com/dr/security.html> (archive.org copy).
- [4] 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 *Proc. of the 19th ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.
- [5] 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.
- [6] David Blythe. The Direct3D 10 System. In *Proc. of the 33rd ACM SIGGRAPH conference*, 2006.
- [7] M. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters, 2003.
- [8] Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.
- [9] N. Feske and C. Helmuth. A Nitpicker’s guide to a minimal-complexity secure GUI. In *In Proceedings of the 21st Annual IEEE Computer Security Applications Conference*, pages 85–94, December 2005.
- [10] 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.
- [11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Networked Systems Design and Implementation NSDI '05*, May 2005.
- [12] Tracking graphics state for networked rendering. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–95, 2000.