Blink: Advanced Display Multiplexing for Virtualized Applications

Jacob Gorm Hansen
Department of Computer Science
University of Copenhagen
Denmark
jacobg@diku.dk

ABSTRACT

Providing untrusted applications with shared and safe 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 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. Blink provides its own stored procedure abstraction for efficient hardware access, but also supports fast emulation of legacy OpenGL programs. To achieve performance and safety, Blink employs just-in-time compilation and simple program inspection.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Security kernels*; D.4.8 [**Operating Systems**]: Performance; I.3.2 [**Computer Graphics**]: Graphics Systems

General Terms

Performance, Experimentation, Security

Keywords

Graphics, virtualization, hardware acceleration, just-in-time compilation, interpretation

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 [7], or for ease of management [5]. VM technology is quickly becoming commoditized, and it is conceivable that future desktop operating systems will ship with VM-style containers as a standard feature. While initially users may be willing to put up with seeing multiple desktops in a simple picture-in-picture fashion, over time the demand for more integrated and seamless experiences will grow. Users will expect virtualized applications to blend in nicely, and will want to make use of graphics hardware acceleration, for games, simulations, and video conferencing. Our work attempts to address this need by providing virtual machines with access to the powerful accelerated drawing features of modern display hardware, without compromising the safety guarantees of the VM model.

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 programming the graphics card, largely without operating system involvement, but at the danger of being able to crash the graphics adapter or lock up the entire machine [11]. Because of this danger, untrusted software running inside VMs should not be given direct hardware access, and a layer of indirection between client and hardware is necessary. Such a layer should also provide a hardware-independent abstraction, to allow a VM to run unmodified across different types of graphics adapters. One way of implementing this layer is by letting clients program to a high-level API, and have a trusted display system translate API commands into programming of the actual hardware. The trusted translation

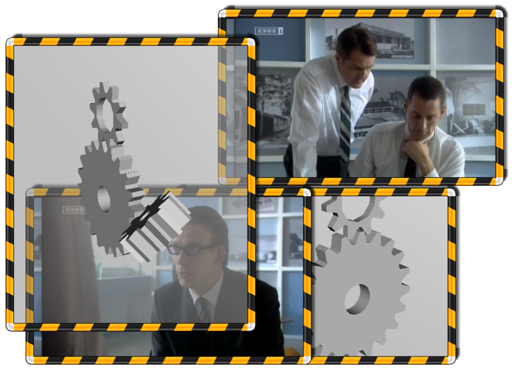
Figure 1: Blink running GLGears and MPlayer.

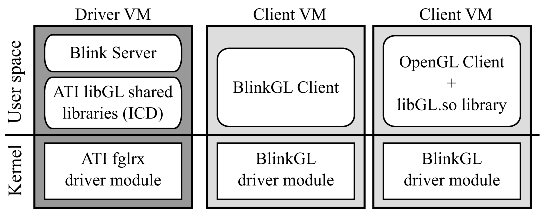

Figure 2: Blink client VMs communicate with the server VM using shared memory. Dedicated BlinkGL clients access the BlinkGL primitives directly, and unmodified OpenGL clients go through an OpenGL driver library.

step verifies the safety of each API command, and then forwards commands to the graphics card driver. In this way, applications can be prevented from bypassing address space protection with malicious DMA operations, or exploiting bugs in graphics hardware or driver APIs. All use of hardware resources can be tracked and subjected to a policy that prevents one application from causing starvation, *e.g.* by consuming all memory on the graphics card.

2. DESIGN AND IMPLEMENTATION

This paper describes Blink, a prototype system for providing untrusted VMs with safe and efficient access to modern display hardware. Blink aims to be as safe, simple and flexible as possible, while at the same time retaining full performance.

The goal of Blink is to serve as the display component of a system where each application is encapsulated inside its own VM. Blink is backwards-compatible with existing software, through emulation layers for X11, kernel framebuffers, or OpenGL, but performance may be enhanced by adjusting applications to bypass these layers and use Blink directly.

The Blink prototype runs on top of the Xen [1] virtual machine monitor. The Blink display server runs as a regular user-application inside a Linux guest VM, using commercially developed device drivers for graphics hardware access. The VM running the Blink server mediates access to the graphics hardware, and clients running inside untrusted VMs talk to this server using a shared-memory protocol, as shown in figure 2.

In the remainder of this section, we first briefly introduce 3D programming using OpenGL, and the challenges faced when trying to make an OpenGL-like abstraction to clients in separate protection domains. From there, we describe the key points of our design, and how they have been implemented in our prototype.

2.1 GPU Programming

Most modern GPU programming is done using APIs such as OpenGL [15] or Direct3D [3]. Our work focuses on OpenGL, a standardized API that is available on most platforms. 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. Some commands modify state such as transformation matrices, lighting and texturing parameters, and some result in immediate drawing to the screen. Drawing commands 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 two parts. The kernel 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. Thus, the output of an OpenGL program can be re-targeted to new hardware, or to a virtualized abstraction, by replacing the ICD installation files.

2.2 Off-Screen Buffers

In a traditional windowing system, each application owns one or more clearly defined rectangular regions of the screen, and such systems expend much effort clipping drawing operations to correctly fall within these regions. With the advent of faster graphics cards with greater amounts of video memory, a new model has become dominant. Pioneered by the MacOSX "Quartz" composited display system, the new approach is to give each application its own *off-screen buffer*, a contiguous range of video memory to which the application may draw freely. The display system runs a special *compositor* process which takes care of painting all of the off-screen buffers to the screen, in back-to-front order. The benefits of this model are simplicity and the support for advanced effects such as translucent or zooming windows. The problem with this model is the great amount of memory required for off-screen buffers, to avoid context-switching back and forth between the server and all clients every time the screen is redrawn. There are also cases where off-screen buffers are likely to contain redundant information, *e.g.* in the case of a video player. A video player 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 our proposed system, the decision of whether and how to use off-screen buffers is left to applications. As we shall see, this flexibility is achieved by letting applications execute code as *Stored Procedures (SPs)* inside the display server.

2.3 BlinkGL Stored Procedures

Virtual machine systems often exploit existing wire protocols when communicating between a VM and the surrounding host environment. For instance, a remote desktop protocol such as RDP or VNC makes connecting a VM to the display relatively straight-forward. VNC and RDP communicate screen updates in terms of pixels and rectangles. Remote display of 3D may be achieved in these systems by rendering the final image server-side [8], though at an additional cost in bandwidth, and with the risk of server CPU or GPU contention.

When wishing to render 3D content locally on the client, the common solution is to serialize rendering commands, into batches of Remote Procedure Calls (RPCs [2]), which are then communicated over the wire protocol. In addition to the cost of communication, this method carries a performance overhead, due to the cost of serializing and de-serializing command streams. In OpenGL, translation costs can be amortized to some extent by the use of *display lists*, macro-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 extends the display list abstraction into more general and flexible *BlinkGL* stored procedures. Because stored procedures are richer in functionality than display lists, they can handle simple user interactions—*e.g.* redrawing the mouse cursor or highlighting a pushbutton in response to a mouse rollover—independently of the application VM.

BlinkGL is a super-set of OpenGL. BlinkGL stored procedures run on the CPU—inside the display server—and in addition to GL commands they can also perform simple arithmetic and control operations. Stored procedures are sequences of serialized BlinkGL 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 that 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 a C-compiler. 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 off-screen buffers to hold client areas can be avoided by drawing client areas with SPs on the fly.

2.4 Program Inspection

Some aspects of SP execution require additional checking, *e.g.* to prevent clients drawing outside of their windows. During JIT compilation, commands and parameters are inspected, and filtered or adjusted according to various policies. This inspection allows the Blink server to weed out 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, then the Z-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. 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.

2.5 Versioned Shared Objects

VMs residing on the same physical host may communicate through shared memory, instead of using wire protocols that are likely to introduce extra data copying, especially problematic for large texture or framebuffer 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

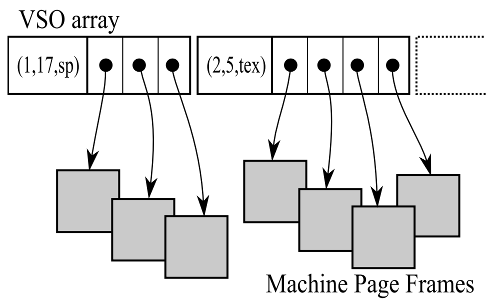

Figure 3: Versioned Shared Objects with OID's 1 and 2 respectively, pointing to physical machine page frames. The first object contains a BlinkGL stored procedure, and the second a texture image.

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. Texture data gets DMA'ed from application to video memory, and inter-VM copying is avoided.

2.6 Virtualizing Standard OpenGL

Rather than expecting all existing OpenGL software to be rewritten as BlinkGL, Blink also contains a compatibility wrapper which allows unmodified OpenGL software to display via Blink. This wrapper is implemented as a custom, client-side ICD, with the help of an additional BlinkGL command, called *glEval()*, in the Blink server.

The *glEval* command invokes an interpreter that understands serialized standard OpenGL and executes it immediately, and by combining client-side driver code with a server-side SP in a producer-consumer pair, it is possible to transparently host unmodified OpenGL software. Like other display systems that compose multiple OpenGL clients to a shared display, the wrapper needs off-screen buffers to avoid flicker or artifacts, and to allow arbitrary effects such as transparent windows. However, this functionality is not supported by the Blink display server. Instead, SPs that are part of the client ICD handle this by executing *glEval()* in the context of an off-screen buffer, and drawing the off-screen buffer onto a texture during the redraw SP. This way, Blink is able to host unmodified OpenGL applications, and to subject them to arbitrary transformations when composing the screen.

For the code interpreted by *glEval()*, 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%, due to a large number of indirect branches [10]. 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. As a more efficient alternative, we designed a new interpretation technique which we refer to as *Stack Replay*.

Stack Replay is a simple and fast interpretation technique designed specifically for the characteristics of OpenGL and similar sequences of batched remote procedure calls. It is based on the observation that a serialized OpenGL program is little more than an array of call-stacks. This means that 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.[1] This approach is platform-specific, but does offer better performance than platform neutral alternatives such as a giant-switch interpreter, which could still be provided as a fall-back on other platforms. The platform-specific parts of the interpreter consist of a seven line main-loop in C, and 10 lines of inline x86 assembly pre- and postambles, used when calling a batch of generated code.

When using Linux as the guest VM, the Blink client driver also supports displaying the Linux kernel framebuffer and X11 on an OpenGL texture. This is accomplished by adding a framebuffer driver, which maps a main memory buffer onto a BlinkGL texture, to the guest Linux kernel. Using this driver it is possible to run legacy text mode and X11 applications on top of Blink. Figure 4 shows X11 running on top of a BlinkGL texture.

2.7 Display State Recovery

The precursor to Blink was the 2D Tahoma [7] Display System (TDS). TDS was completely stateless, with the display acting merely as a cache of client content. Among other benefits, this allowed for checkpointing and migration [6] of VMs. BlinkGL clients can be implemented to be stateless, *e.g.* the server just calls the client's initialization SP to recreate any missing state, but the transparently virtualized OpenGL programs are not stateless out of the box, because OpenGL is itself a stateful protocol. To solve this problem, we have added a state-tracking facility to the client-side OpenGL emulation layer, so that copies of all relevant

[1]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.

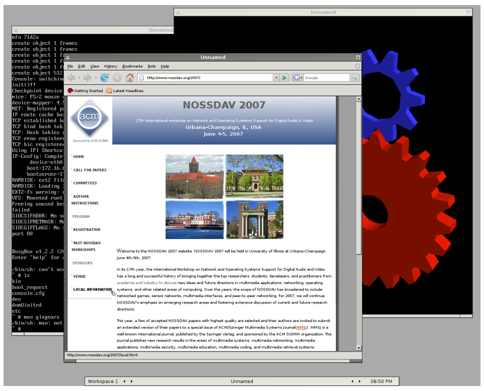

Figure 4: Blink running in root-less mode on top of X11. Here, the kernel framebuffer is shown, both running a text mode console and with X11 in a VM, running a web browser.

state are maintained inside the address space of the calling application. Display lists and textures are captured verbatim, and the effects of transformation to the OpenGL matrix stack are tracked, so that they can later be replayed. Our approach here is in many ways similar to the one described by Buck et.al. [4], but with the purpose of being able to recreate lost state, rather than as a performance optimization.

3. EVALUATION

In this section we attempt to measure key aspects of Blink's performance. At the micro-level we measure the overheads introduced by JIT compilation and interpretation, and at the macro-level we measure overall system throughput for a simple application. These benchmarks do not claim to be an exhaustive evaluation, but they give a good indication of how Blink performs relative to native OpenGL code.

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, using ATI's proprietary 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 six 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 1. We noticed that subsequent invocations (numbers in parentheses) of the compiler

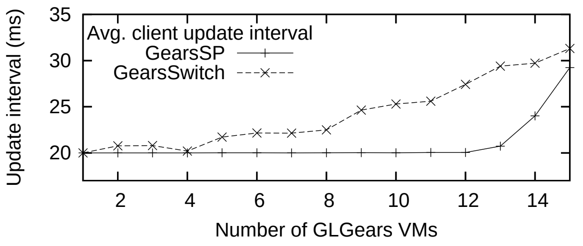

Figure 5: Averaged delay between frame updates of the OpenGL Gears demo. GearsSP uses stored procedures; GearsSwitch context switches between VMs.

Type of input	#Instr.	Compile	Execute
OpenGL JIT	8,034	102 (41) cpi	41 cpi
Arithmetic JIT	8,192	99 (55) cpi	50 cpi
OpenGL interpr.	8,191	0 cpi	59 cpi

Table 1: Cycles-per-instruction for the JIT compiled SPs, and for interpreted OpenGL streams, on a 2GHz Pentium 4. Numbers in parentheses with warm cache.

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 sequence to see similar performance gains. Arithmetic operations on virtual floating point registers are costlier than GL calls, as we make little attempt to optimize them. Finally, we measure the cost of interpretation. We call the interpreter with an input program similar to OpenGL-mix, and measure the cost of interpretation using Stack Replay. As before, all calls are to dummy functions that simply return immediately. We see that the use of interpretation adds about 44% overhead compared with the execution of JIT compiled code (59 vs. 41 cpi respectively). Thus there is benefit to using the JIT compiler in cases where the cost can be amortized, but the interpreter yields reasonable performance as well.

To validate our claim that for GL-calls the JIT compiler produces code of similar-quality as `gcc`, 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 2.

Secondly we measured overall system throughput. For this we 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.

The Blink version of GLGears uses SP register arithmetic to transform the gear objects independently of the client. 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 5 shows time-deltas as a function of the number of VMs. We see that GearsSP is able to maintain a steady 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.

Scenario	Execute
Native ATI driver call (gcc 3.3.6)	552 cpi
Blink Stored Procedure	554 cpi

Table 2: Cycles per OpenGL command, when executed using the native driver, or through JIT compiled code.

Type of input	Redraw rate
Linux Direct Rendering	6834 fps
BlinkGL JIT compiled	6564 fps
Interpreted OpenGL on Blink	4940 fps

Table 3: Frames-per-second, for a 512x512 GLGears demo, on an Intel Core Duo 2, with a 256MB nVidia GeForce 7900GS graphics card.

The final test was run on more modern hardware, a 2.1GHz Intel Core Duo 2 CPU, equipped with a 256MB nVidia GeForce 7900GS graphics card, and nVidia's proprietary graphics driver. Again, we are using the GLGears OpenGL demo as our benchmark, measuring total frames per second. Results are tabulated in table 3. In line with our previous results, BlinkGL stored procedures execute at close to native speed, while the overhead of interpreting command streams and off-screen buffer rendering results in approximately 25% drop in frames per second, for the OpenGL emulation case. We attribute most of this overhead to the extra copying needed for the off-screen buffer.

4. RELATED WORK

The X Window System (X11) facilitates remote display, and the protocol's extensibility means that other protocols such as OpenGL can be tunneled inside X connections. Unfortunately, currently popular versions of X have very large code bases, making them hard to trust security-wise. Efforts to create trusted X implementations [9] have not had lasting impact, and many of the assumptions on which X is based (*e.g.* the need for remote font servers or support for monochrome or color-mapped displays) are no longer relevant. For these reasons, we have chosen not to base our work on X.

Recently, the VMGL [16] project has adopted Chromium [14] to work across VM boundaries, over TCP/IP. Like Blink, VMGL supports state tracking for use in VM checkpointing and migration, and VMGL currently implements a larger subset of OpenGL than Blink. Blink employs JIT compilation and static verification of stored procedures, and saves the overhead of passing all data through a pair of network protocol stacks by optimizing for the common case of client and server being on the same physical machine. VMWare Inc. has also announced experimental support for Direct3D virtualization in their desktop products. Blink does not support Direct3D, but a potential workaround is to run a Direct3D emulator on top of OpenGL, *e.g.* Transgaming's Cedega [2] technology.

Specialized secure 2D display systems for microkernels have been described for L4 [12] and EROS [18]. Both systems make use of shared memory graphics buffers between client and server, and both describe mechanisms for secure

[2]http://www.transgaming.com

labeling of window contents. Our work addresses the growing need for 3D acceleration but currently our labeling mechanism is rather crude.

5. FUTURE WORK

Blink allows an untrusted application to drive a 3D display, and allows the application to be implemented in a stateless manner. We are currently working on combining Blink with the self-migration and checkpointing mechanism described in previous work [13]. This combination allows us to live-checkpoint Blink VMs to stable storage, *e.g.* to a hard- or flash drive, and to fork and live-migrate application VMs across the network. This is similar to the system proposed by Potter et.al. [17], but with the difference that checkpointing is performed from *within* the VM, and may be done in the background without interrupting the user.

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. In addition, Blink remains backwards-compatible by employing an efficient and novel batch RPC interpretation technique, knows as Stack Replay. 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 S. Hansen, Eske Christiansen, Eric Jul, Niraj Tolia, and the anonymous reviewers, for their suggestions and comments on previous versions.

8. REFERENCES

[1] 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. of the 19th ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[3] D. Blythe. The Direct3D 10 System. In *Proc. of the 33rd ACM SIGGRAPH conference*, 2006.

[4] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–95, 2000.

[5] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. 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.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Networked Systems Design and Implementation NSDI '05*, May 2005.

[7] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[8] D. De Winter, P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt, and P. Demeester. A Hybrid Thin-Client protocol for Multimedia Streaming and Interactive Gaming Applications. In *the 16th Annual International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 2006.

[9] J. Epstein, J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, B. Dancer, C. Martin, M. Branstad, G. Benson, and D. Rothnie. A high-assurance window system prototype. *Journal of Computer Security*, 2(2-3):159–190, 1993.

[10] M. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters, 2003.

[11] R. E. Faith and K. E. Martin. A security analysis of the direct rendering infrastructure, 1999. http://precisioninsight.com/dr/security.html (archive.org copy).

[12] 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.

[13] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW 2004)*, pages 126–130, 2004.

[14] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[15] M. 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.

[16] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent graphics acceleration. In *Proceedings of VEE 2007*. ACM Press, June 2007.

[17] S. Potter and J. Nieh. Webpod: persistent web browsing sessions with pocketable storage devices. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 603–612, New York, NY, USA, 2005. ACM Press.

[18] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the Thirteeenth USENIX Security Symposium*, San Diego, CA, August 2004.