Chromium Renderserver: Scalable and Open Remote Rendering Infrastructure

Brian Paul, *Member, IEEE*, Sean Ahern, *Member, IEEE*, E. Wes Bethel, *Member, IEEE*, Eric Brugger, Rich Cook, Jamison Daniel, Ken Lewis, Jens Owen, and Dale Southard

***Abstract*—**
Chromium Renderserver (CRRS) is software infrastructure that provides the ability for one or more users to run and view image output from unmodified, interactive OpenGL and X11 applications on a remote, parallel computational platform equipped with graphics hardware accelerators via industry-standard Layer 7 network protocols and client viewers. The new contributions of this work include a solution to the problem of synchronizing X11 and OpenGL command streams, remote delivery of parallel hardware-accelerated rendering, and a performance analysis of several different optimizations that are generally applicable to a variety of rendering architectures. CRRS is fully operational, Open Source software.

***Index Terms*— remote visualization, remote rendering, parallel rendering, virtual network computer, collaborative visualization, distance visualization**

I. Introduction

The Chromium Renderserver (CRRS) is software infrastructure that provides access to the virtual desktop on a remote computing system. In this regard, it is similar in form and function to previous works, but provides important new capabilities driven by contemporary needs and architectural opportunities. First, CRRS provides seamless presentation of rendering results from both Xlib and hardware-accelerated OpenGL command streams: it is a vehicle for delivering imagery produced by remote, hardware-accelerated rendering platforms. Most previous remote desktop systems are capable of delivering imagery produced only by the Xlib command stream; they typically have no knowledge of or access to the imagery produced by hardware-accelerated OpenGL rendering. Second, it supports operation on parallel backend systems composed of either shared- or distributed-memory computers – it is capable of harnessing the capabilities of a parallel rendering platform for the purpose of generating imagery and sending it to a remote viewer.

B. Paul, K. Lewis and J. Owen are with Tungsten Graphics, Inc., Steamboat Springs, CO. brian.paul@tungstengraphics.com.

E. W. Bethel is with Lawrence Berkeley National Laboratory.

S. Ahern and J. Daniel are with Oak Ridge National Laboratory.

E. Brugger, R. Cook, and D. Southard are with Lawrence Livermore National Laboratory.

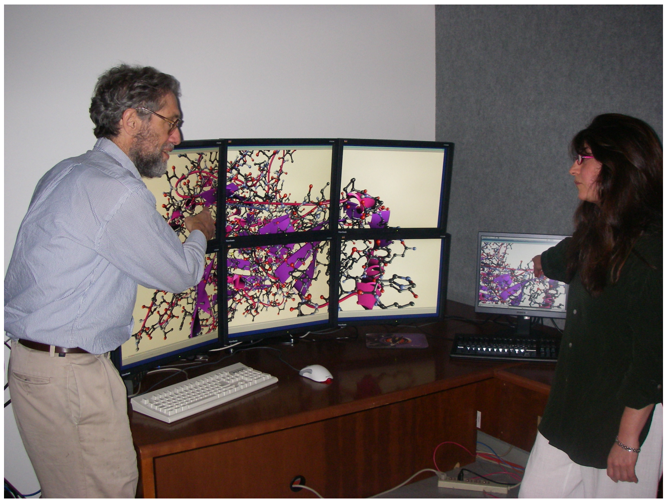

Fig. 1. An unmodified molecular docking application run in parallel on a distributed memory system using CRRS. Here, the cluster is configured for a 3x2 tiled display setup. The monitor for the "remote user machine" in this image is the one on the right. While this example has "remote user" and "central facility" connected via LAN, the typical use model is where the remote user is connected to the central facility via a low-bandwidth, high-latency link. Here, we see the complete 4800x2400 full-resolution image from the application running on the parallel, tiled system appearing in a VNC viewer window on the right. Through the VNC Viewer, the user interacts with the application to change the 3D view and orientation, as well as various menus on the application to select rendering modes. As is typically the case, this application uses Xlib-based calls for GUI elements (menus, etc.) and OpenGL for high performance model rendering.

CRRS design and development has been motivated by the explosive growth in data produced by simulations, collected by experiments and stored at centrally located computing and data warehousing facilities. Correspondingly, as network traffic continues to increase by an order of magnitude every 46 months [3], it is increasingly important to locate data-intensive applications – visualization, rendering and analysis – "close to" the data [2] where such applications have access to more capable resources. This trend is increasingly important as large-scale computational and experimental science teams combine forces to solve the world's most challenging science problems [15].

The rest of this paper is organized as follows. In Section II we describe the background, previous work and motivation for CRRS. Next, in Section III we present the CRRS architecture. We characterize CRRS end-to-end performance in Section IV and show how various optimizations impact overall end-to-end performance. We conclude in Section V with discussion and suggestions for future work.

II. Background and Previous Work

The idea of using remote desktops and applications is not new. Activity over the past decade includes proprietary (SGI's OpenGL Vizserver [9], HP's Remote Graphics Software [5], and Mercury's ThinAnywhere [14]) and Open Source (VNC [17], VirtualGL [4], [18]) solutions. Orthogonal projects, like NX [11], aim to improve the efficiency of Layer 7 communications [21] through a combination of compression, caching, and minimizing round-trips across high-latency links. NX is better thought of as a "protocol accelerator" rather than as "remote desktop infrastructure."

VNC [17] is a client-server system for remote desktop access that uses a simple, platform-independent protocol. We use the term "VNC" to refer to any of the multiple Open Source implementations. The VNC protocol, known as "Remote Framebuffer" (RFB), resides at OSI Layer 7. Via RFB messages, the VNC Viewer authenticates to a VNC Server, sends event messages (mouse and keyboard events) and requests screen updates from the server. The VNC server responds to RFB Update Request messages with RFB Update messages, which contain blocks of pixels that have changed since the time of the previous an RFB Update Request from the client.

There exist many different VNC "servers" that vary in implementation. XF4VNC (xf4vnc.sf.net) provides an X server module that "snoops" Xlib traffic to determine which regions of the screen are modified. Others, like GNOME's vino-server and KDE's krfb server, operate outside of the X server: they periodically grab the desktop image and scan it for changes. The XF4VNC method is more efficient at detecting screen changes, but does not detect changes that result from direct rendering with OpenGL as such changes occur outside the X server. This shortcoming is solved with CRRS.

OpenGL Vizserver [9] is a proprietary product from SGI that consists of a custom display client and server-side software libraries and executables. It allows a remote user to run hardware-accelerated graphics applications on a server then display the resulting imagery on a remote client. Once a user connects from the client to the server, he may run OpenGL and Xlib applications the server system with the resulting Xlib- and OpenGL-based imagery appearing on the client's display. No modification to applications is needed for use in the Vizserver environment. Vizserver "snoops" the OpenGL protocol, and internally triggers a framebuffer capture when it detects the presence of `glFlush`, `glFinish`, or `glXSwapBuffers` [13].

VirtualGL [4], [18] is an Open Source package that provides the ability to deliver hardware-accelerated OpenGL rendering to remote clients. In "direct mode," images pixels resulting from OpenGL rendering are compressed with an optimized JPEG codec and transmitted over a socket to the client; Xlib commands are sent over a separate socket to the client. In this configuration, the client must run an X server as well as a special VirtualGL client application. In "raw mode," OpenGL pixels are written into an uncompressed X-managed bitmap. X itself, or an X proxy (e.g., VNC) manages encoding the framebuffer pixels and transmitting them to the client.

VirtualGL achieves excellent remote image display performance through two vehicles. The first is its use of TurboVNC as a X proxy for mediating the interface between remote client and application image generation. TurboVNC implements some of the same CRRS features we describe later, notably multibuffering, to effectively overlap rendering, image acquisition, encoding and transmission. The second is a highly optimized encoding path: TurboJPEG is a vector-optimized version of the JPEG encoder built with Intel's Integrated Performance Primitives, a set of highly-optimized multimedia libraries for x86 processors. One significant difference between CRRS and VirtualGL is the fact that CRRS is engineered specifically for use in a parallel, hardware-accelerated rendering configuration. Another is that CRRS provides access to a remote desktop; VirtualGL lets a user run a single application. CRRS explicitly supports client applications that run in parallel.

HP's Remote Graphics Software [5] supports remote desktop access, including access to hardware-accelerated graphics applications. This system consists of two software components – a client and server. As with VNC, the client sends keyboard and mouse events over the wire to the server. The server forwards events to desktop applications, then harvests and compresses image updates and sends them along to the client where they are displayed. This system uses a proprietary image codec to achieve high compression rates. It also uses a combination of graphics API call interception (for OpenGL) and framebuffer state monitoring to determine which portions of the display have changed and need to be sent to the client. It provides session-level authentication using Microsoft password authentication protocol NTLM and

Kerberos under Windows and PAM under Unix. RGS also supports multiple viewers simultaneously connected to a single server for a collaborative session mode. While RGS supports "multi-screen" mode, it does not appear to support fully configurable distributed- or shared-memory parallelism like CRRS.

Mercury International Technology's ThinAnywhere [14] products support remote desktop access including OpenGL. In its most flexible configuration, this system uses the proprietary interactive Internet Protocol (iIP) for image delivery from a server to ThinAnywere client(s) running under Linux or Windows. The server supports both RDP/iIP and X11/GLX, so it can provide access to either Windows or Unix application servers. The iIP protocol supports AES-128 encryption for transport armoring, but the mechanism of key generation isn't specified in the public documentation. ThinAnywere also supports RDP and Citrix ICA with client and server plugins, but does not support many-to-one or many-to-many clustered rendering.

IBM's Deep Computing Visualization (DCV) system [8] offers a combination of scalable rendering infrastructure and remote delivery of imagery in a client-server configuration. On the back end, DCV intercepts the OpenGL command stream and routes subsets of the command stream to nodes of a distributed memory system for rendering. Sub-images from each are then collected and combined into a final image, which is then compressed and sent out to a remote client for display. Like VirtualGL, the DCV system sends Xlib protocol directly to the client for processing and the usual host-based ACLs govern security policy. OpenGL pixels are compressed and sent over a separate communication channel to a custom OpenGL client. The communication protocol for transporting OpenGL imagery is proprietary.

While DCV appears to support a mode of operation whereby remote clients connect and interact through a VNC server, it does not appear to support parallel rendering operation in this mode of operation. CRRS, in contrast, implements a general and elegant solution to the problem of providing pixels from both Xlib and OpenGL command streams – from potentially parallel applications – to multiple remote clients.

Chromium is a "drop-in" replacement for OpenGL that provides the ability for any OpenGL application to run on a parallel system equipped with graphics hardware, including distribute memory clusters [7]. It intercepts OpenGL commands issued by the application, and routes them to the appropriate node for rendering. CRRS uses Chromium to implement the distributed memory parallel rendering capability. During the course of CRRS design and development, we added a new

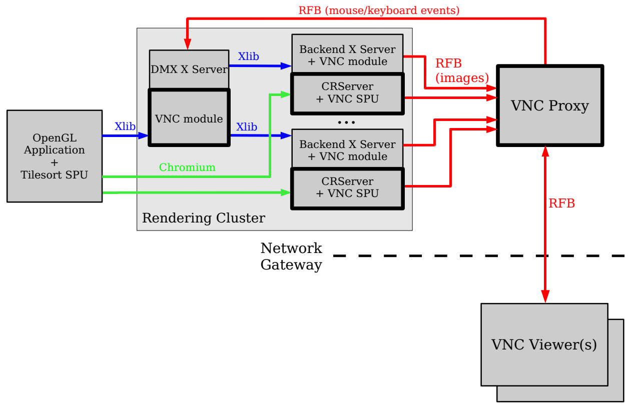

Fig. 2. CRRS system components for a two-tile DMX display wall configuration. Lines indicate primary direction of data flow. System components outlined with a thick line are new elements from this work to implement CRRS; other components outlined with a thin line existed in one form or another prior to the CRRS work.

Chromium Stream Processing Unit (SPU) (see Section III-A.3) – the VNC SPU – that implements OpenGL framebuffer grabs, image encoding and transmission to a remote client via a specialized VNC server known as the VNC Proxy (Section III-A.5).

III. Architecture

CRRS consists of six general system components (below), many of which are built around VNC's RFB protocol. We leverage the RFB protocol because it is well understood and there exist viewers for nearly all current platforms. One of our design goals for CRRS is to allow an unmodified VNC viewer application to be used as the display client in a CRRS application. The CRRS general components are:

- The application, which generates OpenGL and Xlib drawing calls.
- The VNC Viewer, which displays the rendering results from the application.
- The Chromium VNC Stream Processing Unit, which obtains and encodes the image pixels produced by the OpenGL command stream from the application and sends them to a remote viewer.
- Distributed Multihead X (DMX), which is an X-server that provides the ability for an Xlib application to run on a distributed memory parallel cluster.
- The VNC Proxy, which is a specialized VNC Server that takes encoded image input from VNC Servers and VNC SPUs running on each of the parallel rendering nodes and transmits the encoded image data to the remote client(s). The VNC Proxy solves the problem of synchronizing the rendering results of the asynchronous Xlib and OpenGL command streams.
- The VNC Server X Extension is present on the X server at each parallel rendering node. This com-

ponent harvests, encodes and transmits the portion of the framebuffer modified by the Xlib command stream.

A. CRRS System Components

CRRS is a system of components that interact through the Chromium, Xlib and RFB protocols. Figure 2 illustrates the CRRS components and connections for a two-tile DMX display wall configuration. Descriptions of the main components follow.

1) Application: The application is a graphics or visualization program that uses OpenGL and/or Xlib for rendering. Applications need no modifications to run on CRRS, but they must link with the Chromium *faker* library rather than the normal OpenGL library.

2) VNC Viewer: VNC Viewers (or just Viewer for short in this paper) are available for virtually all computer systems. There are no special Viewer requirements – we tested many common Viewers with CRRS. The result is maximum flexibility for users and developers since there is no "special" CRRS client-side viewer other than the standard, ubiquitous VNC Viewer.

3) VNC Chromium Stream Processing Unit: In CRRS, all OpenGL rendering is done through Chromium [7]. A new Chromium SPU, the VNC SPU, intercepts OpenGL rendering commands and returns the rendering results to VNC clients (the Proxy in this case). In a tiled/DMX rendering system, the Chromium Tilesort SPU sends OpenGL rendering to the back-end servers, each of which hosts a VNC SPU.

The VNC SPU, which is derived from Chromium's Passthrough SPU, is multi-threaded to perform rendering and RFB services simultaneously. The main thread renders the incoming OpenGL command stream and then, upon `SwapBuffers` or `glFinish` or `glFlush`, retrieves the rendered image from OpenGL with `glReadPixels` and stores the image pixels in a holding buffer. A second thread acts as a VNC server. It accepts connections from any number of VNC viewers and responds to RFB Update Request messages by encoding RFB Update responses using image pixel data from the holding buffer.

4) DMX: DMX (Distributed Multi-head X) is a special X server for controlling multi-screen displays [16]. To an Xlib application, the DMX X server appears as an ordinary X server. For CRRS, we added several enhancements to the DMX X server to support the RFB protocol. Specifically, the DMX server needed the ability to accept RFB mouse and keyboard events. The DMX server, however, does not need the ability to send RFB images updates since the Proxy handles that task. DMX is not required to run CRRS in a single-tile configuration.

5) VNC Proxy: The VNC Proxy (or just Proxy for short) is derived from the VNC Reflector project[6]. It operates as an agent between the application desktop (which may be a multi-screen DMX display) and some number of VNC Viewers. To the application desktop, the Proxy appears as a VNC client. To the VNC Viewers, it appears as a VNC server.

The Proxy collects images rendered by OpenGL and Xlib into a virtual framebuffer (VFB), which is then used to satisfy requests from the Viewers. When using a DMX display wall, the Proxy queries DMX for the identity of the back-end X servers and their tile boundaries. The Proxy directly connects to each of the back-end servers in order to collect imagery, bypassing the DMX server itself. Otherwise, all RFB traffic would need to make an intermediate trip through the DMX server.

On the Viewer side, the Proxy processes some RFB commands and passes others along. It processes RFB Authentication messages (see Section III-C for security-related discussion) as part of the initial Viewer-Proxy connection. During operation, it sends mouse and keyboard RFB event messages directly to the front-end X/DMX server for further processing. The Proxy answers RFB Update Request messages from all Viewers by encoding RFB Update messages from its internal framebuffer. In turn, the Proxy sends RFB Update Request messages to all back-end VNC Servers. The source of Xlib image data comes from the VNC Server running on each back-end X server, while the OpenGL image data comes from the Chromium VNC SPU, which we describe in Section III-A.3.

All the "parallel aware" aspects of CRRS are consolidated into the Proxy. Thus, any "standard" VNC Server may be used in the CRRS back-end: no special "parallel VNC Server" code is required in the back-end. As the Proxy acts as a message router – brokering communication between a set of hosts in the back-end and one or more Viewers – it has the potential to become a performance bottleneck. This concern led us to pursue optimization strategies (see Section III-B) as well as conduct a battery of performance tests (see Section IV) to better understand end-to-end system performance characteristics. The VNC Proxy has been set up as an Open Source project at SourceForge [19].

B. Optimizations

In this section, we describe a number of optimizations in CRRS that can have a profound impact on end-to-end performance:

- RFB caching. Improves performance by maintaining a cache of RFB Update messages that contain encoded image data. The VNC Proxy responds to

RFB Update Requests by sending cached responses rather than encoding, possibly multiple times, the contents of its internal VFB. This optimization helps to reduce end-to-end latency.
- Bounding box tracking. Here, only the portions of the scene rendered by OpenGL that have changed are rendered and transmitted to the remote client. This optimization helps to reduce the amount of RFB image payload.
- Double buffering. By maintaining a double-buffered VFB in the VNC SPU, two operations can occur simultaneously – application rendering and image encoding/transmission. This optimization helps to reduce end-to-end latency.
- Frame synchronization. While not strictly an optimization, this feature is needed to synchronize parallel rendering streams as well as to prevent frame dropping (spoiling) when images are rendered more quickly than they can be delivered to the remote client.

1) RFB Caching: Normally, the Proxy decodes incoming RFB Updates to its VFB, which is a full resolution image. When a Viewer sends an RFB Update Request to the Proxy, the Proxy responds by extracting pixels from its VFB, compresses/encodes them into an RFB Update response and sends to the Viewer. It is often the case that the Proxy will receive an RFB Update from an X-server VNC module or VNC SPU, decode then store results in its VFB only to shortly thereafter regenerate the same data in response to an RFB Update Request from a Viewer. The purpose of the RFB Caching optimization is to avoid the re-encoding step, thereby reducing latency and processing load, and thus improve the VNC Proxy's overall throughput.

When RFB Caching is enabled, the Proxy saves the incoming RFB Update messages in their encoded format in a cache. When a Viewer requests an update from the Proxy, the Proxy searches its RFB cache to see if the request can be satisfied by a cached entry. If not, the Proxy generates a new RFB update message for the Viewer using its local VFB as the source of pixel data. Each cache entry consists of screen region bounds and a block of encoded/compressed pixel data for that region.

At present, the Proxy caches RFB Update messages only from the VNC SPU(s), and not those from the XF4VNC VNC server(s). This issue is the result of how *zlib* compression is implemented in the *tight* encoder of VNC Servers – we "fixed" the *zlib* compression problem for the VNC Server code in the VNC SPU, but did not propagate those changes to the XF4VNC VNC Server so as to avoid introducing a version dependency in CRRS.

When the Proxy receives a new RFB Update message from a Server, the Proxy searches its RFB cache to determine if any cached regions intersect the new region. When the Proxy finds a cached region intersecting the new region, the older cache entry is discarded and the incoming message is saved as a new cache entry. When the old and new regions partially overlap (e.g., are not identical), this algorithm produces the correct results since for each incoming RFB Update message, the Proxy first decodes the message by modifying its VFB and then caches the new message.

The Proxy appends new entries into the cache and uses a linear cache search algorithm to locate entries in the cache. The linear search algorithm is acceptable since the cache typically contains only a small number of entries. The number of cache entries is less than or equal to the number of DMX screens multiplied by the number of OpenGL display windows. For this reason, the cache memory consumption has an upper bound that is the size of the Proxy's VFB if we assume that the size of a compressed RFB pixel block is always smaller than the uncompressed image in that pixel block.

It is typically the case that the viewer-requested update area does not exactly match the area covered by cached RFB Update messages. In the case of these "partial intersections," the Proxy computes an intersection ratio, which is the quotient of: (1) the intersection area of the requested and cached regions, and (2) the area of the cached region. The closer this quotient is to 1.0, the greater the overlap of the requested and cached regions. The Proxy compares this ratio to a user-tunable intersection ratio threshold parameter to determine if the particular cached entry is a "hit" or a "miss."

For our typical intended use case – animations of 2D or 3D scientific visualization – the RFB cache can satisfy all the viewer's RFB Update Requests, thereby avoiding the cost of an image compression for each update. The positive impact on performance stems from (1) reduced computational cost of the Proxy performing compression and encoding to satisfy the Viewer's RFB Update Request, and (2) a corresponding decrease in latency when the Proxy answers RFB Update Requests.

2) Bounding Box Tracking: A key feature of an efficient VNC server implementation is the ability to track the regions of the screen that have changed and then perform RFB encoding for only those regions. In the Xlib-only configuration where the VNC server runs as an extension to the X server, the VNC server "snoops" the Xlib protocol to track screen regions that change. To achieve the same type of capability when rendering scenes with OpenGL, we have designed and implemented a similar capability in CRRS called "bounding box tracking." This optimization, when enabled, restricts

RFB processing to only those window regions that have changed as a result of OpenGL drawing commands. This approach avoids having to encode and transmit the entire OpenGL window if only a small portion has changed.

In a Chromium configuration, the OpenGL "command snoop" occurs at the *tilesort SPU* level: application-generated OpenGL commands are processed by the *tilesort SPU* where they are "packaged" and then routed to the appropriate *crserver* for rendering. Part of the "packaging" includes bounding box information for the primitive represented by the set of OpenGL commands. When the *tilesort SPU's* bounding box tracking feature is enabled, primitives are sent only to the *crservers* where they will be rendered, thereby saving network bandwidth and processing. This feature existed in Chromium prior to the CRRS project; however, the *crservers* did not pass the bounding box information on to their hosted SPU(s). The implication is that no other SPUs in the SPU chain on the rendering hosts would be able to use bounding box information. We enhanced the *crserver* so that such bounding box data is passed along to the first SPU in the chain. In the CRRS case, the first SPU in the chain is the VNC SPU. The VNC SPU uses the bounding box information in an effort to send smaller RFB update messages to the client (the Proxy). That is, if only a small part of the OpenGL window is changing, only that small part of the window should need to be encoded and sent to the VNC Proxy, not the whole window.

The VNC SPU's method for using bounding box information is as follows. As the OpenGL commands are rendered, the VNC SPU accumulates bounding boxes in an "accumulated dirty region" data structure. That data structure maintains information indicating which regions of the window have been modified by OpenGL drawing commands. When the VNC SPU needs to send an RFB Update message to the Proxy, it sends data only from the accumulated dirty regions rather than the entire window region.

The `glClear` command requires special handling. A naive implementation would simply treat `glClear` as a rendering command that changes the entire OpenGL window: the entire window is the modified region, not just the region defined by the previous bounding boxes. This would defeat the purpose of tracking the accumulated dirty region.

Our approach is to maintain two separate dirty regions: the current frame's region and the previous frame's region. When `glClear` is executed (at the start of the frame), the current region is assigned to the previous region, and the current region is emptied. During rendering, the current region accumulates the bounding box data. At the end of the frame, we compute the union of the previous and current region. The union specifies which window areas need to be read and placed in the SPU's virtual framebuffer.

If only the current region were read back, the image of the object from the previous frame would linger in the VNC SPU's VFB. By including the previous frame's region in readback, we effectively erase the old image data with the window's current background color (set by `glClearColor`). This algorithm ensures that the VNC SPU's VFB is always up to date with respect to the current window contents.

The VNC SPU also must cope with the situation in which it is sending RFB updates at a different (slower) rate than the application frame rate. For example, in the time it takes the VNC SPU to send one RFB Update message to the Proxy, the application may have rendered many frames. If only the previously described union region were sent to the Proxy, the Proxy's VFB may not be updated correctly. Instead, the VNC SPU needs to send to the Proxy all framebuffer regions that have potentially changed since the previous update was sent to the Proxy. To solve this problem, the VNC SPU builds and maintains an accumulated dirty region. This region contains the accumulation of all bounding boxes between the times when the Proxy sends RFB Updates to clients. Thus, subsequent RFB Updates sent from the VNC SPU to the Proxy contain all window regions that have changed since the previous RFB Update message was sent.

There are several special cases in which the bounding boxes are ignored and the entire window contents are read back and sent. Examples include the first time a window is rendered to, whenever a window is moved or resized, and whenever the clear color is changed.

The effectiveness of bounding box tracking for reducing the size of RFB Update messages depends on how much of the window changes from frame to frame. In turn, the amount the window contents change from frame to frame is highly application dependent. In cases where the entire window contents change between each frame, as would be the case with video games, bounding box tracking will provide no performance gain. In other cases, like scientific visualization applications that often display a 3D model in the center of the window, a sizable fraction of the window contents might not change from frame to frame. In these cases, there is opportunity for performance gains due to bounding box tracking.

3) Double Buffering: The VNC SPU uses an internal image buffer – its VFB – to store the pixels it obtains from each OpenGL window via `glReadPixels`. That buffer also serves as the source for the RFB image compressors (tight, hextile, etc.).

There are two threads in the VNC SPU: the main application thread and the VNC server thread. The main application thread will periodically call `SwapBuffers` or `glFinish` or `glFlush`. The VNC SPU intercepts those functions and updates its VFB with the current window image using `glReadPixels`. Meanwhile, the second thread accepts RFB Update Request messages from the Proxy and replies to them with encoded RFB Update messages. To prevent encoding errors that could result from the encoder reading the VFB while it is being updated with new contents, the VFB must not be modified while encoding is in progress. Since the first thread's `glReadPixels` calls do exactly that, we must synchronize access to the VFB by the two processing threads.

Our initial VNC SPU implementation synchronized two-thread VFB access with a mutex so that only one thread could access the VFB at any time. Performance and profiling analysis showed that each of the two threads spent a significant time amount of time simply waiting for access to the shared framebuffer.

We improved performance – eliminating stalls due to waiting – by using a double-buffered VFB. The main application thread can write to the "back" buffer while the second thread can read from the "front buffer" to perform RFB image encoding. The main thread performs a VFB buffer exchange after `SwapBuffers`, `glFinish`, or `glFlush` whenever the RFB encoder is not reading from the "front buffer." This approach allows the application to run at a faster frame rate than the RFB encoder. If the RFB encoder can't keep up with the application's frame rate, intermediate frames will be dropped (spoiled).

In some situations (such as with frame synchronization, below) we do not want frames to be dropped, so we have implemented a VNC SPU configuration option that will prevent frame dropping. When this option is enabled, there is a handshake signal to regulate VFB buffer swapping. The VNC SPU's main thread raises a signal when a new frame is ready. The VNC SPU's server thread waits for this signal, then encodes the image and sends the RFB Update. During encoding, the VFB is locked so that the main VNC SPU thread cannot update it with new contents. When the server thread is finished encoding, the lock is released and the main VNC SPU thread can update the VFB with new contents before returning control to the application.

4) Frame Synchronization: When using a DMX display, there is a separate VNC SPU for each screen in the tiled display. Further, there is no explicit synchronization between the VNC SPUs and the image streams being sent to the Proxy. The result is that at any given point in time, the Proxy's framebuffer may contain data from tiles produced at slightly different points in time. For an animated scene, the visual result is that a Viewer may be displaying results from different points in time. When the application stops animating, all the SPUs "catch-up" and the last frame's image data appear in the Viewer as expected.

We added a frame synchronization option to the Proxy to address this problem. The key objective is to prevent sending RFB Update messages to the Viewer(s) when the Proxy's framebuffer is mid-way through an aggregate update (i.e., receiving updates from one or more VNC SPUs.) We implement such synchronization using a barrier mechanism in the Proxy that manages the incoming and outgoing RFB streams, which contains state for each incoming and outgoing VNC socket connection. Each incoming (i.e., VNC SPU) socket can be in one of three barrier states: pre-update, mid-update, and post-update. The initial state is pre-update. Each outgoing socket (i.e., VNC Viewer) connection can be in one of two states: blocked and non-blocked. The initial state is non-blocked.

When the Viewer receives the first part of an incoming RFB Update message from the Proxy, the Viewer's socket state is set to mid-update. All of the Proxy's outgoing socket connections also are blocked so no intermediate RFB Updates will be sent to the Viewer. When the Proxy receives the end of the message, it sets the socket state to post-update. The socket also is blocked so no further input will be read and no RFB Update requests will be sent to the VNC SPU. After all the Proxy sockets connected to SPUs have transitioned to the post-update state, the barrier is released and all sockets are unblocked (and the VNC SPU sockets are set to the pre-update state). At that time, the Proxy's outgoing socket(s) will be serviced and an RFB Update will be sent to the viewer(s). There will be no incoming RFB Updates pending on the Proxy's SPU sockets since no requests will have been sent.

Frame synchronization requires that the VNC SPUs do not drop frames as described above. Otherwise, if frames were dropped, one might see different frames of animation in different image tiles. Another requirement is that the VNC SPU always sends an RFB Update at the end of a frame, even when there is no new pixel data to be sent. When there is no new pixel data to send, the VNC SPU sends a null/empty RFB Update message, which has no content, but serves to satisfy the synchronization needs of the barrier.

While "frame synchronization" ensures that the contents of the Proxy's VFB from each VNC SPU are consistent, it does not prevent "image tearing" in the

Viewer. Image tearing occurs when rendering is not synchronized to the refresh rate of a monitor. In the case of the VNC Viewer, image tearing occurs when the decoding and displaying of the image data in an RFB Update message takes longer than the monitor's refresh rate. The traditional solution to this problem in computer graphics involves double-buffering. In this case, double-buffering in the VNC Viewer would alleviate image tearing during Viewer image display. At least one VNC distribution, TurboVNC, is known to offer a double-buffered Viewer. We have not explored using this double-buffered Viewer with CRRS.

Under some circumstances, we have observed CRRS frame synchronization results in improved performance due to more efficient VNC SPU operation. When frame synchronization is enabled, the VNC SPU is prevented from dropping frames as the two threads in the VNC SPU are synchronized. Without this synchronization, the VNC SPU's server thread may sometimes be starved or preempted by the main thread, resulting in an overall decrease in RFB throughput.

C. Security

Because CRRS is a system of components acting though multiple protocols, implementation of security can be problematic. Each of the underlying protocols have their own pre-established authorization mechanisms, and one of the CRRS design requirements is that it is based on open standards. Thus, our security analysis focuses on categorizing the risk of the connections and ensuring that sites have avenues to "tighten the security screws" as required by their policies.

A pictorial view of the connections and their types in CRRS is shown in Figure 3. In that Figure, the connections between one or more Viewers and the Proxy occur in Zone 1. These Zone 1 connections typically occur in the open internet and consist exclusively of RFB traffic. Inside the CRRS back end, connections that occur between the Proxy, the VNC servers in each of the tiles of a DMX display, and DMX itself occur in Zone 2. Traffic in Zone 2 consists of a mixture of RFB and Xlib protocols. Further inside the CRRS back end, the (potentially parallel) rendering application emits OpenGL and Xlib rendering commands. OpenGL commands are routed to one or more Chromium *crservers*, while Xlib traffic is routed to the DMX server, which in turn, snoops the Xlib protocol and routes Xlib commands to the appropriate tile for rendering. This traffic occurs in Zone 3.

For many sites, security in Zone 1 will be the most critical issue. The Zone 1 RFB connection might occur from remote locations and are thus outside the control

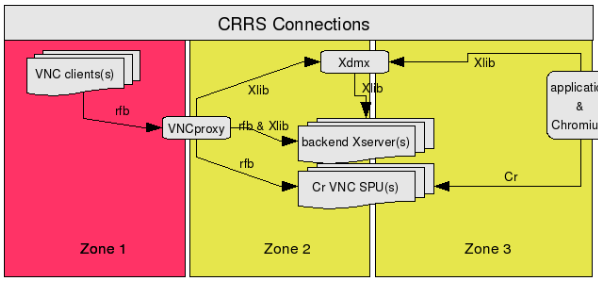

Fig. 3. CRRS Security Zones. Zone 1 traffic consists exclusively of RFB protocol messages, and typically occurs over the open internet. A commonly accepted practice to "secure" this connection is to force all such traffic through SSH tunnels, where site-wide policy can enforce authentication and the connection is armored. Inside the center, Zone 2 traffic consists of a mixture of RFB and Xlib messages between the Proxy and the VNC servers. Zone 3 traffic consists of application-generated OpenGL and Xlib command streams, and is typically where services like MPI are provided for parallel application operation.

of the site. Additionally, the RFB protocol itself has only a basic mechanism for authentication and provides no encryption or other protection during transport. To mitigate security concerns, the Proxy supports binding the listening socket to a specific IP address. That feature can be used in conjunction with SSH or SSL tunneling software to provide both stronger authentication and transport layer encryption. Additionally, the CRRS team has provided client patches that implement SSL directly in the java-based viewer to the upstream maintainers of that code.

The standards for Zone 2 and 3 security likely will be driven by site-specific deployment models and requirements. In general, both Zone 2 and Zone 3 connections occur within the interconnect fabric of a single visualization cluster rather than over the open network. Since such fabrics are private to a cluster, transport security concerns are generally addressed as part of the cluster design, leaving only user authorization as a concern. For the Zone 2 connections, this means using *Xauth* and *vncpasswd* to authorize the connections between the Proxy and other back-end components in order to deny access from other users running on the same node as the Proxy. For sites that use a clustering model that restricts logins to a few front-end nodes, moving the Proxy to an "internal" node can provide further mitigation of security issues by removing the requirement for user-based security in favor of host-based security. For deployments that do not use a cluster model, Zone 2 connections can be addressed via SSL or SSL tunneling in the same way as Zone 1.

For Zone 3 there are typically no additional requirements beyond those for Zone 2. The Zone 3 connection are analogous to MPI messages, for which most sites require at most host-based authorization. Again, the option for tunneling remains, though probably at a substantial cost in performance.

IV. Performance Characterization

A. *Performance Test Applications*

For the purpose of performance evaluation, we employed one network bandwidth measurement tool and two different rendering applications.

Iperf is a tool for measuring effective TCP or UDP bandwidth/throughput over IP networks [1].

City is a lightweight GLUT/OpenGL application that is included the Chromium distribution. It draws a scene consisting of a flat ground surface and a number of textured buildings. The scene rotates a small amount about an axis perpendicular to the ground between each frame. This application is useful in our tests because its rendering cost is relatively low and about 50%-75% of the screen pixels change from frame to frame.

svPerfGL is a lightweight OpenGL benchmark application – similar to SPECViewperf in intent – that loads triangle vertex/normal/color data stored in a netCDF file (such data files are accessible at the *svPerfGL* project site [20]), rotates about about a vertical axis a small amount each frame, runs for a finite duration and reports average frame rate at the end of the run. Like *city*, its rendering will result in about 50%-75% of the screen pixels changing on each frame. Its focus is on benchmarking OpenGL systems under a load characteristic of scientific visualization.

B. *Methodology and Resources for End-to-End Tests*

1) Computing Platforms: For our performance tests, we ran the CRRS back-end components (Zones 2 and 3) on a cluster at Lawrence Berkeley National Laboratory (LBNL). Each of the eight nodes of the LBNL cluster runs SuSE Linux 10.0 and consists of dual-socket, dual-core 2.6Ghz AMD Opteron processors, 16GB of RAM, NVIDIA GeForce 8800 graphics accelerators, Infiniband 4x interconnect for intranode traffic and gigabit ethernet for internet traffic.

At Lawrence Livermore National Laboratory (LLNL), the display platform is a 16-node cluster consisting of dual-socket, single-core 2.0Ghz AMD Opteron processors running the RHEL4+CHAOS Linux distribution. Each node has 4GB of RAM, an NVIDIA GeForce 6600GT graphics accelerator, Infiniband 4x interconnect for intranode traffic and gigabit ethernet for internet traffic.

At Oak Ridge National Laboratory (ORNL), the display platform is a 64-node cluster. Each node consists of dual 2.2Ghz AMD Opteron processors with 2GB of RAM and running the SuSE 9.2 Linux distribution. The fourteen nodes driving the display wall are equipped with NVIDIA QuadroFX 3000G graphics accelerators.

Iperf (Mb/s)	LAN	WAN-R	WAN-H1	WAN-H2
Clear	877.00	2.07	168.00	10.80
Tunnel	380.00	1.97	115.00	49.00
Tunnel -c (JPG)	87.40	2.07	82.00	36.30
Tunnel -c (XWD)	291.00	17.20	150.0	51.00
Ping RTT (ms)	0.01	16.25	2.71	62.25

TABLE I. Results of *iperf* bandwidth measurement of four different networks reported in megabits/second. The networks consist of a Gigabit ethernet LAN, residential broadband (WAN-R), and two different high-speed networks – one between LBNL/LLNL (WAN-H1), the other between LBNL/ORNL (WAN-H2). We had *iperf* transfer known image data between endpoints ("Tunnel -c – JPG" and "Tunnel -c – XWD") to measure the impact of SSH compression on throughput.

2) Networks and Throughput Measurement: To measure network performance, we ran *iperf* in two different modes: (1) a direct connection between client and server, and (2) tunneling the *iperf* connection through SSH. The tunneled connection represents a typical CRRS use scenario, where the connection between the Viewer and Proxy occurs over the open internet. Table I shows the measured throughput over each of these networks.

Since most SSH implementations offer a lossless compression option, we included tests to measure the effective compression when moving pre-compressed and uncompressed image data – the typical CRRS payload. For these tests, we prepared archive files of images in JPEG and XWD format for the pre-compressed and uncompressed image tests, respectively. Each of the JPEG and XWD image sequences show exactly the same thing: an isosurface rotating a small amount each frame about the vertical axis; approximately 50%-75% of the image pixels change on each frame. The results in Table I show that tunneling has a more pronounced adverse impact on throughput on higher speed networks, and that SSH compression produces better throughput when transmitting uncompressed through the tunnel.

The four different networks shown in Table I are as follows: (1) LAN: switched copper gigabit ethernet; (2) WAN-R: residential broadband from LBNL to the residence of one of the authors; (3) WAN-H1: high-speed IP network between LBNL and LLNL over ESnet; (4) WAN-H2: high-speed IP network between LBNL and ORNL over ESnet.

During performance testing, which spanned many months, we discovered a large amount of variation in performance over these networks. This variance was at times quite severe – as much as 50%. Worse, one of the mid-stream service providers for the residential broadband network changed its routing in a way that adversely affected network performance. To work around this situation, we used a bifurcated strategy. For those tests that require multiple collaborative users, we (1) would run *iperf* prior to any testing to verify that network conditions were close to baseline, and (2) then run the given test five times and average the results.

For most other tests, we assembled a network emulator at LBNL. The emulator consists of: (1) the *Nist NET* network emulator software [10], which supports user tunable bandwidth and latency parameters; (2) two workstations – one acts as a gateway and runs the network emulator software, the other is a "remote client" that routes all IP traffic through the gateway machine.

C. Individual Performance Optimizations

Earlier in Section III-B, we described a number of optimizations aimed at improving overall end-to-end CRRS performance. This Section presents results of tests aimed at measuring the impact of each of these individual optimizations when applied separately and together. For these tests, we used the *city* and *svPerfGL* applications. Each was run five times for a duration of sixty seconds at varying resolution/tiling configurations and over different networks configurations using the network emulator. The choices for configurations and applications are intended to provide reasonably broad coverage across optimization opportunities.

In one test configuration, we run both applications under CRRS at SXGA resolution (1280x1024) on a single node and transmit results to a remote Viewer. In another, we run both applications under CRRS at a resolution of 4800x2400 pixels on a 3x2 DMX mural and transmit results to a remote Viewer. In each of these configurations, we vary the optimization and the network.

Both applications report their frames-per-second rendering rate, which is the metric for measuring performance in these tests. In all test configurations, we enable the *frame synchronization* option (Section III-B.4). With *frame synchronization* enabled, the application-reported frames-per-second accurately reflects the rate that frames are delivered to the remote Viewer over the course of the run. We average the frames-per-second from five runs to account for variation in network (emulator) and system performance.

1) No Optimizations: We begin the optimization testing by measuring performance without any optimizations whatsoever. These results, shown in Figure 4, establish the performance baseline for measuring the impact of individual optimizations.

Here we see a noticeable performance difference depending upon how image payload data is encoded. The *hextile* encoder is lossless and inexpensive but results in a lower rate of compression than the *tight* encoder. Therefore, when the network can transmit payload faster than the processor can compress it, the *hextile* encoder is a better choice. Conversely, when network throughput is the bottleneck, better compression results in better frame rates.

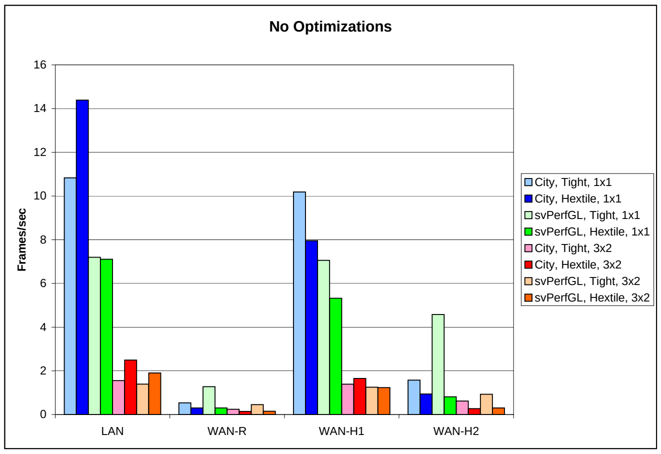

Fig. 4. Frame rate of test applications over different networks, varying resolution, and no CRRS optimizations.

The *tight* encoder is "adaptive" in the sense that it examines blocks of pixels and then encodes the pixel blocks in one of two ways. If the block of pixels is "homogeneous," it uses a form of run-length encoding. If not, then the block of pixels undergoes JPEG compression. The JPEG encoder has tunable attributes to balance image quality versus amount of compression. In these tests, we used an image quality value of 95%, which is often accepted as being "visually lossless," yet producing reasonably good compression ratios.

2) RFB Caching: We instrumented the Proxy to report the effective compression ratio by computing the number of bytes sent divided by the number of pixels at three bytes per pixel in the Proxy's virtual framebuffer. For both test applications, we ran the instrumented Proxy code using both *tight* and *hextile* encoders. The approximate compression ratios for each of these combinations are shown in Table II. While the absolute amount of compression is highly dependent upon the image contents, we observe that the *hextile* compression ratio varies less than that for the *tight* encoder. We also observe that

Application	*City*	*svPerfGL*
Tight, JPEG quality=95	12.0:1	26.1:1
Hextile	3.1:1	3.4:1

TABLE II. Approximate compression ratios for each of the *tight* and *hextile* image encoders when applied to the image output to the two test applications *city* and *svPerfGL*. The *tight* encoder, which uses lossy compression based upon JPEG, provides high levels of compression, and therefore is better suited for use on high latency, low bandwidth network links.

the lossy *tight* encoder is able to achieve much higher compression ratios for these particular applications.

The test results, shown in Figure 5, indicate that RFB caching has a greater positive impact on performance when using the more expensive *tight* encoder by avoiding multiple executions of the encoder on a block of image data. We see more consistent performance improvement in the tiled than serial configurations across all test parameters. In the tiled configuration, CRRS is processing about six times more image data than in the serial configuration; we expect the RFB caching performance impact to be more consistently present in this configuration. We also see less performance variability in the lower latency networks (LAN, WAN-H1) than in the higher latency networks. This result is expected since the overall frame rate is a function of latency – the effects of network and Proxy response are additive.

In some cases, RFB caching seems to have a small, adverse impact on overall application performance. This result occurs when using the fast *hextile* encoder on a relatively small amount of image data. In these configurations, it is likely that the cache lookup and replacement operations, which involving allocating and freeing memory buffers, take longer than simply encoding and transmitting the image data.

3) Bounding Box Tracking: This optimization aims to reduce the size of RFB Update payload by having the VNC SPU encode RFB Update messages that reflect only the portion(s) of the OpenGL display window that actually change from frame to frame. Without this optimization, each VNC SPU will encode the entire contents of the OpenGL display window in response to an RFB Update Request message.

The test results, shown in Figure 6, indicate this optimization has a uniformly positive, albeit modest, impact regardless of encoder type and network. This result is expected since this optimization will reduce the overall amount of RFB traffic for the two test applications in all configurations. The absolute amount of improvement gain is a function of scene contents – scenes where little changes from frame to frame will realize a greater benefit than those that make more extensive inter-frame changes. We also see this optimization has a positive impact regardless of whether used in serial or parallel configurations. This optimization, which propagates bounding box information to the parallel rendering nodes, is capable of producing a positive impact throughout the parallel rendering back end.

4) Double Buffering: This optimization allows the multithreaded VNC SPU to maintain and operate with multiple internal VFB's. One thread responds to RFB Update Request messages and reads/encodes from one buffer, while another thread harvests OpenGL pixels from the other. This optimization is intended to eliminate contention between these two threads and therefore increase overall throughput.

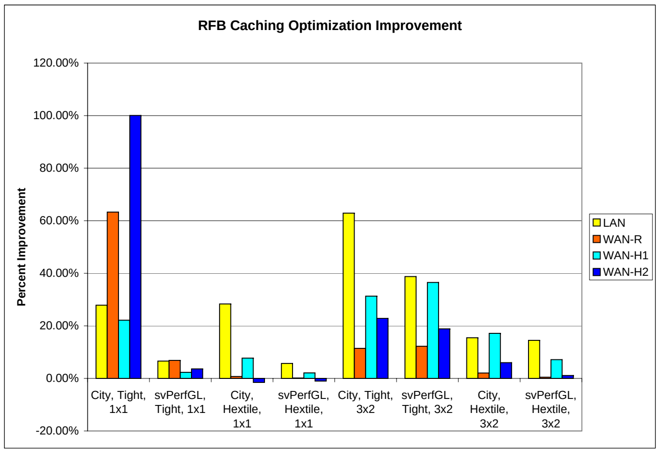

Fig. 5. RFB Caching performance improvement test results.

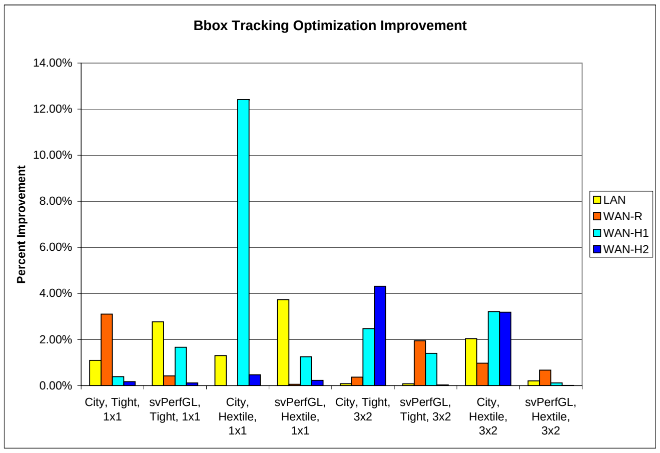

Fig. 6. Bounding box tracking performance improvement test results.

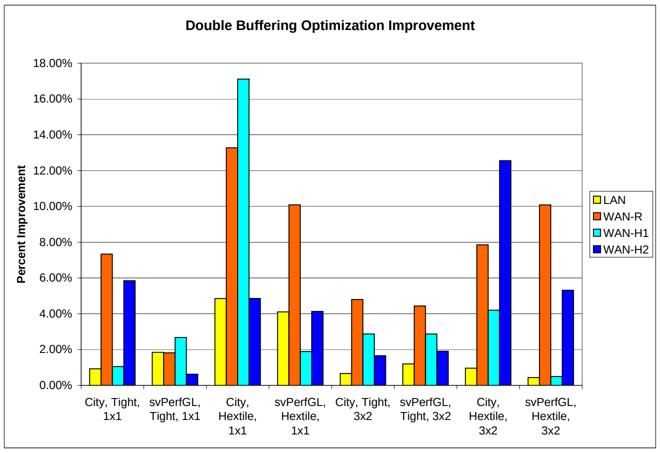

Fig. 7. Double buffering optimization performance improvement test results.

Double buffering has greater positive impact when using the *hextile* rather than the *tight* encoder. Like RFB caching, double buffering serves to reduce end-to-end latency. The greater performance improvement seen with the *hextile* encoder is due to the fact that it can encode image data more quickly than the *tight* encoder.

Test results, shown in Figure 7, indicate performance improvement from this optimization is uniformly positive across both serial and tiled configurations. This result is expected since the performance gain from double buffering, which effectively overlaps application rendering and RFB image encoding, is present at each of the parallel rendering nodes. In contrast, other optimizations, like RFB caching, occur only at the Proxy, which is effectively a serial "junction point" in CRRS between the back-end rendering engines and the Viewer.

We see greater positive impact on higher latency networks (WAN-R, WAN-H2) than on lower latency networks (LAN, WAN-H1). This result is expected due to the additive nature of end-to-end latency.

5) All Optimizations: Here, we run the test battery with all of the above optimizations enabled. Figure 8 presents the results in the same format as for the individual unit tests, while Figure 9 shows the results organized by network and optimization rather than application.

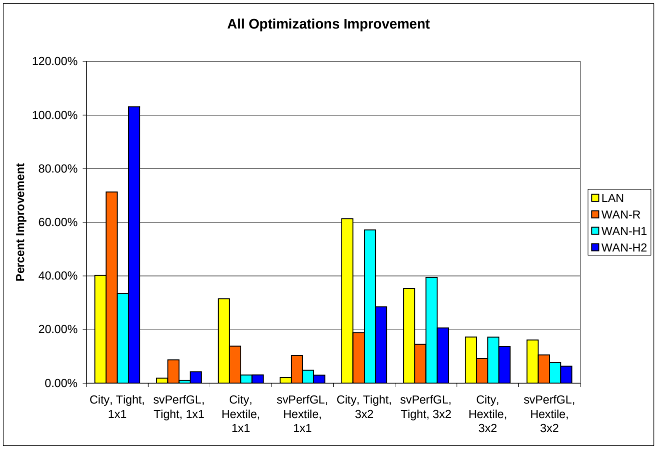

Fig. 8. Relative performance improvement from enabling all optimizations.

Double buffering's amount of positive impact appears to be inversely proportional to the latency of the underlying network. In other words, double buffering provides the greatest amount of performance gain on high-latency networks. RFB caching has a noticeable, positive impact across all test applications, encoders, networks and display resolutions. Bounding box tracking has a modest, but positive, impact on performance in all configurations. Since this performance optimization's impact is a function of scene content, other applications may realize a greater or lesser degree of performance

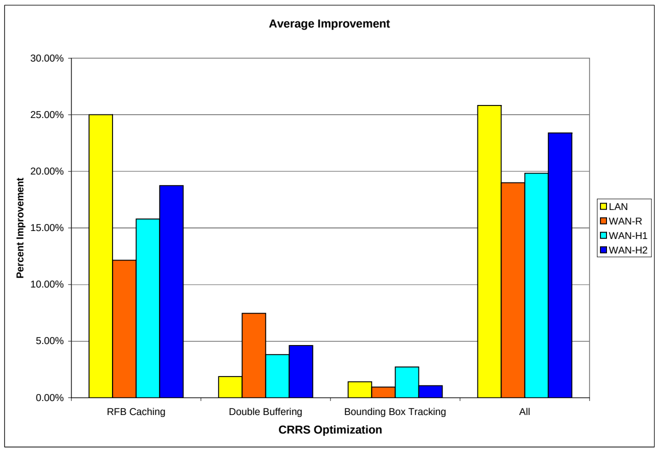

Fig. 9. Relative performance improvement from enabling all optimizations.

gains. Overall, the performance improvement of all optimizations appears to be nearly additive. This result is expected since they all serve to either reduce latency or the amount of RFB Update payload traffic. We conclude from these tests that CRRS performs best when RFB caching, double buffering and bounding box tracking are all enabled for all types of encoders and networks.

V. Conclusion and Future Work

CRRS is software infrastructure that provides the ability for a remotely located user to take advantage of distributed memory parallel rendering resources at a centrally located facility to perform interactive visualization. This type of use model is increasingly common and reflects the need to perform visual data analysis "close to" the data. The new contribution of this work is the design and implementation of software infrastructure that performs parallel, hardware-accelerated rendering for unmodified OpenGL applications with delivery of results via industry-standard mechanisms to one or more collaborative clients. Our study includes detailed performance evaluation of several different system optimizations. These optimizations, while couched in terms of CRRS, will be of benefit to virtually all rendering applications where end-to-end performance is a concern. All CRRS system components are Open Source, freely available, and have undergone extensive testing.

During CRRS development and testing, we did conduct some preliminary scalability studies. In particular, we examined performance when adding more simultaneous collaborative viewers. Due to space limitations, those results are not presented here. One conclusion from that testing is that the benefits from CRRS optimizations, particularly RFB caching, extend to the multi-participant use model. The primary bottleneck in that case is network throughput at the VNC Proxy host.

Interesting future work would include additional studies characterizing performance of strong scaling (adding more rendering nodes for a fixed problem size) and weak scaling (increasing the number of rendering nodes and problem size). Such studies would be instrumental in identifying performance bottlenecks and system limits.

During our testing, we encountered several limitations that would merit further investigation and future work. First, when performing image downsizing at the Proxy – so that the application can render at a high resolution and the results displayed at a much smaller resolution – we often encountered situations in which it was difficult to read text on menus. One of the test users (shown in Figure 1) suggested a "fisheye" style of rendering for focus views. This capability is not trivially implementable in the current system.

Another interesting enhancement would be dynamic resolution selection for remote clients. The idea here would be for the Viewer and Proxy to negotiate a scaling factor to meet some performance criteria. Furthermore, extending this idea to be dynamic over the course of the application session would be useful. At present, the scale factor is specified when the Proxy is launched and remains constant for the duration of the run. Related, extending CRRS with alternate encoding methods like MPEG may help to improve absolute performance; similar work has been explored with success for use with mobile devices [12].

There are several interesting systems that perform remote delivery of imagery produced by hardware-accelerated rendering – it would be interesting to see a head-to-head comparison of absolute frame rate for a standard set of benchmarks for all these systems.

While we have touched on issues related to security, we have not directly solved any of them. An ongoing debate within the community is the extent to which security, specifically authorization and authentication, should be performed by the distributed visualization/rendering system. Related, the centrally located facilities are systems that are shared by many users, but graphics cards and X servers are typically employed in a single-user mode. Some sites address this problem by requiring all interactive visualization jobs to be marshaled by a central batch queue. While this approach works, it can leave much to be desired from the perspective of the user.

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy through the SBIR/STTR program under Contract No. DE-FG02-04ER86198.

References

[1] A. Tirumala and F. Qin and J. Dugan and J. Ferguson and K. Gibbs. Iperf Version 2.0.2. `http://dast.nlanr.net/projects/Iperf`, 2005.

[2] Gordon Bell, Jim Gray, and Alex Szalay. Petascale Computational Systems. *IEEE Computer*, 39(1):110–112, 2006.

[3] Joe Burrescia and William Johnston. ESnet Status Update. Internet2 International Meeting, 2005.

[4] D. R. Commander. VirtualGL: 3D Without Boundaries – The VirtualGL Project. `http://virtualgl.sourceforge.net/`, 2007.

[5] Hewlett Packard Company. Remote Graphics Software. `http://h20331.www2.hp.com/Hpsub/cache/286504-0-0-225-121.html`, 2006.

[6] HorizonLive.com, Inc. VNC Reflector. `http://vnc-reflector.sourceforge.net/`, 2007.

[7] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 693–702, New York, NY, USA, 2002. ACM Press.

[8] IBM Systems and Technology Group. IBM Deep Computing Visualization Technical White Paper. `http://www-03.ibm.com/servers/deepcomputing/visualization/downloads/dcv%techwp.pdf`, 2005.

[9] Silicon Graphics Inc. OpenGL Vizserver. `http://www.sgi.com/products/software/vizserver/`, 2007.

[10] Internetworking Technology Group, National Institute for Standards and Technology. Nist NET. `http://www-x.antd.nist.gov/nistnet/`, 2007.

[11] Kurt Pfeifle. NX Components. `http://openfacts.berlios.de/index-en.phtml?title=NX_Components`, 2007.

[12] F. Lamberti and A. Sanna. A Streaming-Based Solution for Remote Visualization of 3D Graphics on Mobile Devices. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):247–260, March-April 2007.

[13] Jenn McGee and Ken Jones. SGI OpenGL Vizserver Administrator's Guide. `http://techpubs.sgi.com/library/manuals/4000/007-4481-008/pdf/007-4481-%008.pdf`, 2004.

[14] Mercury International Technology, Inc. ThinAnywhere. `http://www.thinanywhere.com`, 2007.

[15] U.S. Department of Energy Office of Science. Scientific Discovery Through Advanced Computing. `http://www.scidac.org`, 2007.

[16] Red Hat, Inc. Distributed Multihead X Project. `http://dmx.sourceforge.net/`, 2007.

[17] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[18] Simon Stegmaier, Marcelo Magallon, and Thomas Ertl. A Generic Solution for Hardware-Accelerated Remote Visualization. In *VISSYM '02: Proceedings of the Symposium on Data Visualisation 2002*, pages 87–94, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[19] Tungsten Graphics, Inc. VNC Proxy. `http://vncproxy.sourceforge.net/`, 2007.

[20] Visualization Group, Lawrence Berkeley National Laboratory. svPerfGL. `http://codeforge.lbl.gov/projects/svperfgl`, 2007.

[21] Hubert Zimmerman. OSI Reference Model – The ISO Model of Model of Architecture for Computer Systems Interconnection.

IEEE Transactions on Communications, 28(4):425–432, April 1980.

Brian Paul is a senior engineer and cofounder of Tungsten Graphics, Inc. His interests include interactive rendering, visualization, and graphics software development. He earned his BS and MS degrees in computer science from the University of Wisconsin in 1990 and 1994, respectively. Brian is a member of the IEEE.

Sean Ahern is a computer scientist and the Visualization Task Leader for the National Center for Computational Sciences at Oak Ridge National Laboratory. He is ORNL's PI within SciDAC's Visualization and Analytics Center for Enabling Technology. Prior to Oak Ridge, he was the Visualization Project Leader within the Advanced Scientific Computing (ASC) program at Lawrence Livermore National Laboratory. He has extensive experience with distributed visualization and data processing on computational clusters. He has won two R&D 100 Awards for his work on the VisIt visualization system and the Chromium cluster rendering framework. He holds degrees in Computer Science and Mathematics from Purdue University.

E. Wes Bethel is a Staff Scientist at Lawrence Berkeley National Laboratory and founding Technical Director of R3vis Corporation. His research interests include high performance remote and distributed visualization algorithms and architectures. He earned his MS in Computer Science in 1986 from the University of Tulsa and is a member of ACM and IEEE.

Eric Brugger is a computer scientist with Lawrence Livermore National Laboratory where he is the project leader for VisIt, an open-source, distributed, parallel scientific visualization and analysis tool. He received an R&D 100 award in 2005 as part of the VisIt development team. Eric has over 15 years of experience in scientific visualization. His areas of expertise include parallel rendering, parallel visualization architectures, scientific data file formats, scientific data modeling, and parallel I/O.

Rich Cook has worked for six years as a computer scientist for the Lawrence Livermore National Laboratory Information Management and Graphics Group. He graduated from University of California, Davis with a Master's degree in computer science in 2001.

Jamison Daniel is presently full-time research staff in the Scientific Computing Group at the National Center for Computational Sciences located within the Oak Ridge National Laboratory.

Ken Lewis is director of engineering for Tungsten Graphics, Inc. He has 28 years experience in the workstation and computer graphics industry. He has a MS degree in computer science from the University of Colorado.

Jens Owen is CEO of Tungsten Graphics, Inc. He has been active in the graphics industry for many years: Director of Engineering and Founder of Precision Insight, Engineering Manager at VA Linux Systems, Engineering Manager for a small X Server development company, and Graphics Software Developer for Hewlett Packard's Workstation Division. Jens holds a BSCS from Colorado State University.

Dale Southard is a computer scientist with Lawrence Livermore National Laboratory. He is currently the hardware architect for visualization and post-processing systems, as well as a developer and security specialist with both visualization and systems groups.