Understanding Screen Contents for Building a High Performance, Real Time Screen Sharing System

Surendar Chandra, Jacob T. Biehl, John Boreczky, Scott Carter, Lawrence A. Rowe
FX Palo Alto Laboratory Inc., 3174 Porter Drive, Palo Alto, CA 94304
surendar@acm.org, {biehl,johnb,carter,rowe}@fxpal.com

ABSTRACT

Faithful sharing of screen contents is an important collaboration feature. Prior systems were designed to operate over constrained networks. They performed poorly even without such bottlenecks. To build a high performance screen sharing system, we empirically analyzed screen contents for a variety of scenarios. We showed that screen updates were sporadic with long periods of inactivity. When active, screens were updated at far higher rates than was supported by earlier systems. The mismatch was pronounced for interactive scenarios. Even during active screen updates, the number of updated pixels were frequently small. We showed that crucial information can be lost if individual updates were merged. When the available system resources could not support high capture rates, we showed ways in which updates can be *effectively* collapsed. We showed that Zlib lossless compression performed poorly for screen updates. By analyzing the screen pixels, we developed a practical transformation that significantly improved compression rates. Our system captured 240 updates per second while only using 4.6 Mbps for interactive scenarios. Still, while playing movies in fullscreen mode, our approach could not achieve higher capture rates than prior systems; the CPU remains the bottleneck. A system that incorporates our findings is deployed within the lab.

Categories and Subject Descriptors

J.m [**Computer Applications**]: Miscellaneous

General Terms

Experimentation, Measurement, Performance

Keywords

ScreenCast, DisplayCast, Screen Capture, Screen Sharing

1. INTRODUCTION

Screen sharing is integral to collaboration scenarios such as auditorium projection, remote learning and debugging, multi-user screen sharing and interaction as well as in screencast archival. Consider an usage scenario: Emily presents her simulation plots from cloud servers on a high resolution auditorium projector. Amy watches this presentation on her laptop using the intranet wireless while Bob joins in using in-flight WiFi. Bob then displays another article on the projector that portends the impact of Emily's work. Impressed by these results, Amy sends these screens to other colleagues.

Even though the participants want to share without any restriction on their shared contents, resource availability places some restrictions. Emily might use a dedicated Gbps wired link between the cloud servers that generate her plots and the projector PC. However, Bob laptop's small display and slow in-flight network limits his ability to watch these plots in high fidelity. We desire a system that can faithfully capture the screens while operating in a resource rich environment and yet be adaptable to modest scenarios.

The screen capture mechanism also plays a crucial role. Emily could share her data for plotting on the projector PC and on Bob's laptop. Similarly, Bob sends his PowerPoint report to the projector PC. These data objects are likely compact and network efficient. However, the recipient is required to have the appropriate software setup; Amy needs a PowerPoint player to read the report shared by Bob. Also, privacy concerns might limit sharing the entire data; Bob only wants to send select portions to the projector PC.

Instead, we focus on sharing screen pixmaps. A pixmap represents the screen image as an array of pixel color values. Pixmaps are generated after any application specific rendering. Graphics rendering frameworks also create pixmaps for display. Most operating systems provide a mechanism to capture and display pixmaps. Hence, sharing pixmaps is portable and popular; systems such as Virtual Network Computing (VNC) [21], Microsoft Remote Desktop (RDP) [5], Google Chrome Remote Desktop and Cisco WebEx (`http://www.webex.com/`) also capture pixmaps.

We target resource rich intranet scenarios. Modern laptops use fast, multicore processors. IEEE 802.11n WLAN supports bandwidths of 600 Mbps while the upcoming IEEE 802.11ac promises bandwidths of over 1 Gbps. Earlier systems were designed to operate in resource constrained environments. They were unable to scale to support scenarios that did not experience such bottlenecks. Using wired networks, Wallace et al. [26] only measured a pixmap delivery rate of 4.5 frames per second for VNC.

To understand the requirements for a high performance screen sharing system, we investigated the nature of screen updates; both in terms of the rate at which they are generated as well as the kinds of pixels that comprise them. The update dynamics is application dependent. Hence, we analyzed the screen dynamics while using various applications ranging from playing movies to computer generated animations, presentations, web browsing and interactive editors. We wanted a portable approach. Hence we performed exper-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MM'12, October 29–November 2, 2012, Nara, Japan.
Copyright 2012 ACM 978-1-4503-1089-5/12/10 ...$15.00.

iments under Windows 7 and Mac OS X Snow Leopard. We used two different laptops to assess the impact of hardware capability.

Screen contents were dynamic and behaved differently than fixed frame rate videos. Pixmap generation rates were sporadic with long inactivity durations. Computer generated animations and interaction artifacts created new pixmaps at rates that were only limited by the laptop processing capability. On modern laptops, pixmaps need to be captured at rates far higher than 100 updates per second (ups) to minimize the pixels lost during capture. Future processor improvements will necessitate even higher update rates. We observed significant differences between applications. Sometimes, the same application (e.g., Adobe Flash game) behaved differently between Windows 7 and Mac OS X. Different applications behaved differently while operating on the same object. For the same H.264 compressed movie, Windows Media player doubled the rendering rate as compared to the Quicktime player. Inexplicably, the captured pixmaps also exhibited less inter-update similarity than with the Quicktime player. Windows 7 created more small pixmaps and a faster processor increased the number of small updates.

We showed the inadequacy of Zlib compression for screen updates. We quantified the redundancy in screen pixmaps and devised schemes to achieve higher compression efficiency by transforming the inter-update pixel temporal redundancy into intra-update spatial redundancy. We improved the compression factor of 5.1:1 achieved by Zlib to 143:1; a 28 fold improvement in compression efficiency.

We built DisplayCast [13], a high performance screen sharing system using our observations. Our system captures and compresses over 240 ups for interactive scenarios. We devised ways of effectively responding to scenarios in which the update rates overwhelm the available resources. We empirically show that in Windows 7, a delay of 16 msec to wait to capture the end of an update flurry was reasonable. Our system natively operates in Windows 7 and in Mac OS X and is deployed in a production setting.

Next, §2 describes prior work. §3 describes the experiment setup. §4 analyzes pixel generation rate for various usage scenarios. §5 analyzes the spatial overlap between updates and devices schemes for *effectively* responding to resource constraints. §6 analyzes the pixels and devices a transformation to Zlib that improves screen pixmap compression efficiency. We briefly describe the DisplayCast system that uses our findings in §7 and conclude in §8.

2. RELATED WORK

Prior systems used several ways for screen capture. Some approaches partitioned applications with the local component sending the encoded objects for remote rendering. For example, Apple Airplay [6] sends audio and video for playback to an Apple TV while Microsoft Windows Media player [10] sends media to the RDP player. Similarly, the Access Grid Distributed PowerPoint tool [25] remotely shares PowerPoint presentations. Since these systems use the original objects, they achieve high network efficiency without requiring additional CPU resources. On the other hand, they require tight integration between the endpoints. Remote components must precisely render the redirected objects. They must also enforce all access control restrictions. Thus, Microsoft's own Mac OS X RDP player does not support player redirection. This lack of generality limits this form of screen sharing.

Another approach captures the rendering commands sent by applications to frameworks such as Apple Cocoa, Microsoft DirectX and OpenGL. For example, GLX [3] extensions to the X windows protocol provides remote rendering capability for OpenGL applications. WireGL [17] renders OpenGL commands across a cluster of servers in a display wall scenario. Estes et al. [15] describe a display abstraction that allows applications to expose some application semantics to remote displays. Note that applications can choose among a variety of frameworks. They can also use more than one framework; limiting the generality of this approach.

Capture appliances such as NCast (`ncast.com`) digitize and encode a VGA stream sent from the laptop to the projector. Using a proprietary point-to-point wireless network, WHDI [7] supports 1080p uncompressed video. The Intel WiDi [4], Apple Airplay mirroring [6] and Sony wireless screen mirroring encode screen updates into a wireless H.264 stream using special encoding hardware. These systems treat the screen contents as a movie and transmit them at fixed update rates. We empirically show that the rate of update creation on typical laptops is far higher than is supported by these systems. The playback devices are also not ubiquitous.

Screen capture as a sequence of pixelmaps is also popular. VNC [21] shares screen pixmap updates using the remote framebuffer (RFB) distribution protocol [20]. Wallace et al. [26], Boyaci et al. [11] and Satoshi [22] extend VNC to share pixmap updates from specific application windows. RFB is a client driven lazy request-response protocol that sends the captured pixmaps uncompressed or compressed using lossless Zlib [19] or lossy JPEG [12]. Clients adapt to network conditions by requesting new updates on receipt of responses for prior requests. The server is not required to respond to a client request within a specific time limit (if at all). Thus, pixmap updates can be missed because the client was too slow in requesting the next update or because the server was unwilling to respond faster. Prior work did not quantify the nature of pixmaps or the interactions between the RFB protocol and the screen capture. We empirically show the inadequacy of the pixmap delivery rate of 4.5 that was measured for traditional VNC servers [26].

Sun et al. [24] note high update rates for interactive thin-client applications. We empirically measure the update rates for various application scenarios. They improve compression efficiency by caching several prior pixmaps on the client and server. We only maintain current and prior screen contents on the server.

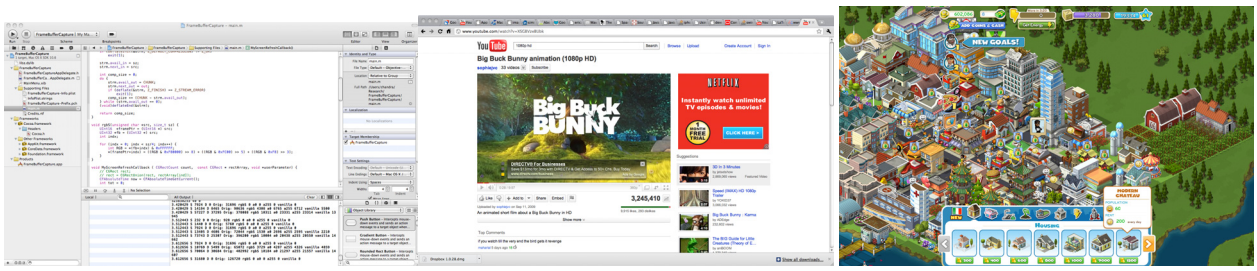
Schmidt et al. [23] describe the SLIM extension to VNC where remote application activity triggers update push from the server. In §5, we show that pixmaps need to be sent at high rates (that are unsupported by VNC) to avoid losing pixels to spatial overlap.

Intel Active Management Technology (AMT) implements a BIOS VNC server that does not rely on the operating system. Since the framebuffer does not retain information about whether any contents were updated, AMT periodically polls the entire framebuffer.

Google Chrome Remote Desktop encodes the screen as a VP8 video. The frame rate depends on VP8 performance on a particular device. H.264 compression is efficient. However, our state-of-the-art laptop achieved less than 10 fps while using hardware accelerated H.264 encoding. OnLive (`onlive.com`) hosts video games and applications on its cloud servers. The output screen is then divided into 16 segments, each of which is compressed separately using a special H.264 encoder to achieve low latency compression.

MS RDP is a proprietary screen sharing system. RDP reconfigures the desktop to ease the screen capture. It supports redirecting audio, GPU, file system, printer, pointer, port, aero glass and Windows Media player using tight integration between the service, operating system and applications. Not all RDP features are available in all OS versions. Our pixmap capture approach is portable and is applicable to the pixmap capture component of RDP.

Screencast is the act of capturing and distributing a podcast video of the display screen contents, especially during lecture presentations. For example, the TechSmith Camtasia tool allows for easy presentation screen capture and distribution through the `http://screencast.com` portal. They create a fixed frame rate video without considering the effect of pixel loss due to spatial overlap.



(a) Xcode (b) YouTube (c) Cityville

Figure 1: Screen capture from three usage scenarios

3. EXPERIMENT SETUP

3.1 Pixmap update collection framework

We used two 15" Macbook Pro laptops: one with a 2.66 GHz Intel Core 2 Duo CPU, 4 GB of 1067 MHz DDR3 memory, NVidia GeForce 9400M and 9600M GPUs with 256 MB of VRAM and a 1440x900 pixel display and another with a 2 GHz quad core Intel i7 CPU, 8 GB of 1067 MHz DDR3 memory, Radeon HD 6490M GPU with 256 MB of GDDR5 memory and Intel HD Graphics 3000 GPU with 384 MB shared memory and a 1680x1050 pixel display. We refer to these laptops as *C2D* and *i7*, respectively.

Each laptop was dual booted to either Windows 7 SP1 or Mac OS X SL (10.6.8) using Apple BootCamp. On Mac OS X, we used the *CGRegisterScreenRefreshCallback()* call to receive asynchronous notifications regarding screen updates, and *CGDisplayCreateImageForRect()* call to retrieve the pixel data. On Windows 7, we used the DemoForge mirror driver [2] to collect pixmap updates. This driver implements the Windows mirroring function [9]. The driver intercepts and stores the boundaries of the 20,000 most recent updates. Our analyzer polled the driver (every msec) to collect these stored updates. It then memory mapped the mirror driver's framebuffer copy to access the pixels that correspond to each pixmap update. Thus, the updates are pushed to our analyzer by Mac OS X while we use a polling model for Windows 7. We refer to each OS setup for the *C2D* and *i7* laptops as *C2D-Win7*, *C2D-Mac*, *i7-Win7* and *i7-Mac*, respectively. Note that applications running on Mac OS X and Windows 7 are optimized for direct display to the local monitor. Their behavior is likely to be different from applications that are optimized for network based X windows display system.

3.2 Activities analyzed

We capture screen pixmaps for the following operations.

- **HD trailer**: We played a full screen Avatar movie trailer using Quicktime X player on Mac OS X and Windows Media player on Windows 7. The H.264 trailer was 4:07 minutes long and was encoded at 1920x1080 pixels, 23.98 fps for a target bitrate of 9 Mbps. We refer to this scenario as *Avatar*.
- **Presentation**: We created a 1024x768 pixel resolution presentation using the Apple Keynote software and converted it to PowerPoint format. We played these presentations using Keynote on Mac OS X and PowerPoint on Windows 7. We transitioned each slide after about 15 seconds. We refer to these scenarios as *Keynote* and *PowerPoint*, respectively.
- **Integrated Development Environment (IDE)**: We analyzed the pixmaps created while developing our software instrumentation tools. We used Apple Xcode 4.0 on Mac and Visual Studio 2010 in Windows. These tools divide the screen into rectangular regions and use them for code editing, file browsing, function help and debugger outputs (Figure 1(a)). We referred to them as *Xcode* and *VS 2010*, respectively.
- **YouTube**: We watched the 1080p HD *"Big Buck Bunny"* animation clip (`http://youtu.be/XSGBVzeBUbk`), both in the default browser spatial resolution as well as in full screen mode using the Chrome browser. With the default spatial resolution, YouTube displays flash advertisements as well as comments and suggested videos around the video clip (Figure 1(b)). While using full screen, advertisements were overlaid on top of the clip. These scenarios are referred to as *YouTube* and *YouTube-FS*, respectively.
- **CNN.com**: We browsed *cnn.com* using Chrome (referred as *CNN-Chrome*). We read a whole article before skimming over others. CNN extensively uses advertisement banners.
- **Cityville game**: Finally, we played the Adobe Flash based Zynga Cityville game (at Level 66), in the native and full screen resolution (referred as *Cityville* and *Cityville-FS*, respectively). Cityville is a casual social city-building simulation game (Figure 1(c)). AppData estimated 100 million active players in January 2011. The game shows animations of people loitering; the rate of such movement depends on the rendering capabilities available to the browser.

4. RATE OF PIXEL GENERATION

Building a high performance real time screen sharing system requires an understanding of the rate of pixel generation, the spatial overlap between updates (§5) and the nature of the pixels (§6).

First we analyse the rate of pixel generation, the number of pixels in each pixmap and the rate at which each pixmap is created. This analysis helps choose between a design optimized to capture 24 full screen updates (to play a movie) vs another that captures 240 pixmaps, each of which has $\frac{1}{10}^{th}$ the number of pixels per second.

For the various applications, we tabulate the mean and standard deviation of the total number of pixels and updates per second in Table 1. The total number of pixels varied between 0.6 and 34.0 on *C2D* and between 1.5 and 49.9 on *i7*. Average update rate varied between 9.6 and 50.1 for *C2D* and between 9.5 and 80.7 for *i7*. The standard deviation of the update rates was high: e.g., *PowerPoint* had a mean update rate of 40.1 and standard deviation of 64.0.

C2D and *i7* should generate 38.9 and 52.9 Mpixels while refreshing the screen at 30 fps, respectively. Thus, the number of pixels were less than would be generated for treating the screen as a fixed frame rate (30 fps) movie: the rate of pixmap generation was higher. The mismatch was pronounced for interactive scenarios such as *Xcode/VS 2010*, *Keynote/PowerPoint* and *CNN* which exhibited high update rates with a small number of updated pixels (<10K pixels). E.g., *PowerPoint* in Windows 7 generated 61.9 updates per second (σ: 113.5) and only 1.7 Mpixels per second.

scenario	Million pixels per second								Updates per second							
	C2D				i7				C2D				i7			
	Mac		Win7		Mac		Win7		Mac		Win7		Mac		Win7	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Avatar	27.4	3.8	26.8	5.5	37.6	4.8	36.6	7.0	24.4	5.1	49.5	13.7	25.1	8.8	51.2	18.2
Cityville	10.1	3.6	10.0	1.5	12.2	4.2	18.2	2.6	30.8	11.8	41.9	23.8	52.3	18.8	63.2	39.6
Cityville-FS	25.3	8.1	20.4	4.4	42.3	10.3	47.8	7.1	22.4	16.1	47.1	41.5	30.1	21.8	47.9	36.0
CNN-Chrome	3.9	5.5	2.5	3.3	9.0	10.9	2.5	4.0	19.9	22.9	50.1	41.6	80.7	62.3	40.9	39.1
Keynote	5.5	1.9			6.2	3.2			9.6	13.6			9.5	13.4		
PowerPoint			2.6	2.3			1.5	1.9			29.7	34.0			40.1	64.0
Xcode	2.0	2.8			1.6	2.1			13.1	14.6			16.3	16.4		
VS 2010			0.6	2.2			1.7	5.9			37.8	39.4			61.9	113.5
YouTube	6.2	0.6	5.5	0.5	6.1	0.7	5.5	0.8	29.3	8.8	31.9	10.8	28.5	13.6	33.8	15.6
YouTube-FS	34.0	5.7	33.6	5.0	46.0	8.9	49.9	7.4	27.5	5.7	28.9	15.6	28.0	6.7	31.4	14.6

Table 1: Pixel generation characteristics

The 24 fps *Avatar* trailer created around 24 and 50 ups while using Mac OS X and Windows 7, respectively. The number of pixels updated was around 27 and 37 Mpixels for Mac OS X and Windows 7 with little variation in the number of pixels between the two laptops. The Windows Media player was aggressively displaying partial frames as they were decoded. *YouTube* (video played by Adobe Flash, both within the browser and in full screen mode) did not exhibit much variation between the different operating systems or laptops. *Cityville* scenario required high update rates and high number of updated pixels especially with Windows 7 requiring more updates for the same number of updated pixels.

Next, we drill down on the number of pixels in each update as well as the duration between updates.

4.1 Number of pixels in each update

For the various usage scenarios, we plot the cumulative distribution of the number of pixels in each update in Figure 2. For *Avatar*, about 5% of the pixmaps contained less than 10,000 pixels. The remaining 95% of the pixmaps occupied 90% of the screen with Mac OS X and 45% of the screen with Windows 7. This movie clip was played by the Quicktime player on Mac OS X and by Windows Media player on Windows 7. In §4.2, we show that the rate of pixmap creation was different between these scenarios. For *YouTube-FS*, less than 4% of pixmaps on Mac OS X and 12% of the pixmaps on Windows 7 were less than 10,000 pixels. The remaining pixmaps updated the entire screen. When letterboxed, unlike Flash player, Quicktime and Windows Media player only updated the active region. We describe its implications in §4.2.

Even though *Cityville-FS* used the Flash player plugin on both operating systems, it produced different screen pixmap behavior. On Windows 7, 50% and 65% of the updates were small while on Mac OS X, only 20% and 10% of the updates were small with *C2D* and *i7* laptops, respectively. The flash plugin likely implements the graphics primitives differently between the two OSs.

For *Keynote*, 55% and 65% of the updates were small on the *C2D* and *i7* laptops, respectively. While using *PowerPoint*, 75% and 95% of the pixmaps were small using *C2D* and *i7* laptops, respectively. Similar results were observed for other scenarios as well. In general, a faster processor allows for faster animation experience which manifests itself in many small updates. In §6.2, we investigate the inter-update redundancy among these updates.

4.2 Duration between updates

Next we plot (Figure 3) a distribution of the duration between pixmaps for the various usage scenarios. For Mac OS X, the discrete nature of the durations (about 18 msec apart) is an artifact of the OS policies for scheduling callbacks.

For *Avatar*, 49% of the pixmaps were sent 50 msec apart while the remaining pixmaps were sent around 32 msecs apart with Mac OS X. The movie was encoded at 24 fps, which corresponds to a duration of 42 msec between frames; the screen pixmaps do not correspond to constant frame creation rates. While using Windows 7, the screen pixmaps were equally distributed between 0 and 42 msecs apart. As we noted in Table 1, on average, this corresponds to twice the frame rate as the encoded rate; Windows Media player appears to progressively decode the movie. With Mac OS X and *YouTube-FS*, 20%, 40% and 40% of the pixmaps were captured at 18, 32 and 50 msec apart, With Windows 7, 50% were received about 42 msec apart while the remaining pixmaps were received between 0 and 42 msec apart. Deducing whether the user is watching a movie by computing the pixmap update geometry and the rate in which they are updated (similar to [11, 18]) would fail.

For interactive scenarios, Windows 7 creates updates at a furious pace. With *VS-2010*, 20% and 40% of the updates were created immediately after each other with the remaining 60% and 50% of the updates were sent within 10 msec of each other for *C2D* and *i7* laptops, respectively. The numbers were similar for *PowerPoint* in which 60% of the updates were sent immediately and 95% of the updates sent within 16 msec of each other. The remaining updates were sent several seconds apart. For *Xcode*, between 40% and 50% of the pixmaps were captured immediately after each other. These values were similar for *Keynote* scenario as well.

To summarize, many applications generated large spikes of screen updates during user interaction. Movies updated the screen at a rate that was largely driven by the media object. However, some movie players showed partially decoded frames. Animations are GPU accelerated which adds to their visual appeal. However, they generated pixmaps at a rate that was only limited by the available computational resources. Improvements in processing power exacerbate this trend. In all scenarios, the number of updated pixels was smaller than for treating the screen as a fixed frame rate movie. Prior systems that used heuristics such as the update dimensions and rate to deduce information about the screen contents (e.g., movie playback) were shown to be fragile.

5. SCREEN SHARING CONSTRAINTS

§4 showed the update rates for various usage scenarios. Storage and network capacity limitations require compressing the screen pixmaps (uncompressed, *i7* requires 1.7 Gbps at 30 fps). However, the available computational capacity might not allow for com-

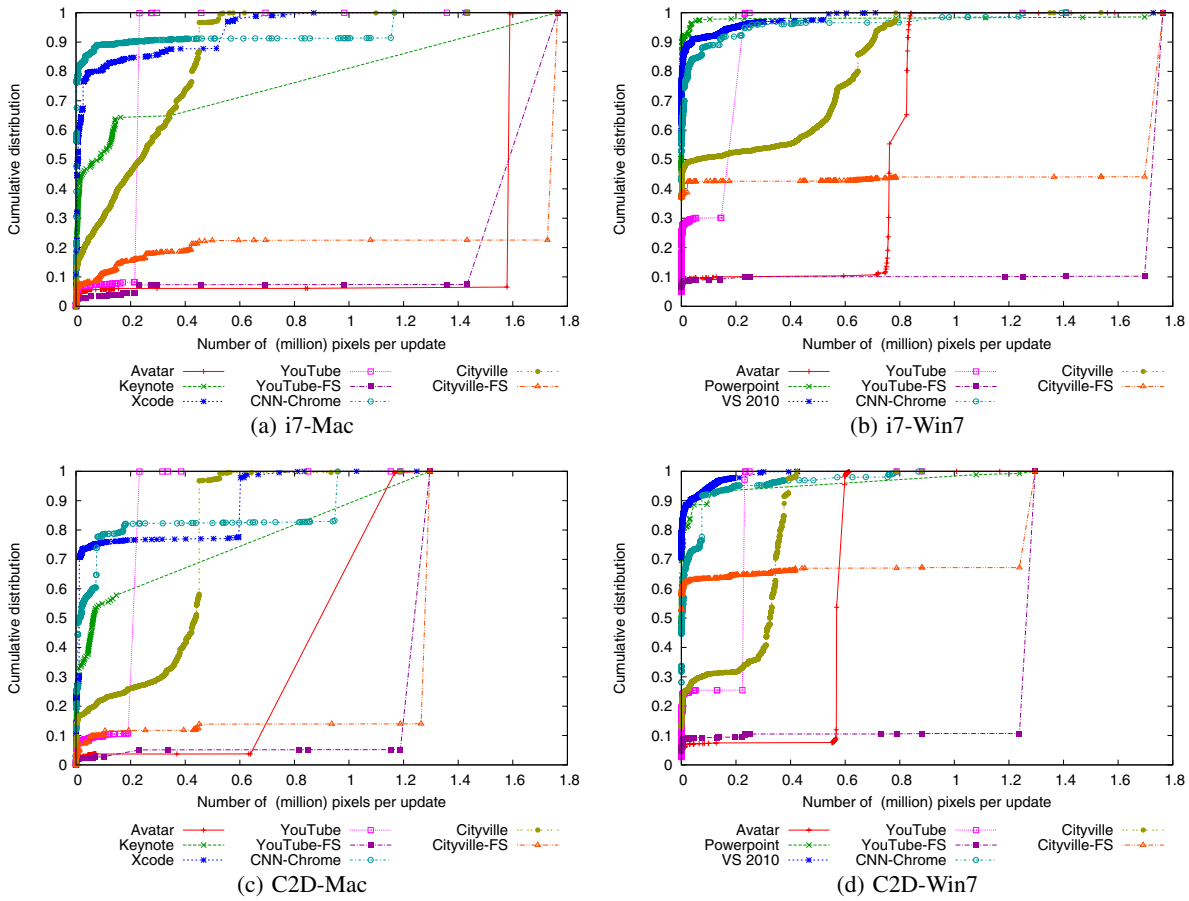


Figure 2: Distribution of number of pixels in each update

pression at high capture rates, especially since this computational need competes with processes that are simultaneously rendering to the screen. Even when they are compressed, the network capacity might be insufficient; e.g., Bob's in-flight WiFi severely constrains his ability to receive many updates from Emily's cloud servers.

X windows applications reduce the network load by rendering offline and then infrequently updating the screen. However, applications designed for desktop platforms (Mac OS X and Windows 7) choose faster update rates (§4). Future trends portend to hardware accelerated screen elements that generate even more updates.

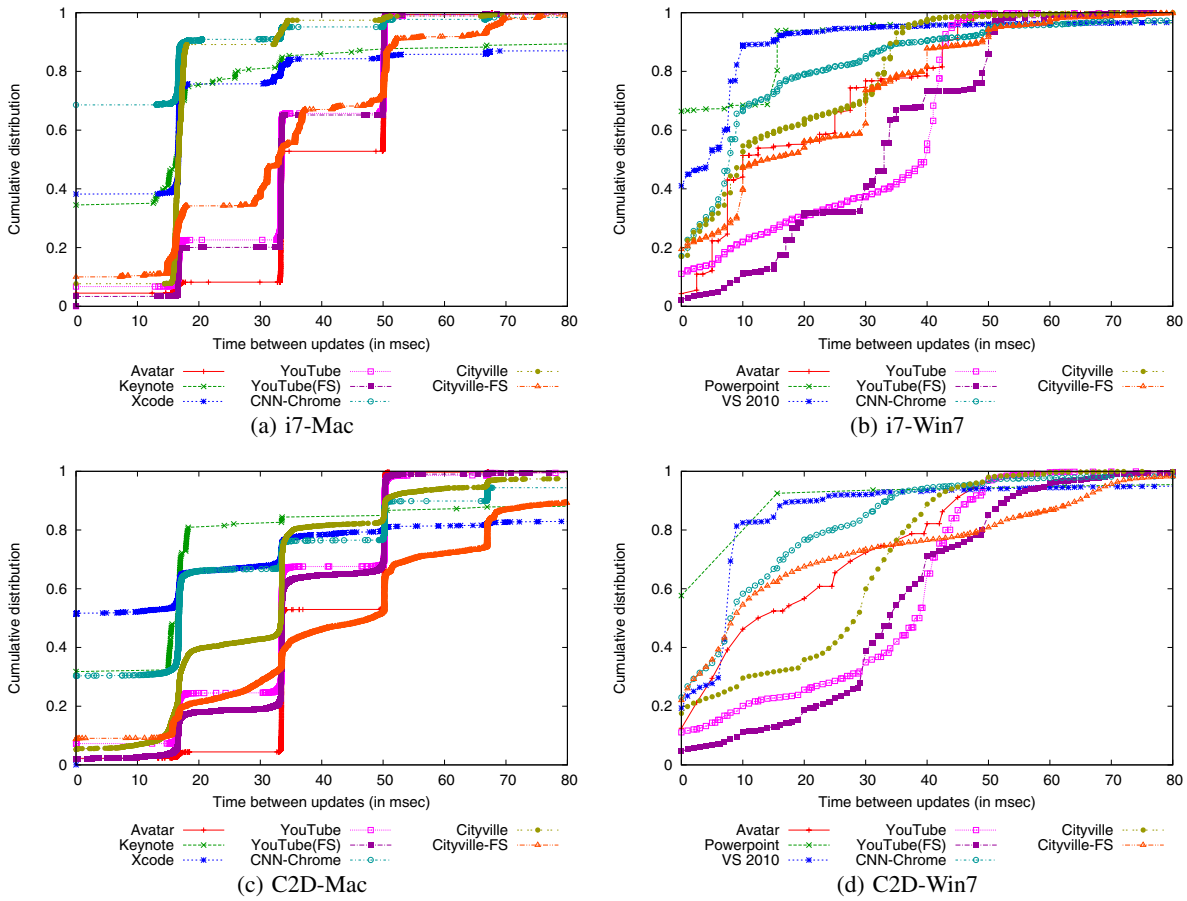
Ideally, sharing systems will capture, compress and deliver the updates at the rate in which they are generated. When resource constraints make this infeasible, they can buffer all updates and eventually transmit them (even if the pixels are super-ceded). This buffering causes CPU, memory and network data spikes during active durations. We investigate ways to address this mis-match.

5.1 Coalescing updates

One approach coalesces and creates a new screen update that encompasses all unsent updates. Unlike disjoint updates, spatially overlapping pixels are lost. The effect of this loss depends on the application scenario. During Quicktime movie playback, a delay in capturing a frame results in a lower frame rate video viewing experience. As noted in §4, Windows Media player aggressively displays partially decoded segments of a movie frame. Any delay causes a low frame rate video viewing experience in which each captured frame might consist of elements from multiple movie frames. In a network constrained remote login scenario, we frequently observed this effect while using VNC/RDP with Windows Media playback from a Mac OS X (the MS RDP player for Mac does not support media player redirection which masks this effect). The pixels lost through spatial overlap also affects systems such as [16] that analyze the screen using optical character recognition techniques. Delaying and removing pixels causes these systems to miss pixels that could be crucial for the recognition task.

We investigate spatial overlap using a fixed period in which all updates are coalesced (similar to Camtasia and Quicktime). We plot the percentage of unseen pixels for capture frequencies of 10, 15, 20, 24, 30, 60 and 100 ups while using Mac OS X and Windows 7 on *i7* in Figure 4. For brevity, we highlight two representative scenarios: interactive (*VS-2010* and *Xcode*) and video (*YouTube-FS*). Applications using *C2D* created fewer updates and hence experienced slightly lower number of lost pixels (not illustrated).

For *YouTube-FS*, slow capture rates of 20 ups, 15 ups and 10 ups lost about 30%, 45% and 65% of the pixels, respectively. Capture frequencies of 60 ups lost 5% of the pixels. Even though the *Avatar* trailer was encoded at 24 fps, our analysis in §4.2 showed that the updates were not always received at uniform intervals on Mac OS X. Thus, from watching the *Avatar* trailer on Mac OS X, capturing the update at 24 ups, sometimes, two updates are received within the fixed capture duration leading to a loss of 50% of the pixels. A higher frequency (e.g., 30 ups) reduces the percentage of unseen pixels. On Windows 7, the media player doubles the rendering rate



(a) i7-Mac (b) i7-Win7

(c) C2D-Mac (d) C2D-Win7

Figure 3: Distribution of duration between each update

to 50 ups. Capture frequencies of 20 ups, 15 ups, and 10 ups lose about 18%, 38% and 59% pixels, respectively.

Scenarios in which screen updates were driven by user interaction (*Xcode*, *VS 2010*, *Keynote*, *PowerPoint* and *CNN-Chrome*), there were durations in which lower capture rates were acceptable. For example, presenters advance to the next slide and then discuss the slide contents for several minutes with no further screen updates. However, when the user interacted with the system, there was a flurry of activity. For example, the *Xcode* scenario lost 20% of the pixels even at a capture rate of 30 ups; we require a rate of over 60 ups to capture all the pixels. For a *Keynote* presentation, the updates were few and far between. However, *Keynote* extensively uses GPU accelerated transitions. During a slide transition, a large number of updates were created in a short duration; over 40% of which can be lost for capture rates of 30 updates per second. Even at 60 ups, 20% of the pixels were lost.

We observed little loss when choosing a capture rate of 100 ups.

5.2 Selectively choosing updates

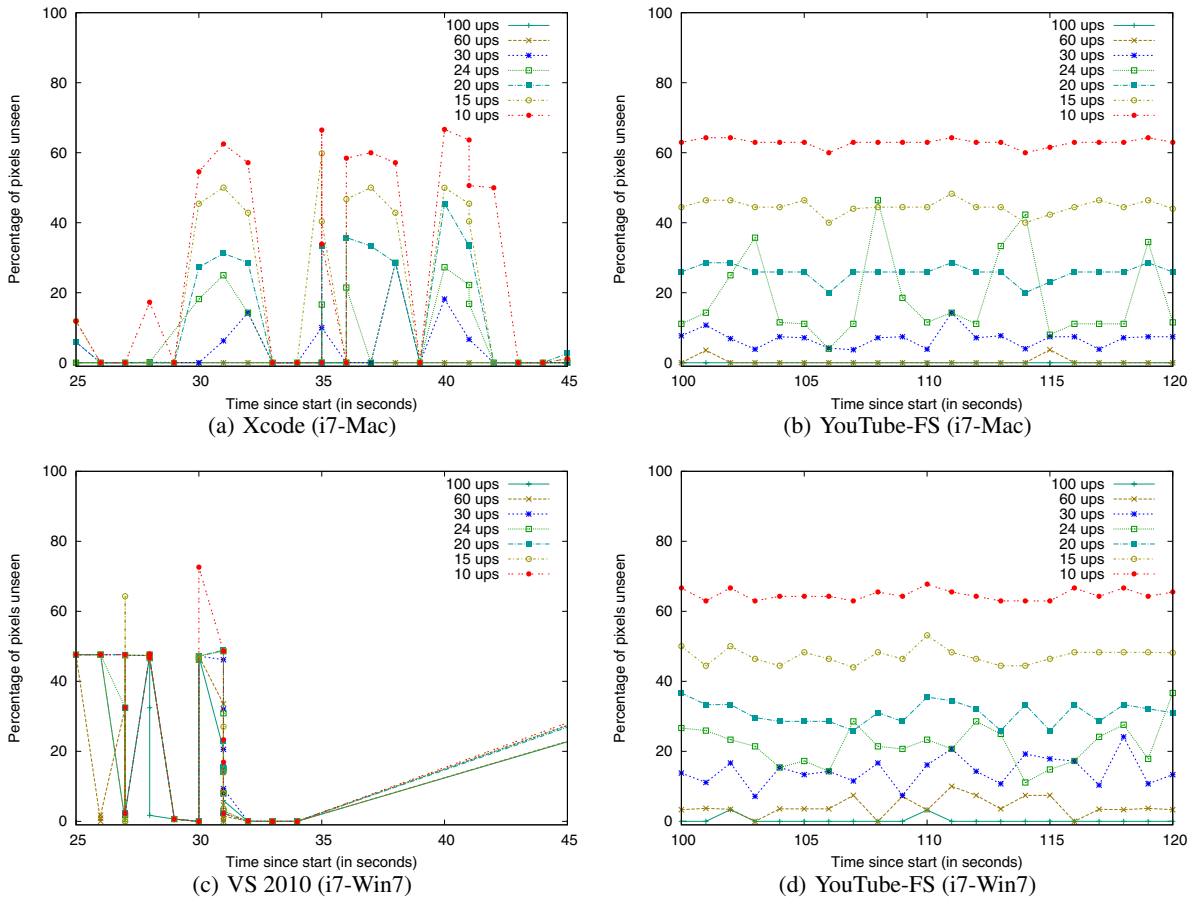
Another approach could prioritize pixmaps and dynamically drop low priority ones. Consider a sequence of updates u_1, u_2, u_3, u_4, u_{10} and u_{20} using the notation that u_n was captured at time n. When the resource availability could only sustain three updates, we might choose u_4, u_{10} and u_{20} instead of u_1, u_2, u_{20}. In the former case, we waited for the flurry of updates (u_1, u_2, u_3 and u_4) to subside and then chose u_4 while the latter case aggressively sends all updates until resource limitation forces it to drop pixmaps. The challenge is to perform these choices without application support, especially when multiple application elements with differing requirements might be overlaid on the same screen. Note that scenarios such as *Cityville* continuously create fast updates; we require a policy that waits for a certain duration before sending the next update. Using our results in Figure 3, we investigate the duration that we should wait before choosing an update for sharing.

§4.2 showed that Mac OS X delivered callbacks at intervals of 18 msec with many updates delivered immediately after each other. Thus, waiting an additional msec after the callback achieves our goal of picking the last update in an animation sequence. If the pixmaps can be compressed and sent during the 18 msec interval between callbacks, Mac OS X can sustain 60 ups.

Besides pixmaps which were delivered immediately after each other, duration between updates varied uniformly for up to 50 msec in Windows 7. Interactive scenarios delivered a majority of their screen updates in a short duration. E.g., with *i7*, 90% and 95% of the updates for *VS 2010* and *PowerPoint* were delivered within ten and 16 msec, respectively. For *C2D*, 90% of the pixmaps were delivered within 16 msec. Hence, 16 msec is a reasonable duration to wait to capture the end of an update flurry in Windows 7.

6. COMPRESSION IMPROVEMENT

Next, we investigate spatial and temporal similarity that is inherent in the screen pixmaps. Compression algorithms achieve high compression factors by exploiting both these forms of redundancy.



(a) Xcode (i7-Mac) (b) YouTube-FS (i7-Mac)

(c) VS 2010 (i7-Win7) (d) YouTube-FS (i7-Win7)

Figure 4: Pixels occluded by coalescing updates

scenario	Mac OS X	Windows 7
Avatar	21.0:1	8.7:1
Cityville	5.3:1	5.1:1
Cityville-FS	3.2:1	2.8:1
CNN-Chrome	5.9:1	12.5:1
Keynote/PowerPoint	25.6:1	9.6:1
Xcode/VS 2010	23.2:1	21.4:1
YouTube	4.8:1	2.4:1
YouTube-FS	10.5:1	5.9:1

Table 2: Average Zlib compression factor

scenario	C2D		i7	
	comp	fps	comp	fps
Avatar	190.6:1	11.9	349.1:1	11.8
Cityville	252.0:1	11.1	171.5:1	14.0
Cityville-FS	102.9:1	11.1	84.9:1	12.6
CNN-Chrome	2283.0:1	7.1	2139.3:1	7.9
Keynote	5956.5:1	5.1	7797.7:1	5.1
Xcode	282.5:1	11.7	2432.3:1	7.3
YouTube	239.5:1	15.0	262.3:1	14.9
YouTube-FS	284.9:1	14.8	499.3:1	14.7

Table 3: H.264 screen capture with Quicktime

Note that screen pixmaps are heterogeneous; mechanisms [27] that were developed for computer generated images need not always achieve good compression factors.

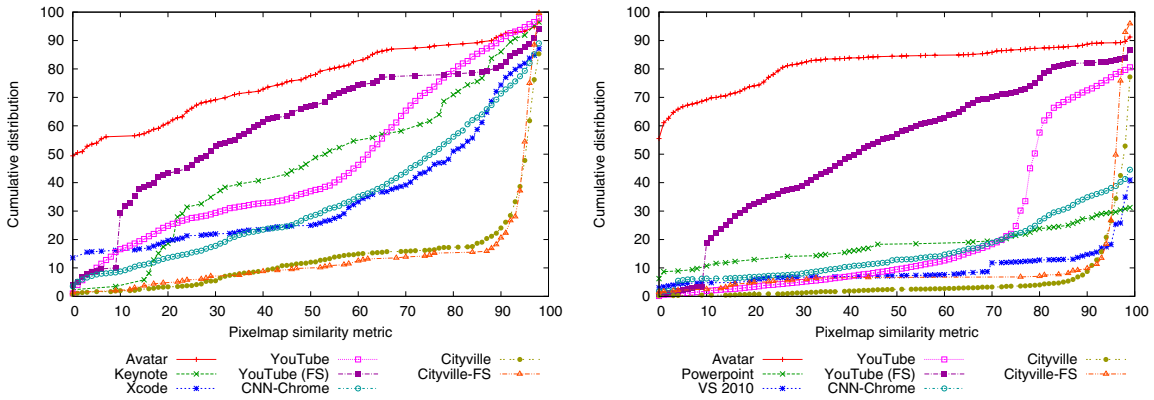
6.1 Inter-update temporal similarity

First, we investigate the similarity between an pixmap and the corresponding displayed pixels. We can avoid sharing fully redundant updates. Partial redundancy requires further processing to avoid sending those redundant pixels. Note that the screen capture mechanism should minimize its share of CPU usage; precluding motion vectors and other mechanisms that locate translated pixels.

6.1.1 Pixmap similarity metric

We measure the inter-update redundancy using a pixmap similarity metric. For each pixmap, we calculate the percentage of pixels that were the same as the corresponding displayed pixel. Thus, a pixel similarity metric of 50 could mean that every other pixel remained the same or the entire top half of the pixmap was similar. These two pixmaps could exhibit different compression performance. We plot the cumulative distribution of the similarity metric for the various usage scenarios in Figure 5.

For the *Avatar* trailer, 9% and 2% of the pixmaps had a similarity metric of 100 while using Windows 7 and Mac OS X, respectively. Even though Windows Media player doubled the pixmap rate (§4.2), the pixmaps covered disjoint regions and did not translate into a similarity metric of 50. Also, a larger share of pixmap updates had less pixel similarity while using Windows 7 than with



(a) i7-Mac (b) i7-Win7

Figure 5: Pixel similarity from prior update

scenario	i7-Mac				i7-Win7			
	alpha-0	*alpha-255*	*RGB(5:6:5)*	*bitmap*	*alpha-0*	*alpha-255*	*RGB(5:6:5)*	*bitmap*
Avatar	39.3:1	39.0:1	46.9:1	89.3:1	14.1:1	12.1:1	14.1:1	35.1:1:1
Cityville	67.0:1	66.2:1	9.1:1	99.0:1	53.6:1	44.0:1	8.2:1	143.7:1
Cityville-FS	34.2:1	34.0:1	5.3:1	40.0:1	28.7:1	25.2:1	6.0:1	53.5:1
CNN-Chrome	49.5:1	48.5:1	11.7:1	80.1:1	49.2:1	39.3:1	18.0:1	134.0:1
Keynote/PowerPoint	35.8:1	36.0:1	50.6:1	42.9:1	23.8:1	20.5:1	13.3:1	43.9:1
Xcode/VS 2010	82.4:1	81.5:1	33.1:1	142.1:1	42.7:1	36.0:1	28.6:1	82.4:1
YouTube	16.6:1	16.5:1	10.9:1	28.1:1	7.8:1	6.9:1	4.5:1	12.7:1
YouTube-FS	44.6:1	44.6:1	20.8:1	104.0:1	12.9:1	10.8:1	10.3:1	29.2:1

Table 4: Compression factors for various pixmap transformations (followed by Zlib)

Mac OS X. For example, 66% of Mac OS X updates and 79% of Windows 7 pixmaps had a pixel similarity metric of 25 or lower.

For all other scenarios, Windows 7 showed a higher similarity metric than Mac OS X. For example, with Windows 7, between 80% (*PowerPoint*) and 97% (*Cityville*) of the pixmaps had a similarity metric higher than 70. With Mac OS X, these values corresponded to between 50% (*CNN-Chrome*) and 85% (*Cityville-FS*). Windows 7 appears to create more updates, each of which were temporally similar to prior updates at the same screen location.

6.2 Intra-update spatial redundancy

Next, we empirically investigate intra-update spatial redundancy and its impact on compressing the update pixmaps using Zlib [14]. Zlib is a popular lossless compression mechanism that uses a variation of LZ77 [29] algorithm. It is widely available in a range of computing platforms and is an inherent component of Java and .NET. VNC uses Zlib for compressing screen updates.

We tabulate the average compression factor using Zlib for the various application scenarios in Table 2. Higher compression factors are preferred. The compression factor depends on the application scenario with better performance on Mac OS X (because of different compression quality vs CPU usage tradeoff, we use .NET 4.0 - *DeflateStream()*). *Avatar* achieves a compression factor of about 21:1 on Mac OS X and only 8.7:1 in Windows 7. *YouTube* and *YouTube-FS* achieve twice the improvement on Mac OS X. Scenarios which require user interaction (e.g., *PowerPoint*, *Keynote*, *Xcode* and *VS 2010*) achieve compression factors of about 20:1. The improvements are more modest for other scenarios.

For comparison, we repeated the various scenarios and performed an H.264 compressed screen capture using the Quicktime X player on Mac OS X. We analyzed these compressed movies and tabulate the compression factors as well as the frame rates achieved in Table 3. Each GOP on these H.264 encoded movies consist of a sequence of IPBPBPBP frames. The compression factors were computed as a factor of the compressed frame size (I, P or B frames) to the full movie spatial dimension (ie, 1440x900 24 bit pixels for the *C2D* laptop and 1680x1080 for the *i7* laptop). Rather than naively compressing the entire screen, it is likely that the Quicktime compressor only considered changed pixel regions. Otherwise, the frame rate would be constant irrespective of the usage scenario; we observed a frame rate range between 5 and 20. Quicktime achieves compression factors between 84:1 and 6000:1. The Quicktime X player is highly optimized to use the laptop GPUs. The captured movies are also heavily compressed (lossy) using H.264 parameters.

Next, we investigate ways to improve the compression efficiency for high capture rates using lossless Zlib.

6.3 Transformation for better compression

§6.2 showed the poor compressibility of screen pixmaps. Zlib compression only uses intra-update redundancy. We improve compression ratios by incorporating inter-update similarity (§6.1). We also exploit the way pixels are represented in pixmaps. Screen pixmaps are represented in a 32 bit RGBA format with opaque alpha value (0xFF). We make the temporally redundant pixel transparent by setting the alpha value to 0x00. We can then change the RGB values of these transparent pixels to any arbitrary value that will improve the pixmap compression factor; we chose either 0x00 or 0xFF for *alpha-0* and *alpha-255*, respectively. For example, a high pixel similarity metric pixmap, will contain a large sequence of 0x00FFFFFF with *alpha-255* and 0x00000000 with *alpha-0*. The receivers recover the original pixel values by alpha compositing the transformed pixmap.

	C2D						i7					
	Mac			Win7			Mac			Win7		
	μ	max	σ	μ	max	σ	μ	max	σ	μ	max	σ
Avatar	43.9	80.2	15.4	69.0	126.1	24.2	49.7	97.8	19.2	94.2	191.4	36.6
Cityville	5.6	24.4	4.6	41.5	52.7	10.5	6.5	27.9	6.2	65.7	78.5	13.7
Cityville-FS	5.1	27.5	4.6	44.1	54.1	9.3	9.0	36.1	4.0	65.6	95.2	12.0
CNN-Chrome	3.1	25.5	5.0	2.6	46.6	6.5	4.6	44.2	7.3	2.8	61.2	8.8
Keynote	0.1	5.4	0.5				0.2	15.7	1.3			
PowerPoint				0.4	8.3	1.4				0.2	4.6	0.7
Xcode	0.6	6.1	1.3				0.5	12.4	1.2			
VS 2010										1.0	12.1	2.1
YouTube	24.4	48.0	10.4	84.9	123.4	18.2	27.8	55.6	12.0	100.6	150.2	26.4
YouTube-FS	41.7	78.9	14.0	69.3	112.8	16.2	48.9	92.6	14.6	116.9	191.8	32.5

Table 5: Bandwidth consumed for *bitmap* encoding

	C2D						i7					
	Mac			Win7			Mac			Win7		
	μ	max	σ	μ	max	σ	μ	max	σ	μ	max	σ
Avatar	4.0	59.0	6.5	8.9	91.0	9.6	4.1	73.0	8.1	10.2	122.0	13.7
Cityville	19.9	174.0	18.3	30.2	119.0	20.0	35.3	100.0	15.1	58.2	157.0	26.5
Cityville-FS	12.8	112.0	9.3	10.6	73.0	10.5	16.3	143.0	20.9	14.4	172.0	27.4
CNN-Chrome	20.8	134.0	26.6	11.8	188.0	26.7	47.2	175.0	54.5	21.3	117.0	18.8
Keynote	1.0	76.0	5.8				1.0	53.0	4.9			
PowerPoint				2.7	115.0	13.8				3.6	240.0	21.4
Xcode	12.8	85.0	15.7				16.1	75.0	18.8			
VS 2010										49.2	207.0	38.6
YouTube	41.1	82.0	11.8	52.1	163.0	8.7	40.5	108.0	11.7	31.9	156.0	16.9
YouTube-FS	4.9	87.0	8.5	4.8	219.0	16.1	4.8	112.0	11.2	5.7	114.0	11.6

Table 6: Frame rates achieved for *bitmap* encoding

We also investigated two other transformations. For *bitmap*, we separated pixmaps into a similarity byte-map and pixel data. We use 0x00 and 0xFF to represent similar and dissimilar pixels in the byte-map, respectively. The pixel data is a sparse array that stores the 24 bit RGB values of dissimilar pixels. We also considered a lossy colormap transformation to *RGB(5:6:5)* format.

The transformed pixmaps were then Zlib compressed. For the various pixmap transformations, we tabulated the average compression factor for *i7-Mac* and *i7-Win7* scenarios in Table 4. The transformations improved the compression factors achieved with Zlib (Table 2) in all scenarios. For Mac OS X and Windows 7, we see the best improvement in all scenarios for the *bitmap* transformation. *alpha-0* provides the next best improvement except *Avatar* where RGB(5:6:5) doubles the compression factor over Zlib.

For *Avatar* in Windows 7, the transformation only improved the compression factor of 8.67:1 achieved by Zlib to 14.13:1, 12.06:1 and 35.08:1 for *alpha-0*, *alpha-255*, and *bitmap* transformations. This improvement was similar for the *YouTube-FS* scenario, where Zlib achieved a compression factor of 5.95:1, while the transformations improved them to 12.85:1, 10.79:1 and 29.22:1 for *alpha-0*, *alpha-255*, and *bitmap* transformations, respectively.

Even though the computer generated animation of *Cityville-FS* was unsuitable for Zlib (compression factor around 3:1), pixmap transformations showed an order of magnitude improvement (40:1 for Mac OS X and 53:1 for Windows 7). These improvements still dwarf the compression factors achievable with lossy H.24 compression (Table 3). On the other hand, we can achieve higher update rates (discussed in more detail in §6.3.1).

We also showed that the transformations were more effective for highly similar pixmaps (not illustrated for lack of space).

6.3.1 *bitmap* transformation performance

Bitmap uses the inter-update redundancy found in most scenarios to improve Zlib performance. Next, we tabulate the mean, standard deviation and maximum bandwidth consumed as well as the pixmap update rates achieved using a *bitmap* based screen sharing system in Mac OS X and Windows 7 in Tables 5 and 6, respectively.

On average, the *Avatar* clip consumed about 47 and 80 Mbps while achieving 4 and 10 ups on Mac OS X and Windows 7, respectively. Since Windows Media player doubled the update rate, this corresponds to a video playback rate of 5 fps. These values were worse than the 9 Mbps and 24 fps achieved by Quicktime screen capture. Similarly, *YouTube-FS* consumed about 45 and 100 Mbps to achieve a update rate of 4.9 and 5.2 for using Mac OS X and Windows 7, respectively. The .NET 4.0 *DeflateStream()* sacrifices compression efficiency for computation overhead. Regardless, the CPU was the bottleneck in both scenarios. We are investigating ways to automatically detect movie playback and then encode movie pixmaps using hardware assisted H.264 encoding (AV Foundation in Mac OS X and Intel Media SDK for Windows 7).

For *Cityville-FS*, we required 7 and 55 Mbps while achieving on average 15 and 12.5 ups on Mac OS X and Windows 7 respectively. However, the system achieved maximum pixmap rates of 172 ups while consuming 95 Mbps. Similarly, *PowerPoint* and *Keynote* sometimes generated up to 240 and 76 ups. We captured up to 86 updates for *Xcode* and 207 updates for *VS 2010*. The maximum bandwidth consumed for all scenarios was within the capacity of IEEE 802.11n wireless network. The upcoming IEEE 802.11ac also promises significant network capacity improvements.

Even though *bitmap* does not achieve the compression factors achievable with H.264, it compresses them at a higher rate than

is even achievable with hardware assisted H.264 encoding. On a wireless LAN network, this tradeoff of higher update rate while using more network bandwidth was acceptable.

7. IMPLEMENTATION EXPERIENCE

Based on the observations described in this paper, we have built and deployed a DisplayCast system [13] for screen capture and distribution. Native implementations in Mac OS X and Windows 7 provide utilities to capture the screen, play them back in real time as well as archive them to a H.264 movie. The various components locate each other using Zeroconf [28]. We provide a HTTP/REST API to control the system. Our system is open sourced.

8. DISCUSSION

We empirically analyzed the performance of pixmap screen capture and showed that the capture rates need to be as high as hundred ups for interactive scenarios. Moore's law advancement in processor capability as well as recent trends of using hardware assisted screen artifacts further exacerbates these rates. We offer practical guidelines for adapting to lower capture rates under resource constrained environments. We devised a simple mechanism to transform inter-update temporal redundancy into intra-update spatial redundancy in order to achieve good compression factors and high capture rates. We incorporate our findings into a practical and open sourced system that works in Mac OS X and in Windows 7 [13].

Newer operating systems such as Mac OS X Mountain Lion [1] and Windows 8 [8] are introducing tablet inspired features such as swiping, pinching, zooming and other operations that are invoked using multi-touch gestures. These operations make extensive use of hardware acceleration capabilities that are built into modern computers. Though these features improve the user experience, they generate enormous amounts of screen updates. For example, a full screen swipe that is tied to the rate in which the user flips the trackpad generates a full screen update that shifts the entire screen by one pixel. High resolution retina displays can also increase the number of pixels in each update. There is a need to implement higher capture rates to avoid losing pixels on these newer systems.

9. REFERENCES

[1] Apple - OS X mountain lion. http://www.apple.com/osx/.

[2] Demoforge mirage driver (dfmirage video hook driver). http://www.demoforge.com/dfmirage.htm.

[3] GLUT and OpenGL utility libraries. http://www.opengl.org/resources/libraries/glx/.

[4] Laptop to TV with Intel® wireless display (WiDi). http://www.intel.com/content/www/us/en/architecture-and-technology/intel-wireless-display.html.

[5] Remote desktop connection. http://windows.microsoft.com/en-US/windows7/products/features/remote-desktop-connection.

[6] Stream movies and music wirelessly with airplay. http://www.apple.com/ipad/features/airplay/.

[7] WHDI - wireless home digital interface. www.whdi.org.

[8] Windows 8 release preview. http://windows.microsoft.com/en-US/windows-8/release-preview.

[9] Windows mirror drivers. http://msdn.microsoft.com/library/ff568315.aspx.

[10] Multimedia redirection improvements in Windows 7 and WS2008 R2. http://goo.gl/W7Uv4, July 2009.

[11] BOYACI, O., AND SCHULZRINNE, H. Bass application sharing system. In *IEEE International Symposium on Multimedia* (Berkeley, CA, USA, 2008), ISM '08, pp. 432–439.

[12] CCITT RECOMMENDATION T.81. *Digital Compression and Coding of Continous-Tone Still Images - Requirements and guidelines*. International Telecommunication Union(ITU), Geneva, Sept. 1992.

[13] CHANDRA, S., AND ROWE, L. A. DisplayCast: a high performance screen sharing system for intranets. In *ACM Multimedia 2012* (Nara, Japan, Oct. 2012). https://github.com/DisplayCast.

[14] DEUTSCH, P. Deflate compressed data format specification version 1.3. RFC 1951, May 1996.

[15] ESTES, C. D., AND MAYER-PATEL, K. The n-dimensional display interface: a more elastic narrow waist for the display pipeline. In *MMSys '12* (Chapel Hill, NC, USA, Feb. 2012), pp. 119–129.

[16] HILBERT, D. M., TURNER, T., DENOUE, L., AND SANKARPANDIAN, K. Autonomous presentation capture in corporate and educational settings. In *IADIS e-Learning'08* (Amsterdam, The Netherlands, July 2008), pp. 239–246.

[17] HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. WireGL: a scalable graphics system for clusters. In *SIGGRAPH '01* (Los Angeles, CA, USA, 2001), pp. 129–140.

[18] KIM, H., JEONG, J., HWANG, J., LEE, J., AND MAENG, S. Scheduler support for video-oriented multimedia on client-side virtualization. In *MMSys '12* (Chapel Hill, NC, USA, 2012), pp. 65–76.

[19] LOUP GAILLY, J., AND ADLER, M. zlib: A massively spiffy yet delicately unobtrusive compression library. zlib.net.

[20] RICHARDSON, T., AND LEVINE, J. The remote framebuffer protocol. RFC 6143, Mar. 2011.

[21] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing 2* (Jan. 1998), 33–38.

[22] SATOSHI, U. MetaVNC - a window aware VNC. http://metavnc.sourceforge.net/.

[23] SCHMIDT, B. K., LAM, M. S., AND NORTHCUTT, J. D. The interactive performance of slim: a stateless, thin-client architecture. In *SOSP '99* (Charleston, SC, USA, 1999), pp. 32–47.

[24] SUN, Y., AND TAY, T.-T. Analysis and reduction of data spikes in thin client computing. *J. Parallel Distrib. Comput. 68*, 11 (Nov. 2008), 1463–1472.

[25] VON HOFFMAN, J. T. *Guide to Distributed PowerPoint*. Boston University, 2001.

[26] WALLACE, G., AND LI, K. Virtually shared displays and user input devices. In *USENIX Annual Technical Conference '07* (Santa Clara, CA, USA, June 2007), pp. 375–380.

[27] YUN, H. C., GUENTER, B. K., AND MERSEREAU, R. M. Lossless compression of computer generated animation frames. *ACM Trans. Graph. 16* (October 1997), 359–396.

[28] Zero configuration networking (zeroconf). http://www.zeroconf.org/.

[29] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337–343.