Kahawai: High-Quality Mobile Gaming Using GPU Offload

Eduardo Cuervo[†], Alec Wolman[†], Landon P. Cox[‡], Kiron Lebeck[*], Ali Razeen[‡], Stefan Saroiu[†], Madanlal Musuvathi[†]

[†]Microsoft Research, [‡]Duke University, [*]University of Washington

ABSTRACT

This paper presents Kahawai[1], a system that provides high-quality gaming on mobile devices, such as tablets and smartphones, by offloading a portion of the GPU computation to server-side infrastructure. In contrast with previous thin-client approaches that require a server-side GPU to render the entire content, Kahawai uses collaborative rendering to combine the output of a mobile GPU and a server-side GPU into the displayed output. Compared to a thin client, collaborative rendering requires significantly less network bandwidth between the mobile device and the server to achieve the same visual quality and, unlike a thin client, collaborative rendering supports disconnected operation, allowing a user to play offline – albeit with reduced visual quality.

Kahawai implements two separate techniques for collaborative rendering: (1) a mobile device can render each frame with reduced detail while a server sends a stream of per-frame differences to transform each frame into a high detail version, or (2) a mobile device can render a subset of the frames while a server provides the missing frames. Both techniques are compatible with the hardware-accelerated H.264 video decoders found on most modern mobile devices. We implemented a Kahawai prototype and integrated it with the idTech 4 open-source game engine, an advanced engine used by many commercial games. In our evaluation, we show that Kahawai can deliver gameplay at an acceptable frame rate, and achieve high visual quality using as little as one-sixth of the bandwidth of the conventional thin-client approach. Furthermore, a 50-person user study with our prototype shows that Kahawai can deliver the same gaming experience as a thin client under excellent network conditions.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*

Keywords

code offload; GPU; computer games; mobile devices

[1]Kahawai is the Hawaiian word for stream.

MobiSys'15, May 18–22, 2015, Florence, Italy.

ACM 978-1-4503-3494-5/15/05 ...$15.00.
http://dx.doi.org/10.1145/2742647.2742657 .

1. INTRODUCTION

Since the advent of consumer mobile devices equipped with high resolution touchscreens along with powerful CPUs and GPUs, gaming has been one of the most popular activities on smartphones and tablets. Recent studies estimate that 46% of mobile users play games, and out of the total time spent using their devices, users play games 49% of the time [15]. As mobile device screens grow larger and screen resolutions increase, finer graphical detail and advanced graphical effects are becoming more important for mobile applications, especially games.

To provide richer visual experiences, mobile devices have seen rapid improvements in their GPU processing capabilities, but today's devices cannot duplicate the sophisticated graphical detail provided by gaming consoles and high-end desktop GPUs. The primary reason for this performance gap is power consumption. A high-end desktop GPU may consume 500 Watts of power, whereas a high-end mobile GPU will consume less than 10 Watts. As a result, mobile GPUs will lag behind their desktop contemporaries for the foreseeable future. The battery capacity of mobile devices is limited and growing slowly, and high power consumption requires sophisticated and bulky thermal dissipation systems that are incompatible with mobile form factors.

Prior research efforts sought to close the performance and energy gaps between mobile devices and server-side infrastructure through remote execution and code offload [3, 7, 9, 14]. However, that work has focused on general purpose workloads running on a mobile device's CPU. Although there has been previous work on remote rendering [28, 37], we are unaware of any prior systems that support GPU offloading from mobile devices to server infrastructure.

Thin-client architectures are another widely explored area of previous work [24, 35], and commercial systems such as OnLive, Playstation Now, and Nvidia Shield take a thin-client approach to mobile gaming. Here, a cloud server with a powerful CPU and GPU execute the game and render its output. The mobile device forwards the user's input to a server, and receives the game's audio-visual output encoded as compressed video. Though popular, thin-client gaming has two drawbacks. First, transmitting game content that meets gamers' expectations with regards to screen resolution, frame rate, and video quality results in high bandwidth requirements. This is particularly worrisome for players connecting over data-capped connections. Second, thin-clients cannot support offline gaming since all game code executes on a remote server.

In this paper, we present *Kahawai*, a GPU offload system that overcomes the drawbacks of thin-client gaming. The main technique used by Kahawai is *collaborative rendering*. Collaborative rendering relies on a mobile GPU to generate low-fidelity output, which when combined with server-side GPU output allows a mobile device to display a high-fidelity result.

The key insight behind collaborative rendering is that while fine-grained details are prohibitively expensive for mobile GPUs to render at an acceptable frame rate, these details represent a small portion of the total information encoded within the displayed output. Thus, collaborative rendering relies on the mobile GPU to render low-fidelity content containing most of the displayed output, and relies on the server infrastructure to fill in the details.

Collaborative rendering in Kahawai addresses thin-client gaming's main shortcomings. First, when the mobile device is connected to a server, Kahawai provides high-quality gaming at significant network bandwidth savings relative to a thin-client. Second, disconnected Kahawai clients can play games offline using their own GPU, albeit with reduced visual quality. To enable collaborative rendering, Kahawai requires careful synchronization between two instances of the same game: one executing on the mobile device and the other on the server. To support this, we integrated partial deterministic replay into a popular open-source game engine, the idTech 4 game engine used by many commercial games.

We have developed two separate collaborative rendering techniques as part of Kahawai, with each targeting a different aspect of the game stream's fidelity (i.e., per-frame detail and frame rate). In our first technique, *delta encoding*, the mobile device produces reduced-fidelity output by generating every frame at a lower level of detail. The server-side GPU concurrently renders two versions of the game output: a high-fidelity, finely detailed version, and a low-fidelity version that matches the mobile device's output. The server uses these two outputs to calculate delta frames representing the visual differences between the high-fidelity and low-fidelity frames. The server then sends a compressed video stream of delta frames to the client, and the mobile device decompresses the stream and applies the deltas to the frames that it rendered locally.

In our second technique, *client-side I-frame rendering*, the mobile device produces reduced-fidelity output by rendering highly detailed frames at a lower rate. The server renders the missing frames at a higher rate, and sends the missing frames as compressed video. The mobile device decodes the video, combines the frames in the correct order, and displays the results. Both of our techniques are compatible with the hardware-accelerated H.264 video decoders built in to many of today's mobile devices.

Our Kahawai prototype is integrated into the idTech 4 game engine, and demonstrates the benefits of collaborative rendering using a commercial game, Doom 3, built on this engine. We show that compared with a thin client using H.264, delta encoding provides far superior visual quality when bandwidth is constrained to less than 0.5 Mbps. Even more impressive, we show that a thin client requires six times as much bandwidth as client-side I-frame rendering to achieve comparable visual quality. Finally, a 50-person user study demonstrates that Kahawai delivers a high-quality gaming experience equivalent to the thin client approach.

2. BACKGROUND

Kahawai relies on concepts and principles from the following related topics: cloud gaming [39, 25, 26]; game structure and performance [20, 12, 16]; video compression [34]; and image quality comparisons [38, 22]. In [8], we present extensive background on the above topics.

Many modern games are built on top of a *game engine*. These engines provide a general platform that greatly simplifies game development. Engines typically separate game-specific content, such as artwork, levels, characters, and weapons, from core functionality such as rendering, physics, sound, and input handling. Some of the most popular game engines for fast-action games are Unreal [12], Unity [36], idTech [20], and the MT Framework. We implemented Kahawai by modifying version 4 of the idTech engine. Fully integrating Kahawai into an engine requires access to the engine's source code, but this does not limit the generality of our approach. Even without access to source code, we successfully applied collaborative rendering to the game Street Fighter 4 [5] by intercepting calls to the custom Capcom engine on which it runs. When adopting a particular game engine, game developers typically obtain a license that includes access to game-engine source code.

In addition to access to the game-engine source code, delta encoding also requires a way to generate a high-detail and low-detail version of each output frame. Desktop PC games commonly provide a large number of settings that control the performance and visual quality of the rendered graphics. Game designers provide these settings so that players with more powerful GPUs can experience a higher-quality version of the game, while those with less powerful GPUs can play at an acceptable frame rate. Game settings control a wide variety of graphical details within the game, including lighting, shadows, fog, texture compression, bump mapping, anti-aliasing, anisotropic filtering, and even the complexity of the 3D models given to the rendering pipeline.

OpenGL [33] and Direct3D [30] are industry standard rendering APIs for programming modern GPUs. These frameworks describe how a given 3D scene should be rendered and abstract away the hardware details of a GPU's implementation. Hardware manufacturers implement support for these APIs in a way that best exploits the characteristics of their hardware. Each API implementation is provided as a binary device driver from the GPU manufacturer [4]. Kahawai does not require access to these driver internals.

It is important to note that given the exact same scene description, rendering APIs do *not* guarantee pixel-exact outputs for different GPUs. In fact, we observe pixel-level differences even for different GPUs produced by the same manufacturer and using the same device driver. These differences are usually imperceptible to the human eye, and in Section 6, we quantify the extent of these differences. We also demonstrate that these differences have a negligible effect on the quality of Kahawai's output.

Finally, to provide the appearance of consistent and smooth motion, most real-time rendering applications like games attempt to render between 30 and 60 frames per second. In practice it is hard to define the exact point when a frame rate is high enough to avoid flicker. However, even at high illumination perceived quality plateaus around 60 Hz [17].

3. Kahawai ARCHITECTURE

This section provides a high-level overview of the two collaborative rendering techniques we have developed for Kahawai.

3.1 Overview

Kahawai supports collaborative rendering by reducing client-side output fidelity along two dimensions: under *delta encoding* the mobile GPU renders low-detail frames at a high rate, while under *client-side I-Frame rendering* the mobile GPU renders high-detail frames at a low rate.

Figure 1 shows the architecture for delta encoding. The basic idea is to utilize game settings to generate two versions of the game output: a high-detail version and a low-detail version. Kahawai exploits graphical similarities between these two versions to compute a highly compressed video of the differences.

The mobile device uses its mobile GPU to render a low-detail stream. The server executes two instances of the game, one for each level of detail, and generates a delta frame by computing the pixel-by-pixel visual difference between its high-detail and low-detail frames. We use *partial deterministic replay*, described in Section 5,

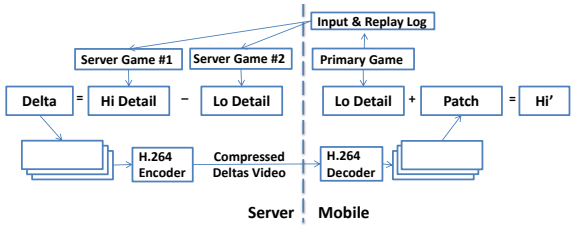


Figure 1: *Architecture of Delta Encoding. The server renders both high-detail and low-detail versions of each frame, calculates the visual difference into a delta frame, and then encodes an H.264 video of the delta frames. The mobile device decodes the video, and uses it to patch the low-detail frames rendered by the mobile GPU. The mobile device sends a log to the server to enable the server-side game instances to correctly follow the mobile client.*

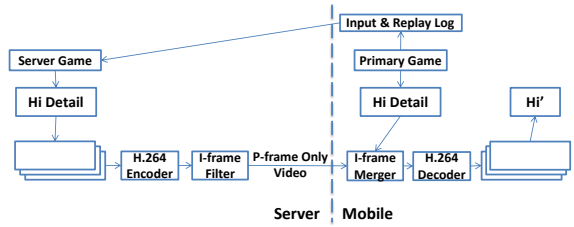


Figure 2: *Architecture of Client-Side I-Frame Rendering. The server renders a hi-detail version of each frame, and encodes the frames as H.264 video. A filter discards the I-Frames from the encoded video, then sends the video back to the client. The client uses the mobile GPU to render I-Frames, and merges them back into the video stream before invoking the H.264 decoder.*

to control concurrent instances of the game running on the server. The server sends a compressed H.264 video of delta frames to the client, which decompresses the video and applies each frame from the received video as a patch to each low-detail frame it renders with its mobile GPU. The end result is an image stream that almost perfectly matches the high-detail output rendered on the server.

Figure 2, shows the architecture for our second technique, client-side I-frame rendering. For this technique, the client generates high-detail game output at a low frame rate, and the server generates high-detail game output at a high rate.

Compressed video is composed of three types of frames: Intra-coded I-frames, Predicted P-frames, and Bi-directional Predicted B-frames. For our purposes, it is only necessary to understand the high-level differences between I-Frames and P-frames. In particular, an I-frame is relatively large and self-contained; an I-frame encodes all of the visual information needed to display its content. On the other hand, P-frames are smaller and contain references to prior frames in the stream. To display a P-frame, a machine must have the P-frame and any other frames it references.

Under client-side I-frame rendering, the server generates an H.264 video of its high-detail, high-rate game output, but drops the I-frames before sending the video to the client. The client receives the video containing only P-frames, recreates the missing I-frames by inserting output from its mobile GPU, and then uses its H.264 decoder to display the final, merged output. The end result, as with delta encoding, is graphical output that almost perfectly matches the high-detail, full frame-rate version rendered by the server GPU. It may be possible to combine these two techniques, but we leave this for future work.

3.2 Requirements

Both collaborative rendering techniques require servers to send H.264-compliant video to clients. H.264 is the de-facto standard for compressed mobile video, and nearly all modern mobile devices have dedicated hardware to accelerate H.264-encoding and -decoding. This hardware improves performance and energy consumption, and frees the mobile CPU and GPU to execute the game instead of encoding and decoding video.

Hardware-accelerated video decoding does not eliminate the need to move video through the memory system, but this overhead is modest compared to the total memory bandwidth of most devices. For example, decoding a 1080p HD video at 60 FPS only requires 400 MB/sec, whereas the memory bus bandwidth on a modern smartphone such as the iPhone 6 is 9 GB/sec.

Both techniques also require frame-by-frame synchronization. Specifically, because we run multiple instances of the game, each rendered object must be in the same position within a given frame for all executions. Low-detail versions may have missing objects found in a high-detail version, but the positions of objects common to both versions must match exactly. We describe in Section 5 how our implementation supports this requirement.

Delta encoding places two specific requirements on the game itself (as opposed to requirements on the game engine). First, the game must provide configuration settings that allow the mobile device to render a low-detail version of the game output at an acceptable frame rate. We view 30 FPS as the minimum acceptable frame rate, and 60 FPS as the maximum. Second, an H.264 video composed of delta frames must be smaller than an H.264 video of high-detail output. The size of the difference represents delta-encoding's bandwidth savings over standard thin-client gaming.

For client-side I-frame rendering, the mobile device must render high-detail frames fast enough and without impacting the game's responsiveness. For example, consider a mobile device that can generate four FPS of high-detail frames. Even if the server GPU is fast enough to generate the missing 56 FPS, the latency of generating each of those four frames on the client will be about 250 ms per frame. Studies of fast-action games have shown that users can tolerate end-to-end latencies on the order of 100 to 200 milliseconds [39, 25], and we estimate that a mobile device must generate at least six FPS of high-detail output for client-side I-frame rendering to be worthwhile.

4. COLLABORATIVE RENDERING

In this section, we describe the design details of Kahawai's two collaborative rendering techniques and characterize when they are most effective.

4.1 Delta encoding

Delta encoding relies on adjusting a game's settings to produce both high-detail and low-detail versions of the game's graphics output. The high-detail version should contain all of the game's visual effects at their maximum level of detail. The low-detail version should allow the mobile GPU to reach an acceptable frame rate, i.e., between 30 and 60 FPS.

The key observation underlying delta encoding is that the high-detail and the low-detail versions of most games' output share the same *visual structure*. That is, games frequently create high- and low-quality versions of a scene using the same logical objects (e.g., enemies, characters, and other items) in the same positions on the screen. Figure 3 shows the high- and low-quality versions of a scene from Street Fighter 4. Such outputs may differ in their level of detail, texture quality, or polygon count, but they convey most of the same information. As a result, encoding the difference between

Figure 3: *Same visual structure. Lo(Left) and Hi(Right)*

Figure 4: *Extra objects at high detail. Lo(Left) and Hi(Right)*

high- and low-quality versions of a frame frequently requires far fewer bits than encoding the high-quality frame itself.

Section 3 outlines the basic steps involved in delta encoding. Delta encoding requires less bandwidth than the conventional thin-client approach as long as the H.264 stream of delta frames is smaller than the H.264 stream of high-detail game output. In the common case, the bandwidth savings of delta encoding will be significant, although there are circumstances in which delta encoding will generate more traffic than a thin-client approach.

It is not obvious that H.264 will effectively compress a video of delta frames, but we have found that delta frames exhibit the same strong temporal locality as the original video frames. As long as this holds, H.264 will deliver high compression ratios for streams of delta frames.

H.264 also provides lossy compression and is designed to hide lost information from the human eye. Information loss is a result of the transformation and quantization (compression) processes in the encoder, which round individual pixel values to improve compression ratios. These losses introduce two challenges when transmitting a video of delta frames: (1) quantization of the delta values introduces visible visual artifacts when the delta is patched into low-detail frames on the mobile device, and (2) the encoded delta does not provide a pixel-exact representation of the difference between a high-detail and low-detail frame.

The challenge of visual artifacts is due to the extended value range that deltas encode. For each pixel, the delta represents the difference between the high-detail value and the low-detail value. The difference can be positive or negative, requiring an extra bit to encode the sign information. Without quantization, one could use modular arithmetic as a simple workaround. Consider an example: if the maximum channel value is 255, high = 10, and low = 253, then delta = -243, which means one could also represent the delta as +12.

However, quantization on the deltas introduces a problem. Consider another example: suppose high = 2 and low = 253, so that the delta = 4 with modular arithmetic. During quantization, the delta could be rounded from 4 to 2. When applying the patch later, low (253) + delta (2) = 255 which means that high is now 255. Quantization of this delta value essentially converts a very dark pixel value into a very bright pixel value, creating a visible artifact.

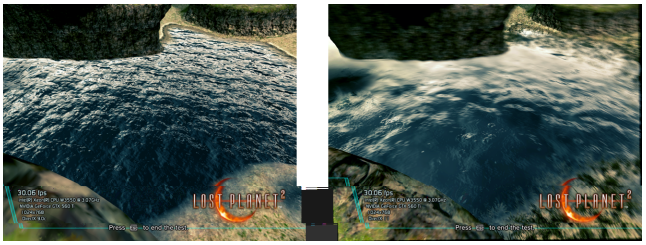


Figure 5: *Higher entropy at low quality. Lo(Left) and Hi(Right)*

We solve this problem by applying a solution proposed by Levoy [28]. Before invoking the H.264 encoder, we apply the transformation in Equation 1.

$$\Delta = \frac{Hi - Lo}{2} + 127 \tag{1}$$

This has the effect of dropping the least significant bit in favor of the sign bit. After H.264 decoding, we apply Equation 2 before applying the patch.

$$Hi = Min(2(\Delta - 127) + Lo, 255) \tag{2}$$

Unfortunately, deltas lose one bit of information as a result of this technique. However, this information loss is small and comparable to the loss normally induced by quantization noise in H.264.

The second challenge we faced with delta encoding is that encoded deltas do not perfectly capture the difference between high-detail and low-detail frames. This is primarily due to lossy compression and, to a lesser degree, to subtle pixel differences between frames generated by different GPU hardware and drivers. Loss due to compression is inversely proportional to how similar the high and lo definition frames are. A delta between two relatively similar frames will result in a smaller distribution of delta values, and will therefore suffer from less quantization noise than a delta from dissimilar frames. In our experiments, we have found that when high- and low-detail frames are sufficiently dissimilar, delta encoding can require more bandwidth or induce worse output quality than simply sending an H.264 encoding of the high-detail output.

To see why, consider the scenes in Figures 4 and 5. First, sometimes games introduce *extra* objects to the high-quality version of a scene. For example, in Figure 4 the high-quality version of a Lost Planet 2 scene includes extra clouds, blades of grass, and other background objects. When this happens, the delta is unlikely to be much smaller than the original high-quality frame.

Similarly, the high-quality version of a frame may be easier to compress than a delta due to how a game generates frames of different qualities. Figure 5 shows how Lost Planet 2 uses a more computationally demanding visual effect to generate smoothly waving water in the high-quality version of a scene, and creates water by statically applying a high-entropy texture to a surface in the low-quality version. Because the high-entropy, low-quality scene has more randomness than the smooth water surface of the high-quality scene, it is harder to compress. As a result, the delta between the high- and low-quality versions is also high entropy and does not compress as well as the original high-quality frame.

These scenes from for Lost Planet 2 show that delta encoding may not always be effective for all games. However, for games such as Doom 3 and Street Fighter 4 at any configuration, delta encoding can be a powerful way to reduce the amount of data sent from a cloud-gaming server to a client.

4.2 Client-side I-frame rendering

Our second collaborative rendering technique is client-side I-frame rendering. For this technique, the client uses its mobile GPU to render high-detail frames at a low rate, and the server renders high-detail frames at a high rate. The server compresses the high-detail frames into a video, replaces the I-frames in the video with empty place-holders, and sends the remaining P-frames to the mobile device. The mobile device then fills in the missing I-frames with the high-detail frames it renders locally.

The bandwidth savings of this technique compared to a thin-client is proportional to the rate at which a client can generate I-frames. This is partially because I-frames are relatively large and suppressing them saves bandwidth. However, more subtly, the faster a client can generate I-frames, the smaller server-generated P-frames become.

The reason P-frames are smaller under client-side I-frame rendering is that as quantization (compression) in H.264 increases, frames become less similar to their original sources. P-frames based on compressed reference frames must encode changes in the current frame as well as make up for loss accumulated over previously compressed frames.

If the encoder is configured to output a low-bandwidth stream, frames will be encoded using heavier quantization and will incur more information loss. This loss accumulates across P-frames until another I-frame arrives in the sequence. This phenomenon is similar to temporal error propagation [6], in which frames dropped from a streaming video affect contiguous frames in the sequence.

In a normal H.264 stream, loss accumulation cannot be solved by inserting I-frames more frequently. More frequent I-frames would lead to higher-quality P-frames under the same level of quantization, but the encoder would exceed its bandwidth budget because I-frames are usually much larger than P-frames. However, with Kahawai's client-side I-frame rendering there is no bandwidth penalty for increasing the frequency of I-frames in the stream because the client generates these frames locally. Thus, by increasing the I-frame rate, Kahawai reduces P-frame loss accumulation.

An additional benefit of client-side I-frame rendering is that client-rendered I-frames are perfect reference frames because they are not quantized. Thus, these I-frames are more similar to subsequent frames in the stream than they would be under a normal H.264 video. This allows an encoder to use fewer bits to generate P-frames referring to previous client-rendered I-frames at the same level of quality.

Finally, it is also possible to achieve a desired I-frame rate on the mobile device, at the expense of a moderate reduction in quality, by decreasing the game settings used to render high-detail frames. This allows the mobile device to render I-frames at a higher rate. If this is not enough to meet the delay and bandwidth constraints of the game, then we consider the game to be unsuitable for Kahawai's client-side I-frame rendering.

5. Kahawai IMPLEMENTATION

To evaluate collaborative rendering, we built two Kahawai prototypes. One is integrated with the open-source version of the idTech 4 Engine, which is also known as the Doom 3 engine. The idTech 4 engine is implemented in C++ and uses OpenGL for graphics. In addition to Doom 3, commercial games like Quake 4, Wolfenstein, Prey, and Brink have also been built on top of the idTech 4 engine.

Our second prototype uses binary interception to implement Kahawai on a closed-source game, Street Fighter 4. Street Fighter 4 is a fighting game with very different gameplay and design than Doom 3. Street Fighter 4 is built on top of the DirectX based Capcom engine that preceded the MT Framework engine used in games like Resident Evil 5, Devil May Cry 4, and Lost Planet 2.

We implemented deterministic graphical output, codec support, and full delta encoding for Street Fighter 4, but we were unable to implement features that required access to source code, such as the client-side portion of I-frame rendering and handling network disconnections. Without access to the game's source code, we could not prevent the client from rendering P-frames. Access to the engine's source would have allowed us to fully implement both collaborative rendering techniques and all other features of Kahawai.

In the rest of this section, we focus on three key challenges: (1) supporting deterministic graphical output, (2) handling input, and (3) implementing the codec support for our two collaborative rendering techniques.

5.1 Deterministic graphical output

Collaborative rendering requires two concurrently executing game instances to produce the same graphical output. For delta encoding, the client game's output must match the output of the low-detail game executing on the server. For client-side I-frame rendering, the client game's I-frames must match the server's I-frames. For game output to match, inputs originating on the mobile client must be deterministically replayed by server-side game instances.

Previous work has proposed combining deterministic replay with computational offload [14], but implementing replay at too low a level can cause high overhead, especially on multi-processor machines. Games are computationally demanding, and slowing down a game will lead to a poor user experience. On the other hand, entangling replay with a game's internal logic would be impractical.

Instead, Kahawai takes a middle ground by integrating *partial* deterministic replay with the idTech 4 engine and the Capcom Engine. The approach relies on the observation that Kahawai does not need to faithfully recreate all aspects of a game's execution as long as replays produce the same graphical output as the original. For example, collaborative rendering can work even if a server-side game instance produces different file writes and thread schedules than those in the original client execution.

We define the graphical output of a game as the sequence of OpenGL or DirectX calls it emits. For example, the idTech engine interacts with the OpenGL graphics subsystem within a single thread, which we call the *render thread*. At a high-level, the main loop for the render thread looks like:

```
while (1) {
    SampleInput();   // process input buffered during
                     // rendering of previous frame
    Frame();         // update the game state
    UpdateScreen();  // make calls to OpenGL, then
                     // glFinish() and SwapBuffer()
}
```

The call to *Frame* is used to update internal game state, and the call to *UpdateScreen* is used to render the new state on the screen. For example, for objects in motion, the game engine uses calls to *GetSystemTime* inside *Frame* to determine how much time has elapsed since the last loop iteration. This elapsed time is used to determine how far to move objects within the scene.

Our goal for partial deterministic replay is to ensure that, for a particular loop iteration rendering frame N, the sequence of OpenGL calls (and their parameters) made within UpdateScreen for frame N is identical across executions. To verify that partial replay executes correctly, we run a game multiple times using a specific input log and a specific configuration of the game settings. We then use *glReadBuffer* to grab the rendered output of each frame for each execution and confirm that there are no pixel differences. Note that we run these tests on the *same* GPU and driver.

Interestingly, Street Fighter 4 appears to rely solely on an internal logical clock. This is evidenced by the fact that the game-play responsiveness scales proportionally with the frame rate, and that user actions (e.g., punching and kicking) take a constant number of frames to execute regardless of frame rate. As a result, Street Fighter 4 only required intercepting requests for the system time and the DirectX equivalent of SwapBuffers (Present) using Detours [19]. Unlike Street Fighter 4, Doom 3 and the idTech engine presented more sources of non-deterministic behavior that required access to the source code to address.

There are three sources of non-determinism that can affect the visual output of the idTech engine's rendering thread. The first source is system time: Kahawai intercepts all calls to *GetSystemTime* from the render and audio threads and ensures that replayed executions receive the same time values. The second source of non-determinism is the pseudo random number generator. Interestingly, we did not need to intercept these calls because correctly replaying keyboard and mouse inputs ensures that random-number generation is deterministic. The last source of non-determinism is the game's music. The rendering thread changes a scene's illumination by directly reading data from the sound buffer. For example, when a game's music gets louder, lights in a game may become brighter. Rather than attempting to make the audio thread deterministic, Kahawai includes a configuration flag to tell the engine to use system time instead of reading from the sound buffer. As far as we can tell, this change did not impact gameplay in any other way.

Finally, our current Kahawai prototype handles mid-game disconnections from the server by falling back to low-detail rendering using only the mobile GPU. When the network allows the mobile device to reconnect to the server, we leverage the idTech 4 game engine's support for game-state checkpoint and restore (many other modern game engines also support game checkpoint and restore). The idTech *save game* mechanism allows players to persist the current game state and to resume at a later time. When the client reconnects, it performs the following steps: (1) it executes until the game can be saved; (2) it pauses execution; (3) it sends the saved state to the server; and (4) it resumes execution after the server confirms it has finished restoring the game state. From that point on, offloaded execution behaves normally with the client and server synchronized. In the future, we plan to implement a background restore mechanism that does not pause game play.

5.2 Input handling

The key challenge for handling game inputs can be seen by looking at the rendering loop shown in the previous section. The thread executing this loop sequentially processes buffered inputs, updates the game state, and renders the scene. While the loop is executing the *Frame* and *UpdateScreen* calls for frame N, the system is buffering input events for frame N+1. To keep the client and server games in sync, Kahawai must ensure that when the client processes an input event at frame N, the server must also process that event at the exact same frame number N. To accomplish this, the client's render thread timestamps inputs with the appropriate frame number before sending them to the server, and the server pauses until it receives inputs for its next frame.

A naive approach to input handling in Kahawai would leave the input sampling as part of the rendering loop, but this creates performance problems. Suppose the one-way delay from a client to a server is 20ms. After the client finishes rendering frame N, it must send all buffered inputs for frame N+1 to the server. The server cannot render the next frame N+1 until at least 20ms after the client has finished rendering frame N and sampled the input for frame N+1. As a result, the game's frame rate becomes inversely proportional

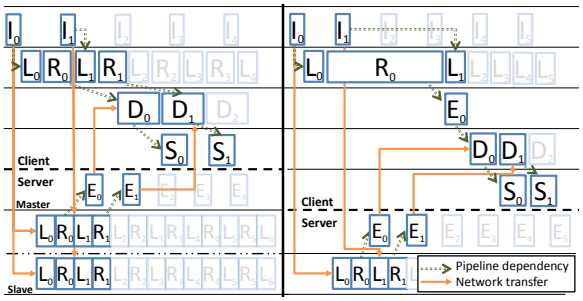


Figure 6: *Pipeline overview and network dependencies for both Delta (left) and I-frame (right). The subscripts indicate the frame number. Green arrows indicate local pipeline dependencies, and purple arrows show network pipeline dependencies.*

to the network latency between the client and server, which is unacceptable for game play.

Kahawai solves this problem with pipelining by decoupling input processing from the main game loop. In fact many of Kahawai's tasks are run asynchronously in a pipelined fashion. This pipeline allows Kahawai to asynchronously process and display frames rather than waiting on the network to make progress. The next section describes the Kahawai input and output pipelines in greater detail.

5.2.1 Pipeline summary

Kahawai executes tasks in parallel whenever possible to reduce their impact on frame rate and input-to-display latency (i.e., the time from when a user generates an input to when the result of the input appears on the screen). Figure 6 shows the pipeline for both collaborative rendering techniques. The main tasks of rendering, encoding, decoding, and showing form the stages of a pipeline. Each stage executes in a different thread and depends on the output of the previous stage. Parallelism emerges from overlapping the stages operating on data from different frames. In Kahawai we have the following asynchronous stages:

- Input (I): In this brief stage, Kahawai queries the keyboard, mouse, and joystick for user input, and queues the results.
- Logic (L): In this stage the CPU executes the game logic using a set of inputs obtained from the input queue, and updates the game state.
- Render (R): In this stage the GPU generates frames using the updated game state. In addition Kahawai captures the rendered output to memory. Client-side I-frame rendering skips the render stage for P-frames on the client.
- Decode (D): This stage transforms an H.264 stream into decoded frames. For delta encoding, the decode stage also requires the incoming delta to be decoded and then applied to the rendered frames.
- Encode (E): For delta encoding, this stage is only performed on the server. For client-side I-frame rendering, the client encodes a locally rendered I-frame by packing the full I-frame into a lossless (I-PCM) H.264 frame through a very cheap transformation.
- Show (S): In the client, this stage presents the game content to the user at the desired frame rate.

The cost of each individual stage is highly variable, depending on the logical (e.g. physics) and graphical (e.g. polygon count) complexity of the frame being processed.

5.2.2 *Input sampling interval*

In a pipelined implementation of Kahawai, sampling the input is the first stage. Figuring out the correct sample rate is critical to overall game performance. If Kahawai samples inputs too often, input-to-display latency will increase and gameplay will suffer. This is because the sampled inputs are placed on a queue, and the thread that executes the logic and render stages dequeues an entry when it has finished its work on the previous frame. If the inputs are being generated faster than the rate of the bottleneck stage in the pipeline, then the length of the input queue will grow and so will the input-to-display latency. If Kahawai samples too infrequently, pipeline throughput will suffer. This is because when the input queue is empty, the logic and render thread will block waiting for the input sampling thread to enqueue a new sampled input.

To ensure that Kahawai samples inputs correctly, we designed an adaptive clocking mechanism that determines how often the input thread samples buffered inputs. Adaptive clocking is needed because network delay and the time to complete each stage in the pipeline are highly variable.

Client-side I-frame creates another complication. Here, the set of stages that make up the pipeline varies depending on the frame type. For each I-frame, there are additional Render and Encode stages on the client that are not required for P-frames. In particular, for I-frames the Render stage will often be the bottleneck stage of the pipeline. Thus, Kahawai must compute the average pipeline performance over a sequence of frames that includes at least one I-frame and the set of dependent P-frames that follow it.

In a pipelined and multi-threaded system with predictable performance, system throughput should be inversely proportional to the time to complete the slowest stage in the pipeline. Our goal for the adaptive clock is to estimate this value over a time window to improve user experience. We found that in practice too short of a time window leads to burstiness that creates jittery gameplay, and too long of a window artificially limits system throughput. Our current implementation uses a window of 50 frames for Delta, and 10 frames for I-frame.

There are three additional complexities that affect our calculation of the adaptive clock. First, part of the time executing the Logic pipeline stage is spent blocking waiting for input from the input queue. Any time spent blocking waiting for input must be subtracted from our estimate of the cost of the Logic stage. Second, we must adjust the clock to compensate for the one-way network delay between the client and server. This affects the system throughput – if input is not sampled early enough, the server will block waiting for input and that will in turn cause the client stages that depend on server output to be delayed. Finally, to compensate for inaccurate timing measurements, we monitor the length of queued inputs, and slow the adaptive clock if the number of queued inputs exceeds a constant threshold.

5.3 Codec support

Both collaborative rendering techniques require capturing GPU output on a frame-by-frame basis. Once we have captured frames from the GPU or received frames over the network, we send them to our collaborative video codec. For delta encoding, two simultaneous instances on the server intercept frames, and share the frames through memory mapped files. The server computes a delta from the two frames, transforms it as described in Section 4.1, and then encodes it using x264 [40] as our H.264 encoder. On the mobile client side, we use ffmpeg [13] as our H.264 decoder, apply the correction as described in Section 4.1 and then apply patches to captured frames from the mobile GPU.

We experimented with the Intel QuickSync [21] hardware H.264 decoder on our tablet, and found that even though it reduced CPU load, it also imposed a substantial latency penalty over using a software decoder. We use software for H.264 encoding as well, due to the lack of flexibility of the hardware encoding APIs to produce streams like those needed by I-frame rendering in which every nth frame is lossless. We expect these APIs will improve over time as real-time hardware encoding and decoding becomes more pervasive. The use of software encoding and decoding limited the resolutions we could support to 720p, hardware support should easily allow full HD (1080p).

For client-side I-frame rendering, we configure x264 to generate I-frames at predictable intervals. We then filter the encoded video to discard the I-frames, while sending the compressed P-frames along with placeholders for the missing I-frames. On the mobile side, we use frame capture to extract the locally rendered I-frames and insert them into the video stream, and finally we send the video to ffmpeg for decoding.

In total we only had to modify 70 lines out of the 600K lines of the idTech 4 engine to enable the functionality of Kahawai. The rest of the Kahawai implementation is in a separate module that we link with the game engine. These modifications introduce determinism, allow the engine to skip the rendering of P-frames, and add mid-game disconnection support. Because these changes are done only to the engine, and not to the actual Doom 3 game, then any other game that runs on idTech 4, such as the upcoming Quadrilateral Cowboy, will work with Kahawai without further modification.

To support Street Fighter 4, where we did not have source code, we created a Detours hook and linked it to the original game binary and our Kahawai module. Implementing this hook required 450 lines of code. It ensures determinism and intercepts the DirectX rendering call *Present*.

6. EVALUATION

To evaluate Kahawai, we explore the following questions:

- How does collaborative rendering affect users' gaming experience compared to a thin-client approach?
- How does the bandwidth, input-to-output latency, and visual quality of the collaborative rendering approach compare to a thin-client approach?
- How do variations in GPU and driver impact image quality?

To answer the first question, we conducted a user study with 50 participants and our Kahawai prototype. To answer the next two questions, we measured the performance of our Kahawai prototype under trace-driven, emulated gameplay.

6.1 User study

To better understand how collaborative rendering affects users' gaming experience, we conducted a user study in which 50 participants were asked to play a small portion of Doom 3 under one of seven system configurations: unmodified Doom 3, thin client (under low and high network latency), delta encoding (under low and high network latency), and client-side I-frame rendering (under low and high network latency). In our results, we refer to the configuration where participants played unmodified Doom on a PC with a high-end GPU as the thick-client configuration, or "thick". The study was performed between December 1st and 3rd, 2014 in an office inside Microsoft's building 112 in Redmond, Washington.

6.1.1 Study design

We asked each participant to complete a portion of the Doom 3 "Central Processing" level. Players begin in a hallway with one monster, continue through a lava-filled room, and end in a room full of monsters. Prior to playing the game, players were given detailed instructions about their goals and the game controls. Each player began the task with full armor and health, as well as a fully loaded plasma gun and a flashlight. All other weapons were disabled.

Participants controlled their on-screen avatars via keyboard and mouse inputs. For example, players could change their avatar's view, make their avatar move, and shoot their avatar's gun. The game screen displayed an avatar's armor, health, stamina, clip-ammunition, and reserve-ammunition levels. After each kill we displayed the number of monsters killed and the number remaining.

Each participant's goal was to kill all 12 monsters, and after completing the task we recorded their remaining health, armor, ammunition, and how long they played. The game stopped as soon as a player has killed all 12 monsters. If a player died before they completed the task, we recorded how many monsters they killed and how long they survived. Players lost health primarily due to attacks by monsters.

Note that players who fell into the lava died instantly, but we allowed them to restart the game once. We included only their post-fall data in our results. Only three of our 50 participants fell into the lava, and interestingly all were self-described gaming experts who often play first-person-shooter (FPS) games. One of these players even brought their own mouse to the study. These participants admitted that because of over-confidence they initially skipped our instructions and rushed straight into the lava.

The tablet used by participants in our thin-client, delta, and I-frame configurations was a Microsoft Surface Pro 3 with a dual core Intel i5-4300U CPU, 8GB of RAM, and 256GB of SSD storage connected by a LAN to a Dell XPS 8700 PC with a quad-core 3.2Ghz Intel i7-4770 CPU, 24GB of RAM, and an Nvidia GTX 780 Ti GPU. Thick-client participants played directly on the PC. Both machines were plugged into identical Dell 24-inch monitors with external mice and keyboards. This prevented participants from knowing which device they were using. For all configurations the game resolution was set to 720p (1280 x 720).

For low-latency configurations, we used an unmodified LAN with approximately 2 ms of round-trip latency between the tablet and server, which corresponds to a typical round-trip time (RTT) over a home LAN. For high-latency configurations, we introduced a 70 ms RTT which corresponds real-world observations over an LTE connection [18].

We asked all participants to complete two surveys (pre-study and post-study). The pre-study survey contained a consent form, asked for basic demographic information, and assessed their interest in and use of technology and gaming. Participants were asked to rate their gaming expertise on a scale of one ("Novice") to three ("Beginner") to four ("Competent") to five ("Proficient") to seven ("Expert"). We used these expertise ratings to create similar player pools for each of our seven system configurations. This survey also asked participants to select their level of agreement with five statements on a scale from one ("Strongly disagree") to seven ("Strongly agree"), including *"I consider myself a gamer."* and *"I often play first-person shooter games (e.g., Halo)."* The pre-study survey also included the following three questions: *"Please list any games that you play regularly."*, *"Please list your all-time favorite games (if any)."*, *"Have you ever played Doom 3?"*.

Participants completed the post-study survey after completing the game. This survey was designed to gauge participants' qualitative evaluation of their game-play experience. It consisted of four statements, and participants were asked to rate their level of agreement with those statements on a scale from one ("Strongly disagree") to seven ("Strongly agree"). In particular, we wanted to know whether the game was fun (i.e., *"The level of Doom 3 I just played was fun."*), was easy, looked good, and ran smoothly.

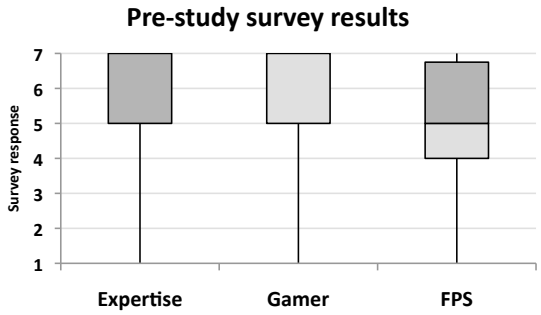


Figure 7: *Results from the pre-study survey in which participants rated themselves on gaming expertise and their agreement with statements "I consider myself a gamer" and "I often play first-person-shooter (FPS) games". For "Expertise", the max and 75th percentile responses were both seven, and the 50th and 25th percentile responses were five. For "Gamer", the max, 75th, and 50th percentile responses were all seven.*

6.1.2 Recruitment and training

After receiving study approval from the Microsoft Research Global Privacy Manager, we recruited potential participants through several gaming-related and general-purpose mailing lists (e.g., Microsoft Computer Gamers, MS Fighting Gamers, MSR Redmond, and Mexicans in Redmond). Our solicitation instructed employees to complete the pre-study survey and promised a $5 meal card as an incentive for participation.

We recruited 67 employees who completed the pre-study survey. We did not tell participants about our research or which system configuration they were assigned. After cancellations, we ended up with 50 of the selected respondents who participated in the study in early December of 2014.

From the pre-study survey, our 50 participants spend an average of two hours per day playing video games, 10 hours per day on a PC, and three hours per day using a smartphone or tablet. 38 participants frequently played games on their PC, and the most commonly owned gaming consoles were the Xbox 360 (owned by 30 participants) and the Xbox One (owned by 27 participants).

Figure 7 shows box plots summarizing answers to three of our pre-study survey questions. For all box plots in the user study, the top and bottom edges of each dark-gray box represent the 75th and 50th percentile responses, whereas the top and bottom edges of each light-gray box represent the 50th and 25th percentile responses. The top and bottom whiskers extend to the max and min responses. Nearly 40% of our participants rated themselves as a gaming "Expert" (i.e., seven, the highest possible response), though the median expertise rating was "Proficient" (i.e., five). Over 50% of participants strongly agreed with the statement that they were a gamer, and over 25% strongly agreed with the statement that they often played FPS games. Overall, the pre-study survey results indicate that our participant pool was both enthusiastic about and familiar with computer games, which was not surprising given the

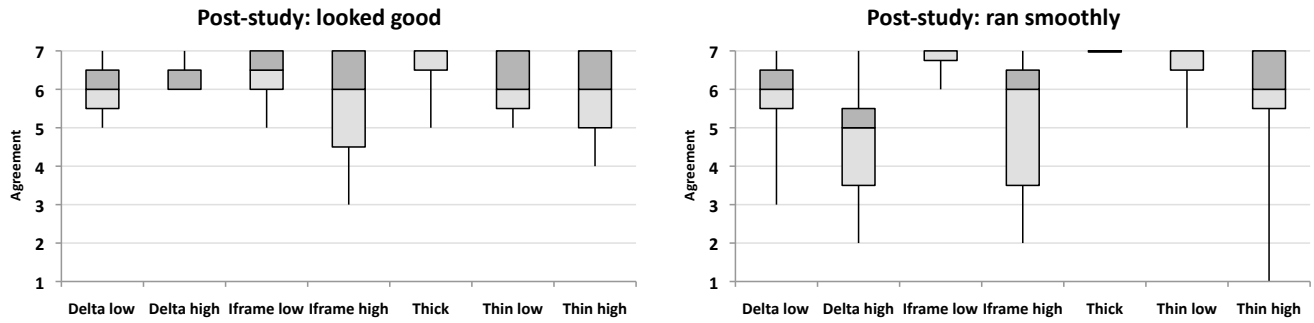


Figure 8: *Results from the post-study survey rating participants agreement with statements that the game looked good and ran smoothly. A one rating corresponds to strong disagreement and seven corresponds to strong agreement. For "thick", all thick-client participants strongly agreed that the game ran smoothly.*

email lists we used to recruit participants. Finally, only five of the 50 participants were female.

6.1.3 Results and analysis

Of the 50 participants, eight played under the I-frame configuration with low latency, while every other configuration had seven players. As mentioned previously, we used the expertise ratings from our pre-study survey to create player groups for each of our seven system configurations that were as similar as possible. Despite our efforts, we could not create groups with identical expertise profiles. However, the delta (high latency), I-frame (low latency), thick-client, thin-client (low latency), and thin-client (high latency) groups all had a median expertise of "Proficient." The delta (low latency) group had a median expertise of "Expert," and I-frame (high latency) group had a median expertise of "Competent."

Our hypothesis at the onset of the study was that collaborative rendering (i.e., delta and I-frame) would provide the equivalent of a thin-client gaming experience under the same network latency. We also suspected that network latency would degrade the gaming experiences experience (i.e., that thick client with zero network latency would provide the best experience and that offloading systems under high latency would provide the worst). We measured gaming experience qualitatively using post-study survey responses and quantitatively using players' game performances.

Perceived quality

Figure 8 shows the results of our post-study surveys, broken down by system configuration. There was strong agreement across all system configurations that Doom 3 "looked good." Not surprisingly, participants who played the thick-client configuration had the highest median response with seven. I-frame with low latency had a median response of 6.5, and all other configurations had median responses of six. These responses indicate that collaborative rendering delivers a gaming experience that users find to look as good as a thin-client approach.

There was more variation in participants' responses to how smoothly the game ran. Thick client, I-frame under low latency, and thin client under low latency all had median responses of seven. Responses for both I-frame and thin client under high latency dropped to six. Delta players provided a median response of six under low latency and a median response of five under high latency. Overall, higher latency appears to have led to worse perceived smoothness. However, both collaborative rendering configurations provided comparable smoothness to thin client: I-frame performed on par with thin client, and delta performed only slightly worse.

Player performance

We were also interested in metrics that measured how collaborative rendering affects participants ability to play the game. Even if collaborative rendering is visually equivalent to a thin-client approach, it should not make games harder to play. To see if system configuration had any noticeable impact on game difficulty, we looked at the time players took to complete their task and their health at the end of the game. Figure 9 shows these results for each configuration.

The median completion time for all configurations was between 58 seconds (I-frame with low latency) and 78 seconds (delta with low latency). For I-frame and thin client, lower latency led to faster completion times. Completion times for both delta configurations were higher than the low-latency I-frame and thin-client configurations. However, overall there is no obvious difference between the completion times on collaborative-rendering configurations and thin-client configurations. Our intuition is that while latency did have a small effect in performance, the actual rendering technique did not. Instead the performance variations were likely caused by individual skills variations among people reporting the same level of expertise, lack of custom input device / keyboard configuration, time of day and luck among others. This can explains why 'thick', our baseline, did not get the lowest completion times.

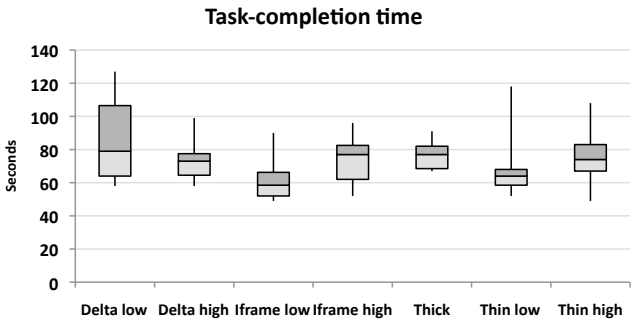
The median health for all configurations was between 78 (thin client with high latency) and 42 (I-frame with high latency). For final health, there appears to be little correlation between health and latency. For delta and thin client, median health suffered when latency was lower, whereas for I-frame, median health improved when latency was lower. Furthermore, thick client had a worse median health (56) than all offloading configurations except I-frame with high latency. Similar to completion time, other factors appear to have influenced health other than latency and overall, system configuration did not appear to have any obvious impact on a player's final health.

In summary, our user study indicates that collaborative rendering does not make the game harder to play than a thin-client approach.

Public Deployment. We deployed Kahawai publicly at MobiSys 2014 demo session [10]. Around thirty participants played the system. The response was positive.

6.2 Trace-driven experiments

Our user study indicates that collaborative rendering provides the same gaming experience as a thin-client approach. In this section, we use trace-driven inputs to our Kahawai prototype to characterize the bandwidth consumption, input-to-display latency, and visual



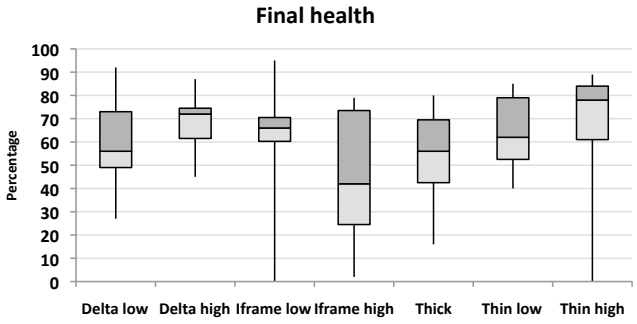


Figure 9: *We recorded the time each participant took to complete a portion of Doom 3 and their health levels after they completed the task.*

Mean Opinion Score	SSIM Score	SSIM (dB) Score
1 (Bad)	0 to .5	0 to 3
2 (Poor)	.5 to .86	3 to 8.53
3 (Fair)	.86 to .9	8.53 to 10
4 (Good)	.9 to .97	10 to 15.22
5 (Excellent, Identical)	.98 to 1	Above 15.23

Table 1: *Comparison between mean opinion scores and values in the SSIM scale.*

quality of the collaborative rendering approach, and we use those same metrics to compare directly with a thin client approach.

6.2.1 Methodology

We used the same client and server in our user study and trace-driven experiments. For consistent output in our video quality experiments, we generated Doom 3 sequences using idTech 4's demo recording feature. For our client-side I-frame experiments, our mobile client generates I-frames at 12 FPS.

To assess Kahawai's image quality. we use structural similarity (SSIM) [38] to measure image quality. SSIM allows us to determine how similar a frame generated by Kahawai is to the highest quality version of that frame. Unlike other image quality metrics like peak signal-to-noise ratio (PSNR) and mean-squared error (MSE), SSIM was designed to model the human eye's perception [23]. A raw SSIM score of greater than 0.9 indicates a strong level of similarity between two images, matching a PSNR of above 50 according to Hore [1]. An SSIM score of one indicates no difference between two images. Table 1 shows the SSIM scale ranges.

The workload we use to evaluate our Kahawai prototype consists of three different scenes in the Doom 3 game. We use the "Demo1" script, consisting of a player moving through a dimly lit environment (*Demo1*), a script of a player moving slowly through a brightly lit and subtly textured environment (*light room*), and a battle scene against the final boss featuring varied lighting conditions and a complex composition (*CyberDemon*). We chose these scenes to expose Kahawai to a range of graphical content. Each scene contains 500 frames at 720p. We used the pre-recorded demo sequences to test video quality, and we played emulating those sequences to test the performance and input latency of our prototype.

For our Doom 3 experiments, we compare three approaches to mobile gaming: delta-encoding (*Delta*), client-side I-frame rendering (*I-Frame*), H.264 streaming (*Thin-client*), and running the game locally in the server (*Thick-client*).

We configure the idTech 4 game engine using two graphics configurations. For our high-detail settings, we used the "Ultra" quality

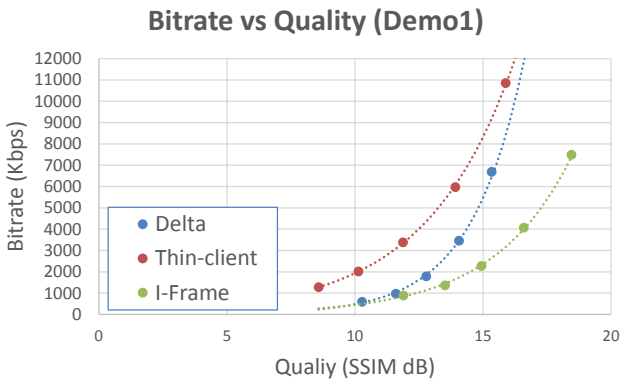


Figure 10: *Bitrate vs quality for the* Demo1 *scene.*

preset, and for our low quality settings we use a modified version of the "Low" quality preset, with bump mapping disabled. X264, our H.264 video encoder, can be configured using a wide range of parameters. We choose a single typical configuration for our experiments, based on the medium preset and the zero latency tune. For our latency and performance evaluation we used the faster preset to compensate for the current lack of low latency hardware encoding support and added 4x multisampling to the highest definition settings to stress the client's hardware. Our client can render locally an average of 15 FPS under this configuration, making it unplayable.

We only encoded video without audio and compared the size of the video streams alone. As we mentioned in Section 5, for Delta and I-Frame we don't need to send the audio stream. The client handles sound locally. In a real world deployment, adding sound would require 128 Kbps or more for thin client on top of the bitrate used for the video stream.

6.2.2 Doom 3 experiments

Given this setup, we sought to answer the following questions:

- For each of Kahawai's collaborative-rendering techniques, how do the bandwidth requirements and visual quality compare to a thin-client approach?
- For delta encoding, how does increasing the client GPU power, as represented by improving the quality of rendering, affect the bandwidth requirements?
- For client-side I-Frame rendering, how does increasing the client GPU power, as represented by increasing the FPS of I-frames, affect the bandwidth requirements?

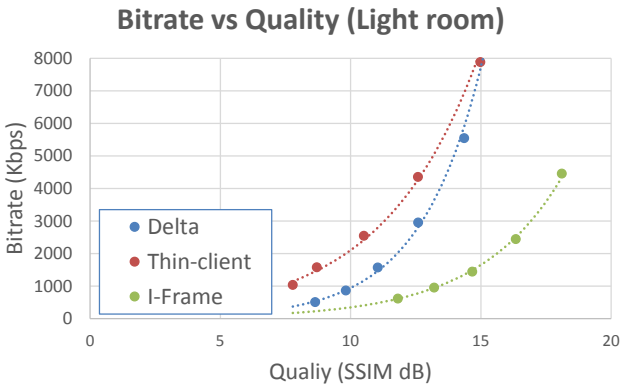


Figure 11: *Bitrate vs quality for the* light room *scene.*

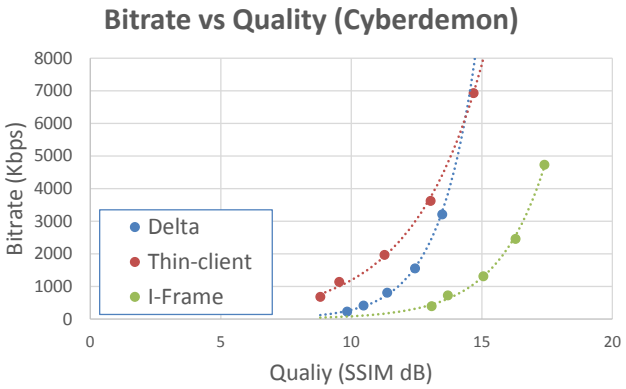


Figure 12: *Bitrate vs quality for the* CyberDemon *scene.*

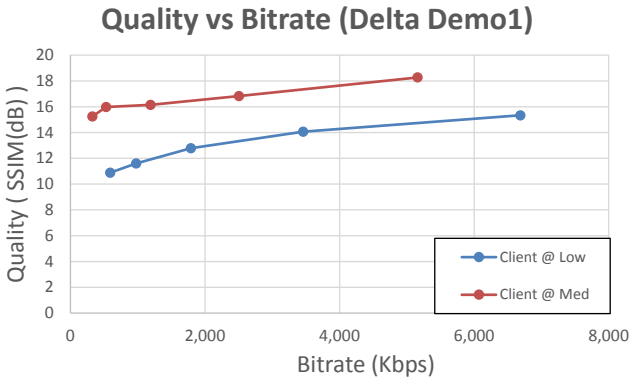


Figure 13: *Quality versus Bitrate for different quality fidelities.*

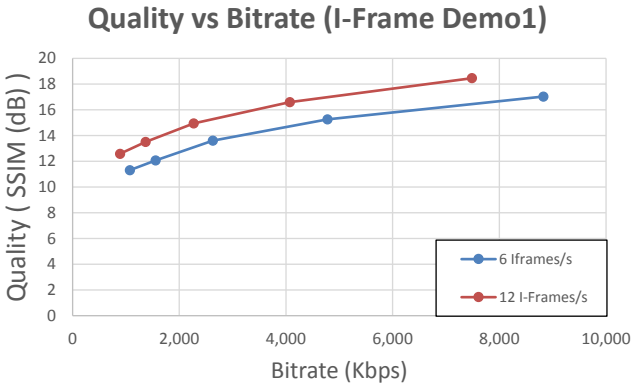


Figure 14: *Quality versus Bitrate for different temporal fidelities.*

- Can Kahawai's collaborative-rendering techniques provide full framerate while keeping latency similar to that of the thin-client approach.

To answer these questions, we play each of our scenes using three approaches (deltas, I-frames, and thin-client) while varying the H.264 constant rate-factor (CRF). Specifying a particular CRF value sets the target quality for a video stream, which the encoder tries to achieve using the lowest possible bitrate (i.e., the required bandwidth). Lower CRF values correlate with higher target quality, and higher CRF values correlate with lower target quality. For our experiments, we use CRF values of 20 (highest target quality), 25, 30 (middle high target quality), 35 (middle low) and 40 (lowest target quality). We use the SSIM (dB) logarithmic scale to measure the actual quality of each frame by comparing the frame the client displays with the lossless original frame that the server generates. See Table 1 for a summary of how to interpret SSIM (dB) scores.

To achieve a target quality, the encoder reduces the required bandwidth by opportunistically dropping information from fast-moving frames. The idea is that when frames change quickly, losing information will be less noticeable to the human eye. The higher the CRF, the more aggressively the encoder will drop information. Thus, CRF impacts both the measured quality of each frame (i.e., SSIM) and the size of the video stream (i.e, required bandwidth). This motivates our choice to use multiple CRF values when evaluating each technique.

Figures 10, 11 and 12 show our bitrate and SSIM results for each technique in the demo1, light room and CyberDemon scenes, respectively. Each line represents a different technique, and each data point on each line represents a different CRF value (from right to left, the points correspond to CRF values of 20, 25, 30, 35 and 40). It is important to notice that using a higher CRF does not mean that any stream encoded with it will have a lower bitrate, it just means that the compression factor is higher. For example, in Figure 12, the bitrate of the Delta stream with a CRF of 20 is lower than the bitrate for the thin-client with a CRF of 25.

Our results are impressive, especially for client-side I-frame rendering. For the Demo1 scene, client-side I-frame rendering can achieve a high video quality, averaging a good SSIM (dB) score of 11.894 using under 900 Kbps (CRF of 40), and an excellent score of 16.593 using only 4 Mbps. In comparison, the thin client approach needs almost three times the bandwidth (around 6 Mbps more) to achieve a similar, but lower score of 15.88 SSIM (dB). Similarly, the CyberDemon scene shows that client-side I-frame rendering can achieve similar quality to the thin-client approach using less than one-fifth of the bandwidth, looking at the data points for a CRF of 30 client-side I-frame rendering versus a CRF of 20 for the thin-client. Finally, the light room scene behaves similarly to the other two scenes, but shows even higher savings due to the strong temporal locality present in the slow moving scene. Here, client-side I-frame rendering achieves similar quality (14.67) using slightly over one-sixth of the bandwidth used by thin-client (14.98).

Delta encoding thrives at the lower end of the bitrate spectrum, providing much better quality than the thin-client approach when bandwidth is constrained. For example, it can be seen in Demo1 that we can achieve good video quality (SSIM of 10.31) using only

Scene	Metric	AMD 6450		Intel HD 3000		Nvidia 8200		H.264 CRF=25	
High Detail	% of changed pixels / stdev	34.1	4.9	32.9	4.6	0.2	0.2	91.2	2.7
	SSIM (dB) / stdev	26.0	2.2	25.8	2.3	47.0	4.7	12.6	1.6
Low Detail	% of changed pixels / stdev	40.3	4.6	38.0	4.2	0.1	0.1	92.2	2.8
	SSIM (dB) / stdev	27.3	0.5	27.7	0.5	47.0	5.1	16.0	1.8

Table 2: *Pixel and Quality Differences Between GPU Types.*

590 Kbps. Similarly, in the other two scenes, still produces good quality video for less than 1 Mbps.

It is important to notice that as the bitrate increases, the savings that delta encoding provides over thin-client decrease. For a very high bitrate, thin-client should actually outperform delta encoding, due to the information that is lost during delta encoding (Section 4.1). However, because the purpose of delta encoding is to work at low bitrates, this should not be cause for concern.

We show the impact of the client's output fidelity in the resulting bitrate. For delta encoding, the client's fidelity is measured as how much detail can the client include in its "'low"' quality setting. Figure 13 shows a comparison using delta encoding on the Demo1 scene. One line shows the client rendering the game using our modified low settings *Delta-Low*, while the other shows the client rendering the game using the built-in medium settings. Our results show that generating a delta from two scenes that are more similar to each other will result in more compressible deltas. In this scenario, delta encoding can achieve an excellent video quality (an average SSIM (dB) of 15.986) using as little as 528 Kbps, placing it slightly above the operating range of widely adopted high-quality music streaming services.

Increasing the client's temporal fidelity (the number of I-frames rendered per second), when using client-side I-frame rendering also reduces the required bitrate. Figure 14 shows a comparison using client-side I-frame rendering on the demo1 scene. One line shows the client rendering at 12 I-frames per second, as we use in all our other experiments, while the other line shows the client rendering only 6 I-frames per second. In this scenario, I-Frame with a client rendering 12 FPS settings can achieve an excellent SSIM (dB) of 16.59 using only 4 Mbps while it takes almost 9 Mbps to do it with a client rendering at 6 FPS.

6.3 GPU and driver variation

Rendering APIs such as OpenGL do *not* guarantee pixel-exact outputs even when the input scene descriptions are identical. In Table 2, we characterize the extent of this effect in terms of both the pixel differences and the image quality as measured by SSIM (dB). For this experiment, we render a scene on our server using the Nvidia GTX 580 GPU. We then compare the rendered scene on a frame-by-frame basis with the same scene rendered on three other GPUs: the AMD 6450, the Intel HD3000, and the Nvidia 8200 GTS. We perform this experiment twice: once using the scene rendered at high detail, and again using the scene rendered at low detail. We also compare the original scene with an H.264 compressed version using a CRF of 25.

Table 2 shows the results of this experiment. We see that the absolute percentage of pixels that are different is quite large: GPUs from other manufacturers show pixel differences of 33% to 40%, whereas a different GPU model from the same manufacturer shows very small pixel differences of .11% to .22%. Even across manufacturers, where the absolute percentage of changed pixels is large, the impact of those pixel differences on image quality is extremely small. We see that the SSIM (dB) values for the GPU from the same manufacturer are above 45, and for GPUs from different manufacturers are still above 25. Comparing these results to the effects of H.264 compression on image quality, we see that the impact of GPU and driver variation on image quality appears to be substantially less than the effects of H.264 compression on image quality.

6.4 Latency and performance tests

A major concern with offloading rendering to a server is the impact on latency and throughput, and most importantly how that affects gameplay. The key measure for throughput is FPS, and for latency is the time from when a user performs an input to when the effect of that input becomes visible. To determine whether Kahawai provides good frame rates with modest latency overhead above either the network RTT (for delta encoding) or the time to render a high quality I-frame (for client-side I-frame), we perform the following experiment.

We run Doom 3 interactively in four different scenes. These scenes correspond to the same three scripted scenes we used in our trace-driven evaluation, plus the scene we used for our user study. We play each scene for a total of 60 seconds. For throughput, we measure the average framerate across each scene as well as the standard deviation. To characterize latency, we measure the time elapsed between when Kahawai samples an input event and when Kahawai shows the frame reflecting the effect of that input event. For latency we also report the mean and the standard deviation. The results are shown in Figures 15, 16, 17 and 18.

First, these figures show that Kahawai performs well over a low-latency network connection, as shown by the blue bars in Figure 15 through Figure 18. These results are consistent with our user study, in which players rated low-latency configurations higher. The average framerates for delta-encoding are just above 40 FPS, and for I-frame they exceed 80 FPS. Figure 16 and Figure 18 show that I-frame's latency and throughput vary more than than delta-encoding's. This is not surprising since the work needed to process P-frames on the client is much less than the time to process I-frames. Figure 17 and Figure 18 show that the additional latency imposed by delta-encoding is higher than that imposed by I-frame, but that both are modest when the underlying network RTT is low.

The higher latency for delta-encoding is caused by our implementation, but is not fundamental. Delta-encoding requires the client GPU to capture each rendered frame, and perform RGB to YUV color conversion on it. We need the image in YUV format for input to the delta-patching algorithm described in Section 4.1. We implement RGB to YUV in OpenCL on the tablet GPU, but the Intel OpenCL runtime has a bug that causes us to perform an extra GPU to GPU memory copy for each frame. In addition, we have not yet implemented delta-patching in OpenCL on the GPU. This change would eliminate the additional memory copy from GPU to host memory and back.

Finally, Figure 17 and Figure 18 show that when the network RTT is high, the input-to-display latency exceeds 100ms. These latency values are high enough that users can perceive a loss in quality. This is consistent with players' rating the visual quality and smoothness of the thick client as better than any of the offloading systems in our user study. However, our user study also found that players perceived the visual quality and smoothness of collaborative rendering to be equivalent to a thin client. Figure 9 shows

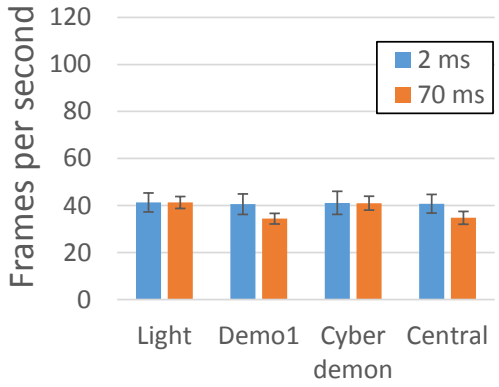


Figure 15: *Delta throughput*

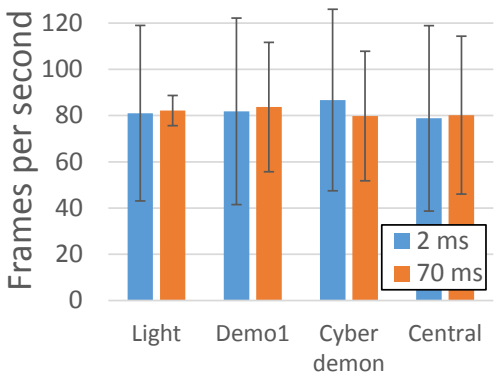


Figure 16: *I-Frame throughput*

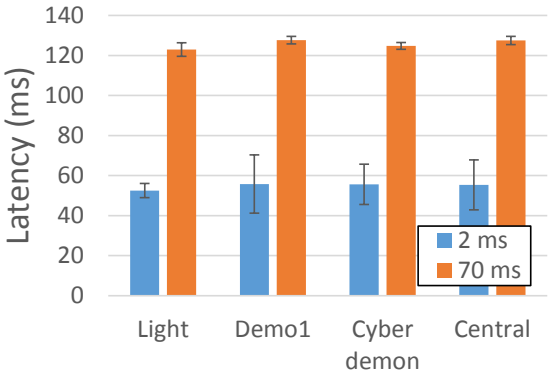


Figure 17: *Delta latency*

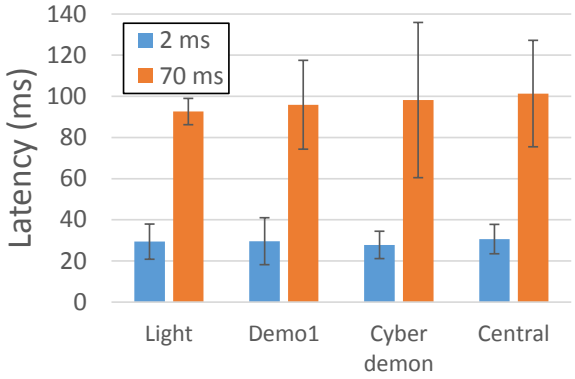


Figure 18: *I-Frame latency*

that collaborative rendering also did not impact players' game performance during our user study. Nonetheless, these results suggest that Kahawai will perform best when offloading over a LAN or to a server within a nearby CDN.

7. RELATED WORK

Adjusting data fidelity to match the quality of a mobile client's connectivity is a well-studied technique in mobile computing. For example, in Odyssey [32] depending on the network conditions, a mobile client can specify the fidelity at which a server should send it data items. Narayanan et al.[31] previously observed that different fidelities of the same 3D rendering expose the same visual structure, allowing a low fidelity rendering to be useful enough under resource constraints. Collaborative rendering also relies on low-fidelity data representations to reduce network usage, but combines locally-computed low-fidelity data with information from a server to give mobile clients access to high-fidelity data.

Overcoming the resource constraints of mobile devices by partitioning programs between a device and more powerful server infrastructure is also a well-studied area. Chroma [3] allows programmers to specify program partitions and conditions under which to adopt those partitions. CloneCloud [7] and MAUI [9] leverage features of managed-language runtimes to automatically partition a program depending on runtime prediction of the partition's energy and performance benefits. Unfortunately, both of these systems are designed for general-purpose CPU workloads and are inappropriate for applications like fast-action games that heavily rely on a GPU.

Thin-client computing is an extreme form of code offload in which all of a program's computation runs on a remote server [35, 24]. This approach has been embraced by companies like OnLive, Playstation Now, and Nvidia Shield as a way to deliver high-fidelity gaming experiences on mobile devices. Collaborative rendering has two advantages over thin-client gaming: (1) it offers substantial bandwidth savings, and (2) it allows mobile clients to use their own GPU to provide low-fidelity game play when a device is poorly connected or completely disconnected.

We have also worked on Outatime [27], a cloud-gaming system designed to address another important problem: network latency. Outatime produces speculative frames of possible outcomes and delivers them to the client ahead of time. Outatime predicts inputs and compensates for misprediction errors in the client through a graphics technique called view interpolation. While Kahawai greatly reduces the bandwidth required for cloud gaming by moderately increasing input latency, Outatime masks a substantial amount of network latency at the cost of extra bandwidth consumption. In the future, we may integrate the two system into a single low-bandwidth, latency-resilient cloud-gaming system.

Delta-encoding's use of synchronized, duplicate processes is similar to techniques developed for primary/backup fault tolerance[2] and deterministic replay [11, 14]. However, the work that most closely resembles delta-encoding is that of Levoy[28]. In Polygon-assisted compression of images, Levoy proposed selectively disabling detail and effects that are complex to render in clients and sending them as compressed deltas. Delta encoding takes a similar approach, and we have borrowed some of this work's calculations. The primary difference between our work and Levoy's is our efficient use of H.264. In particular, delta-encoding

is well suited to H.264 because of the temporal locality among delta frames exhibited by fast-action games.

Mann's work on Selective Pixel Transmission[29] also tries to reduce network communication by extrapolating frames through locally rendered, textureless models and frequent reference frame deltas. A high frequency of the reference frame deltas is necessary to limit the temporal error propagation caused by extrapolating frames from the basic model. Furthermore, in order to obtain a better balance between bandwidth and image quality, only a subset of the pixels in the delta are transmitted.

This approach resembles some of the techniques used in client-side I-frame rendering. However, instead of extrapolating frames, we constantly transfer progressive differences between each reference frame. We use the deltas to make up for limitations in the mobile GPU instead of limiting the amount of error. The high frequency of I-frames in our approach is used to reduce the size of the P-frames rather than to sync the images in the server and client.

8. CONCLUSIONS

This paper presents Kahawai, an approach to GPU offload for mobile games that relies on collaborative rendering. To implement collaborative rendering, a mobile device renders low-fidelity graphics output and combines it with high-fidelity information sent by the server to create a high-fidelity result. In Kahawai, we have developed two collaborative rendering techniques: delta encoding and client-side I-frame rendering.

Experiments with our Kahawai prototype demonstrate its potential to provide high visual quality with modest bandwidth requirements. We show that with access to the source code of only the game engine and with minimal changes, we can enable Kahawai for any game. Furthermore, we show that even without any access to source code, it is possible to enable Kahawai for some games. We show that our delta encoding technique, when compared with an H.264 thin client, provides much better visual quality when bandwidth is low – less than 0.6 Mbps. Even more impressive, we show that an H.264 thin client requires up to six times as much bandwidth as client-side I-frame rendering to achieve high visual quality. Finally, a 50-person user study with our Kahawai prototype demonstrates that collaborative rendering provides the same user experience as a thin client.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Mahadev Satyanarayanan, for their insightful feedback as well as our colleague Dinei Florencio for sharing his expertise on video compression. This work was partially funded by the National Science Foundation under award CNS-0747283.

10. REFERENCES

[1] H. Alain and D. Ziou. Image quality metrics: PSNR vs. SSIM. In *Proc. of ICPR*, 2010.

[2] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proc. of ICSE*, 1976.

[3] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying Cyber Foraging for Mobile Devices. In *Proc. of MobiSys*, 2007.

[4] H. Bao. Real-time Graphics Rendering Engine. *Most*, pages 1–6, 2011.

[5] Capcom. Street Fighter IV, 2004. `http://www.streetfighter.com/us/usfiv`.

[6] L. Cheng, A. Bhushan, R. Pajarola, and M. El Zarki. Real-time 3D Graphics Streaming using MPEG-4. In *BroadWISE*, July 2004.

[7] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proc. of EuroSys*, 2011.

[8] E. Cuervo. *Enhancing Mobile Devices through Code Offload*. PhD thesis, Duke University, 2012.

[9] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. of MobiSys*, 2010.

[10] E. Cuervo, A. Wolman, L. Cox, S. Saroiu, M. Musuvathi, and A. Razeen. Demo: Kahawai: High-Quality Mobile Gaming Using GPU. *MobiSys*, 2014.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proc. of OSDI*, 2002.

[12] Epic Games. Unreal game engine. `http://unrealengine.com/`.

[13] Ffmpeg. `http://ffmpeg.org/`.

[14] J. Flinn and Z. M. Mao. Can deterministic replay be an enabling tool for mobile computing? In *Proc. of HotMobile*, 2011.

[15] Flurry. `http://blog.flurry.com/`, Jan 2012.

[16] J. Gregory. *Game Engine Architecture, Second Edition*. A K Peters, 2014.

[17] S. Hecht. Intermittent stimulation by light. *The Journal of general physiology*, 1935:965–977, 1936.

[18] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proc. of MobiSys*, 2012.

[19] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proc. of the 3rd USENIX Windows NT Symposium*, July 1999.

[20] IDSoftware. Doom 3 source, 2011. `https://github.com/TTimo/doom3.gpl`.

[21] Intel QuickSync. `http://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html`.

[22] ITU-T Recommendation J.247. *Objective Perceptual Multimedia Video Quality Measurement in the Presence of a Full Reference*. ITU-T, Aug. 2008.

[23] Z. Kotevski and P. Mitrevski. Experimental Comparison of PSNR and SSIM Metrics for Video Quality Estimation. In *ICT Innovations 2009*, pages 357–366. Springer, 2010.

[24] A. Lai and J. Nieh. Limits of Wide-Area Thin-Client Computing. In *Proc. of SIGMETRICS*, 2002.

[25] R. Leadbetter. Console Gaming: The Lag Factor, Sept. 2009. `http://www.eurogamer.net`.

[26] R. Leadbetter. OnLive Latency: The Reckoning, July 2010. `http://www.eurogamer.net/articles/digitalfoundry-onlive-lag-analysis`.

[27] K. Lee, D. Chu, E. Cuervo, Y. Degtyarev, S. Grizan, J. Kopf, A. Wolman, and J. Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *Proc. of MobiSys*, 2015.

[28] M. Levoy. Polygon-assisted JPEG and MPEG compression of synthetic images. *Proc. of SIGGRAPH*, 1995.

[29] Y. Mann and D. Cohen-Or. Selective Pixel Transmission for Navigating in Remote Virtual Environments. *Computer Graphics Forum*, 16(3), Sept. 1997.

[30] Microsoft. DirectX Developer Center, 2012. `http://msdn.microsoft.com/en-us/directx`.

[31] D. Narayanan and M. Satyanarayanan. Predictive Resource Management for Wearable Computing. In *Proc. of MobiSys*, 2003.

[32] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. of SOSP*, 1997.

[33] OpenGL. OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org/`.

[34] I. Richardson. *The H.264 Advanced Video Compression Standard*. John Wiley & Sons, 2010.

[35] N. Tolia, D. G. Andersen, and M. Satyanarayanan. Quantifying Interactive User Experience on Thin Clients. *IEEE Computer*, 39(3), Mar. 2006.

[36] Unity. Unity 3D Engine 5. `http://unity3d.com/5`.

[37] VirtualGL. `http://www.virtualgl.org/`.

[38] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[39] M. West. Measuring Responsiveness in Video Games, July 2008. `http://www.gamasutra.com`.

[40] x264. `http://www.videolan.org/developers/x264.html`.