A Playout Buffer for Moonlight Cloud-Based Game Streaming to Smooth out Network Jitter

A Major Qualifying Project submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted By:
Nicholas Heineman - Computer Science
Yu-Chi Liang - Computer Science
Aaron Nguyen - Computer Science
William Ryan - Computer Science

Date:

Thursday, April 25, 2024

Report Submitted to:

Professor Mark Claypool

Worcester Polytechnic Institute

Acknowledgement

We would like to thank our Advisor Professor Claypool and our Graduate Assistant Xiaokun Xu for greatly assisting and supporting our project. This project would have been unachievable without their help and expertise in the field.

Advisor

Mark Claypool claypool@wpi.edu

Graduate Assistant

Xiaokun Xu xxuu11@wpi.edu

Abstract

Cloud game streaming allows a game to be run on a server and streamed over the Internet to a lightweight client that plays the game like a video. This setup is sensitive to delay in the network and network jitter that can cause frame jitter, which produces visible interrupts in the playout, affecting quality of experience. To address this, we added a client-side playout buffer to open-source streaming software, which stores frames to smooth out network jitter. Then we created several algorithms that regulated the size of this buffer to protect against jitter. We tested our system on a streaming testbed setup by introducing delay jitter. We found that having a frame buffer and a queue monitoring system lowered frame jitter with controlled amounts of delay.

Table of Contents

List of Figures and Tables

1.0 Introduction

As technology advances, computer hardware is also getting more powerful. Video games have benefitted from these advancements as a result. Video game graphics are getting better and better, but this also means they demand stronger CPUs and GPUs. A game played at 4K resolution and 60FPS needs a computer with high end components. However, these advancements mean people need to have high-end setups to play games in traditional systems. This is the problem that cloud-based gaming aims to solve.

Unlike traditional gaming setups, cloud-based gaming does not rely only on local hardware. Instead, cloud-based gaming shifts the processing power to remote servers, allowing players to stream games over the Internet without the need of high-end gaming setups. While cloud-based gaming helps lessen the demand on local hardware, to be able to stream games in a stable state, a fast and stable Internet connection is needed. As a result, cloud-based gaming is affected by network latency. A delay can occur in the time between sending the packet and being displayed on screen. In addition, network latency can vary over time, and when this happens it causes jitter or interruptions during the streaming session. Jitter in the network takes several forms, but notable types are delay jitter and bandwidth jitter. Delay jitter is caused by packets being delayed between the server and client. Bandwidth jitter is caused by the amount of data being sent varying, changing the bitrate. Both of these cause frame jitter, which is when the stream is interrupted and freezes for some time. Since there is no frame to be played, the client can only display the last frame until the new frame has arrived. Both delay and frame jitter are problematic because players need up-to-date information and smoothness in the streaming during their gaming experience. Both of these factors can negatively affect the Quality of Experience

(QoE) of the user as it lowers the smoothness of the stream and increases the amount of delay. It is necessary to alleviate both of these issues in order to make cloud gaming more viable in the future.

Our project focused on handling frame jitter caused by delay jitter spikes. In order to test possible solutions, we worked using Moonlight, an open-source streaming client, and Sunshine, its paired open-source server. We implemented a buffer that takes frames assembled by the Moonlight software and stores them in a queue. In the event of a delay jitter spike, frame playout is not immediately affected since there are frames in reserve. Additionally, we created a system that regulates the size of the queue to allow for a delay-frame jitter tradeoff. We studied existing buffering algorithms used in other fields, such as video conferencing, looking at how they deal with delay jitter and implemented algorithms. The E-Policy dequeues frames at an average rate, allowing a buffer to build naturally as jitter spikes occur. The I-Policy also dequeues frames at a consistent rate, but it also discards any frames that arrive later than is expected, keeping.
We ran experiments evaluating our system through two computers connected by Ethernet, running a Sunshine server streaming content to Moonlight running on the other. To mimic the client-server relationship in cloud game streaming, we used a Raspberry Pi as a switch to add varying amounts of network jitter. Our experiments consisted of streaming a game's benchmark from the server to the client with different jitter settings for 1 minute. We tested base moonlight, and two versions of our complete E-policy algorithm, 50 times at each jitter setting to get meaningful results.

Our results showed that adding a buffer can reduce or eliminate frame jitter caused by delay jitter. In the base Moonlight settings, as the amount of delay jitter increased, the frame jitter increased alongside it, making the amount of interruptions in gameplay increase. However,

when our adjusted E-policy system is added with a low queue target, the majority of frame jitter is removed at lower settings. The spikes remained at the high delay jitter setting. When it is added with a high queue target, nearly all frame jitter is removed. However with our algorithm, the larger the queue target gets, the larger the overall delay in the system becomes.

The remainder of the paper is divided as follows: Chapter 2 provides background information on cloud streaming and jitter compensation techniques; Chapter 3 describes the setup of the buffer and experimentation process; Chapter 4 covers the analysis and results of the experiments; Chapter 5 concludes the findings and provides key takeaways; and Chapter 6 provides potential avenues for future research and current limitations.

2.0 Background

Throughout the next chapter, we are going to discuss the history of cloud gaming, its benefits and disadvantages, the tools we plan to use, and a variety of buffering techniques and algorithms.

2.1 History of Cloud Gaming

Cloud game streaming goes back to the early 2000s, with G-Cluster being the prominent founding startup of the industry. G-Cluster relied on Quality of Service support from network providers, which can be attributed to the less mature Internet at the time (Cai et al., 2016). It never really took off and services designed with "Over-the-top" (OTT) network designs, such as OnLive and GaiKai began to emerge in the late 2000s. OTT designs tend to go directly from the host to the user, rather than rely fully on infrastructure from Internet Service Providers (Cai et al., 2016). OnLive struggled to gain support from publishers because the idea of its subscription model was not common in the industry at the time ("Cloud Gaming", 2022). Sony acquired GaiKai in 2012, and began development on its Playstation Now platform, and acquired the patents from OnLive in 2015 (Cai et al., 2016). PlayStation Now and Sony's acquisition of GaiKai was a signal to the rest of the games industry that cloud game streaming had a future (Cai et al., 2016). NVIDIA soon followed suit with NVIDIA GRID, their cloud streaming solution geared towards their Shield TV devices. It would later be rebranded to GeforceNow and redeveloped for more platforms. In 2018. Microsoft and Google began development on their own cloud game streaming solutions, Project xCloud and Project Stream respectively ("Cloud Gaming", 2022). Google would launch Google Stadia in 2019 and Microsoft launched Xbox Cloud Gaming in late 2020. Amazon also entered cloud game streaming with its Luna game

streaming service in 2020 (“Cloud Gaming”, 2022). Despite the loss of Google Stadia in early 2023, the cloud gaming industry is still growing in support, with platforms expanding and constantly updating to better the experiences for the end users.

2.2 The Benefits and Disadvantages of Cloud Gaming

In general, gaming is known to be one of the most computationally heavy tasks the average user runs. In cloud computing, the actual computing power is in a remote data center. Cloud game streaming utilizes a client-server relationship with a nearby data center rendering the game and computing all logic on a remote server, then streaming the gameplay to a thin client. For example, GeforceNow, NVIDIA’s cloud game streaming solution, finds the nearest data center to the player and streams their game from there. This allows gamers to use much lower-power hardware, as only an Internet connection is needed to connect to the server and start playing, drastically reducing the barrier to entry. The cloud infrastructure does introduce some potential issues that would not be as present otherwise; specifically, network latency and network jitter begin to impact the experience whereas they are not present in a locally played game.

Latency, the delay between transmissions on the network, is one of the common factors that hurt the QOE of a cloud game stream. This delay is experienced by players through input by the user as the discrepancy between the player input and visual output grows (De-Yu Chan, 2017).

Network jitter is the variance in network conditions. Jitter spikes occur when there are spikes of high latency, or drops in bandwidth in the network connection (Hassan Iqbal et al., 2021). Traditionally, a playout buffer is used in video streaming to account for jitter spikes. YouTube for example will buffer portions of the video before it begins its initial playback in

order to accommodate network jitter. For use in game streaming, a playout buffer causes an increase in latency that scales linearly with the number of frames in the buffer.

Both latency and jitter impact the game streaming experience. Latency causes a temporal desync with the gameplay experience. While latency does not impact visual quality or smoothness in similar ways to Jitter, higher latency environments make the user feel like they are not playing the game in real-time, due to the perceivable delay with their inputs (De-Yu Chan, 2017). Jitter spikes impact cloud gaming by causing freezes and stutters in the video playback (Hassan Iqbal et al., 2021). We implemented a playback buffer like those used in traditional video streaming and set out to apply it to cloud game streaming.

2.3 Moonlight and Sunshine

Moonlight was built as an open-source client for NVIDIA's gamestream protocol, allowing PC gamers to take their games anywhere on their network, or even over the internet. Sunshine is an open-source game stream server, and it was built to work with Moonlight in particular. Sunshine serves as a drop-in replacement for NVIDIA's proprietary game stream server, and when paired with Moonlight, the two tools create an open-source self-hosted game stream solution. Together they enable gamers with high-end PCs to host a cloud game streaming server and use any light weight device as the client. These are important because the basis of our project revolves around Moonlight. Our goal is to implement a playout buffer in Moonlight that balances gameplay smoothness and latency for an improved game streaming solution.

2.4 Playout Buffering Algorithms

Playout buffers manage the flow of data within the system in which they are implemented. Playout buffers are vital in reducing the effects of latency, by reducing interrupts in play and maintaining a consistent frame rate.

2.4.1 E-Policy

The policy starts off with a low initial display latency and is increased based on any late arriving frame (Stone & Jeffay, 2002). This builds a natural buffer as frames are played out at a consistent rate and delays occur. This can be seen in Figure 1 where frame c arrives late by over two frame times. Because of this the buffer is increased to 3 frame times for all subsequent frames. With the E-Policy no frames are discarded in the process, and the number of interrupts is reduced over time as the buffer size increases to a large enough value to handle future jitter spikes. The drawback is that the buffer increases with the delay permanently.

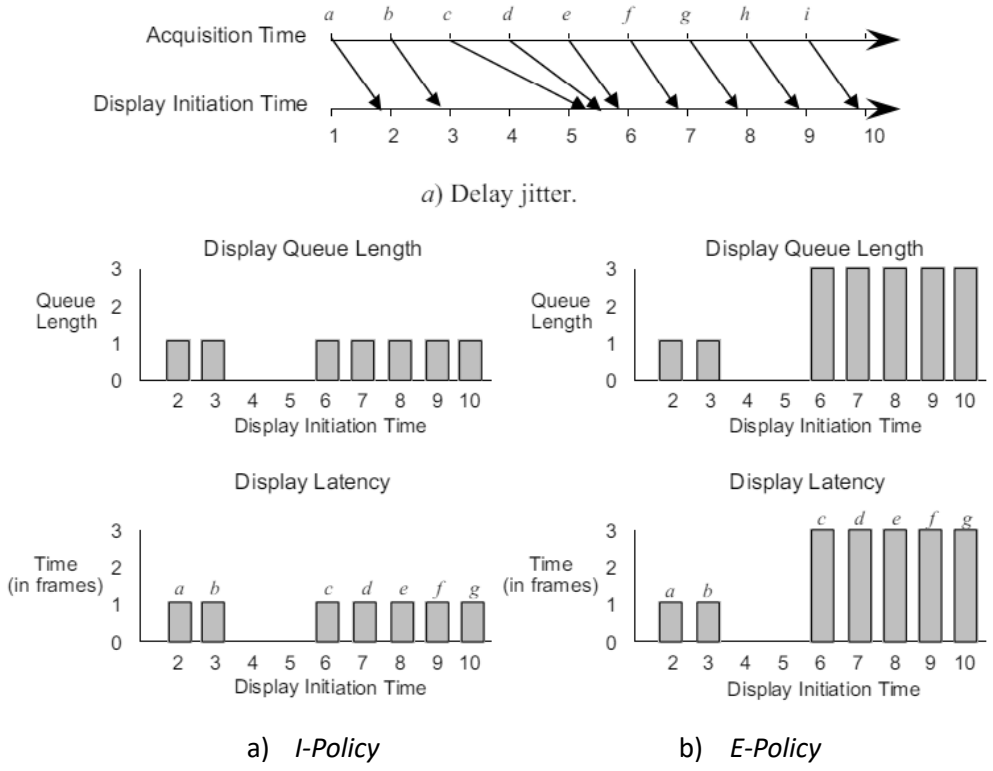

Figure 1: I-Policy and E-Policy after jitter spike (Stone & Jeffay, 2002)

2.4.2 I-Policy

The I-Policy is another playout buffer used within video conferences; similarly to the E-Policy, it plays out frames at a consistent rate. The way the I-Policy differs is the buffer comes with a fixed display latency meaning the delay is the same for every frame that is played achieving this by discarding any frames that arrive later than the expected arrival time (Stone & Jeffay, 2002). This process can be seen in Figure 2 when frame b arrives two frame times late and is discarded in the I-Policy, leaving a gap between when frame a and frame b is played. The pros of the I-Policy are that it causes lower latency overall and it shows frames that are more up-to-date based on the time they arrive compared to the E-Policy. The cons of the I-Policy are a

larger number of interrupts compared to the E-Policy and some information is lost when the frames are discarded.

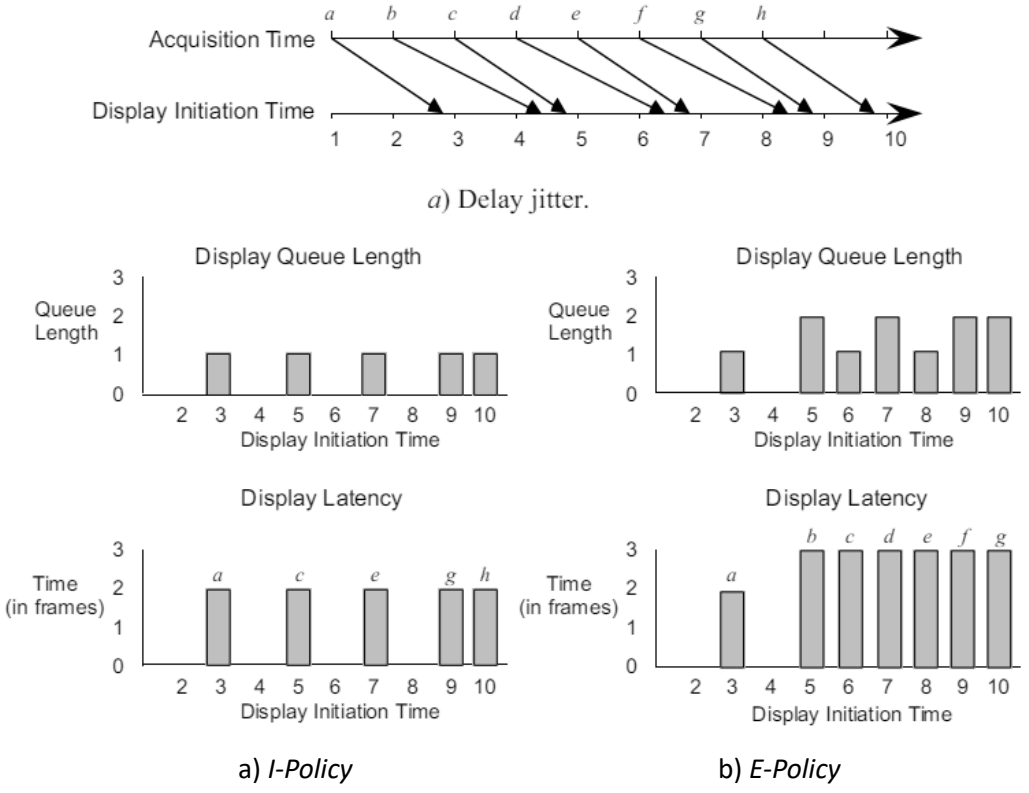

Figure 2: I-Policy and E-Policy (Stone & Jeffay, 2002)

2.4.3 Queue Monitoring

Queue Monitoring is a type of playout buffer that dynamically changes based on network traffic on how many frames are arriving at a time. In this policy threshold values are implemented which are the length in frame times for a queue length where if the queue has more frames then the specified queue length the display latency will be reduced(Stone & Jeffay, 2002). The buffer then discards the oldest frame in the queue and displays the oldest frame left. The pros of a monitor are that it is flexible based on different threshold values which can cause higher latency and fewer interruptions or lower latency with more interruptions. The con is that the buffer only changes based on late-arriving frames.

3.0 Methodology

This chapter describes the process of implementing our buffering algorithms in Moonlight. In addition, it describes the methods we used to evaluate our buffer algorithms by logging and graphing the data collected through the scripted runs of the War Thunder benchmark on various jitter settings.

3.1 Buffer Implementation and Placement

Before we implemented our buffer, we needed to decide whether to place the buffer on the frame or packet level of the program. If placed at the packet level our program would regulate the individual data packets sent by the server to the client and let Moonlight handle assembly and rendering of frames. At the frame level, our code would regulate fully assembled frames during rendering, leaving Moonlight to handle packet management. We decided to place the buffer at the frame level, because we thought it would give us better control of the user experience, since users are seeing frames when playing. Our initial implementation of the buffer was placed directly before the rendering pipeline, taking the frame that moonlight assembled and using it in our own code.

The initial design was on the same thread as the render, and would queue the frame into the buffer and immediately dequeue. The reason for this was to test our buffer to see if it would interfere with anything from base Moonlight's functionality. After this initial debugging period, to get the buffer to work we split the base Moonlight render function into two separate threads: one that would queue the frames into the buffer, and another to dequeue the frames to be rendered as shown in Code Listings 1 and Code Listings 2.

3.2 Logger

A logger was necessary for the experiment because we need specific key values as evaluation metric and a mean for debugging. We need a logger that can be accessed from anywhere and freely add new variables we want to record. In addition, we want this logger to produce data in a form that can easily be graphed. Such a logger increases efficiency during our design process.

Creating the logger involved implementing a singleton instance of a Logger class, which all parts of the Moonlight code could then reference and use. Mutex locks controlling access to the logger instance were used to avoid race conditions. Our first iteration of this logger only involved storing output messages in a text file. However, we also wanted to make a system that would create easily graphable data, so we constructed a CSV creator that defined each column as a variable we wanted to track. Each entry was paired with a timestamp registered directly at the moment of logging, so that threading would not create inconsistent times. With this system, we were easily able to create graphs with the CSV, plotting the stored time values against whichever variables we chose to log.

3.3 Implement Buffering Algorithms

We implemented and tested two different buffering algorithms, the I-Policy and E-Policy. We also implemented Queue Monitoring as a setting that would adjust both of these algorithms based on the on queue size.

3.3.1 E-Policy

The first policy that we implemented was the E-Policy, which is the simplest algorithm. It works by storing frames in a buffer before they are sent to be rendered, and then dequeuing the frame at a regular interval. We tested two different methods to decide the length the buffer would

hold onto a frame. The first method used a constant time of 16.670 ms, this is due to moonlight running at 60 frames per second, which means a frame is rendered every 16.670 ms. Our second method was a weighted moving average, which takes the average interframe time over the entire run of the program and adjusts the time based on each incoming interframe time. The formula for the weighted moving average is shown in Equation 1. The value of the alpha is between 0 and 1. It determines the influence of the new frame time. We used 0.95 as the alpha value, as seen in Code Listings 1, to ensure that the average frame time does not increase too quickly. The average value is adjusted after each run based on the expected sleep time and the actual sleep time, which is calculated by measuring the time before the program goes to sleep and the time after and calculates the difference. This value is then used during the next time a frame would be rendered and is subtracted from the current sleep time to make up for the added time from the previous run as seen in Code Listings 2. Before any frames are sent to be rendered the dequeue thread checks if the buffer is currently dequeuing as seen in Code Listings 2, which checks if the buffer is in a fill state when frames are added to the buffer with no dequeue or a drain state when frames are able to be dequeued.

Equation 1: Formula for calculating an exponentially weighted moving average

Average = (Average * α) + (Current * 1 - α)

Code Listing 1: Pseudo code of E-Policy queueing function.

```
Queuing Thread:
E-Policy (Frame):
        time_arrived = current time
        time_between_frames = time_arrived - last_frame_arrival_time
        lock queue mutex
        add frame to buffer queue
        unlock queue mutex
        lock sleep mutex
        average_frame_time = (average * α) + (time_between_frames *1-α)
        unlock sleep mutex
        last_frame_arrival_time = time_arrived
```

Code Listing 2: Pseudo code of rendering function in dequeue thread

```
Dequeue Thread:
render_frame_dequeue_thread():
repeat forever
        if DRAIN state then
                start_time = current time
                frame = buffer[front]
                send frame to render
                end_time = current time
                run_time = end_time - start_time
                average_sleep = average interframe time
                if queue_monitoring_flag = true then adjust sleep offset value
                sleep_off_set = current sleep offset value
                expected_sleep = average_sleep - run_time - sleep_difference -
sleep_off_set
                if expected_sleep > 0 then
                        begin_sleep_time = current time
                        sleep for expected_sleep
                        end_sleep_time = current time
                        real_sleep_time = end_sleep_time-begin_sleep_time
                        sleep_difference = real_sleep - expected_sleep
                else
                        sleep_difference = 0
        else
                sleep for 500 microseconds
```

3.3.2 I-Policy

The I-Policy works similar to the E-Policy, except that it discards any frame that arrives later than it is expected to. It also has a fill and drain state: at the start of the program the buffer will be in the fill state and fill to the set amount of frames before it starts dequeuing. It then enters the drain state for the remaining run of the program, and will dequeue at the same moving average rate from E-policy. The value we set for the expected arrival rate of each frame is double the dequeue rate of 16.670 ms discarding any frames that arrive later besides keyframes as they

are needed to render any following frames, as seen in Code Listings 3.

Code Listing 3: Pseudo code of I-Policy queueing function

```
Queuing Thread:
I-Policy (Frame):
        time_arrived = current time
        time_between_frames = time_arrived - last_frame_arrival_time
        if time_between_frames > 33333 microseconds and not a key frame then
                discard the frame
        else
                lock queue mutex
                add frame to buffer queue
                last_frame_arrival_time = time_arrived
                unlock queue mutex
```

3.3.3 Queue Monitoring

Queue Monitoring is a setting that can be used by any of the two policies previously mentioned. This monitoring evolved over time. At the start, we set the value as a constant offset to the average calculation. Later, we adapted it to have a target queue size: when the queue is too small or too big, the offset value would be added or subtracted until it restabilizes. Then, we made the value of the offset more adaptive, scaled by the difference between current and target queue size. Finally, we settled on a system that relies on built in constants. The system sets the offset at a large value of 4000μs when the queue is large, a smaller value of 2500μs when the queue is too small, and at a stabilizing value of 1250μs when the queue is within acceptable parameters as seen in Code Listings 4. The reasoning for using these constants will be discussed

in the next chapter.

Code Listing4: Pseudo code of playout offset function for queue monitoring

```
Adjust Offset Value ():
        queue_length = current queue size
        lock offset mutex
        if queue_length > queue_monitor_target then
                sleep_offset_value = LARGE
        else if queue_length < queue_monitor_target then
                sleep_offset_value = STABLE
        else
                sleep_offset_value = SMALL
        unlock offset mutex
```

3.4 Experiment Setup

This next section is going to cover our experiment setup and testing environment. It will also include how we designed the experiment and evaluation metrics we used.

3.4.1 Benchmark

For our experiments, we have considered two options, a user study or game benchmarking. A user study allows us to gather feedback on how the participants feel about the different policies in varying network settings. One concern with this approach is consistency. Since the goal is to evaluate and compare the performance of different policies, it is necessary to select a game that creates the same gameplay for all trials. Given that most games create variance during its gameplay from run to run and the participants have varying skill levels, it was difficult for us to settle on a game. On the other hand, benchmarking was the more consistent approach. During development and testing, we have determined that the type of game we use does not impact the performance. We tested games with benchmarks that are available to us such as

Shadow of the Tomb Raider and War Thunder before reaching this conclusion. The capture for the two different games using base Moonlight can be seen in Figure 3 with the top graph as War Thunder and bottom graph as Tomb Raider. While the Tomb Raider run had more variance than War Thunder, it is within the normal range. Finally, we decided to use War Thunder's built-in benchmark as the base testing game. While not inherently better than other benchmarks, the benchmark of War Thunder is easily accessible from the main screen and sets up an efficient testing environment. While Shadow of the Tomb Raider has a built-in benchmark, it is harder to access compared to War Thunder. To access the benchmark for Shadow of the Tomb Raider, we have to navigate to the graphics settings and then enter the benchmark tab. After that, we can enter the benchmark. The issue is that when exiting the benchmark, it does not always return to the menu before entering the benchmark. We cannot reenter the benchmark quickly after each run. The exit timing for this benchmark is also inconsistent. These are all factors that make creating the experiment script harder. In comparison, War Thunder has the benchmark located in its main menu. To access it, we need to go over to the Battles tab and select the benchmark in the drop down menu. After exiting the benchmark, we are always back in the main menu with a benchmark summary pop up. We then press enter to return. The way War Thunder exits the benchmark is more consistent and easy to automate.

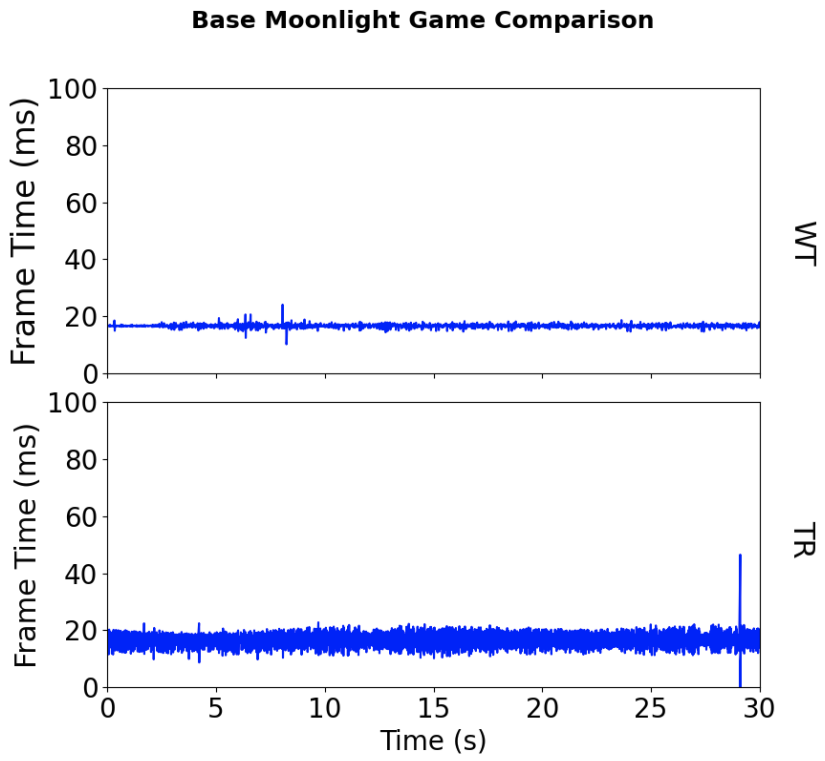

Figure 3: Base Moonlight Capture between War Thunder and Tomb Raider

3.4.2 Data Collection

As mentioned in the earlier section, we have built a logger for data collection purposes. Specifically, we are logging the queue occupancy, frame enqueue and dequeue rate. These are useful metrics for observing the performance of each policy. The queue occupancy reveals how many frames are in the queue and the delay introduced by the buffer. Since we are streaming at a rate of 60 FPS, each frame time is about 16.67 ms (1/60). This means a queue occupancy of, number adds number delay. The frame enqueue rate is used to study the effect of delay jitter on frame arrival rate. Earlier, we discussed how a delay jitter event will subsequently cause frame jitter. The enqueue rate captures this behavior and visually demonstrates the effect. The dequeue rate is used to represent the playout for the stream. We want to make a comparison between the arrival and playout rate to show that the policies can mitigate the effects of delay jitter events.

Besides our logger, we are also using an additional tool called PresentMon (Appendix X). This tool is used for capturing and analyzing the performance of graphics applications. PresentMon also generates its own resulting CSV. Among the data PresentMon is able to track, a useful value is “msBetweenPresents”. This is the same information as the frame dequeue rate. We use PresentMon as a means to verify our captured data. This information is also used for calculating the evaluation metrics.

3.4.3 Experimentation

The experiment is done in a lab setting where two computers are connected by ethernet with a Raspberry Pi acting as a switch as shown in Figure 5. The computers we are using have an i7-8700k CPU and a Nvidia GTX1080 graphics card. They also have 64GB of ram. The stream is done in 1080p resolution with 60 FPS. The benchmark we are using in War Thunder is “Pacific war (Day)” and the specific graphics settings we are using is in Figure 4. In the 1 minute capture, after the loading screen, it will show two scenes with just the sky and cloud. Next, it presents multiple scenes with transitions where different types of planes are flying at different angles. Towards the end of the capture, we are starting to see battles between the planes with gunfire and explosions. The scene and their order will be the exact same in each run. This creates the consistency that could be lacking from a user study.We are testing 4 different policies total with base Moonlight as the default comparison. We have included E-Policy and Queue Monitoring at queue size of 2 and 10. The purpose of including two versions of Queue Monitoring is to study the effect of a shallow vs deep buffer in the context of cloud game streaming.

Figure 4: War Thunder In-Game Graphics Setting

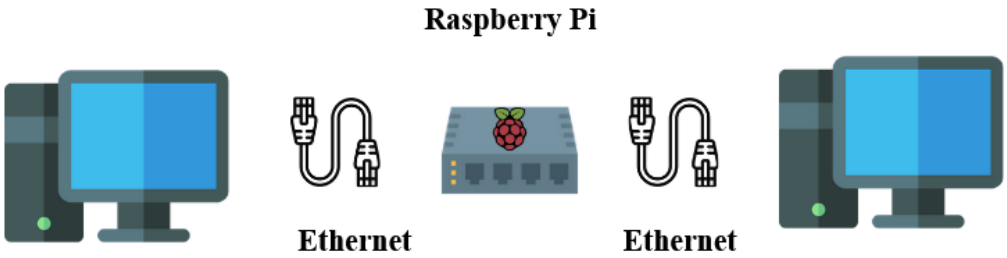

Figure 5: Example Lab Setting

Originally, we were going to include I-Policy in the experiment since we did design and implement a version of it. However, after some updates and code changes, it started to have abnormal behavior. The stream starts off as normal, but as soon as it enters the benchmark, it

does not render any new frames. We suspected that this was due to a discard of the key frames, but after more testing, we cannot find a solution. Due to time constraints on the project, we decided to omit I-Policy from the experiment, but encourage future works to investigate and test this policy.

For adding delay jitter to the network, we are using tc with Netem. Tc stands for traffic control and Netem is a Linux tool for network emulation. The core principle of adding delay jitter is through running a command and adding qdisc, or queueing discipline. By specifying the type of qdisc, we are giving instructions to the network on how to handle the flow of packets. This creates the effect of delay and jitter. We set up 3 different types of scripts that apply the specified delay jitter settings through Netem with a jitter stop script, see Figure 6 for reference.

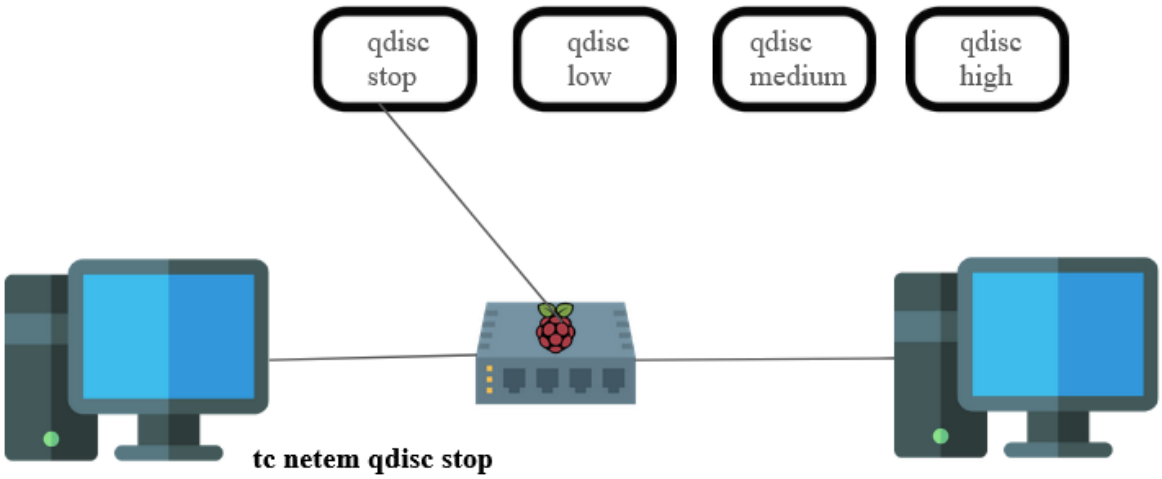

Figure 6: Tc Netem command example.

We tested a variety of network settings. These settings are no jitter, low jitter, medium jitter, and high jitter. The purpose of a no jitter and high jitter environment is to test the two extreme conditions for the different policies. The two remaining settings are to mimic common network conditions in everyday activity. The main change per jitter setting is the magnitude. Magnitude refers to the intensity of the jitter. The values of the jitter setting from low to high is: 25 ms, 40 ms, and 100 ms. All settings are applying jitter every 0.1ms, or 10 times a second. These settings are the independent variable of the experiment.

To run the experiment, we have developed a Python script that automatically enters the War Thunder benchmark with PresentMon capture and changes the policy and jitter settings. First, the script generates all possible pairs between policies and network settings. It then iterates through all pairs and runs the benchmark at the jitter setting 50 times. The duration of each capture is one minute, and it then exits Moonlight and restarts the capture process. After the end of each run, the script appends a number tag to the resulting CSV files created by our logger and PresentMon. This is to correctly match the file from each run so we are comparing files from the same run. After the 50 runs are over, it generates a summary file with calculated evaluation metrics from each run. Each policy-setting pair also has a specific folder to store all the data and resulting CSV.

3.5 Evaluation Metrics

We are going to use using 3 evaluation metrics, dashboards, magnitude and interrupts. Dashboards are our developed evaluations metrics while magnitude and interrupts are common metrics for Quality of Experience.

3.5.1 Dashboard

One of the evaluation metrics we developed is a dashboard-like graph that displays the frame enqueue and dequeue rate with the queue size. An example of this can be seen in the results section. This is to visually show the effectiveness of each policy and how it can smooth out the frame jitter event that occurs during the run. For base Moonlight, we are only displaying the PresenMon's capture result. This is because Moonlight does not have a designated queue and we are unable to log that information. Thus, we assume the queue is constantly at zero as Moonlight dequeues the frame as soon as it arrives.

3.5.2 Magnitude and Interrupt

The other evaluation metrics we used are magnitude and interrupts. Interrupt occurs when the frame's arrival time has exceeded double the expected time. The expected frame time will be different depending on the streaming frame rate. For our experiment, since we are streaming at 60 FPS, each frame time is about 16.67ms. So, an interrupt occurs when the frame arrives later than 33.34ms. The number of interrupts will tell us how the network condition affects the streaming session. We compute interrupts as a rate across the experiment run, and it has a unit of interrupts/s. Magnitude is the strength/intensity of the given delay jitter event. Magnitude is calculated from interrupts. Once we have determined where all the interrupts occur during the run, we find the difference between the frame time and double the standard frame time. Then we sum that difference for all interrupts and divide by the run duration. These metrics will be able to determine how effective the policy is at mitigating a frame jitter spike. The policy is performing better when it has a low number of interrupts and magnitude.

4.0 Results

In this chapter, we present the results of the experiment through the defined evaluation metrics. We depict the dashboard of a single run for each delay jitter setting per policy and review the performance graphs. We represent jitter events visually and how the policies are affected. Then, we calculate interrupts, magnitude, and queue occupancy over 50 runs for each policy-setting pair.

4.1 Dashboard

As mentioned in methodology, we built a system that quickly constructs a dashboard of three key graphs based on data gathered from a single run. The default run conditions were a War Thunder benchmark with a capture duration for 60 seconds. The X-axis is the capture time in seconds. From top to bottom, the Y-axes are: The frame enqueue (in milliseconds), the queue occupancy (the number of frames), and frame dequeue rate (in milliseconds). If the Enqueue and Dequeue graphs exhibit near constant values, appearing as a straight line, it means that the we are enqueueing or dequeueing at a constant rate. This is what we describe as a smooth playout. If the graph does not show a constant value but goes up and drops back down quickly, it indicates a spike. The target dequeue rate is 16.67 ms, as this translates to the 60 f/s we are expecting based on our settings. For queue size, users prefer a stable queue that dips during lag spikes and rises back up again afterwards, with the queue occupancy at a specified value.

4.1.1 E-Policy

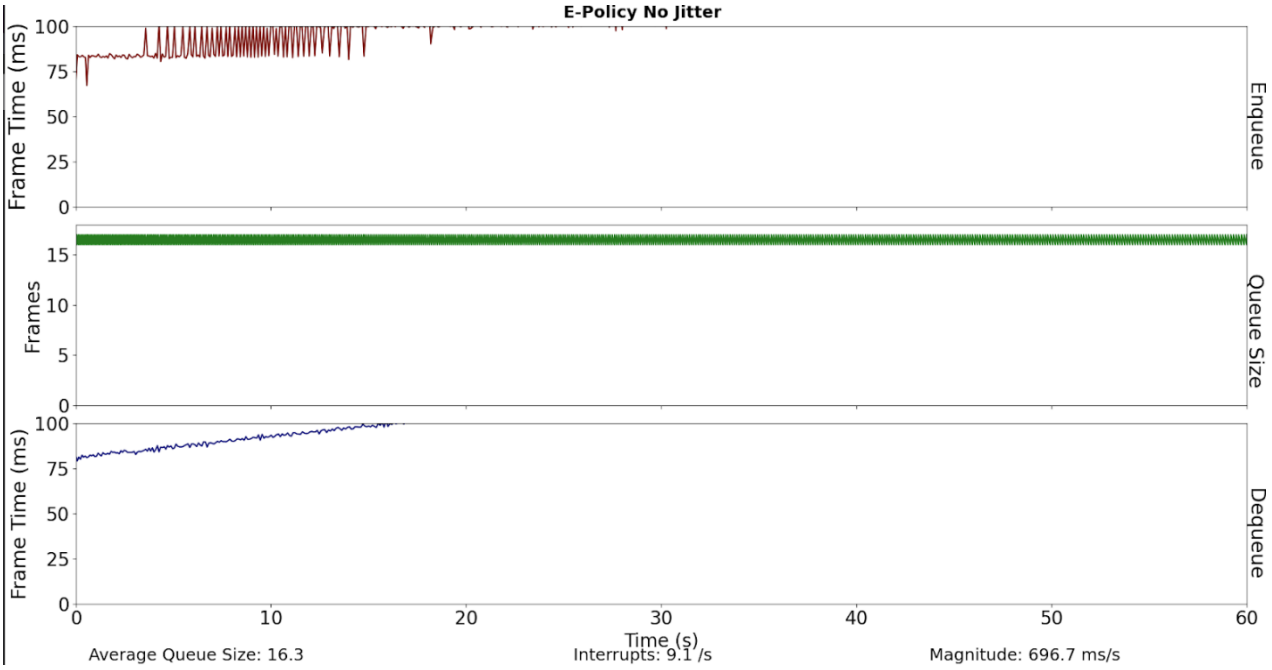

Figure 7: E-Policy dashboard at no delay jitter setting

As mentioned in the methodology chapter, we initially planned to include E-Policy with no queue monitoring during the experiment. However, as shown in Figure 7, the frame enqueue and dequeue frame time consistently increased. The queue size also increased and stayed at a constant between 16 and 17. The frame time increased to the point where it can no longer be contained in the graph. This caused the bitrate of the stream to drop significantly and become pixelated. We suspect this behavior is a mechanism of Moonlight. Once the bitrate and delay reach a certain threshold, Moonlight may control the amount of incoming frames and drop frames accordingly. The increase in delay is caused by the way we designed the E-Policy algorithm. In its current form, there is a negative feedback loop. Once the dequeue rate increases in delay, it causes the future frames to delay. This would explain why both the enqueue and dequeue rate are increasing. In the end, we were unable to identify a flaw in our system that

created this feedback loop. Due to this behavior, we decided that it is best to not include the base E-policy model as part of the evaluation.

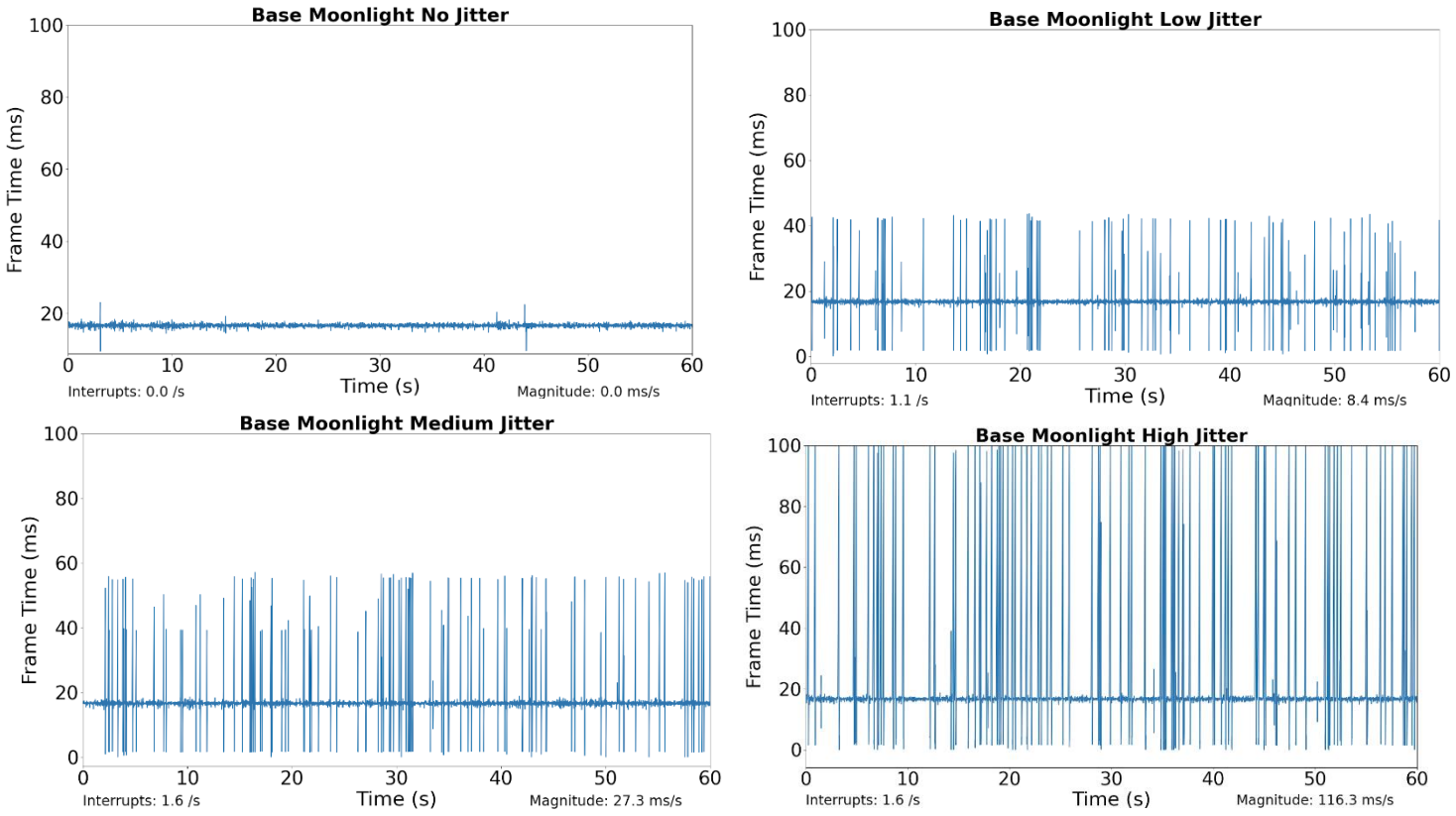

Figure 8: Dashboard for base Moonlight playout at different delay jitter settings

4.1.2 Base Moonlight

The dashboard we designed for base Moonlight is different from our algorithms because base Moonlight does not have a playout queue. Figure 8 only shows the playout rate. In the top left of the figure, we observed the value of the playout rate is near a constant value, which is what we described as smooth. Moonlight performs well in a no jitter environment, but as the magnitude of the delay jitter increases, the number of spikes in the playout also increases. The statistical summary at the bottom indicates an upward trend in interrupts per second. These spikes indicate a jitter event, and each spike is considered to be an interruption. Because base

Moonlight does not have a built-in queue, it plays a frame as soon as it arrives. So upon encountering an interrupt, it leaves a gap in the playout and no frame is rendered at the moment. With high delay jitter, this gap happens more often and the impact of jitter is more apparent. During the streaming session, we can visually observe pauses in playout. It is clear that Moonlight suffers in a high delay jitter environment as it was difficult to have a session without interrupts. The magnitude also reflects how intense each interrupt is. We also gathered summary statistics from 200 runs of base moonlight, with 50 runs of each delay jitter setting. This is displayed in Table 1. We measured average queue size, number of interrupts, and magnitude of jitter spikes on no, low, medium, and high delay jitter. Each column in the table represents the average values 50 runs, with the uncertainty of each calculation listed. Similar to our sample run, moonlight performs well at low jitter levels, but gets progressively worse the higher jitter gets. By using both data and observation, cloud game streaming with base Moonlight in a network with jitter does not provide a smooth user experience.

Table 1: Summary statistics of 50 experiment runs for base Moonlight

Delay Jitter Setting	None	Low	Medium	High
Average Queue Size	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
Interrupts/s	0.00 (0.01)	1.06 (0.13)	1.41 (0.14)	1.44 (0.17)
Magnitude(ms/s)	0.03 (0.07)	8.28 (1.04)	24.82 (2.46)	103.30 (11.86)

4.1.3 Adjusted E-Policy - Target Queue Size 2

While we described implementing a queue monitoring algorithm to be applied to E-Policy, we did not completely implement that design. While we were designing the algorithm, we were unable to find a proper way to calculate an offset to adjust the playout rate. We mainly used

algebraic based formulas to calculate the offset and they were unsuccessful. Our calculations created a dequeue rate that fluctuates up and down around the target, as depicted in the Appendix X. This is due to how fast the frames are arriving, but our calculation is too slow to keep up. We were always behind in the offset calibration and not able to maintain the queue at the correct value. As a result, to implement the behavior of queue monitoring, we decided to use hard coded values. These values were 2500, 1200, and 4000 μs. The 2500 μs is the regular offset, meaning that when the queue is at the specified size, we will subtract the value from the playout rate as adjustment. 1200 μs is the minimum playout adjustment. When the queue size is below the target size, subtracting a smaller value will make the playout slower and allow the queue time to be filled. Lastly, the 4000 μs is the maximum playout adjustment. In Chapter 8, code listings 4, each of these adjustments are applied to the playout rate on lines 5, 7, and 9. If the queue occupancy is too high, then making the playout faster can drain the queue. We used μs in our code to achieve higher accuracy with the decimal point. These are constant values that are used across all runs of our experiment. Since we are no longer using an adaptive algorithm to calculate an offset for the playout rate, it is not appropriate to call our implementation queue monitoring anymore. This is a version of E-Policy with a specified queue target. So from this point in the paper, we are going to address our implemented algorithm as Adjusted E-Policy instead of queue monitoring.

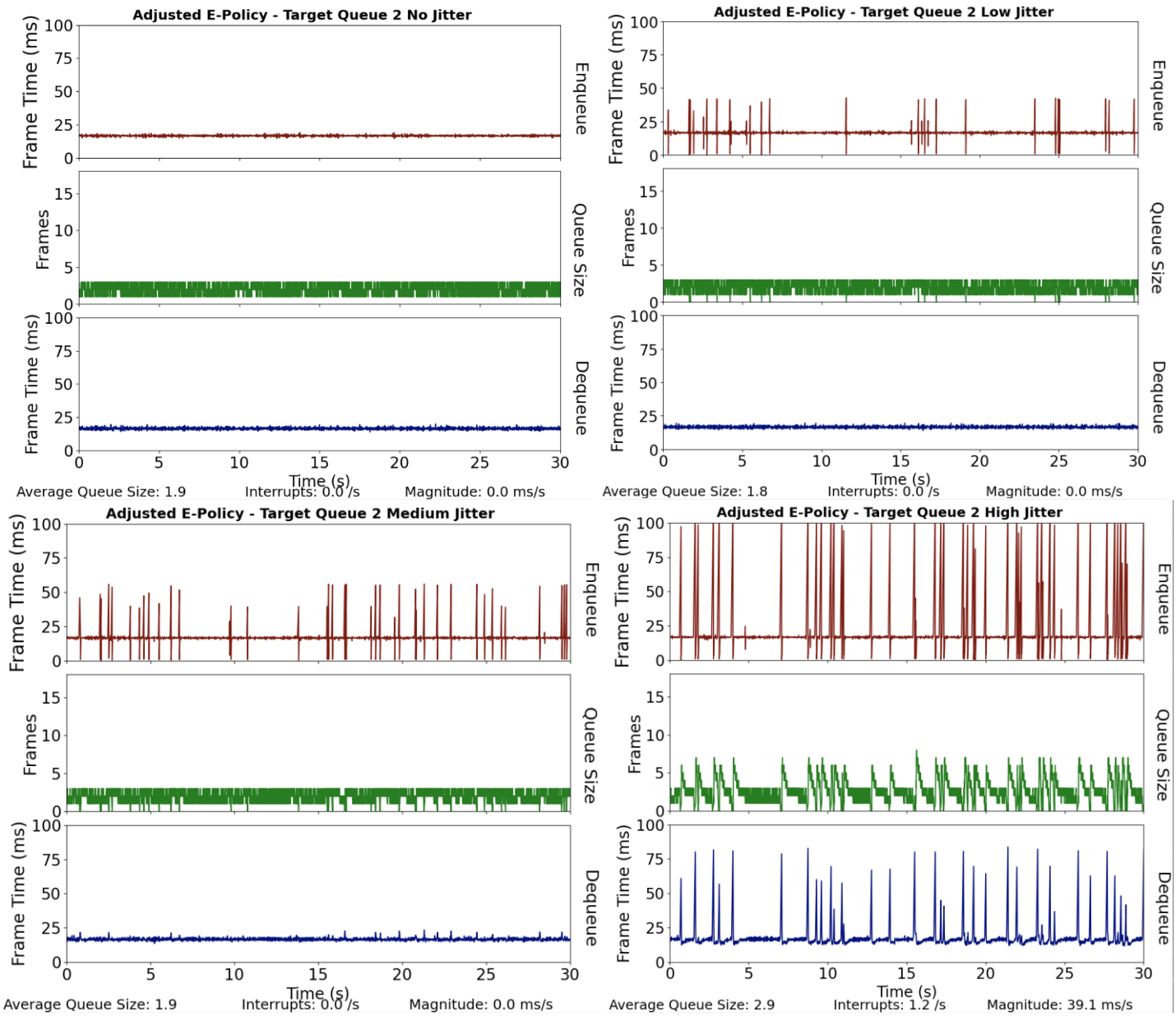

Figure 9: Adjusted E-Policy Target Queue 2 dashboard at different delay jitter settings

Figure 9 shows the playout performance of adjust E-Policy with a target queue of 2 for

each delay jitter setting. With no jitter, the performance is similar to base Moonlight. However, at low and medium delay jitter, the playout is smooth compared to playouts in Figure 8. If we observe the dashboard in the bottom right of the Figure 9, we can see the presence of spikes in both enqueue and playout. The queue is not deep enough to handle large jitter events. Based on the queue size graph, during large jitter spikes, the queue is drained to zero. The queue is not set up to handle a jitter event where the magnitude is greater than two frame times. This creates the gap in playout mentioned earlier. Although it appears a queue size of two does not seem adequate, the magnitude of the interrupts are lower than the applied setting. The magnitude for the high delay jitter setting is 100ms while the resulting average magnitude is only 40ms/s for our policy. With a queue size of two, we can effectively mitigate all frame jitter in low and medium settings at values 25ms and 40ms. For high delay jitter, it can decrease the magnitude of the frame jitter, but cannot completely avoid it. We also performed 200 runs with this queue size, with 50 runs for each delay jitter. This data is displayed in Table 2. The data organization is the same as Table 1. Similar to the sample run for this policy, the system handles low and medium levels of frame jitter but cannot fully counteract high levels of frame jitter. Our adjusted E-Policy is working correctly based on our design and hypothesis.

Table 2: Summary statistics of 50 experiment runs for Adjusted E-Policy Target Queue 2

	None	Low	Medium	High
Average Queue Size (Frames)	1.88 (0.01)	1.83 (0.01)	1.89 (0.02)	2.81 (0.07)
Interrupts/s	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	1.08 (0.11)
Magnitude(ms/s)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	37.35 (2.82)

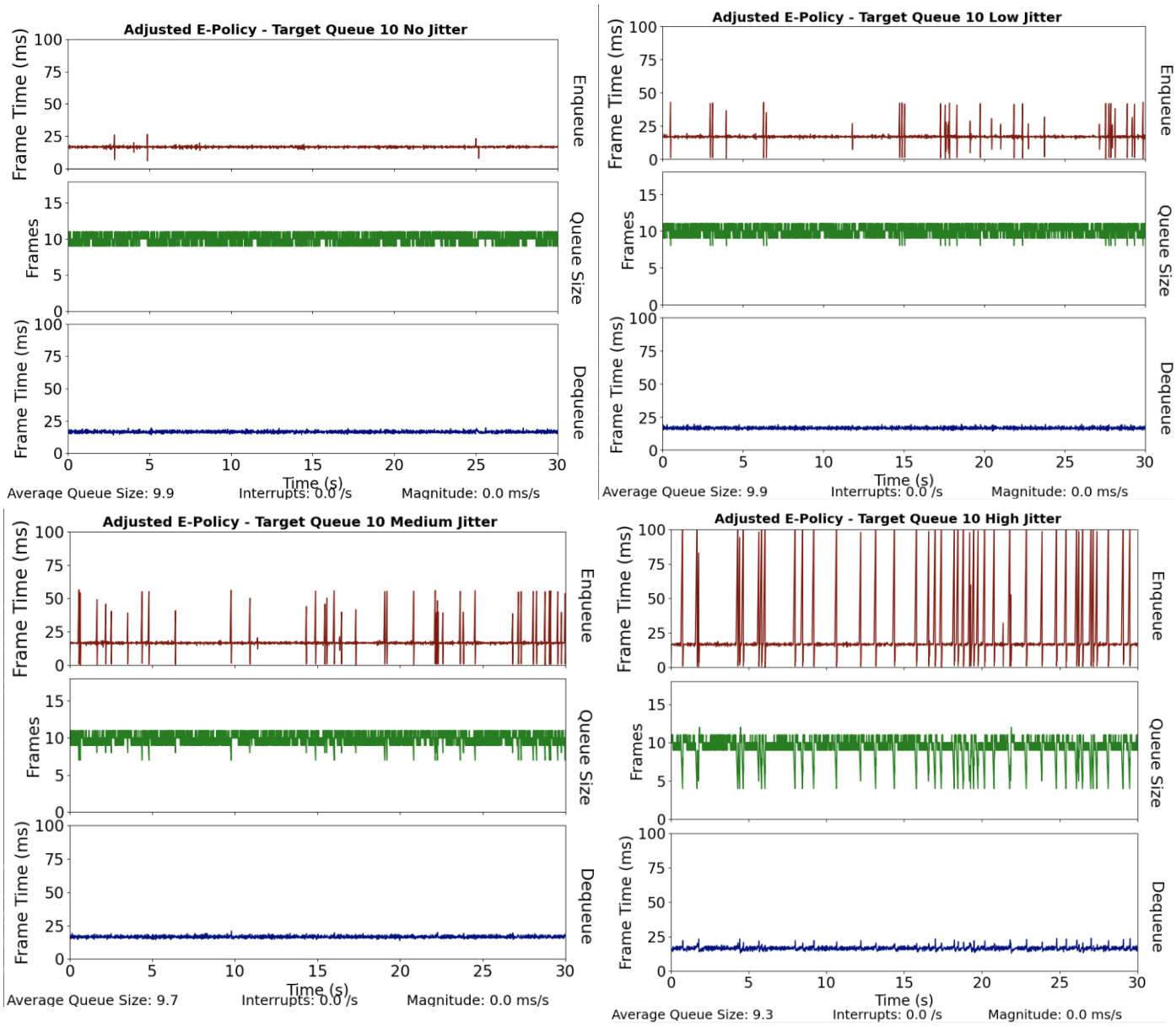

Figure 10: Adjust E-Policy Target Queue 10 dashboard at different delay jitter settings

4.1.4 Adjusted E-Policy - Target Queue Size 10

Here, with Figure 10, we are showing the playout performance of adjusted E-Policy with a queue size of 10. The performance from low to medium delay jitter is the same as with queue size of 2. For high delay jitter setting though, the larger queue size eliminated the chance of any frame jitter event from happening. The queue size graph shows that the queue does not drain to

zero but at four or five frames. This is expected because 100ms of delay jitter is about six frames. A deeper queue is more equipped to handle large delay jitter events. The results of 200 runs with this queue size are depicted in Table 3. The data organization is the same as Tables 1 and 2. This policy can successfully handle frame jitter in all conditions tested. These dashboards show our queue can control the smoothness of frame playout in a network with delay jitter.

Table 3: Summary statistics of 50 experiment runs for Adjusted E-Policy Target Queue 10

	None	Low	Medium	High
Average Queue Size	9.88 (0.01)	9.83 (0.02)	9.75 (0.02)	9.30 (0.05)
Interrupts/s	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
Magnitude(ms/s)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

We next show the results of 50 runs per policy-setting pair. This is to show that the policies are not only working once, but also showing statistically significant results.

4.2 Aggregate Performance Across 50 Runs

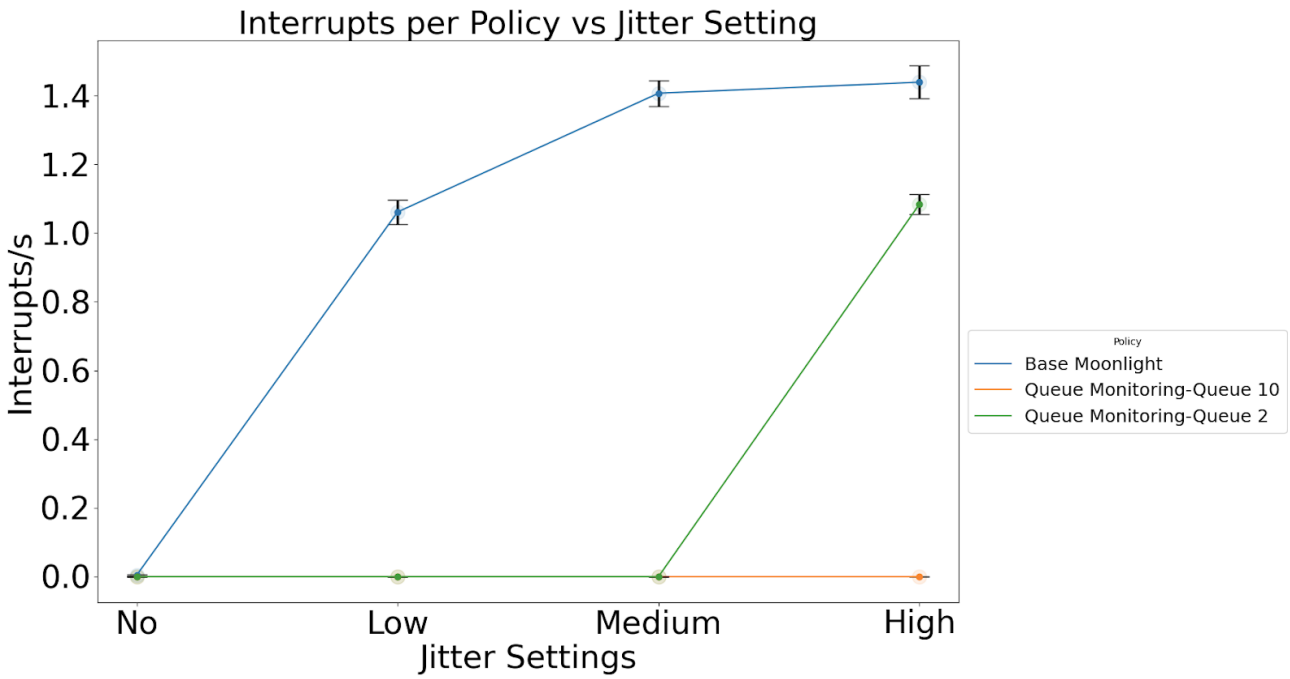

Figure 11: Interrupts/s for each policy at different delay jitter settings

After we finished the experiment, we calculated the average interrupts per second for each policy at the different delay jitter settings. We plotted the result with 95% confidence interval in Figure 11. As we have shown with the dashboards, the interrupts for base Moonlight steadily increase as we apply more intense delay jitter. Adjusted E-Policy with both shallow and deep queues can effectively compensate for frame jitter at low and medium settings. Only a large queue can adequately compensate for frame jitter caused by high delay jitter.

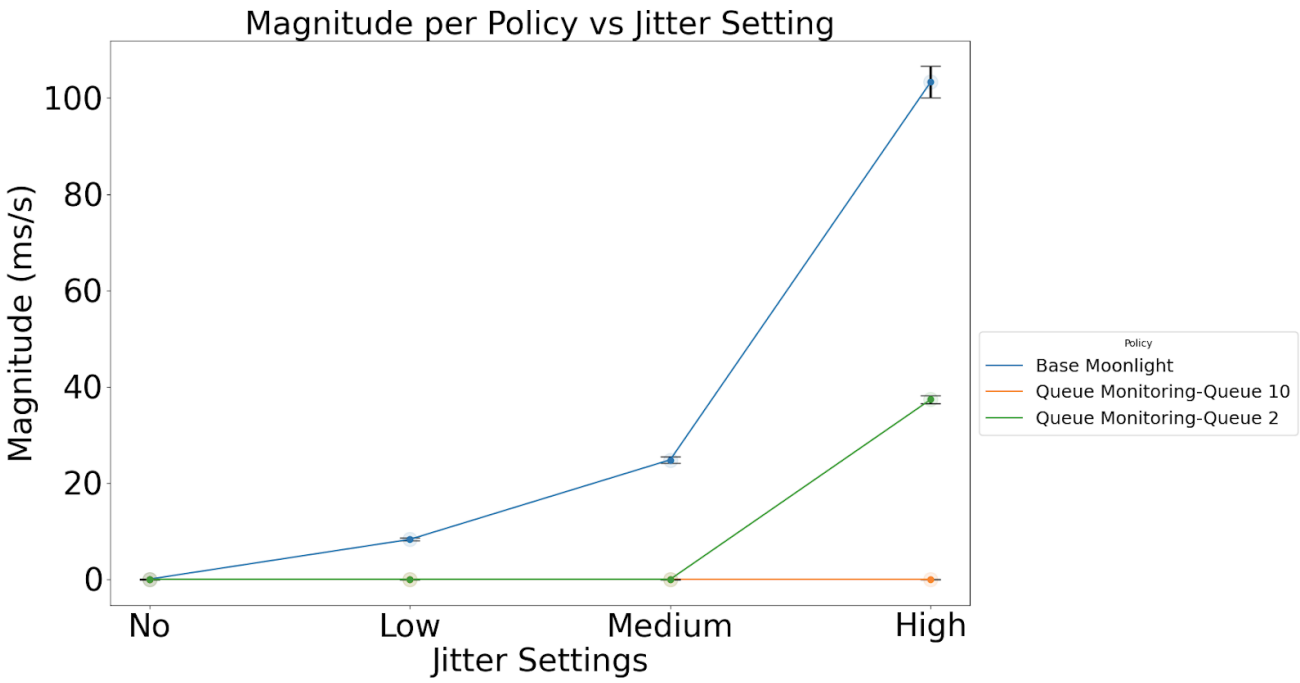

Figure 12: Magnitude for each policy at different delay jitter settings

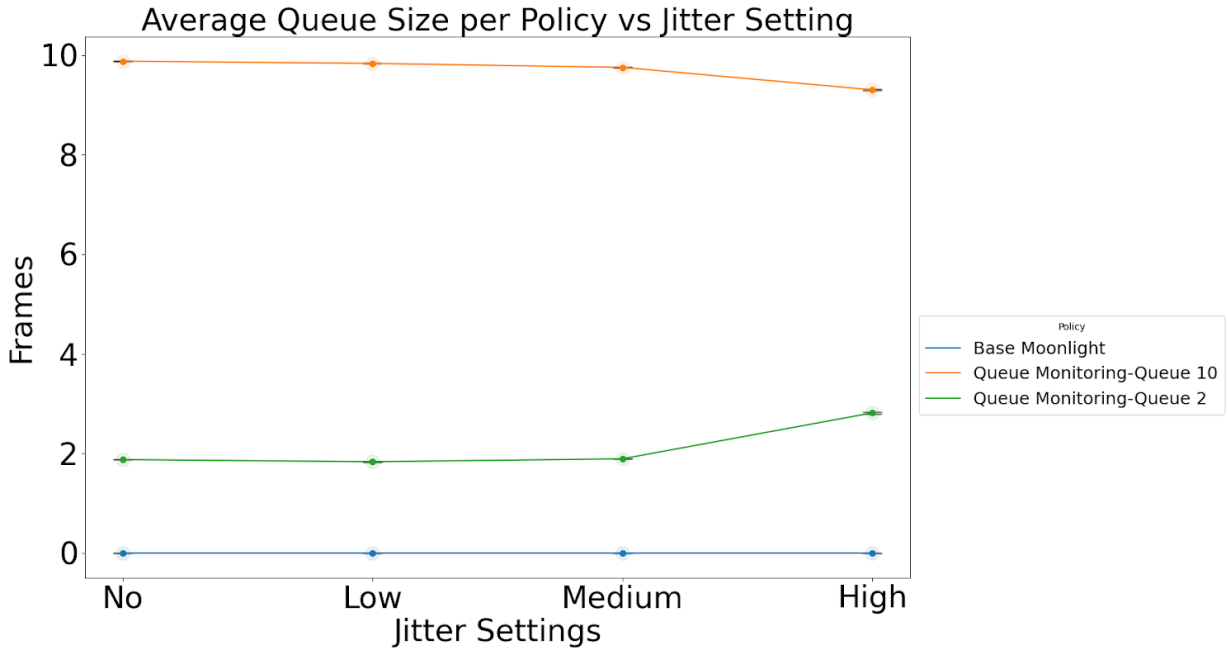

Figure 13: Average queue size for each policy at different delay jitter settings

We calculated the average magnitude at each interrupt in Figure 12. Since each frame inside the queue has a frame time of 16.67ms, each queued frame adds that much delay. Hence, we calculate the total delay added to the playout by multiplying frame time with queue size. For our experiment, we are adding an additional 33.34ms by implementing a queue of two and 166.7ms worth of delay for a queue size of ten. In Figure 13, we show the queue occupancy of each policy, and our adjusted E-Policy can maintain the specific target queue size for all settings. For a queue size of ten, the added delay may be preferable than large frame jitter spikes, but it also maintains that queue size for no frame jitter. With no frame jitter, this added delay will significantly reduce the quality of experience for the user. On the other hand, with the queue size of two, we are not adding as much delay, not affecting the experience nearly as much. The cost for handling frame jitter in our approach is adding additional delay to the playout, especially in a fixed target size queue.

4.3 Limitations

Our experiment has several limitations. First, due to technical difficulty and time constraints, we were unable to thoroughly test all the existing buffering algorithms. This limits our scope and only shows that an algorithm with a queue can improve the performance of Moonlight. In addition, our version of the queue monitoring in the experiment is incomplete. The goal of queue monitoring is maintaining the queue occupancy at the specified value by adjusting the playout rate to adapt to the delay jitter events. However, we have attempted multiple calculations for such calibration, but all have resulted in increasing or decreasing the rate too quickly and recovering too slowly. Ultimately we decided to use hard coded values in order to achieve the desired design. Our implementation is not a true adaptive version.

Next, as we mentioned in the last section, there is a cost of having a large queue. Since our queue size is determined beforehand, it would be hard to determine the best queue size without adding a large amount of delay. Additionally, we are simulating a network condition where the delay jitter is constantly at a specific level. In normal network conditions, we should be seeing varying magnitudes of delay jitter. The queue size must be able to handle every delay jitter condition in a single run. Finally, while we determined that the type of game we use does not have an impact on the performance of the algorithms, that only applies to benchmarking. Since we did not do a user study, we cannot fully evaluate the extent to which the added delay impacts user experience. Different types of games may require different types of buffering algorithms. We were able to see that adding a buffer can boost the performance of cloud game streaming in different delay jitter settings, but more extensive research is needed to find the ideal buffering algorithm.

5.0 Conclusion

Cloud-based gaming reduces the need for the player to invest in upgrading or purchasing expensive computer hardware. The cloud server handles the heavy rendering and processing while the client focuses on decoding frames and sending inputs. However, to be effective, cloud-based game streaming is reliant on the connection between the server and client. Delay jitter is the variation in latency between the server and client. Bandwidth jitter is resulted from the variance in the bandwidth that changes the amount of data transmitted. Both delay jitter and bandwidth jitter can cause frame jitter. Frame jitter is the variance during playout which causes interrupts during the stream and degrades the Quality of Experience of the gaming session. It is necessary to develop a way to deal with the different types of jitter events to achieve better Quality of Experience.

Our work primarily focused on handling frame jitter caused by delay jitter. Since delay jitter causes the transmitted frame to arrive late at the client, it results in frame jitter. We implemented a buffer at the client side to store frames. With this approach, in the event where a delay jitter happens, we still have frames to play and the stream remains smooth. The buffer drains when a delay jitter event occurs and fills it back up when it is over. When sufficiently large, the buffer can absorb all magnitudes of delay. This solution comes at the cost of introducing additional delay into the system, however, which decreases playability and this delay will cause the frame to be displayed later than when it was rendered on the server. It also makes player input slower and not matching the most up-to-date event in the game.

Our experiments show adjusted E-Policy, with a target queue size of two, can alleviate frame jitter in low and medium jitter settings. While it cannot fully handle a high jitter environment, the algorithm still reduced the magnitude of the jitter by over half compared to the magnitude of jitter applied, keeping the stream more smooth. We needed a queue with a target size of ten to fully smooth out the playout in a high jitter environment. Despite the smooth playout, this comes at a cost of added delay to the system, an additional 16.67ms worth of delay per frame.

6.0 Future Works

This chapter describes further research that could expand on our project.

6.1 Imperfect Average Calculation

All of our algorithms included a weighted moving average calculation that was supposed to converge towards a playout rate that would keep the queue in a stable state. However, when left unchecked, the value would always be more than required and increased to the point of extreme latency. Our solution to this was implementing a constant offset value that kept the playout stable. This offset value was difficult to manage, and not very precise; adjusting the playout rate naturally would likely have made this process easier. We were unable to identify the source that causes excess growth in latency; further research could delve deeper into finding another way to calculate and measure the playout rate. This also includes finding other sources that could contribute to this issue. With the problem identified, an algorithm could be developed that relies only on a dynamic average calculation, and doesn't need hardcoded average values.

6.2 Moonlight Inconsistencies

At the beginning of the project, we assumed that Moonlight had no lag compensation techniques. However, we found this may not be true. As the queue gets higher, moonlight reduces the bitrate. At a certain point, the reduction of bitrate is large enough that the queue stabilizes, usually around 16-18 frames. This is mostly not relevant to our project, as our frame goals were always well under these values and our working implementations were never affected by this problem. We suspect that this could mean Moonlight has some metric for determining delay in playout, and lowers the bitrate so the stream can proceed. Another possibility is that the

16-18 frames are related to the ratio between dependent frames and a keyframe. Ultimately we are not sure of the cause, and replicating and understanding this behavior would be crucial for any future Moonlight research. Outlining and tweaking this system could be an additional source of stability for the system, or it could be a more flexible way of reducing lag than the algorithms we have developed.

6.3 Location of Buffer System

We chose to place the buffer towards the end of the Moonlight rendering pipeline. With this system, the frames were already assembled from their individual packets. However, there may be a better location to put our buffer structure. The Moonlight client has its own queue to take in information and assemble the frames; modifying this system may be a better place to construct a buffer. Additionally, throughout the project we had to deal with Moonlight quirks, including an inability to render with threads due to OpenGL being outdated. We suggest that future projects could find additional locations where the buffer can be placed and compare their performances.

6.4 Variable Queue Target

The majority of our work was spent tweaking our average calculation and queue monitoring to achieve a constant queue occupancy across runs. We tested the effectiveness of each under consistent jitter settings. However, in home use, the amount of jitter fluctuates due to varying internet status, and a better approach is to have the system adapt to that variance. There are several directions to tackle this idea. One way is to consider the amount of frames in the buffer, or the rate of change of frames. This could be used to analyze jitter conditions and set an appropriate queue size target. Alternatively, other metrics such as frame enqueue rate could be

used to calculate the magnitude of current jitter conditions and alter the queue size target. Further research could test one or both of these systems, and compare them against our system to see if an adapting system is better than simply choosing an adequate queue target.

6.5 User Studies

Our research was focused on how our system dealt with queue size, spike magnitude, and number of interrupts. Additionally, to limit variance, we tested our program on a static benchmark that performed roughly the same each run. Because of this, we have limited knowledge about how our system works under normal gaming conditions. Therefore, future research should consider testing our system and subsequent systems with user studies. Statistics to test include using multiple games, having different goals in each of the games, and getting a spread of gaming skill among the participants. For our specific system, it would be beneficial to test both differing levels of jitter, and different queue sizes. Measurement techniques could include in-game metrics such as points earned, and also user-based metrics such as a questionnaire asking how much lag affected gameplay. Ultimately any gaming-based jitter compensation should be thoroughly tested with users before being deemed effective.

7.0 References

Cai, Wei, et al. “The Future of Cloud Gaming [Point of View].” *Proceedings of the IEEE*, vol. 104, no. 4, Apr. 2016, pp. 687–91, https://doi.org/10.1109/JPROC.2016.2539418.

“Cloud Gaming: The Past, The Present And The Future.” *Gameopedia*, 18 Feb. 2022, https://www.gameopedia.com/cloud-gaming/.

De-Yu Chan. “Impact of Information Buffering on a Flexible Cloud Gaming System.” *Magda El-Zarki*, 2017, pp. 1–6, https://doi.org/10.1109/NetGames.2017.7991543.

GameTechDev. (n.d.). PresentMon (Version 2.00). GitHub. https://github.com/GameTechDev/PresentMon

Hassan Iqbal, et al. “Dissecting Cloud Gaming Performance with DECAF.” *Association for Computing Machinery*, vol. 5, no. 3, Dec. 2021, https://doi.org/10.1145/3491043.

Moonlight Game Streaming: Play Your PC Games Remotely. https://moonlight-stream.org/. Accessed 9 Apr. 2024.

Stone, Donald L., and Kevin Jeffay. “An Empirical Study of Delay Jitter Management Policies” *Readings in Multimedia Computing and Networking*, Elsevier, 2002, pp. 525–37, https://doi.org/10.1016/B978-155860651-7/50131-5.

Sunshine Documentation. https://docs.lizardbyte.dev/projects/sunshine/en/latest/. Accessed 15 Sept. 2023.

8.0 Appendices

Appendix A

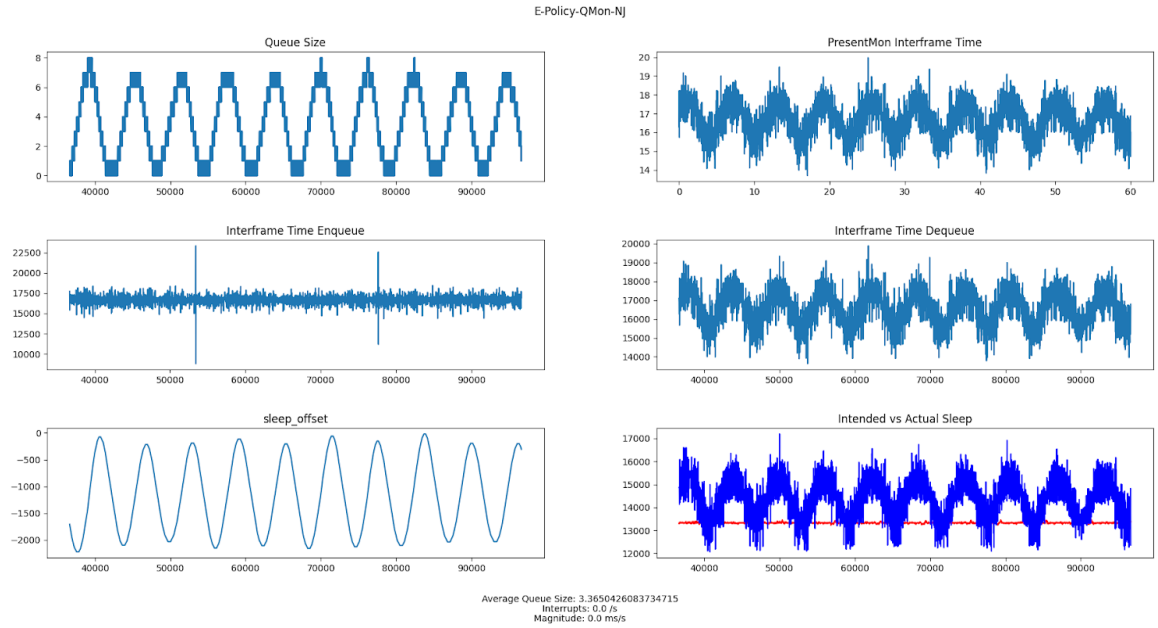

Figure 14: E-Policy Dashboard from Adaptive Queue Monitoring

In Figure 14 we present a more detailed dashboard showing the different recorded metrics with PresentMon's capture. Specifically, the graph for queue size, sleep offset value, which is the adjustment we make, and the frame times are all wavelike. These are produced from our attempts at an adaptive queue monitoring algorithm. We tried to alter the playout rate based on the difference between the queue target and the current queue size. However, the adjustment adapted far too quickly and caused our queue to overshoot the target, and the same thing would repeat when the queue was above its target. After multiple attempts, we were unable to find a mathematical calculation that would allow us to achieve the desired dynamic adjustment.

Appendix B

PresentMon is a capturing and performance analysis tool developed by Intel (GameTechDev, n.d.). PresentMon has two different versions; one is a user interface and the other is an executable. During early testing, we used the interface version. The interface is nicely designed but starting and ending capture needs to be done manually through hotkey. We found this to be inconsistent and when graphing, we would notice a massive spike that resembles a frame jitter. That spike is due to us ending the stream and the capture. To avoid this problem, we switched to using the executable version as it does not have the described problem. The executable version also adds more customization options such as altering the capture duration, file name and location, and modifying the result CSV it produces. The customization also allows easy organization through specifying the file location and name. We heavily suggest spending time on reading the PresentMon documentation and testing with it.

Another important note is that the executable version may not have privileges to access the running applications, and PresentMon will not be able to log data. There are two methods to handle this error. One is going through the group policies and editing the permission. The other approach is running the script with administration, which is what we did. By running our experiment script inside a command prompt opened with the administrator, it will not have a permission error.