Scheduling Policies for 360° Video Servers

Ben Civjan
University of Illinois at Urbana-Champaign
Champaign, Illinois
bcivjan2@illinois.edu

Abstract

In the rapidly evolving field of multimedia technology, 360° videos have emerged as a groundbreaking format that offers immersive experiences. However, it poses significant challenges for video servers, such as high bandwidth demand, large video size, and heterogeneous user requirements. In this paper, I propose two novel scheduling policies for 360° video servers that aim to optimize the quality of experience (QoE) for users while minimizing the server cost. The instructor/observer policy takes into account different user roles to more effectively prioritize content distribution. The field-of-view policy leverages the viewing angle of the user to determine which threads to prioritize based on whether "content of interest" is being viewed.

I evaluate these algorithms by implementing a HTTP video server and user-space thread scheduler to conduct a series of experiments to evaluate average thread runtime. I conclude that if clients seek to download the video as fast as possible, it is beneficial to choose a large base time quantum. The field-of-view scheduling policy works best when the video's horizon is of interest to many users. But when seeking to reduce the download time of high-priority clients, the instructor/observer policy with a high instructor multiplier is the best option.

1 Introduction

360° videos have dramatically increased in popularity over the last few years largely due to advances in immersive technologies like virtual reality (VR) and augmented reality (AR). Major video platforms like YouTube, Meta, and Netflix have put efforts into promoting these services. Most current applications are related to gaming and entertainment, but there are applications in areas such as education, immersive telepresence, sports, and others.

Scheduling, in the context of threads, is the process of deciding which thread should execute or get a resource in the operating system. The scheduling of threads can be done using different policies, such as round robin (RR), first come first serve (FCFS), shortest job first (SJF), longest job first (LJF), and priority scheduling. These policies have different objectives, such as maximizing throughput, minimizing latency or response time, or maximizing fairness[3].

Virtual threads, or Green threads, are a type of thread that is managed by a program in user-space, rather than by the operating system. They are not native to the platform, but rather implemented by a runtime library or a virtual machine. Virtual threads can be used to create and run multiple concurrent tasks in a program, without relying on the OS's support for threads. A key advantage of virtual threads is that they can be completely managed and scheduled in user-space without needing to make modifications to the OS kernel. Virtual threads can be scheduled to run inside one or more OS threads.

2 Related Work and Motivation

There are various strategies that have been developed to mitigate the higher resource demands of 360° video. A simple way to stream 360° video is by using viewport-independent streaming. With viewport-independent streaming the entire omnidirectional frame is sent to the client at a constant level of quality, similar to how a standard 2D video would be streamed. This can be desirable due to its ease of implementation, but suffers from worse performance than optimized techniques [11]. An optimized version of this strategy is viewport-dependent streaming. In viewport-dependent streaming, the client only receives a subsection of the frame corresponding to the current viewing direction. This subsection is equal to or greater than the viewing angle of viewport[11][13] and is more efficient than sending the entire frame.

Another common strategy for managing bandwidth in 360° videos is tiling. This technique involves dividing a video into spatial sections, enabling the transmission of specific sections to end users independently. The focus is on streaming high-resolution format only for the tiles within the user's field of view, allowing for the delivery of other tiles in lower quality or even omitting transmission entirely[12] [14].

Work has been done on projection schemes, encoding, adaptive 360° video streaming schemes, and networking strategies for 360° video [11]. However, limited investigation and work has been done on the management of processes within 360° video servers. There is a need for novel scheduling algorithms that can address the challenges associated with serving 360° video and optimize the performance of such servers. These algorithms should be able to improve the QoE and user satisfaction, as well as enhance the scalability and the robustness of the 360° video server.

3 Novel Scheduling Policies

Scheduling policies are crucial for ensuring fair and efficient allocation of resources in computing systems. In this paper,

I propose two novel scheduling policies for a video streaming application that take into account different heuristics to provide weights to different threads. The first policy, instructor/observer, assigns higher priority to users who act as instructors, i.e. those who might have multiple viewers of their screen, in order to provide guidance or feedback to other users. The second policy, field-of-view, assigns higher priority to users who are viewing the horizon of the video, i.e. those who may be viewing a more interesting perspective of the scene.

3.1 Instructor/Observer

One of the applications of 360° video is for educational or instructional purposes. Consider the following scenarios of video streaming:

- There is a teacher in a classroom with several students. The teacher is attempting to show the students what the inside of a surgery room looks like, but due to time constraints and patient privacy laws they are unable to get physical access to a hospital. The teacher instead opts to run a demonstration with a pre-approved immersive omnidirectional video. The students have the option to follow along on their own laptops in case they want to view different parts of the room that the teacher isn't currently looking at.
- A fire department is running a training based on 360° footage captured during a real incident. The training session instructor is streaming the video on a large screen to show what the fireman in the video was looking at. The firemen in the training session stream the same video to follow along with the instructor while being able to view sections that aren't currently in the field-of-view that is being displayed on the big screen.

In both situations, there are two clear roles: an instructor and an observer. If all these videos are being sent from the same video server, the more observers who log on to stream the video, the less resources will be allocated to the instructor. As a result, if too many observers are present it may cause the instructor's video to buffer and cause a worse QoE for all the learners.

The goal of the instructor/observer policy is to make sure that the worker threads on the server receive the appropriate amount resources based on their role. To enforce this, an instructor multiplier, α, is introduced in the following manner. Note that $\delta :=$ *Unit Time Quantum*.

$$\begin{aligned} \delta_i' &= \alpha * \delta \\ \delta_o' &= \delta \end{aligned} \tag{1}$$

The parameter α determines the weight assigned to an instructor thread in the scheduler. A larger value means that the instructor thread receives a larger execution time quantum compared to an observer thread. If α is too large then the observer threads could be starved of resources. If α is small then the instructor threads will not experience a performance boost compared to the observer threads.

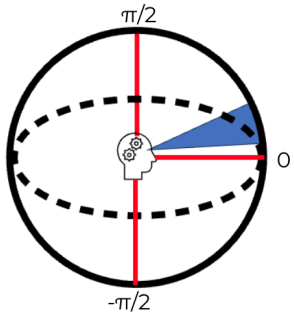

Figure 1. Viewing Position Range

3.2 Field-of-View

The field-of-view scheduling policy is based on the vertical viewing position of the user. It rewards users who are looking closer to the horizon of the video (i.e. vertical viewing position of 0°) by awarding them a larger time quantum in the scheduler.

The viewing position x-axis ranges from $[0, 2\pi]$ and the y-axis ranges from $[-\frac{\pi}{2}, \frac{\pi}{2}]$, as shown in Figure 1. Note that $\delta :=$ *Unit Time Quantum*, $v_y :=$ *Vertical Viewing Position*, and $v_y \in [\frac{-\pi}{2}, \frac{\pi}{2}]$. The field-of-view multiplier, β, will equal 1 when $|v_y| = \frac{\pi}{2}$ and will equal $1 + 2\pi$ when $v_y = 0$.

$$\begin{aligned} \beta &= (1 + 2\pi) - |4 * v_y| \\ \delta_i' &= \alpha * \delta \\ \delta_o' &= \delta \end{aligned} \tag{2}$$

3.2.1 Assumptions. The major assumption of this policy is that the most interesting events in the video occur close to the horizon. A straightforward example of this is the video recorded by Google street view. The objects of interest (buildings, people, cars, etc.) would all appear near the horizon of the video. Less interesting parts of the video, like sky or sidewalk, would appear towards the vertical poles of video. In cases like this, it would make sense to prioritize clients who are observing with a 0° viewing angle.

However, not all videos follow the same pattern. This is a limitation of the field-of-view policy. Possible improvements and solutions to overcome this assumption are discussed in the *Future Work* section.

4 Design and Implementation

In order to examine the performance of the novel scheduling policies described above, I had to implement a few key components. The first was a user-space thread scheduler that uses virtual threads. Generally, the scheduling of operating-system level threads is outsourced to the OS kernel

itself. This makes it impossible as a user-space application developer to have fine-grained control over the allocation of computing resources to individual threads. The second was a HTTP video server that could accept incoming requests from a client and spawn virtual threads to read and send video byte chunks. The server uses the native TCPStream library in Rust to implement parts of the HTTP byte chunking protocol.

4.1 Client and Server Setup

The first step in the development of the 360° video server was choosing an appropriate 360° video client. I settled on *Simple 360 Video Player* created by Sean Lawrence[7]. This software assumes that the received video is a series of complete omnidirectional frames (i.e. no tiling involved). It takes the omnidirectional video frames and maps them into a smaller 2D field-of-view that the user can maneuver using their arrow keys. A demonstration of the video player is available in the Github repository.

To get the client working with my server, I simply updated the HTML5 video tag's *src* attribute to point at my server's IP address. This instructs the browser to use HTTP byte-range requests to retrieve the video from the *src* IP address. It is worth noting that all network requests are handled through the browser's implementation of the HTML5 video tag, which is closed source.

Next, I created a TCP listener using Rust's native TCPStream library. Since the client uses HTTP, I perform some parsing to collect the protocol headers and fields. Once the request has been determined to be valid, the server sends a valid HTTP response header and begins streaming video bytes to the client.

The video bytes are sent in the following manner:

1. Attempt to read 1MB chunk from the video file into a buffer
2. If no bytes were read (i.e. video is done sending), finish the operation
3. If the TCP stream is valid, write the 1MB chunk to the stream
4. If thread execution time is greater than scheduled time quantum, yield the thread. Otherwise continue from Step 1.

4.2 Developing the scheduler

The purpose of a scheduler in the context of a 360° video server is to control the run-time of threads that are serving video content to clients. The three most necessary abilities of the scheduler are to pause/resume the execution of a thread, keep track of the thread execution time, and manage the order of thread execution.

Since all the algorithms that I wanted to implement were variations of non-preemptive round-robin, the scheduler could use a simple FCFS queue to manage the order of thread execution. To keep track of thread execution time, I simply get the current time when the thread starts executing and check the total elapsed time after each iteration of the *WriteChunk* loop.

The most challenging component to implement was the pausing and resuming of executing code to actually enforce the scheduling policy. To do so I took advantage of Rust's concurrency model: Async [1]. Async Rust allows the programmer to asynchronously wait on functions in a way that looks like synchronous code. The Rust language offers the keywords `async` and `.await`, which transform a block of code into a state machine that implements a trait called `Future`.

A synchronous method that calls a blocking function will stop the whole thread from running, but blocked `Futures` will give up the thread's control so that other `Futures` can run. A `Future` must implement a function `poll`.

Rust `Futures` are lazy and won't do anything unless actively driven to completion by an executor. At a high level, the `poll` function attempts to resolve the `Future` into a final value. If the value is not ready, the task is scheduled to be woken up when it can make more progress. It either returns `Poll::Pending`, meaning the `Future` is not ready to continue executing, or `Poll::Ready` meaning the `Future` is ready to return its result. In order to poll futures, an executor must create a `Waker`. A `Waker` is in charge of telling the executor that the task is ready to progress when the `wake` function is called. A simple `Waker` implementation could be to just put its associated task back on a queue. [1]

I exploit this state machine behavior to implement a user-space thread scheduler. First, I create a simple executor which contains a FCFS ready queue. The executor's implementation of `wake` is to put the caller on the back of the ready queue. When running, the executor takes tasks off the ready queue and polls them.

I then create a simple `Future`, called `YieldNow`, where on the first call to `poll`, it calls `wake` and returns `Poll::Pending`, and on subsequent calls returns `Poll::Ready`. This means that when asynchronous functions `.await` on `YieldNow`, the code execution is stopped at that point and the task is placed on the back of the ready queue in the executor.

This is all that is needed to create a user-space scheduler in Rust. When a new connection is created, a new task is spawned with code described in Section 4.1 and the thread is yielded by awaiting on the `YieldNow` future.

5 Evaluation

5.1 Methodology

The goal of the evaluations are to determine the effectiveness of each scheduling policy in relation to one another.

Due to the limitations of simultaneous connections to the same origin from a browser, the benchmarking had to be completed independently of a client. I created a simple

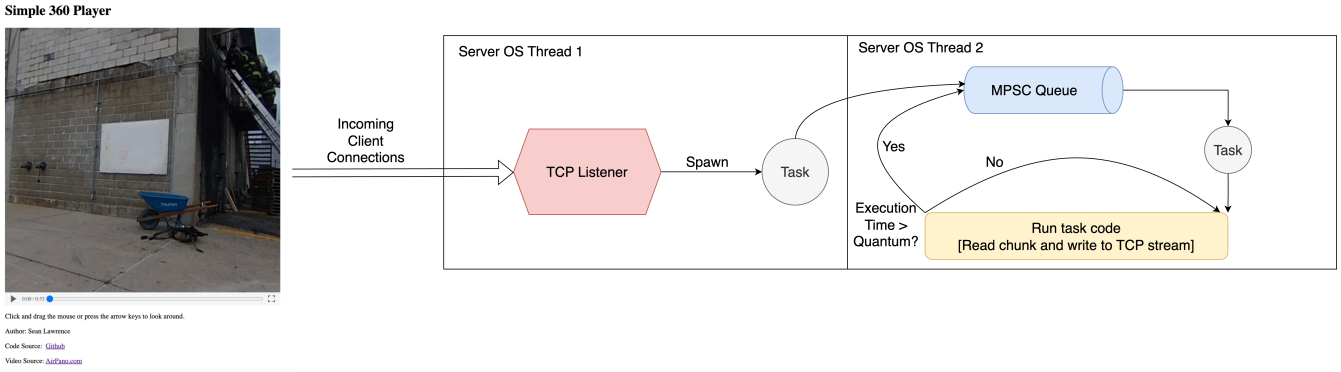

Figure 2. Server Architecture

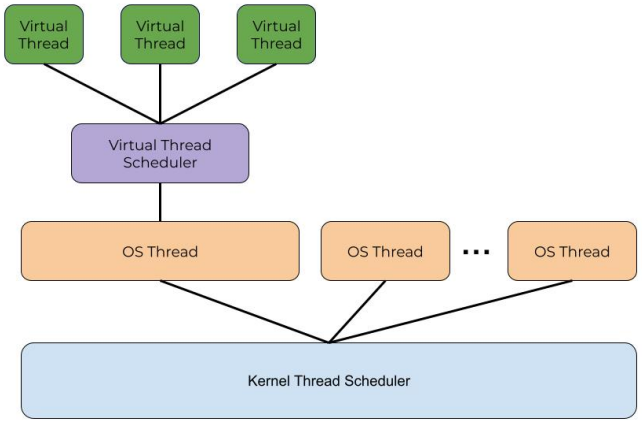

Figure 3. Virtual Threads within OS Threads

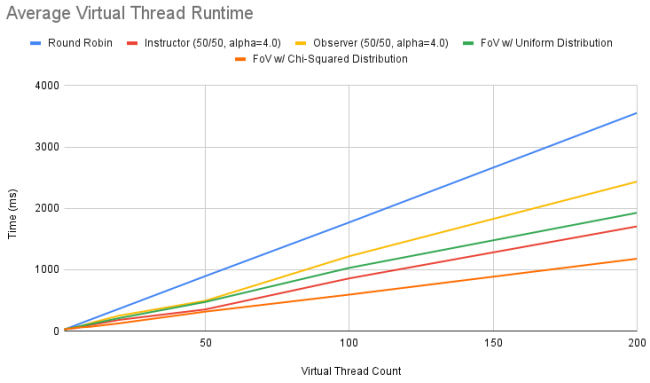

Figure 4. Virtual Thread Benchmarks

wrapper function around the read/write loop (described in Section 4.1). The wrapper function, in addition to running the read/write loop, times its total execution time and writes the execution time to a thread local array. Each index of the thread local array corresponds to a virtual thread index. To create a simulated client a new virtual thread is spawned with the wrapper function, a unique thread id, and execution time quantum. The time quantum determines which algorithm is being tested. Since there is no actual TCP connection created in the experiments, the primary computation cost in each thread is the reading of bytes from the video file. No data is actually sent over TCP in the benchmarks.

All experiments were conducted on a 2023 MacBook Pro with the 12-core Apple M2 Max processor and 32 GB of RAM. Each experiment was conducted using a video file of 359 MB, chunk size of 1 MB, and unit time quantum of 4 ms.

5.2 Assumptions

I define the runtime of a thread as the amount of time that elapses between when the thread first begins executing and its completion. This includes both the execution time and the periods of time spent on the ready queue. This does not include the time spent on the ready queue *before* the thread begins execution.

The threads in all graphs refer to the *virtual threads*, not OS threads. The virtual threads and OS threads have a hierarchical relationship, as shown in Figure 3. In all experiments the virtual threads were spawned within a single OS thread. The scheduling of the OS thread is completely handled by the operating system kernel and could be preempted at any point by another OS thread if the kernel chooses to do so. It is also important to note that the total runtime of a thread does not mean that a client will not receive any data before that point; rather the runtime signifies that the client will have *finished* receiving data at the end of the time period.

5.2.1 Round Robin. The independent variable was the number of spawned virtual threads, and the measured variable was average thread runtime.

5.2.2 Instructor/Observer. The independent variables were the number of spawned virtual threads, ratio of instructor to observer threads (i.e. 50% instructor, 50% observer)

and α, the instructor multiplier. The measured variable was average thread runtime.

5.2.3 Field-of-View. The independent variables were the number of spawned virtual threads and the current viewing angle of the client. The measured variable was average thread runtime. In the field-of-view experiments, it was imperative to be able to simulate clients observing the video from different viewing angles. To accomplish this, I randomly sampled user viewing angles within a range of $[-\frac{\pi}{2}, \frac{\pi}{2}]$ from a uniform distribution and chi-squared distribution with 2 degrees of freedom [10].

A uniform sampling distribution will simulate clients having viewpoints spread equally across the entire length of the video. However, in practice it is likely that most clients will have a viewing angle that is closer to the horizon. The chi-squared distribution is right-skew, so there is a higher chance of sampling a value close to 0 (i.e. close to the horizon) and a lower chance of sampling higher values (i.e. close to vertical pole).

5.3 Round Robin Analysis

We can see the performance of the standard round-robin algorithm in Figure 4. The relationship between virtual thread count and average runtime is linear, as expected. Since we are using only a single OS thread to schedule the virtual threads, they are taken off the ready queue one at a time. If more threads are in the queue at once, all threads will have to wait longer to execute since there is no preemption. With 50 virtual threads the average execution time is 895 ms, with 100 virtual threads it is 1771 ms, and with 200 virtual threads it is 3555 ms.

5.4 Instructor/Observer Analysis

The two main independent variables that I tested with the instructor/observer policy were α, the instructor multiplier, and the proportion of instructor threads to observer threads. In Figure 4, the average instructor and observer times are shown using $\alpha = 4.0$ and an equal number of instructor threads and observer threads. Figure 5 includes 4 separate graphs of the average runtime of instructor/observer threads with both parameters being modified. I ran the policy with 50% instructor threads, 50% observer threads, and $\alpha = 4.0$, with total thread counts ranging from 2 - 200. With 40 total threads, instructors took 352 ms and observers took 493 ms. With 100 total threads, instructors took 858 ms and observers took 1219 ms. With 200 total threads, instructors took 1705 ms and observers took 2435 ms.

We can see in Figure 5 that adjusting the proportion of instructor threads to observer threads has an effect on the average runtime. In the case where I use 20% instructor threads and 80% observer threads, 100 total threads took instructors 659 ms and observers 1559 ms. With 200 total threads it took instructors 1308 ms and observers 3116 ms. This was a 24% decrease in instructor runtime and 28% increase in observer runtime. We can conclude that a lower concentration of instructor threads will result in faster instructor thread runtime but slower observer thread runtime, and visa-versa. This makes intuitive sense because when there are less instructor threads, their respective weights are a larger percentage of the total weight on the scheduler.

We can also see in Figure 5 that modifying α has a dramatic impact on the runtime of the threads. If we compare the experiments with a 50/50 distribution of instructor threads and observer threads, the $\alpha = 6.0$ policy outperformed the $\alpha = 2.0$ policy in both instructor thread runtime and observer thread runtime. With 100 threads, $\alpha = 2.0$ instructor threads ran on average 1310 ms and observer threads ran 1581 ms. With 100 threads, $\alpha = 6.0$ instructor threads ran on average 140 ms and observer threads ran 897 ms. Thus, there is about an 89% decrease in instructor thread runtime and about 43% decrease in observer thread runtime when using $\alpha = 6.0$ instead of $\alpha = 2.0$.

5.5 Field-of-View Analysis

The field-of-view results are highlighted in Figure 4. The uniform distribution incurred average thread runtimes of 474 ms with 50 threads, 1029 ms with 100 threads, and 1927 ms with 200 threads. The chi-squared distribution incurred average thread runtimes of 316 ms with 50 threads, 593 ms with 100 threads, and 1178 ms with 200 threads. In other words, field-of-view was 50% slower with 50 threads, 73.5% slower with 100 threads, and 63.5% slower with 200 threads when using a uniform distribution instead of a chi-squared distribution.

5.6 Relationship between Runtime and Quantum

One similarity between all of the scheduling policies is that the larger the time quantum per thread, the shorter the average total thread time is. We see this phenomenon in the field-of-view experiments with different sampling distributions as well as in the instructor/observer when changing α.

As stated in the *Field-of-View Analysis* section, sampling viewpoints from a uniform distribution resulted in at least 50% slower runtime compared to sampling from a chi-squared distribution. This large separation is likely due to the fact that sampling from the chi-squared distribution will yield more viewpoints close to 0°, or the horizon. In the field-of-view scheduling policy this results in threads receiving larger weights and thus receiving a larger execution time quantum in the scheduler. Threads with a larger quantum will finish in less total trips around the ready queue, which reduces wait time.

Similarly, when increasing α in the instructor/observer policy we see a dramatic decrease in thread runtime. Since $\textit{instructor quantum} = \alpha * \textit{base quantum}$, selecting a larger value of α will result in proportionally larger instructor time

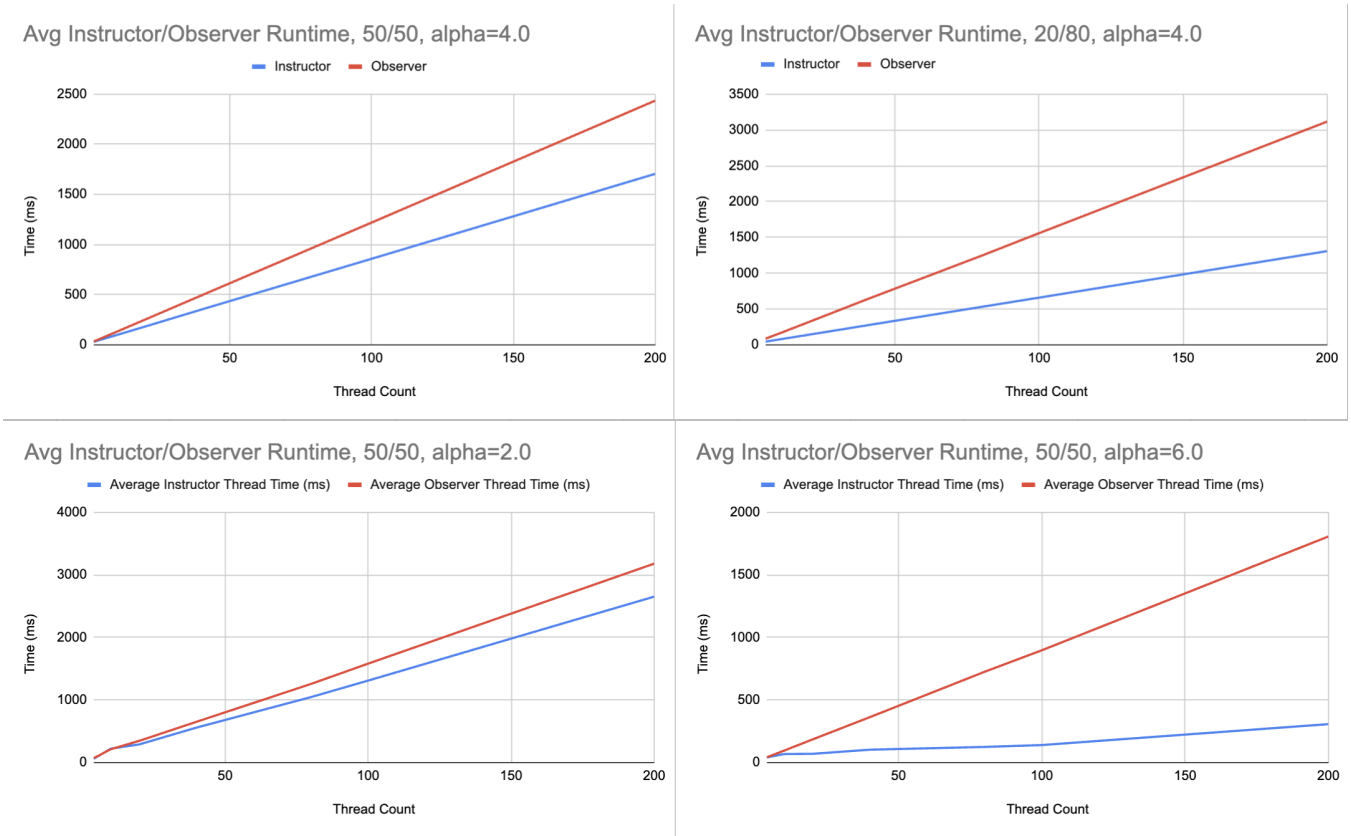

Figure 5. Instructor/Observer Virtual Thread Benchmarks

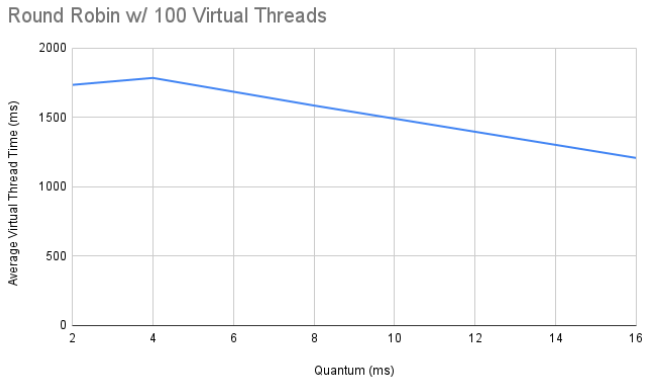

Figure 6. Virtual Thread Benchmarks with Varied Quanta

quantum. With larger quanta, instructor threads will finish in less trips around the ready queue. Finishing in less trips around the queue eliminates wait time and frees up resources on the scheduler. This will not only reduce the runtime of the instructor threads but also will reduce the runtime of the observer threads, because once the instructor threads complete the observers will have less competition for resources.

To verify this claim, I ran additional tests using the round-robin policy with 100 virtual threads and a varied time quantum (as shown in Figure 6). There is a clear downward trend in runtime as the time quantum increases. With 100 virtual threads and a quantum of 2 ms the average runtime was 1735 ms and with a quantum of 16 ms the average runtime was 1208 ms. In other words, there is about a 30% decrease in runtime when changing the time quantum from 2 ms to 16 ms.

6 Conclusion

The deliverables of this project are a prototype 360° video server that is able to serve video to browsers over HTTP [4], a user-space thread scheduler implemented in Rust [4], and implementations of *round-robin*, *instructor/observer*, and *field-of-view* scheduling policies. Using these implementations I performed in-depth experiments and analysis of each policy.

Through these experiments, I found that the field-of-view policy had the fastest average thread runtime, executing 200 virtual threads with an average runtime of 1178 ms. I found that in the same experiment the instructor/observer policy with $\alpha = 6.0$ had the fastest runtime for instructor threads (307 ms) but was slower that field-of-view for observer threads (1806 ms).

I conclude that, if prioritizing total video download speed, it is best to select a large base time quantum and choose algorithms that will produce larger time quanta per thread. If it is anticipated that many users will be looking toward the horizon of the video, then the field-of-view scheduling policy will have the best performance. However, in situations where minimizing the runtime of high-priority threads is important, the instructor/observer policy with a high value of α will have the best performance.

6.1 Future Work

6.1.1 Latency Analysis. Immediate future work I see for this project is examining the thread latency (i.e. the time between thread creation and first execution time). I predict that there will be a trade-off when using a larger thread quantum, which will likely result in increased latency. However, additional experiments would need to be conducted to verify this claim.

6.1.2 QoE Analysis. More analysis should be conducted into which actual thread characteristics (runtime, latency, etc.) have the largest effect on user QoE. As the saying goes, "there is no such thing as a free lunch", and maximizing one characteristic will likely come at a higher cost to others. Determining which are most impactful to QoE will allow for more concrete conclusions to be made about the best overall scheduling algorithms to use for 360° video servers.

6.1.3 M:N Thread Distribution. To boost the performance of the scheduler, rather than spawn all virtual threads within a single OS thread, it is possible to distribute the virtual threads among multiple OS threads. In other words we can multiplex M virtual threads to N OS threads. This is how many asynchronous Rust runtimes work [2].

6.1.4 Item-of-Interest Scheduling. Additional scheduling policies related to 360° video could be implemented. One in particular that I consider an improved version of field-of-view is item-of-interest scheduling. Rather than assuming that the most interesting part of the video occurs at the horizon (or 0° viewing angle), one could preprocess the video to determine objects of interest. Objects of interest would be sections within the frame of the video that are important to highlight or that user viewpoints gravitate towards. Based on the user's current viewing distance from an object-of-interest, they could receive a larger/smaller time quantum in the scheduler.

References

[1] [n. d.]. *Asynchronous Programming in Rust.* https://rust-lang.github.io/async-book/02_execution/04_executor.html

[2] async.rs team. [n. d.]. *async-std.* https://async.rs

[3] G. Chitralekha. 2021. A Survey on Different Scheduling Algorithms in Operating System. In *Computer Networks and Inventive Communication Technologies*, S. Smys, Ram Palanisamy, Álvaro Rocha, and Grigorios N. Beligiannis (Eds.). Springer Nature Singapore, Singapore, 637–651.

[4] Ben Civjan. 2023. Video Scheduler 360. https://github.com/bencivjan/video-scheduler-360

[5] A. Dan, D. Sitaram, and P. Shahabuddin. 1994. Scheduling Policies for an On-Demand Video Server with Batching. In *Proceedings of the Second ACM International Conference on Multimedia* (San Francisco, California, USA) *(MULTIMEDIA '94)*. Association for Computing Machinery, New York, NY, USA, 15–23. https://doi.org/10.1145/192593.192614

[6] Ching-Ling Fan, Wen-Chih Lo, Yu-Tung Pai, and Cheng-Hsin Hsu. 2019. A Survey on 360° Video Streaming: Acquisition, Transmission, and Display. *ACM Comput. Surv.* 52, 4, Article 71 (aug 2019), 36 pages. https://doi.org/10.1145/3329119

[7] Sean Lawrence. 2017. Simple 360 Video Player. https://github.com/slawrence/simple-360-player

[8] Md Milon Uddin and Jounsup Park. 2021. 360 Degree Video Caching with LRU LFU. In *2021 IEEE 12th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. 0045–0050. https://doi.org/10.1109/UEMCON53757.2021.9666668

[9] D. Salomon. 2007. *Transformations and Projections in Computer Graphics*. Springer London. https://books.google.com/books?id=6CzuydFlMpwC

[10] Eric W. Weisstein. [n. d.]. *Chi-Squared Distribution.* https://mathworld.wolfram.com/Chi-SquaredDistribution.html

[11] Abid Yaqoob, Ting Bi, and Gabriel-Miro Muntean. 2020. A Survey on Adaptive 360° Video Streaming: Solutions, Challenges and Opportunities. *IEEE Communications Surveys Tutorials* 22, 4 (2020), 2801–2838. https://doi.org/10.1109/COMST.2020.3006999

[12] Alireza Zare, Alireza Aminlou, Miska M. Hannuksela, and Moncef Gabbouj. 2016. HEVC-Compliant Tile-Based Streaming of Panoramic Video for Virtual Reality Applications. In *Proceedings of the 24th ACM International Conference on Multimedia* (Amsterdam, The Netherlands) *(MM '16)*. Association for Computing Machinery, New York, NY, USA, 601–605. https://doi.org/10.1145/2964284.2967292

[13] Chao Zhou, Zhenhua Li, and Yao Liu. 2017. A Measurement Study of Oculus 360 Degree Video Streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference* (Taipei, Taiwan) *(MMSys'17)*. Association for Computing Machinery, New York, NY, USA, 27–37. https://doi.org/10.1145/3083187.3083190

[14] Junni Zou, Chenglin Li, Chengming Liu, Qin Yang, Hongkai Xiong, and Eckehard Steinbach. 2020. Probabilistic Tile Visibility-Based Server-Side Rate Adaptation for Adaptive 360-Degree Video Streaming. *IEEE Journal of Selected Topics in Signal Processing* 14, 1 (2020), 161–176. https://doi.org/10.1109/JSTSP.2019.2956716